CSC265 Homework 1
Fall 2021
Student Name: Haojun Qiu
Student Number: 1006900622

(I am so sorry that I have no enough time to shorten my solution. I will definitely avoid exceed page limit in next homework.)

Q1

- Data structure used to represent set defined in the question:
  A type of array that contains an attributes *length* that represents the length of this fixed-length array. Therefore, accessing the length of array takes constant time by accessing the variable length. Also, assume that accessing an element of an array takes constant time, via indexing.
- I will use a type of singly linked list that has an attribute $first$ - a pointer to the first element of the list, and a pointer *last* - a pointer to the last element of the list. Therefore, it can have an *append* method that does adding an element to the end of the list, only taking constant time. Implementation of *append*: 1) access the original end element in by accessing variable, and then has its *next* attribute points to the new appended element, 2) and then we make the attribute *last* of this list points to the newly appended element. We initialize such an empty linked list, by writing $newLinkedList()$.

A high level idea of my algorithm:
Our main strategy would be finding out intersection between $\{s\}$ and S' for every $s \in S$. And since S and S' are disjoint respectively, summing up those counters we gives the right result. We would use similar algorithm as merge in mergesort to implicitly "merge" all elements in S and S' such that they are sorted by y-intercept. It's implicitly merge because we don't actually need the merged collection, but the index of elements of S in the merged collection (if we actually accumulate that collection) would be sufficient. And when tie appears, we follow some sorting convention when noting down the position of element s in S. We do the same thing at the $x = 1$ side. And then, we take take the difference of position of s at $x = 0$ side and position of s at $x = 1$ side, this will be the number of intersection points between $\{s\} and S'$ (with some edged cases dealt below) since it represents the number of lines segments from the other set S' that starts lower (or higher) at $x = 0$ and ends higher (or lower) at $x = 1$ than s itself.

ORDER-LEFT-END(S, S')

```
1  i = 0
2  j = 0
3  L = new array of size S.length (i.e., n)
4  while i < S.length and j < S'.length
5        if S[i + 1].b < S'[j + 1].b
6              L.append(i + j + 1)
7              i = i + 1
8        elseif S[i + 1].b == S[j + 1].b
9              if S[i + 1].m + S[i + 1].b ≥ S'[j + 1].m + S'[j + 1].b // ..
10                   L.append(i + j + 1) // "normal" edge case
11             else L.append(i + j + 2). // "abnormal" edge case
12             i = i + 1
13             j = j + 1
14       else j = j + 1
15 if j ≥ S'.length      // S' is just exhausted
16       if S[i + j]
17       for r = i + j + 2 to S.length + S'.length
18             L.append(k)
19 return L
```

ORDER-RIGHT-END(S, S')

```
1  i = 0
2  j = 0
3  L = new array of size S.length (i.e., n)
4  while i < S.length and j < S'.length
5        if S[i+1].m + S[i+1].b < S'[j+1].m + S'[j+1].b
6              L.append(i + j + 1)
7              i = i + 1
8        elseif S[i+1].m + S[i+1].b == S'[j+1].m + S'[j+1].b
9              if S[i+1].b ≥ S'[j+1].b // ..
10                   L.append(i + j + 1) // "normal" edge case
11             else L.append(i + j + 2). // "abnormal" edge case
12             i = i + 1
13             j = j + 1
14       else j = j + 1
15 if j ≥ S'.length      // S' is exhausted
16       for r = i + j + 1 to S.length + S'.length
17             L.append(r)
18 return L
```

CALCULATE-INTERSECTING-PAIRS(S, S')

```
1  L_1 = ORDER-RIGHT-END(S, S')
2  L_2 = ORDER-LEFT-END(S, S')
3  res = 0
4  for k = 1 to S.length
5        res = res + ABSOLUTE(L_1[k] − L_2[k])
6  return res
```

Runtime Analysis of CALCULATE-INTERSECTING-PAIRS(S, S').

Let S and S' be deifned as in Q1 so that $|S| = n$, and $|S'| = n'$.

- Firstly, let's take a look at the runtime of the helper function call ORDER-RIGHT-END(S, S') at line 1. Lines 4 to 14 takes at most $O(n+n')$ since it in essence access elements in S and S' at every iteration with constant runtime body. And lines 15 to 17 takes at most $O(n)$ since it at most has n iteration given $j = S'.length = n'$ (this is justified latter, in lemma2), and every iteration takes constant time. All the other lines takes constant as well. So, together, it runs in $O(n + n')$ time.
- The same argument applies to the helper function call ORDER-RIGHT-END(S, S') at line 2 since the are almost identical runtime-wize. So, it takes $O(n + n')$ in runtime as well.
- The for loop from lines 4-5 runs $|S| = n$ iteration, each iteration takes constant time, so runs in O(n) time.
- Lines 3 and 6 takes constant time.

So, to sum up, CALCULATE-INTERSECTING-PAIRS(S, S') takes $O(n + n')$ in runtime, as required. ■

Q2

Lets' see an example first. If a collection S_L that contains all elements form S and S' sorted like $\langle s_1, s'_1, s'_2, s_2, s_3, \dots \rangle$, L_1 by its elements' y-intercept, then the returned value of ORDER-LEFT-END(S, S') should be $[1, 4, 5 \dots]$. We formalize it as a lemma. My implementation will much like "Merge" helper in the "MegerSort". However, we are not interested in merging them, but to note down the positions of elements in S in a bigger collection that contains all elements from both S and S', sorted by y-intercept.

We define a sorting convention for tie case first, facilitating the lemma 1 statement. I will be defining a few more sorting convention for tie, for some lemmas, but actually several of them has the same idea as this one (just with end points change).

Sorting Convention for Tie 1. *Let $s \in S$ and $s' \in S'$. If $s.b = s.b'$, then*

- *s appears ealiaer in the collection, if $s.m + s.b \geq s'.m + s'.b$; and*
- *s appears ealier in the collection if $s.m + s.b < s'.m + s'.b$.*

Lemma 1. *Suppose the input S and S' is as specified in Q1. The algorithm* Order-Left-End(S, S') *returns a list of length n, representing the positions of $\{s_1, s_2, \ldots, s_n\} = S$ (in-order) in an ordered combining list of elements S_L that contains all the elements in S and S', sorted in non-decreasing order by y-intercept, and when tie appears, it follows the sorting convention 1.*

Proof. To prove this lemma, we firstly a loop invariant for the from lines 4-15 of Order-Left-End(S, S'). - **Loop Invariant:** We denote the variable at the end of k-th iteration with a subscript k (e.g., L_k means L at the end of k-th iteration). We define the predicate $LI(k)$ as follows: If the loop runs at least k iterations, then

1. $i_k \leq n$
2. $j_k \leq n'$
3. L_k is a list of integers representing the positions of $\{s_1, \ldots, s_{i_k}\}$ in S_L described in lemma 1.
4. first i_k elements in S and j_k elements in S' has been visited (implicitly positioned) in S_L, (in the form of updating L_k so that it satisfies 3)
5. first i_k elements in S and first j_k elements in S' all has smaller or equal y-intercept to the elements in $S[i_k + 1 \ldots n]$ and $S'[j_k + 1 \ldots n]$.

I will show that for all $k \in \mathbb{N}$, $LI(k)$ holds, using simple induction. Initially, let $k = 0$, then $i_0 = 0 \leq n$ (i before entering the loop) and $j_0 = 0 \leq n'$. (3) and (4) are vacously true since $i_k = j_k = 0$. And that L_0 is an empty list. So, $LI(0)$ holds. Then, let $k \in \mathbb{N}$. We assume that $LI(k)$ holds, and show that $LI(k + 1)$ holds. Assume there is at least $k+1$ iterations, then so does k. So, part (1) - (5) of $LI(k)$ holds. We need to show that $i_{k+1} \leq n$, $j_{k+1} \leq n'$ and L_{k+1} is a list of integers representing the positions of $\{s_1, \ldots, s_{i_{k+1}}\}$ in S_L described in lemma 1.
Firstly, the while loop condition and existence of $(k + 1)$-th iteration implies that $i_k < S.length = n$ and $j_k < S'.length = n'$. Then, since both i and j at most gets incremented by 1 in every iteration, we know that $i_{k+1} \leq n$ and $j_{k+1} \leq n'$. So (1) and (2) of $LI(k + 1)$ holds. Then, we show (3) of $LI(k + 1)$. By (4) of $LI(k)$, first i_k elements in S and j_k elements in S' has been implicitly positioned. Then, next element is $(i_k + j_k + 1)$-th element to be ranked. By (5) of $LI(k)$, first i_k elements in S and first j_k elements in S' are all smaller or equal y-intercept to the elements in $S[i_k + 1 \ldots n]$ and $S'[j_k + 1 \ldots n]$, and that $S[i_k + 1]$ and $S'[j_k + 1]$ has smaller y-intercept in elements $S'[j_k + 2 \ldots n]$ if exists, respectively. Hence, we know that next position is for $S[i_k + 1]$ or $S[i_j + 1]$ (or next two positions for both of them in the tie case). Then,
Case 1: if $s_{i_k+1}.b < S'_{j_k+1}$. Then by lines 5-6, $i_k + j_k + 1$ is appended into L_k to form L_{k+1}, which corresponds list of positions of $\{s_1, \ldots, s_{i_k}, s_{i_k+1}\}$ in S_L described above, since $s_{i_k+1}.b$ should comes next as a result of combining case condition and above reasoning. Hence, part (3) of $LI(k + 1)$, i.e., L_{k+1} is as required by both part (3) of LI(k) and $s_{i_k+1}\} = s_{i_{k+1}}\}$ (line 7).
Case 2: $s_{i+1}.b = s'_{j+1}$. Then by lines 9 - 11, the tie case positioning follows exactly what sorting convention for tie 1 specified. And either $i_k + j_k + 1$ or $i_k + j_k + 2$ is appended as position of $s_{i+1} = s_i + 1$ in S_L. Hence (3) of $LI(k + 1)$ is satisfied by combining with (3) of $LI(k)$.
Case 3: $s_{i_k}.b < s'_{j_k}.b$. Then by line 14, nothing is appended to L_k, $L_{k+1} = L_k$. This, however, justifies (3) of $LI(k + 1)$ since i does not change as well ($i_k + 1 = i_k$). So part 3 directly follows from truth of part (3) in $LI(k)$.
So we have shown that (3) of $LI(k + 1)$ is true.
For (4) of $LI(k + 1)$, it follows directly from the code in different cases. In the if and else branches in line 5 and 15, only one of i_{k+1} and j_{k+1} has been "implicitly" position, shown by the increment by 1. For the if

branch in line 5, first i_{k+1} elements in S (shown by increment of i in line 7) and first $j_k = j_{k+1}$ elements in S' have been implicitly positioned, as wanted. For the elseif branch in line 8, first i_{k+1} elements in S and $i_k + 1$ elements in S (both $s_{i_{k+1}}$ and $s'_{j_{k+1}}$) are implicitly positioned), as needed. In the else branch at line 14, only $s'_{j_{k+1}}$) is implicitly positioned shown in increment of j. And hence the first $i_{k+1} = i_k$ elements in S and first j_{k+1} elements in S' is implicitly positioned, as wanted. So, part (4) of $LI(k+1)$ holds.
Now, we show (5) of $LI(k+1)$. Since we know that LI(k) part (5) holds, we only need to show that $S[i_k + 1]$ and $S[j_k + 1]$ has smaller y-intercept to the elements in $S[i_k + 2 \dots n]$ and $S[j_k + 1 \dots n']$. If $s_{i_{k+1}}.b < s'_{j_{k+1}}$, we have $s_{i_{k+1}}.b < s'_{j_k+1}$. Hence part (5) holds as $s_{i_{k+1}}.b < s'_{j_{k+1}}$ by line 7, but both less than has smaller y-intercept to the elements in $S[i_{k+1} + 1 \dots n]$ and $S[j_{k+1} + 1 \dots n']$ since themselves are sorted by y-intercept. If $s_{i_k+1}.b = s'_{j_k+1}$, then $s_{i_{k+1}}.b = s'_{j_{k+1}}$ will both has smaller y-intercept to the elements in $S[i_{k+1} + 1 \dots n]$ and $S[j_{k+1} + 1 \dots n']$. If $s_{i_k+1}.b > s'_{j_k+1}$, by line 14 $s_{i_{k+1}}.b > s'_{j_{k+1}}$. Both of them will be smaller y-intercept to elements in $S[i_{k+1} + 1 \dots n]$ and $S[j_{k+1} + 1 \dots n']$ too. So part (5) of $LI(k+1)$ holds.

Hence we have shown that $LI(k+1)$ holds. Then, we show that the loop terminates.

- Loop Termination: Let $m = n + n' - i - j$. By loop invariant, since $i \leq n$ and $j \leq n'$ and all of them are natural number, we know that m is always a natural number. And m gets decreases by 1 or 2 at every iteration (as i or j or both increases). Hence, value of m at every iteration forms a decreasing sequence of natural number, which is finite. Hence the loop on from lines 4-15 terminates.

- After the loop from lines 4-14: Since the loop terminates, we know it runs a finite number of iterations, say $k \in \mathbb{N}$. We know that $LI(k)$ holds from above. Also since the loop terminates, we know that $i_k \geq S.length = n$ or $j_k \geq S'.length = n'$.

Case 1: $j_k \geq S.length = n'$.
In this case, combined with $LI(k)$ part (2), we know that $j_k = n'$. Also since line 17 checks to true, it runs and finally the algorithm returns $L_k + [i + j + 1, \dots, n + n']$ (list concatenation). By $LI(k)$, L_k is list of intergers representing the positons (rank) of $\{s_1, \dots, s_{i_k}\}$. We have exausted ("ranked") all $j_k = n'$ elements in S' and i_k elements in S.
Also, we observe that it must be that at the last iteration (the k-th), line 8 **elseif** branch or **else** gets executed, since otherwize j has not changed hence there should be even no (k-1)-th iteration, which is a contradiction. So, (1) if the **elseif** branch gets executed at $k - th$ iteration, we know that $S[i_{k-1} + 1].b = S[j_{k-1} + 1].b$ by condition in line 8, i.e., $S[i_k].b = S[j_k].b$ since by lines 12 and 13 we have $i_k = i_{k-1} + 1$ and $j_k = j_{k-1} + 1$. Then we have $S[i_k + 1].b > S[j_k].b$ (since $S[i_k + 1].b > S[i_k].b$). (2) If the **else** branch gets executed at $k - th$ iteration, we know that $S[i_{k-1}+1].b > S[j_{k-1}+1].b$ by implicit else condition in line 14, i.e., $S[i_k+1].b > S[j_k].b$ since $i_k = i_{k-1}$ and $j_k = j_{k+1} + 1$ by line 14. In both cases, we can conclude that $S[i_k + 1].b > S[j_k].b$, i.e., $s_{i_k+1}.b > s.n'.b$
All together, we know that firstly s_{i_k+1} should follows right after those already (implicitly) appended $i + j$ elements in S_L, and since $s_{i_k+1}.b < \dots < s_n$.b, they should be ranked one another in S_L. This is exactly what for loop from lines 16-17 does, ranked $\{s_{i_k+1}, \dots s_n\}$, in order, from $i_k + j_k + 1 = (i_k + 1) + n'$ to $n + n'$ in S_L. Therefore, we have shown that returned value is as desired.

Case 2: $j_k < S.length = n'$ and $i_k \geq S.length = n$.
In this case, L_k is returned by the algorithm since line 16 check to false. Notice that $LI(k)$ holds and k iteration exists implies $i_k \leq n$. So, we have $i_k = n$. Then, by part (3) of $LI(k)$, L_k is a list of integers representing the positions of $\{s_1, \dots, s_n\}$ (since $i_k = n$) in S_L described in lemma 1. Since L_k is returned, this what we wanted to prove! ■

Sorting Convention for Tie 2. *Let $s \in S$ and $s' \in S'$. If $s.m + s.b = s'.m + s.b'$, then*

- *s appears ealier in the collection, if $s.b \geq s'.b$;*
- *s appears ealier in the collection if $s.b < s'.b$.*

Lemma 2. *Suppose the input S and S' is as specified in Q1. The algorithm* ORDER-RIGHT-END(S, S') *returns a list representing the positions of $\{s_1, s_2, \dots, s_n\} = S$ (in-order) in an ordered combining list of elements S_R that contains all the elements in S and S', sorted in non-decreasing order by y-value at $x = 1$, and when tie appears, it follows the sorting convention 2.*

Proof. The proof would be almost the same as lemma 1. ■

Theorem 1. *Let input S and S' is as specified in the Q1. Then, algorithm* CALCULATE-INTERSECTING-PAIRS(S, S') *returns the number of pairs of lines in $S \cup S'$ that intersect, where two completely overlapping line segment count as one line segment and is not count as a pair of intersecting line segment.*

Proof. We will use lemma 3, S and S are disjoint, and the loop in lines 4 and 5 of the algorithm CALCULATE-INTERSECTING-PAIRS(S, S') to prove this theorem. In line 4, the loop variable k gets 1 to $S.length = n$ at every iteration. So, by line 5 (the loop body), the variable res is equal to the sum

$$|L_1[1] - L_2[1]| + |L_1[1] - L_2[1]| + \cdots + |L_1[n] - L_2[n]|,$$

when it returns, which by lemma 3, is equal to

$$\begin{aligned} &\text{the number of pairs of lines in } \{s_1\} \cup S' \\ +\ &\text{the number of pairs of lines in } \{s_2\} \cup S' \\ +\ &\ldots \\ +\ &\text{the number of pairs of lines in } \{s_n\} \cup S' \end{aligned} \quad ,$$

which turns out to be the number of pairs of lines in $S \cup S'$ that intersect, since the set S and S' themselves are disjoint so that there will be no intersecting pair in S itself and in S' itself, respectively. Therefore, the return value is as desired. ■

Lemma 3. *Let input S and S' is as specified in the Q1. Let S_L and S_R be ones specified in lemma 1 and lemma 2 (the two ordered collection). Let L_1 and L_2 the variable gets assigned on lines 1 and 2 of* CALCULATE-INTERSECTING-PAIRS(S, S'). *Let $i \in \{1, 2, \ldots, n\}$. Then, $|L_1[i] - L_2[i]|$ equals the number of pairs of lines in $\{s_i\} \cup S'$ that intersect, where two completely overlapping line segment count as one line segment and is not count as a pair of intersecting line segment.*

Proof. We will split the proof into different cases.
Case 1: s_i does NOT intersect with any element in S' at $x = 0$ and $x = 1$, i.e., at any of the two end points. This is the "nice" case. Then by lemma 1 and 2, observe that $|L_1[i] - L_2[i]|$ is the difference of line segments before $s_i \in S_L$ and $s_i \in S_R$. Futher this will be the difference of line segments from S' before $s_i \in S_L$ and $s_i \in S_R$, since s_i is disjoint with other line segments in S so have same number of lines ranked before it in S_L and S_R. Geometrically, and by the definition of S_L and S_R, it is either that there will be $L_1[i] - L_2[i]$ line segments from S' that starts lower at $x = 0$ (ranks lower than s_i in S_L) and ends higher at $x = 1$ (ranks higher than s_i in S_R) when $L_1[i] > L_2[i]$, and the other way around $(L_2[i] - L_1[i])$ when $L1[i] < L_2[i]$. This geometrically implies the number of intersecting pairs in $\{s_i\} \cup S'$ has to be $|L_1[i] - L_2[i]|$, as needed.
Case 2: s_i intersect with an element in S' at $x = 0$ or $x = 1$ at one side only. (Notice that it is only possible to intersect with **one** element in S' rather than anymore since S' is disjoint.)
We have prove in Case 1 that our algorithm behaves well when counting the number of intersecting pair located other than the end points (i.e., in between lines $x = 0$ and $x = 1$). So, in this case, we only need to show that it successfully counts the intersecting pair happened at the end point ($x = 0$ or $x = 1$).

- Case 2.1: s_i intersect with an element $s' \in S'$ at $x = 0$ such that $s_i.m + s_i.b > s'.m + s'.b$.
  Then, by lemma 1 with sorting convention for tie 1, s_i will be ranked one position lower than s' in S_L (geometrically it is treated as s_i starts lower than s' at $x = 0$ in terms of ranking, although they indeed have same end point at $x = 0$), and since s_i has higher end point at $x = 1$ i.e., higher position in S_R (since $s_i.m + s_i.b > s'.m + s'.b$), this one intersecting pair of s_i and s' will be counted into $|L_1[i] - L_2[i]|$.
- Case 2.2: s_i intersect with an element $s' \in S'$ at $x = 0$ such that $s_i.m + s_i.b < s'.m + s'.b$.
  Then, by lemma 1 with sorting convention for tie 1, s_i will be ranked one position higher than s' in S_L (geometrically it is treated as s_i starts higher than s' at $x = 0$ in terms of ranking, although they indeed have same end point at $x = 0$), and since s_i has lower end point at $x = 1$ i.e., lower position in S_R (since $s_i.m + s_i.b > s'.m + s'.b$), this one intersecting pair of s_i and s' will be counted into $|L_1[i] - L_2[i]|$.

- Case 2.3: s_i intersect with an element $s' \in S'$ at $x = 1$ such that $s_i.b < s'.b$.
  Similar as Case 1 but is justified using lemma 2 and sorting convention for tie 2.
- Case 2.4: s_i intersect with an element $s' \in S'$ at $x = 0$ such that $s_i.b > s'.b$.
  Similar as Case 2 but is justified using lemma 2 and sorting convention for tie 2.

Case 3: s_i intersect with an element s' in S' at $x = 0$ and $x = 1$, i.e., s' fully overlap with s.
In this case, by lemma 1 with sorting convention for tie 1 and lemma 2 with sorting convention for tie 2, s_i will be ranked one position lower than s' in both S_L and S_R. Since both S and S' are disjoint internally, s_i will be the the same position in L_1 and L_2 and not intersecting with other lines in S' as well (so intersecting pairs between $\{s_i\} \cup S'$ should be 0, otherwise S' will not be disjoint, reaching contradiction). So, $L_1[i] = L_2[i]$, this implies $|L_1[i] - L_2[i]| = 0$, as needed. ■

Q3 A high level idea of my algorithm:

Our algorithm for Q3 will count a pair of overlapping line segment or one line segment contained in the other as an intersecting pair, different from Q1 and Q2, this is because the algorithm would look cleaner this way, in this case.

The strategy now is still to find out number of intersecting pair between $\{s\}$ and S', where $s \in S$. We do this for every $s \in S$, and using the disjoint property to sum up these counters, we get the total number of intersecting pairs. The main idea is related to Q1 but not the same.

When s is non vertical, we use binary search-kind of algorithm to find out the position of right end point of s if it were "inserted" in S', sorting by y-value at $s.r$ first (the position in line $x = s.l$, analogy to $x = 0$ in Q1). We do the similar thing at the side of $x = s.r$ (analogy to $x = 1$). And when tie appears during finding position, we adopt some sorting convention, which is actually just the sorting for tie 1 with an tiny alternation. And since the approach and mechanism behind the algorithm, the part of the proof of correctness will have similar argument as in lemma 3. My algorithm for this part would works for case of $s.l = s.r$, i.e., s is a point.

When s is vertical, we still use binary search-kind of algorithm to find out the position of s if it were "inserted" twice into S' sorted by y-value of line segment when $x = s.b$, where first position of s representing where would lower end point of s located, and second position of s representing where would higher end point of s located. With some sorting convention of tie applies, and just take the difference between the two position, which will roughly be the number of line segment this vertical s cross through, i.e., number of intersecting pairs. My algorithm for this part would works for case of $s.l = s.r$, i.e., s is a point.

CALC-Y(s, x)

1 **return** $s.m \times x + s.b$

```
FIND-POSITION-AT-RIGHT(S', s, i, e)
 1  if S' is empty
 2       return 1
 3  while i ≠ e
 4       mid = (i + e)//2     // Integer division
 5       if CALC-Y(s, s.r) ≤ CALC-Y(S'[mid], s.r)
 6            e = mid
 7       else
 8            i = mid + 1
 9  if CALC-Y(s, s.r) > CALC-Y(S'[i], s.r)
10       return i + 1
11  elseif CALC-Y(s, s.r) = CALC-Y(S'[i], s.r)
12       if CALC-Y(s, s.l) > CALC-Y(s, s.r)
13            return i
14       elseif CALC-Y(s, s.l) < CALC-Y(s, s.r)
15            return i + 1
16       else
17            return "contained"
18  else   // CALC-Y(s, s.r) < CALC-Y(S'[i], s.r)
19       return i
```

```
FIND-POSITION-AT-LEFT(S', s, i, e)
 1  if S' is empty
 2       return 1
 3  while i ≠ e
 4       mid = (i + e)//2     // Integer division
 5       if CALC-Y(s, s.l) ≤ CALC-Y(S'[mid], s.l)
 6            e = mid
 7       else
 8            i = mid + 1
 9  if CALC-Y(s, s.l) > CALC-Y(S'[i], s.l)
10       return i + 1
11  elseif CALC-Y(s, s.l) = CALC-Y(S'[i], s.l)
12       if CALC-Y(s, s.r) > CALC-Y(s, s.r)
13            return i
14       elseif CALC-Y(s, s.r) < CALC-Y(s, s.r)
15            return i + 1
16       else
17            return "contained"
18  else   // CALC-Y(s, s.l) < CALC-Y(S'[i], s.l)
19       return i
```

```
NON-VERTICAL-S-INTERSECTING-PAIR(s, S')
1  l = FIND-POSITION-AT-LEFT(S', s, 1, S'.length)
2  k = FIND-POSITION-AT-RIGHT(S', s, 1, S'.length)
3  if l = k = "contained"
4       return 1
5  return ABSOLUTE(l − k)
```

```
VERTICAL-LOWER-END-POSITION(S', s, i, e)
 1  if S' is empty
 2      return 0
 3  while i ≠ e
 4      mid = (i + e)//2
 5      if s.l > CALC-Y(S'[mid], s.b)
 6          e = mid
 7      else
 8          i = mid + 1
 9  if s.l > CALC-Y(S'[i], s.b)
10      return i + 1
11  else    // when s.l ≤ CALC-Y(S'[i], s.b)
12      return i − 1
```

```
VERTICAL-UPPER-END-POSITION(S', s, i, e)
 1  if S' is empty
 2      return 0
 3  while i ≠ e
 4      mid = (i + e)//2
 5      if s.r > CALC-Y(S'[mid], s.b)
 6          e = mid
 7      else
 8          i = mid + 1
 9  if s.r < CALC-Y(S'[i], s.b)
10      return i − 1
11  else    // when s.r ≥ CALC-Y(S'[i], s.b)
12      return i + 1
```

```
VERTICAL-S-INTERSECTING-PAIR(s, S')
1  l = VERTICAL-UPPER-END-POSITION(S', s, 1, S'.length)
2  k = VERTICAL-LOWER-END-POSITION(S', s, 1, S'.length)
3  return l − k
```

```
CALCULATE-INTERSECTING-PAIR2(S, S')
 1  result = 0
 2  i = 1
 3  n = S.length
 4  while i ≤ n
 5      if S[i].m = ∞:
 6          result = result + VERTICAL-S-INTERSECTING-PAIR(S[i], S')
 7      else
 8          result = result + NON-VERTICAL-S-INTERSECTING-PAIR(S[i], S')
 9      i = i + 1
10  return result
```

Runtime Analysis of CALCULATE-INTERSECTING-PAIR2(S, S').
Let S and S' be defined as in Q3 so that $|S| = n$ and $|S'| = n'$.

- The while loop from lines 4 to 9 runs at most n iterations. Each iteration would run in $O(log(n'))$ time.
- The runtime of each iteration is $O(log(n'))$ since either line 6 or line 8 is reached, where both of them takes $O(log(n'))$. Firstly for NON-VERTICAL-S-INTERSECTING-PAIR$(S[i], S')$, it calls two helper inside its body,

which are FIND-POSITION-AT-RIGHT$(S', s, 1, S'.length)$ and FIND-POSITION-AT-LEFT$(S', s, 1, S'.length)$. They have the similar implementation, that starts with exactly what iterative binary search over the set S' does, and have more constant operations after the loop ends. So, each of them takes $O(log(n'))$ as claimed. So, NON-VERTICAL-S-INTERSECTING-PAIR$(S[i], S')$ takes $2O(log(n'))$ and some constant operations, which is still $O(log(n'))$ in runtime. This is almost identical for VERTICAL-S-INTERSECTING-PAIR$(S[i], S')$ (since VERTICAL-UPPER-END-POSITION$(S', s, 1, S'.length)$ and VERTICAL-LOWER-END-POSITION$(S', s, 1, S'.length)$ are similar to FIND-POSITION-AT-RIGHT$(S', s, 1, S'.length)$ - iteratively binary search through S' and only differ in constant time operations, so both taking $O(log(n'))$ as well.)

Hence, multiplying the number of iterations and loop body runtime, we know the while loop from lines 4 to 9 in CALCULATE-INTERSECTING-PAIR2(S, S') runs in $O(nlog(n'))$ time. The other lines are takes constant time, so the whole algorithm CALCULATE-INTERSECTING-PAIR2(S, S') runs in $O(nlog(n'))$ time, as desired. ■

Q4

Lemma 4. *The algorithm* CALC-Y(s, x) *returns the y value of input line segment s at input x-coordinate value x.*

Proof. It follows directly from the formula $y = mx + b$ and the line 1 of the algorithm. ■

Lemma 5. *Let $x \in \mathbb{R}$ such that $0 \leq x \leq 1$. Let S' be given as in Q3. Then, S' is already sorted by y-value at x, in increasing order.*

Proof. It's true at both end points $x = 0$ and $x = 1$ immediately by the question specification.

It is true for $0 < x < 1$ since S' is disjoint. (If there is such that x that makes sorting S' by y-value at x gives different sorting than it does at the end points $x = 0$ (stricly increasing), there must exists intersection(s) between elements in S' so S', that makes S' would not be disjoint, reaching a contradiction.) ■

Notation: // means integer division.

Lemma 6. *Let i, e be two integers such that $i < e$, $i \leq (i + e)//2 < 2$.*

Proof. Since $e > i$, we know that $i + e//2 \geq (i + i)//2 = (2i)//2 = i$. Also, by definition of integer division and $i < e$ we have $(i + e)//2 \leq (i + e)/2 \leq \big((e - 1) + l)/2 = e - 1/2 < e$. ■

This time I proved "left side" (lemma 7) of my algorithm works and the "right side" will be almost identical, as lemma 8 states.

Sorting Convention for Tie 3. *Let $s \in S$ and $s' \in S'$. If $s.m \cdot s.r + s.b = s'.m \cdot s'.r + s'.b$, then*

- *s appears earlier in the collection, if $s.m \cdot s.l + s.b > s'.m \cdot s.l + s'.b$; and*
- *s appears earlier in the collection if $s.m \cdot s.l + s.b < s'.m \cdot s.l + s'.b$.*

Lemma 7. *Let $s \in S$ be a non-vertical line segment and S' be as specified in Q3. Let $S_{R(s)}$ be an ordered collection that contains s and all the elements in S' such that they are sorted in non-decreasing order by y-value at $x = s.r$, and sorting convention for tie 3 applies when tie appears. Let $i = 1$ and $e = n' = |S'|$. Then,* FIND-POSITION-AT-RIGHT(S', s, i, e) *returns 1 when S' is empty; the position of s in $S_{R(s)}$ when s is not contained in any of element in S'; a string "contained" otherwise.*

Proof. From lemma 5, we know that S' itself is already sorted by y-value at $x = s.r$. Let's prove an loop invariant first. Consider the loop from lines 3-8 in FIND-POSITION-AT-RIGHT(S', s, i, e). Let predicate $LI(k)$ be defined as: If there is at least k iterations, then

1. $1 \leq i_k \leq e_k \leq |S'| = n$

2. if CALC-Y$(S[1], s.r) \leq$ CALC-Y$(s, s.r)$ and $S'[i_k - 1]$ exists, then CALC-Y$(S'[i_k - 1], s.r) \leq$ CALC-Y$(s, s.r)$
3. if CALC-Y$(s, s.r) \leq$ CALC-Y$(S'[n'], s.r)$ and $S'[e_k + 1]$ exists, then CALC-Y$(s, s.r) \leq$ CALC-Y$(S'[e_k + 1], s.r)$

I will show that $LI(k)$ holds for all $k \in \mathbb{N}$. Base case: Let $k = 0$. (1) follows directly from the precondition specified in lemma 7. (2), (3) are immdiately true since both the if parts evaluate to false as $S'[0]$ and $S[n' + 1]$ does not exist.

Assume that $LI(k)$ holds. I will show $LI(k + 1)$ holds. Assume there is at least $k + 1$ iterations (thus at least k). I will show 3 parts of $LI(k + 1)$ holds

Part (1) of $LI(k + 1)$: Since k iteration exists, $i_k \neq e_k$. By $LI(k)$ we have $1 \leq i_k \leq e_k \leq n$. And thus $i_k < e_k$, so by lemma 6, we have $i_k \leq mid_{j+1} \leq e_k$. Then, by lines 5 and 8, we have two cases. 1) $i_{k+1} = i_k$ and $e_{k+1} = mid_{k+1}$, OR 2) $i_{k+1} = mid_{k+1} + 1$ and $e_{k+1} = ek$. Together (combined with $1 \leq i_k \leq e_k \leq n$) we have $1 \leq i_{k+1} \leq e_{k+1} \leq n$, as desired.

Part (2) of $LI(k + 1)$:

Part (3) of $LI(k + 1)$:

Loop Termination: Let $m = e - i$ to be the loop measure. Notice that m decreases at every iteration since assuming there is (k+1) iteration then either $e_{k+1} = mid < e$ by lemma 6 (e decreases) or $i_{k+1} = mid + 1 > i$ by lemma 6 (i increases). Also since i and e are both natural number, together we know that k is always a natural number. So value m at each iteration forms decreasing sequence of natural number, which is finite, thus the loop terminates.

Let's say the loop terminates after $k \in \mathbb{N}$ iteration, then we know that $i_k = e_k$ by the termination condition.

Notice that in all cases all elements in S' already been sorted in increasing order of y-value at $x = s.r$, so we just need to place s in between them with this observation (we can return the correct position if we can find answer from $S[i_k]$'s neighbor).

Case 1: Assume that CALC-Y$(s, s.r) >$ CALC-Y$(S'[i_{k+1}], s.r)$.
Case 1.1: Assume that $S[i_k]$ is the end (i.e., $i_k = n$), then $S[i + 1]$ does not exist, we should place s after all the elements of S' in $S_{R(s)}$ thus $i_k + 1 = n$ is returned, as desired.
Case 1.2: Assume that $S[i_k]$ ($i_k < n$) is not the end. It must be one of the interior element in S'. Then, since in this case s has smaller y-value at $x = s.r$ than which of $S[e_k + 1] = S[i_k + 1]$ (by part (3) of LI), it should be replaced in between $S[i]$ and $S[i + 1]$, thus $i + 1$ is returned as desired.

Case 2: Assume that CALC-Y$(s, s.r) =$ CALC-Y$(S'[i_{k+1}], s.r)$.
Then, the code from lines 12-17 follows well with the sorting convention for tie 3. If s has y-value at $x = s.l$ greater than which of $S[i_k]$'s, i is returned as it is "appears" earlier in the collection than S'$[i_k]$, and $i + 1$ if it's a less than, and return an "*contained*" when y value of s and $S'[i_k]$ is the same at $x = s.l$, this is as required as well since it these two line segments either has a contained or overlapping relationship.

Case 3: Assume that CALC-Y$(s, s.r) <$ CALC-Y$(S'[i_{k+1}], s.r)$.
Case 3.1: $S[i_k]$ is the start of S' (i.e., $i = 1$). Then we should since CALC-Y$(s, s.r) >$ CALC-Y$(S'[i_{k+1}], s.r)$, we should return 1 (in prior to all of them), as desired since $i_k = 1$.
Case 3.2: $S[i_k]$ is not the start of S' (i.e., $i = 1$). Then, it should be in the interior of S'. Then, since we know that CALC-Y$(S[i_k - 1], s.r)$CALC-Y$(S[i_k], s.r)$ (by LI (2)), we should return i as its position in $S_{R(s)}$, as wanted.
■

Sorting Convention for Tie 4. *Let $s \in S$ and $s' \in S'$. If $s.m \cdot s.l + s.b = s'.m \cdot s'.l + s'.b$, then*

- *s appears earlier in the collection, if $s.m \cdot s.r + s.b > s'.m \cdot s.r + s'.b$; and*
- *s appears earlier in the collection if $s.m \cdot s.r + s.b < s'.m \cdot s.r + s'.b$.*

Lemma 8. *Let $s \in S$ be a non-vertical line segment. Let $S_{L(s)}$ be an ordered collection that contains s and all the elements in S' such that they are sorted in non-decreasing order by y-value at $x = s.l$, and sorting convention for tie 4 applies when tie appears Suppose the input S' is specified as in Q3, and s is an element of S as specified in Q3, and $i = 1$ and $e = n' = |S'|$. Then,* FIND-LEFT-POSITION(S', s, i, e) *returns the position of s in $S_{PLs)}$ when s is not contained in any element in s', a string "contained" otherwise.*

Proof. The argument is almost the same as proof of lemma 7. ■

Lemma 9. *Let S and S' be as specified in Q3, and let $s \in S$ be a non-vertical line segment. The algorithm* Non-Vertical-s-Intersecting-Pair(s, S') *returns the number of intersecting pairs between $\{s\}$ and S', where s fully overlap with an element in S' or is contained by an element of S' is counted as an pair of intersecting line segment.*

Proof. We prove by cases.
Case 1: s_i does NOT intersect with any element in S' at $x = s.r$ and $x = s.l$, i.e., at any of the two end points. The argument goes almost identical as in case 1 of proof of lemma 2.
Case 2: s_i intersect with an element in S' at $x = s.r$ or $x = s.l$ at one side only. (Notice that it is only possible to intersect with **one** element in S' rather than anymore since S' is disjoint.)
The argument goes exactly the same way as in case 2 of proof of lemma 2, as the sorting convention 1 and 3 are just the same but with replacement of end points, and sorting convention 2 and 3 are the same.
Case 3: s_i intersect with an element in S' at both $x = s.r$ and $x = s.l$.
Then from lemma 7 and 8, lines 1 and 2 both l and k are assigned to string "*contained*" so we return 1. This is as required since we count this as a pair of intersection. (There will only be 1 for this two since S and S' are disjoint). ■

Lemma 10. *Let $s \in S$ be a vertical line segment. Let $S_{VL(s)}$ be an ordered collection that contains s and all the elements in S' such that they are sorted in non-decreasing order by the magnitude of y-value at $x = s.b$ (where s is using $s.l$ to compare). When tie appears between s and some $s' \in S'$, s is placed in collection **before** s. Suppose the input S' is specified as in Q3, and s is an element of S as specified in Q3, and $i = 1$ and $e = n' = |S'|$. Then,* Vertical-Lower-End-Position(S', s, i, e) *returns the position of s in $S_{VL(s)}$; or 0 if S' is empty.*

Proof. todo ■

Lemma 11. *Let $s \in S$ be a vertical line segment. Let $S_{VU(s)}$ be an ordered collection that contains s and all the elements in S' such that they are sorted in non-decreasing order by the magnitude of y-value at $x = s.b$ (where s is using $s.l$ to compare). When tie appears between s and some $s' \in S'$, s is placed in collection **after** s. **later**. Suppose the input S' is specified as in Q3, and s is an element of S as specified in Q3, and $i = 1$ and $e = n' = |S'|$. Then,* Vertical-Upper-End-Position(S', s, i, e) *returns the position of s in $S_{VU(s)}$ when s is not contained in any element in s'; or 0 if S' is empty.*

Proof. The argument is similar to lemma 10. ■

Lemma 12. *Let S and S' be as specified in Q3, and let $s \in S$ be a non-vertical line segment. The algorithm* Vertical-s-Intersecting-Pair(s, S') *returns the number of intersecting pairs between $\{s\}$ and S', where s fully overlap with an element in S' or is contained by an element of S' is counted as an pair of intersecting line segment.*

Proof. By lemma 10 and 11, we know that the positions of line segments from S' with y-value at $x = s.b$ in $S_{V(s)}$ is in between l and k, there are $l - k - 1$ such positions (geometrically this implies vertical s "cross" through that many line segments from S'), so the correct value is returned as the number of intersection between $\{s\}$ and S'. ■

Theorem 2. *Let inputs S and S' be as specified as Q3. Then the algorithm* Calculate-Intersecting-Pair2(S, S') *return the number of intersections between S and S',*

Proof. I will prove a loop invariant first. I will show that for all $k \in \mathbb{N}$, if the loop from lines 4-9 in Calculate-Intersecting-Pair2(S, S') runs at least k iteration, then $i_k \leq n + 1$ and

$$result_k = \sum_{j=1}^{i_k - 1} (\text{number of intersections between } \{s_j\} \text{ and } S').$$

Initially, let $k = 0$. We know that $i_k = 1 \leq n + 1$, and $result_k = 0$ and the summation also equal to zero since $i_k = 0$. Then, let $k \in \mathbb{N}$. Assume that if there is at least k iteration then $i_k \leq n + 1$ and $result_k = \sum_{j=1}^{i_k - 1}$(number of intersections between $\{s_j\}$ and S'). Next, assume there is $(k + 1)$ iteration (so at least k as well), I want to show that $result_{k+1} = \sum_{j=1}^{i_{k+1} - 1}$(number of intersections between $\{s_j\}$ and S').

$$
\begin{aligned}
result_{k+1} =& result_k + (\text{number of intersections between } \{s_{i_k - 1}\} \text{ and } S') \quad \text{(by line 5 \& 7 and lemma 9 \& 12)} \\
=& \sum_{j=1}^{i_k - 1} (\text{number of intersections between } \{s_j\} \text{ and } S') + \\
& (\text{number of intersections between } \{s_{i_k}\} \text{ and } S') \qquad \text{(by induction hypothesis)} \\
=& \sum_{j=1}^{i_k} (\text{number of intersections between } \{s_j\} \text{ and } S') \\
=& \sum_{j=1}^{i_{k+1} - 1} (\text{number of intersections between } \{s_j\} \text{ and } S'), \qquad \text{(by line 8)}
\end{aligned}
$$

as wanted. So the loop invariant holds for all iterations. Next, we show the loop terminates. Let $m = n + 1 - i$ be the loop measure. Notice that since i increases at every iteration by 1, m decreases at every iteration by 1. And since n and i are natural number with $i \leq n + 1$ (by loop invariant above), we know that m must always be a natural number as well. So, value m at every iteration forms a decreasing sequence of natural number, which is finite. That means the loop should terminate!

So, we know that it runs a finite number of iteration, say $K \in \mathbb{N}$, from loop invariant we know that $i_k \leq n + 1$ and $result_k = \sum_{j=1}^{i_k - 1}$(number of intersections between $\{s_j\}$ and S'). And from while loop termination we know that $i_k > n$. So, we have $i_k = n + 1$, i.e., $i_k - 1 = n$, which then implies

$$
result_k = \sum_{j=1}^{n} (\text{number of intersections between } \{s_j\} \text{ and } S')
$$

is returned at line 10. This is indeed is the number of intersection between S and S' since S and S' themselves are disjoint respectively. This completes the proof. ■