Solutions for 611 Homework 1

Mahim Agarwal, Hanwen Xiong, and Jackson Warley

September 22, 2017

1 Solution to Question 1

Let $A = \{a_1, \ldots, a_n\}$ be the array that we need to count the inversion for. Consider the following algorithm ALG for counting the number of inverted pairs in A:

1. Divide the array into 2 equal parts: $A_l = \{a_1, \ldots, a_{n/2}\}$ and $A_r = \{a_{n/2+1}, \ldots, a_n\}$
2. Recursively count the number of inversions in each half by calling the algorithm on A_l and A_r
3. Merge the two halves into a sorted whole(increasing order) and count the number of inversions across the two halves (from A_l to A_r).
4. Return the sum of three counts.

Procedure:
CountInversion($\{a_1, \ldots, a_n\}$)
if n==1 return count = 0

else:
$count_l$ = CountInversion($\{a_1, \ldots, a_{n/2}\}$)
$count_r$ = CountInversion($\{a_{n/2+1}, \ldots, a_n\}$)
return $count_l$ + $count_r$ + Merge(sorted A_l, sorted A_r)
Merge Step:
Let us suppose we have two sorted halves A and B in increasing order of elements. Then do the following:
Start with the first element of A and B and move towards right.
Compare a_i and b_j
If $a_i < b_j$, count = count + 0. No inversion of a_i. Move on to next element of A.
If $a_i > b_j$, count = count + all elements remaining in A. Move on to the next element of A and next element of B
Keep appending the smaller of the two elements a_i and b_j to the sorted list C.
Accuracy explanation:

We will use the following lemma to prove the correctness of ALG.

Lemma 1.1. *If $A = \{a_1, \ldots, a_{n/2}\}$ and $B = \{b_1, \ldots, b_{n/2}\}$ are two sorted arrays in increasing order where A is the left half while B is the right half, then*
if $a_i < b_j$: inversion count for a_i beyond this point = 0
while if $a_i > b_j$: inversion count for this b_j = rest of the elements in A. ($\frac{n}{2}$ - i)

Proof. Since A and B are sorted in increasing order:
$a_1 < a_2 < a_3 < \ldots < a_{n/2}$
and $b_1 < b_2 < b_3 < \ldots < b_{n/2}$

Since we are dividing the array into two halves, all elements in the left half will have index less than all elements in the right half. Thus, for inversion, element in the left half should be greater than element in the right half.

So, if $a_i > b_j$ and $a_i < a_{i+1} < \ldots < a_{n/2}$
$(\Rightarrow)$ $a_{n/2} > a_{n/2-1} > \ldots > a_{i+1} > a_i > b_j$

So count of inversion will increase by number of elements remaining in A beyond i. We now will move on to a_{i+1} and b_{j+1}
if $a_i < b_j$ and $b_j < b_{j+1} < \ldots < b_{n/2}$
$(\Rightarrow)$ $a_i < b_j < b_{j+1} < \ldots < b_{n/2}$

So count of inversion will increase by zero, since index of a_i is less and its value is also smaller. There will be no more inversion pair for a_i and thus we move on to a_{i+1}.

Thus, we will find the total inversion count accross A and B by traversing the array once (travresing completely in the worst case).
Adding three inversion count will give us the total inversion count for whole array. □

Time Complexity:
Step 1: We split the array A into 2 subarrays A and B of half the size. T(n) = 2 T(n/2) + $\mathcal{O}(Merge)$
Step 2: To merge, we only need to traverse the halves once, and for a maximum of n elements at each recursion level.
So, T(n) = 2 T(n/2) + $\mathcal{O}(n)$
By Master theorem: time complexity is $\mathcal{O}(n \log n)$

2 Solution to Question 2

For clarity, let the elements of A and B be indexed as $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$. Consider the following algorithm ALG for determining the nth smallest element of $A \cup B$:

1. Let $i = j = \frac{n}{2}$ be indices into A and B, respectively, which may be updated in later steps.
2. If it is the case that $a_i < b_{j+1}$ and $b_j < a_{i+1}$, return $\max\{a_i, b_j\}$.
3. Otherwise, if $a_i > b_{j+1}$, update i and j so that $i \mapsto \frac{1}{2}i$ and $j \mapsto \frac{1}{2}(j+n)$. If, on the other hand, $b_j > a_{i+1}$, update the indices so that $i \mapsto \frac{1}{2}(i+n)$ and $j \mapsto \frac{1}{2}j$. If i (resp. j) is equal to n, return a_i (resp. b_j); otherwise, return to step 2.

We will use the following lemma to prove the correctness of ALG.

Lemma 2.1. *Suppose that $i+j=n$ and let $m=\max\{a_i,b_j\}$. Then m is the nth smallest element of $A\cup B$ if and only if $a_i<b_{j+1}$ and $b_j<a_{i+1}$.*

Proof. ($\Rightarrow$) Suppose that m is the nth smallest element of $A\cup B$. Then m is larger than exactly $n-1$ elements of $A\cup B$. Where can these elements possibly be located in their lists? If $m=a_i$, then m is greater than exactly $i-1$ elements of A (since A is sorted). Thus m must be greater than exactly $(n-1)-(i-1)=(i+j-1)-(i-1)=j$ elements of B. In fact, since B is sorted, m must be greater than the first j elements of B. In particular, this means we have $a_{i+1}>a_i=m>b_j$, giving us one of the desired inequalities. Moreover, $m>b_{j+1}$ would imply the presence of an nth element of $A\cup B$ which is smaller than m, contradicting that m is the nth smallest element of $A\cup B$. Thus, we have shown $a_i=m\leq b_{j+1}$, and, since the elements are distinct, this inequality must be strict, as desired. A symmetrical argument yields the same result in the case that $m=b_j$, completing one direction of the proof.

($\Leftarrow$) Now suppose that $a_i<b_{j+1}$ and $b_j<a_{i+1}$. Since $m\geq a_i$, it is greater than at least $i-1$ elements of A. Likewise, since $m\geq b_j$, it is greater than at least $j-1$ elements of B. In fact, since $m=\max\{a_i,b_j\}$, one of these inequalities is strict and the other is actually an equality. Without loss of generality (the other possible choice is dealt with by a symmetrical argument), suppose $m=a_i$ and $m>b_j$. Then m is greater than exactly $i-1$ elements of A and at least j elements of B, so m is greater than at lest $(i-1)+j=n-1$ elements of $A\cup B$. We initially supposed that $m=a_i<b_{j+1}$, so this lower bound is actually tight. Since m is greater than exactly $n-1$ elements of $A\cup B$, m is the nth smallest element of $A\cup B$. □

Claim 1: ALG always terminates and returns the nth largest element of $A\cup B$.

Proof. Firstly, note that initially we have $i=j=\frac{n}{2}$, so $i+j=n$. When we update the indices, we do so by $(i,j)\mapsto\left(\frac{1}{2}i,\frac{1}{2}(j+n)\right)$. Since

$$\frac{1}{2}i+\frac{1}{2}(j+n)=\frac{1}{2}(i+j)+\frac{1}{2}n=n$$

we have that $i+j=n$ is an invariant of ALG. Finally, because n is a power of 2, each subsequent image of i is a multiple of 2, so we always have $i,j\in\mathbb{N}$, and in particular i and j are always valid as indices. This allows us to apply Lemma 2.1 at any time during our analysis of ALG.

The updating of i and j amounts to a binary search for the correct index configuration. Since the position of j is completely determined by the position of i (lest we violate the invariant $i+j=n$), it suffices to constrain our analysis to the trajectory of i under iteration of the map that updates the indices over the course of a run. The value of i begins at $\frac{n}{2}$ and is only ever updated by either $i\mapsto\frac{i}{2}$ or $i\mapsto\frac{1}{2}(i+n)$. In partiuclar, i begins in the inverval $I_1=[1,n]$, and is mapped into an interval $I_2\in\left\{[1,\frac{n}{2}],[\frac{n}{2}+1,n]\right\}$ after the first update. In general, the kth time it is updated, i is mapped into an interval $I_{k+1}\subseteq I_k$ of size $\frac{|I_k|}{2}$. Since the intervals containing i must be nested, and n is finite, there may only be a finite number of updates in a single run of ALG. Since we must return if we cannot update i, ALG always terminates.

Now, we claim that if ALG terminates, it returns the correct answer. In the case that we return in Step 2, our answer is correct by Lemma 2.1. If we return in Step 3, then $i=n$ without loss of generality. Since A is sorted, $i=n$ implies the existence of $n-1$ elements of A which are smaller than a_i. If $b_j=b_1<a_i$, then in the previous iteration we would have had $b_1<a_{i+1}$, so we would

have terminated in Step 2.[1] Thus we have shown there are exactly $n-1$ elements of $A \cup B$ which are less than a_n, so a_n must be the nth smallest element. This completes the proof of correctness. □

Claim 2: ALG runs in $\mathcal{O}(\log n)$ time.

Proof. For each k, the kth index update halves the size of the interval I_k in which the desired index lies, so there may be at most $\log_2 n = \mathcal{O}(\log n)$ iterations before the algorithm terminates due to the extremal case of $i \in \{1, n\}$. □

3 Solution to Question 3

For clarity, let the elements of A be indexed as $A = \{a_1, \ldots, a_n\}$. Consider the following algorithm ALG for determining whether A is boring or not:

1. Divide the array into 2 equal parts: $A_l = \{a_1, \ldots, a_{n/2}\}$ and $A_r = \{a_{n/2+1}, \ldots, a_n\}$
2. Recursively solve the two halves by calling the algorithm on A_l and A_r.
3. Let's say that element that make A_l boring is a_l and that make A_r boring is a_r, then return a_l if a_l=a_r else return NULL
   ($\Rightarrow$) return $a_l == a_r$? a_l : NULL. However if any one is NULL and other is not null then return the not null element.
4. If the final answer(a_f) is NULL then array is not boring else check for the count of a_f in original array and if it is greater than $n/2$, array is boring otherwise non-boring.

Procedure:
CheckIfBoring($\{a_1, \ldots, a_n\}$)
if n==1 return a_1

else:
a_l = checkIfBoring($\{a_1, \ldots, a_{n/2}\}$)
a_r = checkIfBoring($\{a_{n/2+1}, \ldots, a_n\}$)
if a_l = NULL and a_r = NULL return NULL (Non-boring array)
else if a_l = NULL and a_r NOT NULL return a_r
else if a_l NOT NULL and a_r = NULL return a_l
else return a_l==a_r ? a_l : NULL
Count the occurences of returned element in the whole array and if it is greater than $\frac{n}{2}$ array is boring else non- boring.
Accuracy explanation:
We will use the following lemma to prove the correctness of ALG.

Lemma 3.1. *If $A = \{a_1, \ldots, a_n\}$ is an array, then this algorithm(through divide and conquer) will always compare a pair of consecutive elements and ignore(or delete) them if they are non-matching. It will keep on doing this recursively until it is left with one or no element(NULL) at which point it will count the number of occurences of final element in the original array and return if the array*

[1] This looks off-by-one at first glance, but because the lists are 1-indexed, this final iteration maps $j = 1$ to $j = 1$ and $i = n - 1$ to $i = n$.

is boring or not. Suppose $A = \{a_1, \ldots, a_n\}$ is a boring array with element a_f. Now, to prove the correctness of this algorithm, it is sufficient to prove that if a_f exists then this algorithm will always return a_f:

Proof. If a_f makes the array boring and it occurs at a total of k times, then:
k $>\frac{n}{2}$
Now, consider the following claims:
Claim 1: There will always be atleast one pair with both elements a_f

Proof. **proof by contradiction:** Let us assume that there are no pair of a_f in A. That means at the maximum a_f can only occur at every alternate index. So total count of $a_f = \frac{n}{2}$
But we know that a_f occurs more than $\frac{n}{2}$ times. hence, contradiction to our assumption. Thus there is atleast one pair of a_f. □

Claim 2: If there are x number of pairs such that both elements are equal to a value other than a_f then there will be x+1 number of pairs of a_f.

Proof. **proof by contradiction:** Let us assume that there are y pair of a_f in A and y <x+1.
So total number of a_f = 2y
and total number of elements other than a_y = 2x
Note: we don't have to consider other elements as no other pair exists thus every other pair will be of the form (a_f, a_f) , in other words, one element will be a_f while other with be something other than a_f.
So, ratio of a_f in total array = $\frac{2x}{2x+2y}$
we know that a_f occurs more than n/2 times. So,

$\frac{2y}{2x+2y} > \frac{1}{2}$
thus 2y >x + y

y >x But we assumed that y <x + 1 and y is an integer which is not possible
Thus, contradiciton to our assumption.
Hence, there will be atleast x+1 pair of a_f in A if there are x pairs of elements other than a_f. □

Thus, for every element(or element pair) not equal to a_f there is atleast one a_f (or one pair of a_f) that will cancel it out and finally we will be left with one extra pair of a_f.
Hence, if a_f exists, then alogrithm will always return a_f and no other elements. Then we can determine the count of a_f in the array and check whether the array is boring or not. □

Time Complexity:
Step 1: We split the array A into 2 subarrays A_l and A_r of half the size. We choose the majority element a_l and a_r and compare them. If they are same we return the element else we return NULL
Note: comparison takes constant time.
T(n) = 2 T(n/2) + $\mathcal{O}(1)$
By Master theorem: time complexity of this step is $\mathcal{O}(\log n)$
Step 2: We count the number of times the returned element a_f occur in the array A
This step takes $\mathcal{O}(n)$ time
Thus time complexity is $\mathcal{O}(logn)$+ $\mathcal{O}(n)$
= $\mathcal{O}(n)$

4 Solution to Question 4

Let L and R denote the left and right subtrees of T, respectively. We define an algorithm ALG to return the weight of a minimum vertex cover by recursively comparing the weights of minimum covers of L and R. To this end, ALG takes an additional parameter – a sum type encoding a Yes/No/Maybe distinction – which determines whether it includes the root of the current tree in its cover. Finally, we take the minimum weight over all combinations of including or excluding the root in the cover.

function ALG($T = (r, L, R)$, includeRoot)
  if includeRoot = Yes **then**
    if $|T| \leq 3$ **then**
      return $w(r)$
    else
      return $w(r) +$ ALG$(L,$ Maybe$) +$ ALG$(R,$ Maybe$)$
    end if
  else if includeRoot = No **then**
    if $|T| \leq 3$ **then**
      return sum of weights of leaves of T
    else
      return ALG$(L,$ Yes$) +$ ALG$(R,$ Yes$)$
    end if
  else if includeRoot = Maybe **then**
    if $|T| \leq 3$ **then**
      return $\min\{w(r),$ sum of weights of leaves of $T\}$
    else
      return min of:

$$W_0 = w(r) + \text{ALG}(L, \text{Yes}) + \text{ALG}(R, \text{No})$$
$$W_1 = w(r) + \text{ALG}(L, \text{No}) + \text{ALG}(R, \text{Yes})$$
$$W_2 = w(r) + \text{ALG}(L, \text{No}) + \text{ALG}(R, \text{No})$$
$$W_3 = \text{ALG}(L, \text{Yes}) + \text{ALG}(R, \text{Yes})$$

    end if
  end if
end function

Claim 1: ALG$(T,$ Maybe$)$ will return the minimum weight of a cover of T.

Proof. For the sake of analysis, it is convenient to talk about ALG as though it computes a cover, even though it technically only computes the weight of that cover. We could easily modify ALG to return the cover itself by storing the identity of a vertex when it includes its weight in a sum, so the convenience of identifying a vertex with its weight does not cause a problem for the argument. Our strategy is to show first that ALG finds minimum covers in the base case of $n \leq 3$, and second that whenever ALG merges the covers of two subtrees, it does so in a way that results in a minimum cover of the parent tree. If $|T| \leq 3$, then ALG either returns the root or the two leaves. Both of these sets are covers of T, so ALG returns a cover for trees where $n \leq 3$. Furthermore, if $n \leq 3$ and ALG$(T,$ Maybe$)$ is called, ALG will choose the result with minimum weight. Therefore, ALG returns minimum covers for base-case trees.

Now, it suffices to show that when ALG merges two subtree covers, the resulting set is a minimum

cover of the parent tree. Whenever ALG is called on a tree with $n \geq 3$ and a Maybe argument, it returns one of the covers associated with W_0, W_1, W_2, or W_3 – whichever has the smallest weight. By inspection, we can see that all of these options are indeed covers. Therefore, ALG returns a minimum cover unless there is some additional cover $C \notin \{W_0, W_1, W_2, W_3\}$, which has lesser weight. However, since the lower covers in the tree are fixed, the only other possibilities are $C = \{r, \text{root}(L), \text{root}(R)\}$ and $|C| \leq 1$ with $C \neq \{r\}$. In the first case, $w(C) > w(W_0)$ since $W_0 \subset C$. In the second case, C is not actually a cover, because at least one of $(r, \text{root}(L))$ and $(r, \text{root}(R))$ is uncovered. Therefore the result of a merge event must always be a cover of minimum weight. □

Claim 2: ALG runs in $\mathcal{O}(n^2)$ time.

Proof. Since ALG has the structure of a typical divide-and-conquer algorithm, we can appeal to the Master Theorem. ALG works by returning the minimum of four sums composed of the quantities $w(r)$, ALG(L, No), ALG(L, Yes), ALG(R, No), and ALG(R, Yes). The first, $w(r)$ takes constant time to compute. The remaining four involve recursive calls to ALG. Since T is balanced, each of these four calls is made on a tree with size $\mathcal{O}(\frac{n}{2})$. The remaining work – summing up the results and taking the minimum – is $\mathcal{O}(1)$. In other words, we divide the problem into four sub-problems of size $\frac{n}{2}$ each, with constant-time work to recombine their solutions into a solution for an instance of size n. Therefore, the Master Theorem gives that since $2 > 2^0$, ALG runs in $\mathcal{O}(n^{\log_2(4)}) = \mathcal{O}(n^2)$. □

5 Solution to Question 5

solution A:

Suppose we have n lines $L = \{ l_1, l_2, \ldots, l_n \}$
Sort the lines in increasing order of slope:
let's say the new array is: $\{ l_1, l_2, \ldots, l_n\}$ where $l_1 < l_2 < \ldots < l_n$

1. Divide the array into two halves.

2. Call the *alg* recursively on both the halves to return set of visible lines in each half

3. Merge both the halves to get final set of visible lines

Merge Step:
Lets say we have two halves:

$L_l = \{ l_1, l_2, \ldots, l_{n/2} \}$ and $L_r = \{ l_{n/2+1}, \ldots, l_n \}$
where slope of $l_1 < l_2 < \ldots < l_n$

Now, for each i^{th} element in L_l (start from right to left) and j^{th} element in L_r (start from left to right)
Compare intersection point of $(l_i, l_{i-1}) = P_0$ and $(l_i, r_j) = P_1$ and do the following:

1. If P_1 is to the right of P_0:
   Append r_j to the right of left array, L_l and increment j by 1 (move on to next element of right array L_r)

2. If P_1 is to the left of P_0:
   Delete all entries to the right of i in left array L_l and all entries to the left of j in right array L_r
   In other words delete $\{ l_i, l_{i+1}, \ldots, l_{n/2} \}$ and $\{ l_{n/2+1}, \ldots, l_{j-1} \}$
   append j^{th} element of right array, r_j, to the right most of left array, L_l, and move on to next element of right array (j=j+1).

The final array will give us all the lines which are visible.
Accuracy explanation:
We will use the following lemma to prove the correctness of ALG.

Lemma 5.1. *If $V = \{l_1, l_2, \ldots, l_k\}$ is a set of all the visible lines sorted in increasing order of their slopes then the visible portion of smaller slope line will always exists to the left of visible potion of higher slope lines.*

Proof. Consider three lines l_1 : y = a_1x + b_1 , l_2 : y = a_2x + b_2 and l_3 : y = a_3x + b_3 where a_1 $<a_2$ $<a_3$. If lines l_2 and l_3 intersects at a point P_{23} (x_0, y_0) then to the right of $P_2$3 l_3 $>l_2$ always since it has larger slope.
On the other hand, If lines l_1 and l_2 intersects at a point P_{12} (x_1, y_1) then to the left of P_{12} l_1 $>l_2$ always since it has smaller slope.

So, if x_0 $<x_1$ then to its right, l_3 $>l_2$ while to its left l_1 $>l_2$
Thus, l_2 cannot be visible anymore.
hence, visible region of smaller slope line always exists to the left of larger slope lines. □

Now, consider the following two claims which will establish the correctness of the algorithm:

Claim 1: If a line on the right half cuts a line on the left half such that the intersection point of two lines lies to the left of visible portion(segment in visible part) of line in the left, then it will eclipse all the lines having smaller slopes than itself but larger slopes than the line under consideration.
In other words, we need to prove that if If P_1 is to the left of P_0 then we should delete all lines to the right of i(left half) and all lines to the left of j(right half)

Proof. This follows directly from Lemma 5.1. Since it is true for any pair of lines and all lines from l_i to r_{j-1} have smaller slope than r_j
for all the pairs of r_j with any one line from $\{ l_i, l_{i+1}, \ldots, l_{n/2} \}$ and $\{ r_1, r_2, \ldots, r_{j-1} \}$ if intersection point lies to the left of visible portion of smaller slope implies the larger slope line will eclipse the smaller slope line. □

Claim 2: If a line on the right half cuts a line on the left half such that the intersection point of two lines lies to the right of visible portion(segment in visible part) of line in the left, then it can never eclipse any line that exists in the left half.

Proof. At the intersection point of r_j and l_i, sincce slope of l_i <slope of r_j at any point to the left, y-coordinate of l_i >y of r_j
and since all other lines to the left of l_i have even smaller slope and lies to further left of intersection point (since they are visible lines from last recursion)
there will be no point to the left where y coordinate of r_j >y of l_i. □

Both these claims combine to prove the correctness of the merge algirithm.

Time Complexity: (Same as merge sort)
Step 1: We split the array L into 2 subarrays L_l and L_r of half the size and recursively solve each of them. So, T(n) = 2 T(n/2) + ...
Step 2: For each recursion, the merge step traverse the array once, and in the worst ccase traverse whole of it.
So, this step takes $\mathcal{O}(n)$ time.
Thus, T(n) = 2 T(n/2) + $\mathcal{O}(n)$
By Master theorem: time complexity of this step is $\mathcal{O}(n \log n)$

Solution B:

5.1 Algorithm

Useful definition:

- l_{max}: the line with maximum slope
- l_{min}: the line with minimum slope
- $l_{nextmax}$: the line with maximum slope in the rest of two subarrays
- $l_{nextmin}$: the line with minimum slope in the rest of two subarrays
- "visible area": $\{y >= l_{max}, y >= l_{min}\}$
- P_{ri}: x coordinate of intersection of l_i and l_{max}
- P_{li}: x coordinate of intersection of l_i and l_{min}

Algorithm:

1. Divide the array of lines into 2 halves evenly
2. Sort the two subarrays by slope in non-increasing order
3. Find the intersection of l_{max} and l_{min}, say(m, n)
4. Compare $l_{nextmax}(m)$ with n, if $l_{nextmax}(m) < n$, throw it; if $l_{nextmax}(m) > n$, put l_{max} into "left_visible" array and update $l_{max} := l_{nextmax}$. Go to step 3 if there's $l_{nextmax}$, put the rest 2 lines into "left_visible" array and go to next step when there's no lines left
5. Find the intersection of l_{min} and l_{max}, say(m, n)

6. Compare $l_{nextmin}(m)$ with n, if $l_{nextmin}(m) < n$, throw it; if $l_{nextmin}(m) > n$, put l_{min} into "right_visible" array and update $l_{min} := l_{nextmin}$. Go to step 5 if there's $l_{nextmin}$, put the rest 2 lines into "right_visible" array and go to next step when there's no lines left

7. "visible" array="left_visible" array $\cap$ "right_visible" array. Basically, "left_visible" array is in non-increasing order slope-wise and "right_visible" array is in non-decreasing order slope-wise. Thus, it's easy to do the intersection by traversing the two arrays reversely

5.2 Time Complexity

Divide: Since we divide the array into 2 subarrays and solve them recursively, we have 2 subproblems of size $n/2$
Conquer: We have to find at most $2(n-1)$ intersections and always $2(n-1)$ comparisons, which takes $\mathcal{O}(n)$. Besides, we have to intersect two arrays, which includes at most n comparisons, also $\mathcal{O}(n)$
So overall we have $T(n) = 2T(n/2)+\mathcal{O}(n)$, according to Master's Theorem we get $T(n) = \mathcal{O}(nlogn)$

5.3 Proof

Lemma 5.3.1: l_{max} and l_{min} are always "visible" within its correspoding scope(i.e. array)
Lemma 5.3.2: l is "visible" if and only if it dominates at the intersection of l_{max} and l_{min}
Lemma 5.3.3: A list of non-increasing(slope-wise) lines $\{l_1, l_2, ..., l_n\}$ are visible if and only if the every single line is "visible" in "visible area" and the list of $\{P_{r1}, P_{r2}, ..., P_{rn}\}$ is also in non-increasing order and the list of $\{P_{l1}, P_{l2}, ..., P_{ln}\}$ is in non-decreasing order

Proof of 5.3.1: Let's first consider l_{max}: $l_{max} = a_{max}x + b$, here b is the biggest intercept of a series of parallel lines. Pick any line of a set of non-vertical lines $l = a'x + b'$. There's always an intersection $(\frac{b'-b}{a_{max}-a'}, \frac{a_{max}b'-a'b}{a_{max}-a'})$. Let $x' = max\{\frac{b'-b}{a_{max}-a'}\}$, then for any $x > x'$, l_{max} always dominates. Similarly, there's always a x'' on which l_{min} always dominates when $x < x''$.

Proof of 5.3.2: Assume there's a "visible" line l that doesn't dominate at the intersection of l_{max} and l_{min}, say(x', y'). Therefore, $l(x') < y'$. Since it's "visible", we know there's some $x = t$ on which l dominates. Then we can infer $a_l = \frac{l(t)-l(x')}{t-x'}$. However, when $t > x'$, $a_l > a_{max}$; Similarly, when $t < x'$, $a_l < a_{min}$. That's contradictory to our assumption, which means l must dominate at $x = x'$.

Proof of 5.3.3: It's obvious that every single line should be "visible" in the "visible area" otherwise it'll be dominated by l_{max} or l_{min} forever. Now assume the orderings of the two x coordinate lists are not as suggested, which means there's some $i > j$, $P_{ri} > P_{rj}$ and $P_{li} > P_{lj}$. That way, l_i will dominate l_j in the "visible area" and thus makes it "invisible"

Proof of correctness to our algorithm: According to *Lemma 5.3.1* we know it's always safe to keep the two special lines(overall l_{min}, l_{max}). According to *Lemma 5.3.2*, every time we throw a line, that line is forever dominated by the current l_{max} and l_{min}. By doing the throwing step we're getting as many "invisible" lines as possible. Finally, our algorithm will generate a set of lines conforming *Lemma 5.3.3*. So we know the array we get has all the "visible" lines.