Finding the Convex Hull of points in a plane

We tackle the problem using a divide and conquer approach. The dividing step, as usual involves breaking the problem into two sub problems. The conquer part (trickier) involves making a convex hull using two smaller convex hulls. The whole procedure and implementation is explained in the folowing sections.

Algorithm:

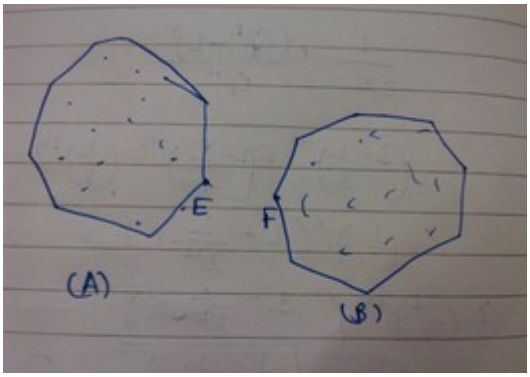
First, all the points are sorted in order of x coordinate.

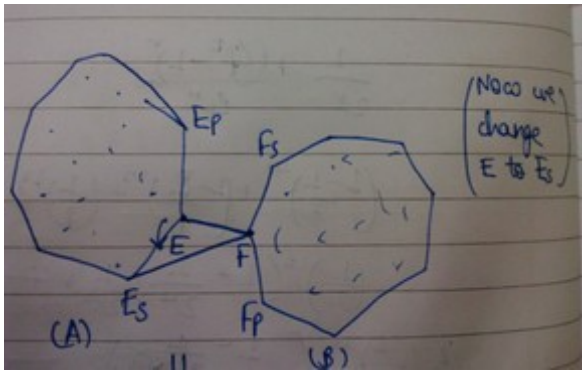
Then, the division involves dividing the points in two halves, one half containing points whose x coordinate is greater than the median and the other half contains the other points.

Now the function is called recursively and we finally get two convex hulls, one for the first set of points, and the other for the other set.

Notice that we have two convex polygons which do not contain nor intersect each other. Also appreciate that finding the convex hull of all the n points is equivalent to finding the convex hull of these two polygons.

Consider the figure given below, it contains the two convex hulls from the recursive calls,





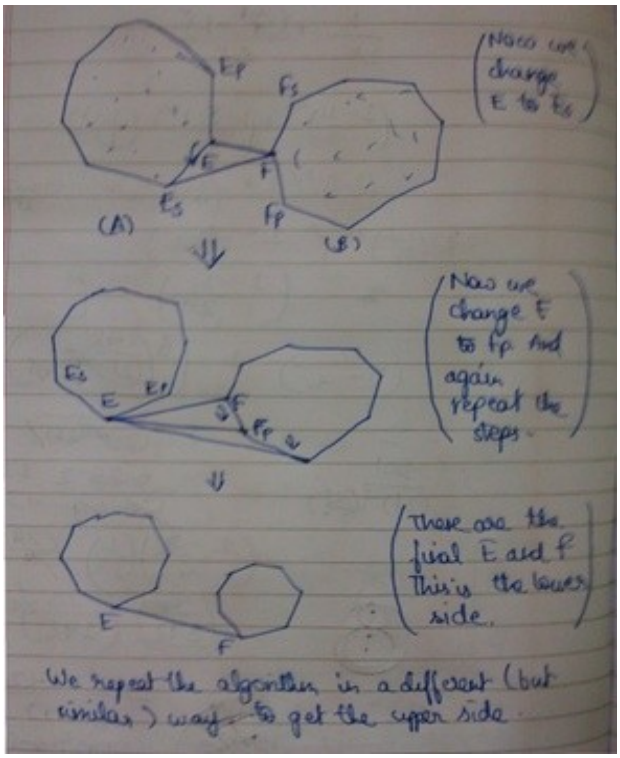
Now, we take the right most vertex of the hull A (let this be E) and the left most vertex of hull B (let this be F) (well, any two reasonable vertices would work).

In a manner similar to merge sort, if the angle between EF and EE_S (in clockwise direction) is less than 180 degrees we replace E by E_s. And if the angle between FF_p and FE (in clockwise direction) is less than 180 degrees we replace F by F_s. See figure for how it looks.

We end this when no replcement is needed. This will give us the lower side (a rough term).

For the upper side, the algorithm is roughly the same. But now we consider, E_pE and EF, and FE and Ff_s. Again we terminate when no replacement is required. See the figure for a test run for the algorithm,

Implementation:



The points are stored in an array, sorted according to x-coordinate.

The convex hulls are stored as circular linked lists, the points are stored in clockwise order which can be done easily using the recursive structure of our function calls.

So when we join two polygons, the circular linked lists can also be updated.

Finally we get a circular linked list containing our convex hull.

Space Analysis:

The total space taken is O(n). But the space in the recursive stacks kicks it up to O(n*logn).

Time Analysis:

Initially we sort the array in O(n*logn).

Then during the function call, the recurrence relation for the number of operations is,
$T(n) = O(n) + 2*T(n/2)$
Which means, $T(n) = O(n*\log n)$

Thus the time taken is O(n*logn).

Point in Convex Polygon

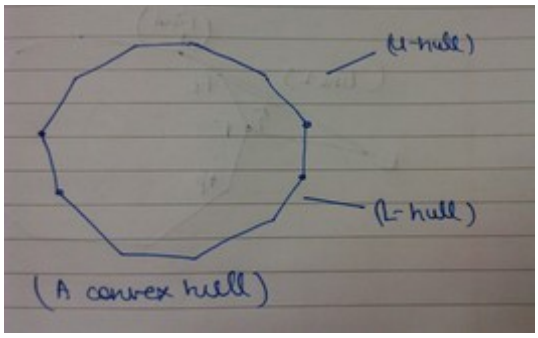
Introduction:

This is a fundamental problem used in future algorithms and is therefore discussed here.

Data Structures used:

Binary Search Trees:

We store the convex polygon as a binary search tree, actually, two BST's (the search takes place on the x coordinate). The details are as follows,



Consider the convex hull, take the right most and left most point. Notice that this divides the hull into two halves. Let one be the Upper hull (U Hull), then the other one is the lower hull (L Hull).

Now make a BST using the points of U hull, and another BST using points of L hull.
See the figure for a clearer view.

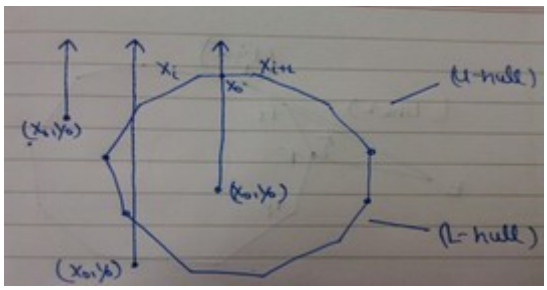
Algorithm:

Given a point (x_0,y_0), we need to find if it lies in the polygon or not.

We use the following theorum:
Shoot a ray in any direction. If it intersects the polygon an odd number of times, then it lies inside it. Otherwise it lies outside it.

So we shoot a ray $x=x_0$, upwards. Now we need to know at how many points does it intersect. But due to the construction of U hull and L hull, the ray will meet each of them at most once.

We use binary search on the BST's (U hull and L hull) to find the possible intersection with the ray. We search for x_0 in U hull. When the search finishes, we will have an x_i and x_{i+1}, where,
$x_i <= x_0 <= x_{i+1}$
Notice that the line segment joining these two points is the only place where the ray can possibly intersect. So we just need to check if it intersects or not. Similarly for L hull.



Thus we get a count of how many times the ray intersects the polygon, and thus we know whether it lies inside or not.

See the figure for a clearer view.

Time complexity analysis:

This algorithm involves only a couple of binary searches, so it takes O(logn) time.

Efficiently maintaining Convex Hull under addition of points

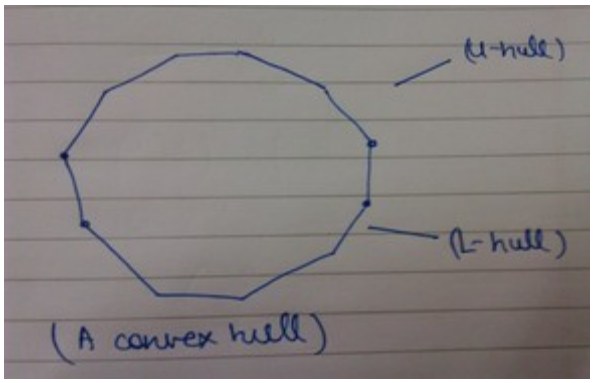
Introduction:

This was an extra question solved by us during the project. It made us think a lot, and also helped us to think about solutions to the future problems. Looking at the solution, we also realised the importance of data structures i.e. representing data in some specific format can make the problem quite easier. Since, it is an extra problem, we will just discuss the basics of the algorithm without going into too much detail.

Data Structures used:

Binary Search Trees:

We store the convex hull as a binary search tree, actually, two BST's (the search takes place on the x coordinate). The details are as follows,

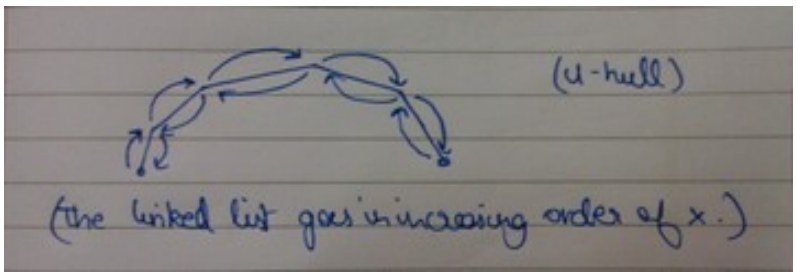


Consider the convex hull, take the right most and left most point. Notice that this divides the hull into two halves. Let one be the Upper hull (U Hull), then the other one is the lower hull (L Hull).
Now make a BST using the points of U hull, and another BST using points of L hull.
See the figure for a clearer view.

Linked Lists:

We store the points in U hull and L hull in two linked lists. This is mainly done for efficient Successor and Predecessor operations. The linked list stores points in a clockwise manner.

See the figure on how the linked lists are stored,



Algorithm:

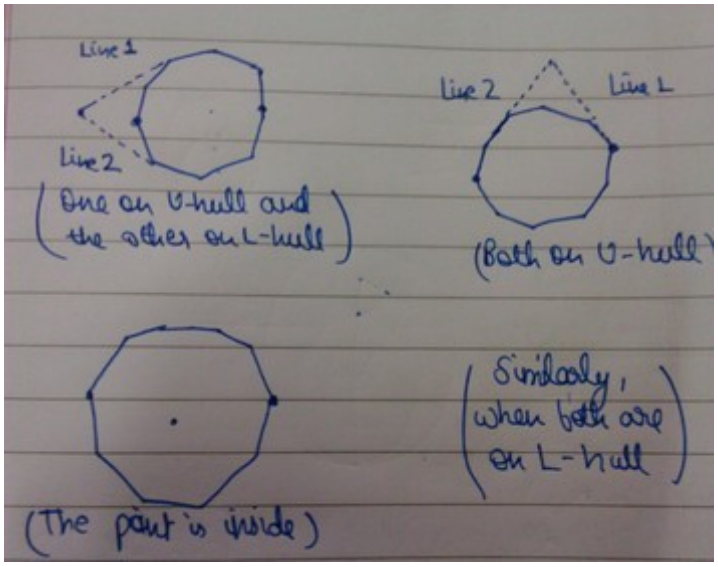
The convex hull should support efficient addition. Consider the time when a point is going to be added. We have a convex hull of the some points, now we

have to add a new point and change our convex hull accordingly.

First, we check if the new point is inside the convex hull or not. This can be done efficiently using the algorithm mentioned earlier.

If it's inside, then there is no change. But otherwise, the convex hull must be modified. Lets handle this case.

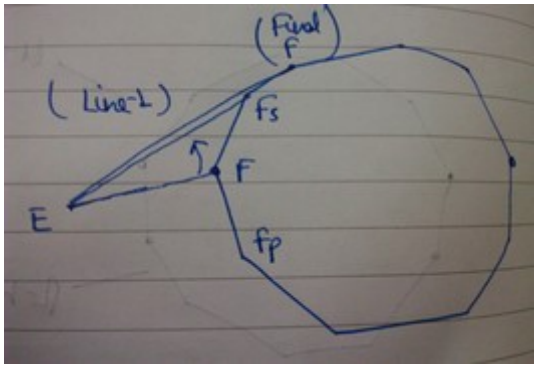
Consider any point and any convex polygon, the figure shows their possible orientations.



Let us define line-1 to be the upper line (U line) and line-2 as the lower line (L line).

While the angle between FE and FF_s is less than 180 degrees, replace F by F_s (Algorithm A). This gives us U line. A similar algorithm will give us L line, but we consider FF_p and FE in it (Algorithm B).

The figure below describes the algorithm to find the point of contact in the first case.



Notice that when you run algorithm A on a hull which does not have a point of contact of the U line lying on it, it will give us an invalid point (or NULL if the return value is modified). Similarly for Algorithm B.
Thus, instead of doing some nasty case work, we can run both Algorithm A and B on both U hull and L hull, and consider only the valid points returned by the functions.

It's not over yet, there is still a lot which needs to be done. Now that we have points C and D (the points of contact), we need to update our hull.

There's a lot of case work here, but the rough idea is as follows,

If C lies on U hull, then we split U hull through C, then discard the greater tree or lesser tree according to what is needed. Similarly for D.
If both C and D lie on U hull, then we once split through C, and once through D. Then take the required parts accordingly.
Finally we add the new point in either U hull or L hull (whichever is suitable).

The BST's are updated, now updating the linked list is quite trivial, since we just have to play around with the next pointer of the nodes.

Time complexity analysis:

Finding whether the point lies inside or not takes O(logn) time.

Then, finding C and D (points of contact) also takes O(logn) time, since the method is similar to binary search.

Splitting a BST also takes O(logn) time. Finally adding the new point in a BST also takes O(logn) time.

Thus the overall complexity is O(logn).

Line Polygon Intersection

Introduction:

This is another fundamental problem used in future algorithms and is therefore discussed here.

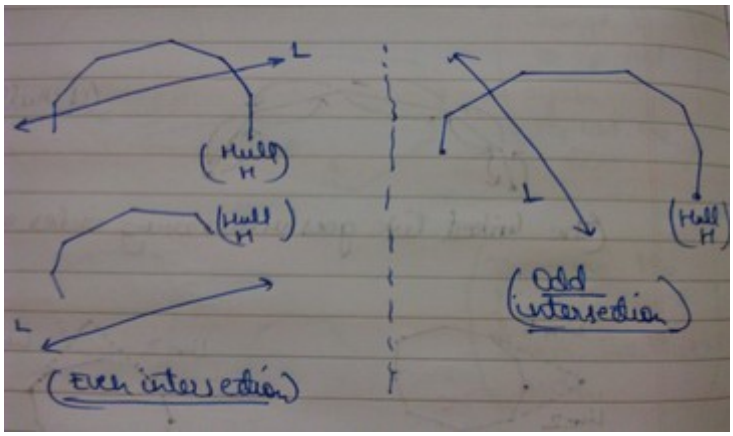
Data Structures used:

Binary Search Trees:

As done earlier, we store the convex polygon as a binary search tree, actually, two BST's (the search takes place on the x coordinate). The details are as follows,
Consider the convex hull, take the right most and left most point. Notice that this divides the hull into two halves. Let one be the Upper hull (U Hull), then the other one is the lower hull (L Hull).
Now make a BST using the points of U hull, and another BST using points of L hull.

Algorithm:

Consider any hull (U hull or L hull) and any line. There can be two possiblities, which are shown in the figure,



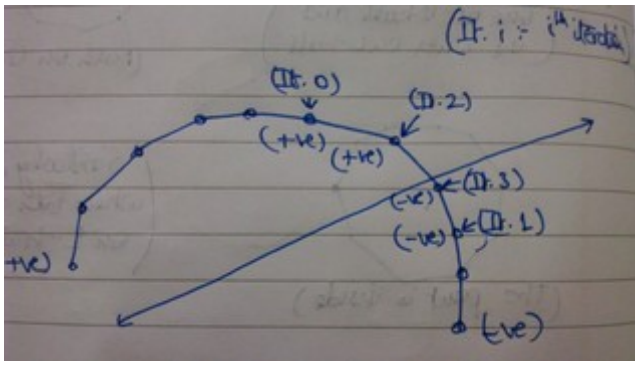
In the first case, the line intersects the hull two times, while in the other it intersects it one time.

So it can broadly be classified into, odd or even number of intersections.

The theorum used in this algorithm is,
When two points (x_1,y_1) and (x_2,y_2) are substituted in the equation of the line i.e. $y = f(x)$, then the sign of $(y - f(x))$ is same if they lie on the same side of the line.

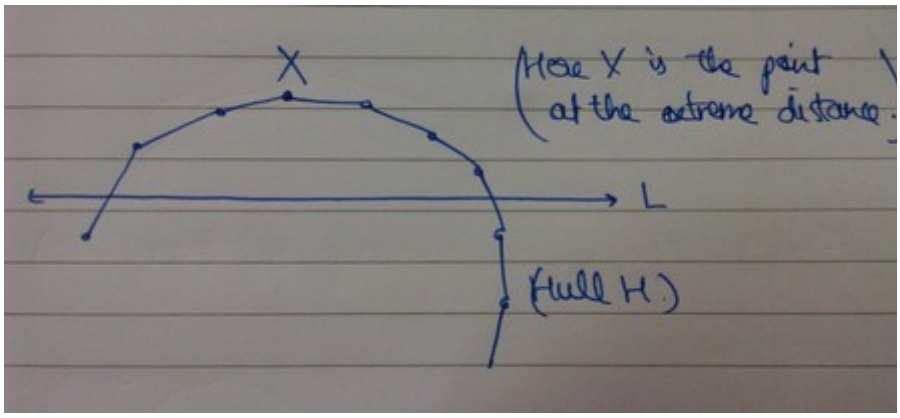
First, lets consider the case when there is only one intersection.

Notice that the extremes of the hull are on opposite sides of the line i.e. if I substitute them in the equation of the line, they will give opposite signs (since they are on opposite sides). So, we do a binary search in which we move away from the extreme which has the same sign. The figure explains the whole procedure,



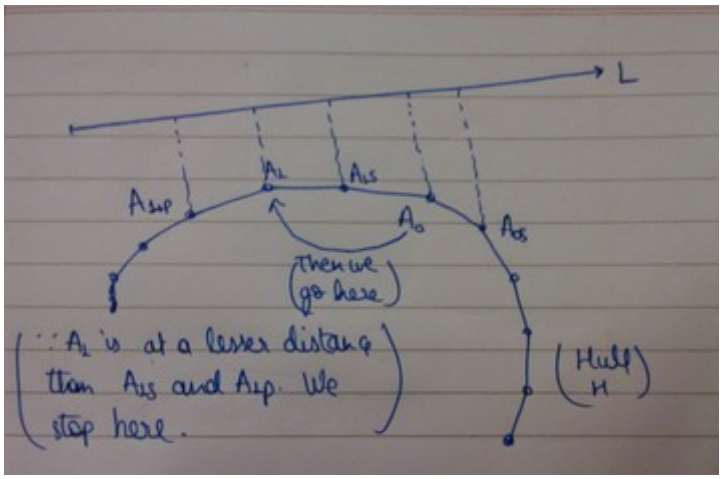
Now, lets handle the case when there are two intersections. Here the two extremes will give the same sign.

We try to find a point which is on the other side of the line (different side from the extremes). When we have this point, lets call it X, notice that we will have exactly one intersection to the left side of X, and one to the right of X. So if we split our hull (BST) through X, then we get two trees with exactly one intersection in them. And this is equivalent to the previous case.



So, to find this point X, we try to find the point which is at an extreme distance from our line.

See the figure for any clarifications,



Again we use a binary search approach for this, remember that the hull is convex. Let our current point be A, we consider the distances of A and A_s from the line. Then we either go to the left subtree or the right subtree depending on which side the extremes lie. The figure will give you a better view of the algorithm,

Thus, with this we can find X and follow the steps described in the scond case. Note that we don't really need to split the hull, we can modify the binary search while ensuring that we never cross the split path. Or a more simple solution is to use an array.

Time Complexity Analysis:

The algorithm mainly uses approaches similar to binary search, thus it take O(logn) time.

Half plane query - Number of points on a given side of a line

Introduction:

This is the first major problem which was solved by us and required quite a lot of thinking. The data structure used is quite unobvious, but it has it's own advantages and disadvantages. We will be discussing the data structure used and the related algorithms in this section.

Data Structures used:

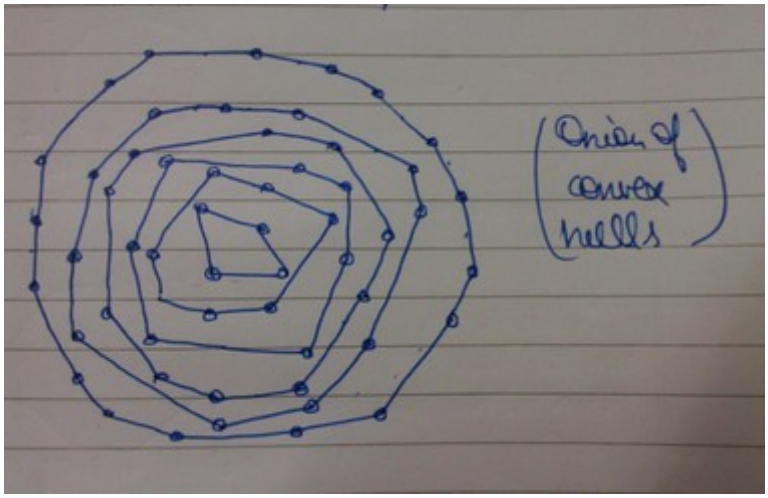
Onion of Convex Hulls:

Onion, it's a wonderful name as it aptly describes what this data structure looks like. An onion is infact a layer by layer joint of many convex hulls. The details for construction are as follows:

First find the convex hull for our set (the n points).
Now remove all the points which are present in the convex hull from the set.
Again find the convex hull for this new set.
Keep removing and repeating until there is no point left.

After this, we will have some convex hulls C_1 , C_2, ... , C_k such that,
C_i contains all C_j ($j < i$)

These C_i's form our onion, with C_k being the outermost layer of the onion and C_1 being the innermost one.



Notice that any set of n points will have a unique onion (Evident from the uniqueness of a convex hull). The figure given below shows an onion,

The convex hulls in the onion are stored as described previously using,

Binary Search Trees:

We store the convex hull as a binary search tree, actually, two BST's (the search takes place on the x coordinate).

Linked Lists:

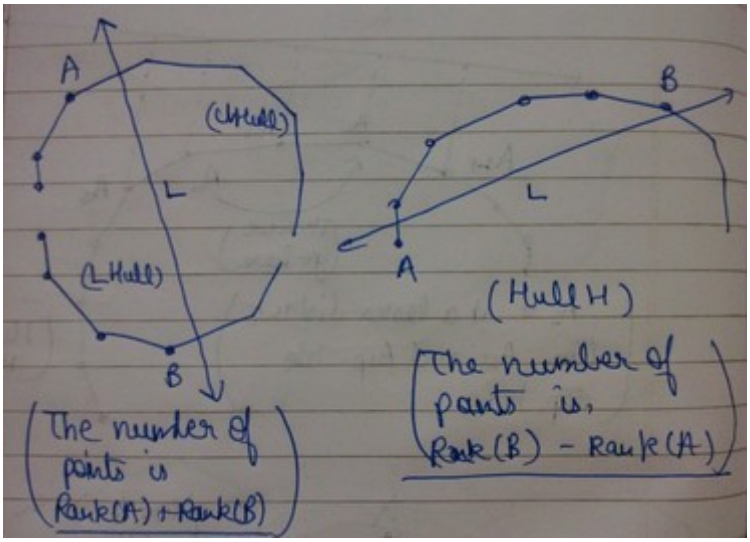
We store the points in U hull and L hull in two linked lists. This is mainly done for efficient Successor and Predecessor operations. The linked list stores points in a clockwise manner.

Algorithm:

The question boils down to finding the number of points on a given side of a line and a convex polygon.

The main idea is as follows,
The BST stores some other data also i.e. it's an augmented data structure. It stores the rank of the node in the BST. Now if the points of intersection of the line are A and B. Then the number of points on that side of the line is |Rank(A) – Rank(B)| or something depending on their ranks.

Now formally, if A and B are the points of intersection. Then we have the following cases,



A lies on U hull and B lies on L hull (and the complementary case)
Both A and B lie on the same hull (let it be U hull).

In the first case, the number of points on a given side is Rank(A) + Rank(B).
And in the second case, it's | Rank(A) – Rank(B)|.

See the figure for a rough explanation,

So, we first find A and B (the points of intersection), it can be done using the method discussed in the earlier sections. Then use the method described above.

To print all the points we just have to go from A to B in a sequential order, which can be done using the linked lists we had made.

Space Complexity Analysis:

First, let's consider the space required for storing a hull (in terms of a BST and a linked list). The BST takes O(k) space, where k is the number of points in the hull. The linked list also takes O(k) space. Thus, storing a hull takes O(k) space.

Now, in reality we have a whole bunch of hulls, i.e. an onion. Let the number of points in the first layer (hull) be k_1, in the second one be k_2 and so on. So the total space taken is,

$O(k_1) + O(k_2) + \ldots + O(k_c)$, where c is the number of layers.
Which is, O(n)
i.e. it's linear in the number of points. So the onion is actually a very space efficient data structure.

Thus the space required is O(n).

Time Complexity Analysis:

We break this into two parts, the first one is Pre processing time, and the other one is the Query time.

First, the pre-processing time. During the pre processing step, we make our onion. Let's analyse the time taken for that.

In the first iteration, we find the convex hull for n points. This takes O(n*logn) time. Then we remove the points which have made it into the convex hull. This takes O(k) or O(n) time (depending on the implementation), where k is the number of points in the convex hull.
In the next iteration, we find the convex hull of the remaining (n-k) points. Thus takes O((n-k)*log(n-k)) time. And the pattern continues.

So the overall time taken is,
$O(n*\log n) + O((n-k_1)*\log(n-k_1)) + \ldots + O(1)$
Now, if we want a loose bound, then the time complexity will be,
$O(n^2 * \log n)$

Thus making the onion i.e. the pre processing step takes $O(n^2 * \log n)$ time.

Now for the query step, there are two queries : one which involves printing all the points on any side, and the other which just asks for the number.

First, consider the query in which we print everything.
We open the onion layer by layer, in each step we consider the present layer i.e. hull and find the number of points lying on a given side. For this, we use the previously discussed line polygon intersection algorithm. It takes O(log k) time, where k is the number of points in the polygon. Then we move sequentially from point A to point B (intersection points), and print all the points on the way. This takes O(l) time, where l is the number of points lying on the given side.
This takes, O(log k) + O(l) time.
Then we repeat this for each hull, until we reach a hull with no intersection.

Thus the total time taken is,
$O(\log k_1) + O(l_1) + O(\log k_2) + O(l_2) + \ldots + O(\log k_c) + O(l_c)$

Now, this is a highly varying sum, and is highgly dependent on the line we choose. So, the best we can do is find a worst case bound for this, which happens when we encounter only one point ont he given side in every hull. So we would have to traverse 'm' hulls before terminating (where m is the number of points on a given side). So the worst case time becomes,
$O(\log k_1) + O(l_1) + O(\log k_2) + O(l_2) + \ldots + O(\log k_m) + O(l_m)$
which is, O(m*log n) + O(m) i.e. O(m*log n)

Thus the query takes O(m*log n).

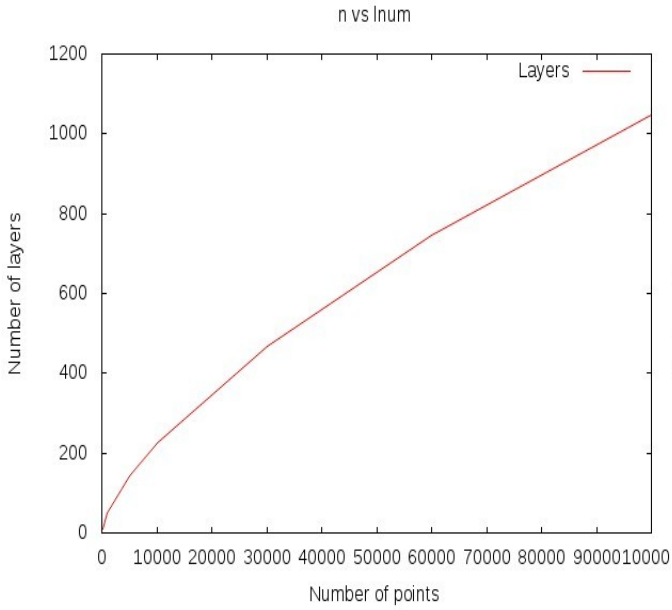
The analysis for the number of points query is also similar, it performs well on average. But the worst case is similar to the above i.e. O(min(m,k)*log n), where k is the number of layers in the onion and m is as defined above.

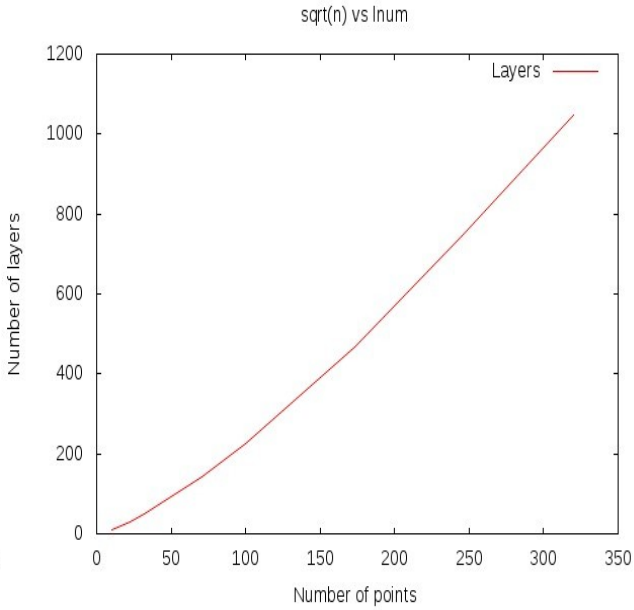
Thus the time taken by both type of queries is O(min(m,k)*log n) per query

Experimental Results:

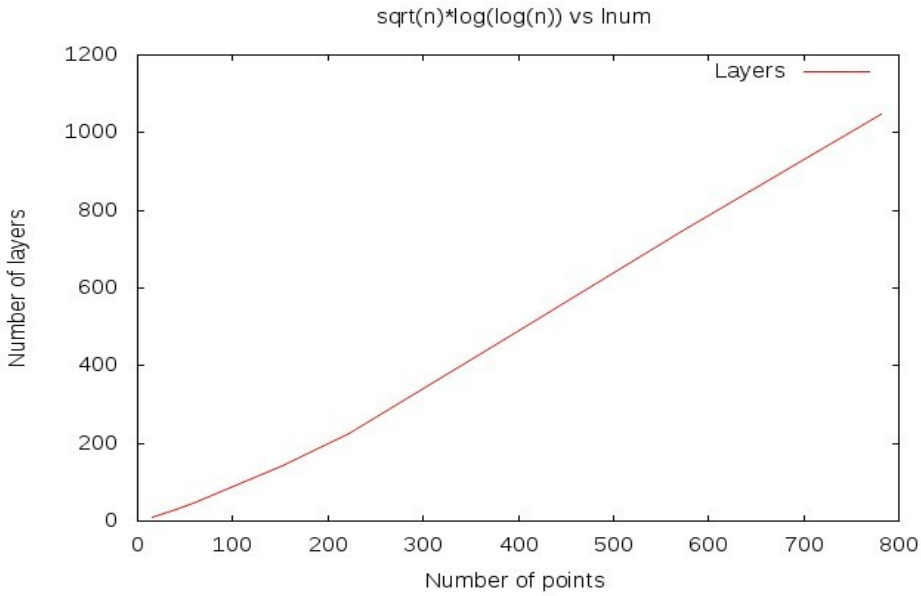
Number of layers in the onion:

We noticed that the number of hulls is almost constant for a given 'n'. So the plotted graph looks like,



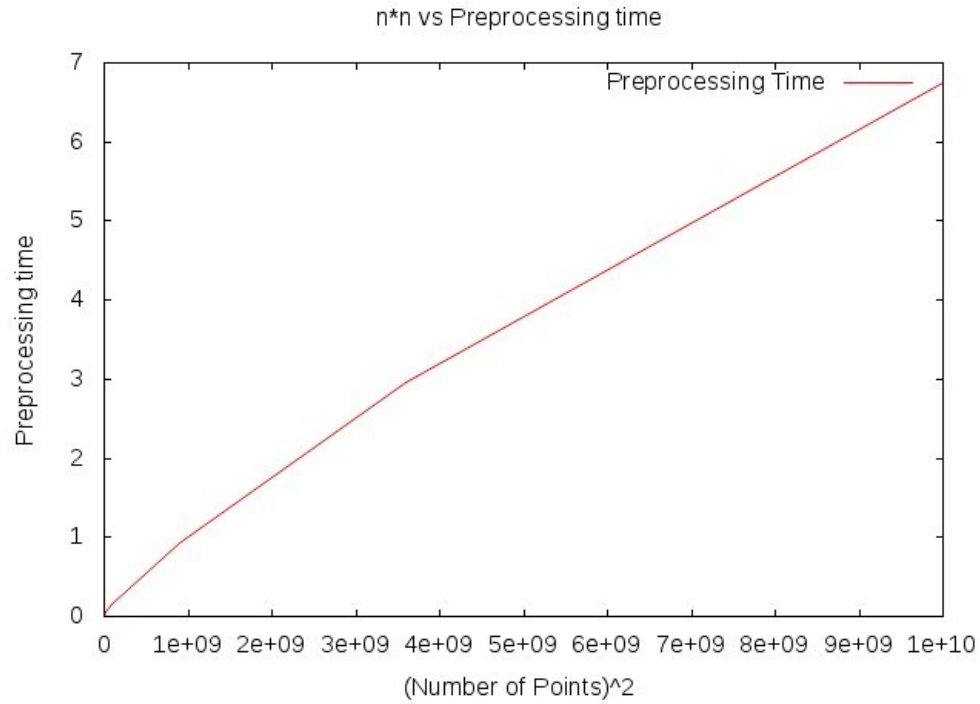


The above graphs are lnum vs n, and lnum vs sqrt(n) respectively. So, after another approximation, we felt that sqrt(n)*log(log(n)) is quite close to lnum. The graph is,



Preprocessing Time:

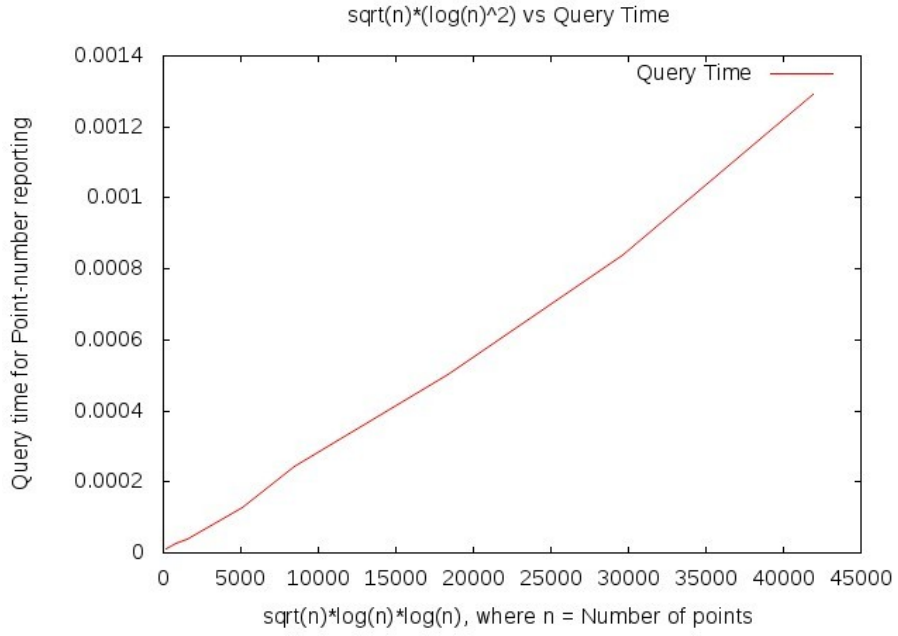
The graph of Preprocessing time vs n*n looks like,



So, due to the concave nature of the graph we can say that it is bounded above by n*n.

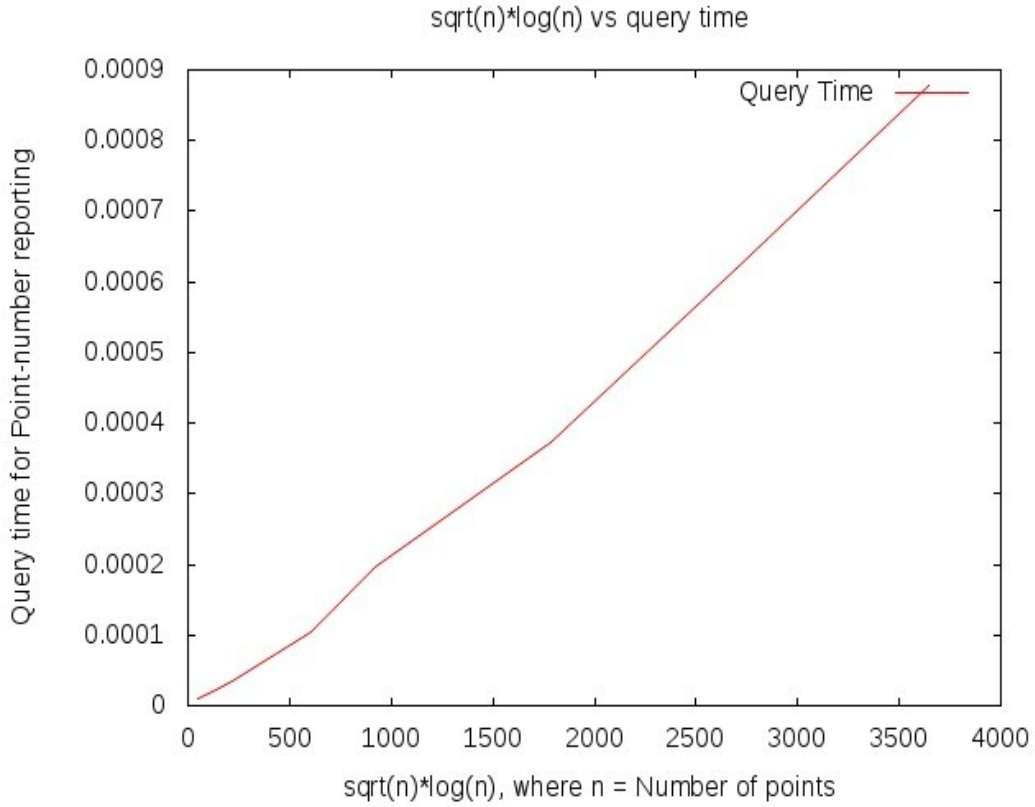
Query Time:

For point reporting the graph of time vs sqrt(n)*(log(n)*log(n)) looks like,



So we can say that the time is proportional to the ((number of points) * log(n)).

For point number reporting, the graph of time vs sqrt(n)*log(n) looks like,



Fractional Cascading:

Fractional cascading is a technique to make binary search in k different lists of similar data very efficient.

If the total number of elements in all the list in n, then trivial binary search leads to O(k*log(n/k)) solution.

By using fractional cascading and merging the lists, the task can be achieved in O(log(n) + k) time.

This technique can also be used in our half plane problem using convex hull layers. The time taken to search for the points of intersection can be sped up and the resulting time taken per query will O(log(n) + k).

The space complexity will be O(n*log(n)).

Thus, the performance of our solution can be improved to O(log(n) + k).

Rectangular query : Number of points in a rectangle

Introduction:

This is an extra problem solved by us during the pursuit of an efficient solution for the half plane problem. So we will just discuss some details of our solution.

Data Structures used:

The main data structure used is similar to a segment tree. Consider all the n points, they will be of the form (x,y). So we first sort all the points according to the y coordinate, let's say this array is A.

Now, imagine that we run merge sort on A, to sort this array according to the x coordinate. Then, in the first all all pairs of elements will be sorted, in the next call, all set of four elements would be sorted and so on. So we store all these miniature sorted arrays in our data structure.

So, this data structure can give us the sorted sub-array (i,j). Since (i,j) can be decomposed into intervals which are a power of 2, and since our data structures stores the sorted subarray for that case, thus we can find it quickly.

Algorithm:

Let's say we need to find the numbe rof points in the rectangle defined by (x_1,y_1) and (x_2,y_2). So, this can be broken into a sub problem in which we need to find the number of points in the rectangle defined by (0,0) and (x,y).

To do this, we first search for 'y' in array A, using binary search. Now, we have a sub array of A in which all points have y coordinate less than y. Now we want to find the number of points which also have x coordinate less than x. For this we use our data structure, using binary search for x in all the decomposed sub arrays.

Thus, in the end we have the number of points in the rectangle.

Space Complexity Analysis:

Let's find the space taken by our data structure. Notice there will be O(log n) levels, in which we store n elements at each level. So the space complexity is O(n*log n).

Time Complexity Analysis:

Let's consider the pre processing time. It takes the same time as merge sort i.e. O(n*log n).

Now for the query step, we first use a binary search for y in array A. It takes O(log n).
Next we decompse our sub array into intervals (which are powers of 2). There can be maximum O(log n) of them. We perform binary seach in each of them. So the time taken at each of it is, O(logn). Thus the total time is O((log n)^2).

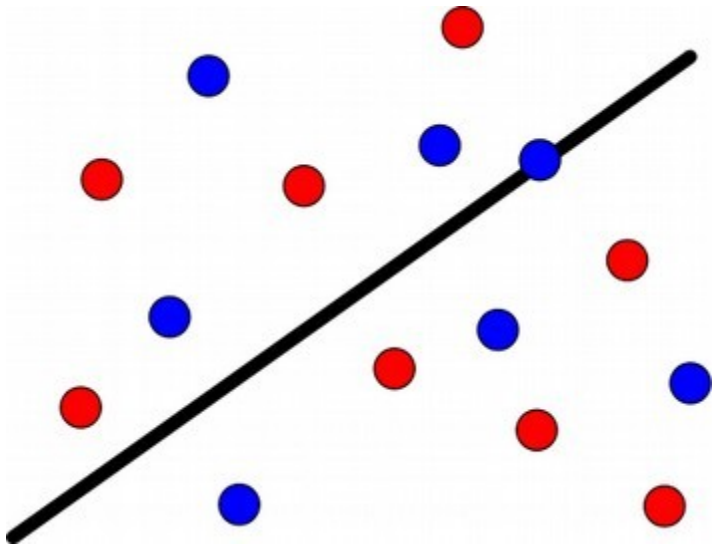
The overall time complexity is O((log n)^2).

Ham Sandwich Theorum

Ham Sandwich theorum is actually a general for 'n' dimensions. But, we will be using only the two dimension version, which is,

For a finite set of points in the plane, each colored "red" or "blue", there is a line that simultaneously bisects the red points and bisects the blue points, that is, the number of red points on either side of the line is equal and the number of blue points on either side of the line is equal.

It is depicted in the following figure,



The line which divides it in equal parts is called a Ham Sandwich cut. We use the HS (Ham Sandwich) theorum to prove the existence of two lines, which divide the plane into four quarters, each of which contains the same amount of points.

The approach is further explained in the following sections.

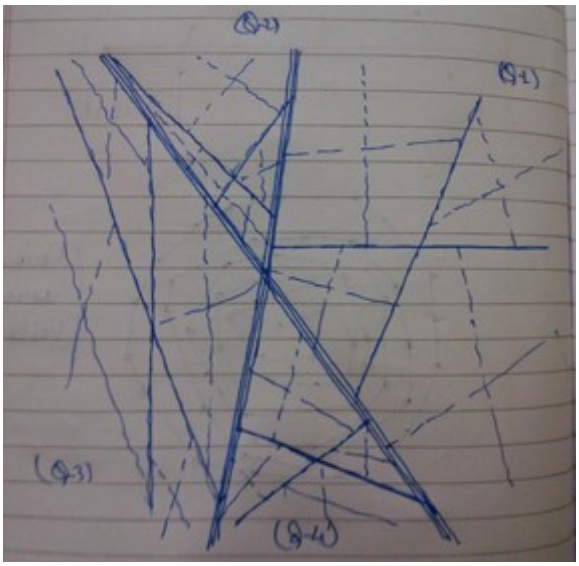
Half Plane query : Using Ham sandwich cuts

Introduction:

This is an alternative solution to the half plane problem discussed earlier. This solution involves Ham Sandwich cuts which we've just discussed. This is a sub prolem used to solve the main problem of our project. We will be discussing the data structure used and the related algorithms in this section.

Data Structures used:

Originally we have n points. From the discussion in the last section, we know that there are two lines which divide the plane into four quadrants, such that all the quadrants contain the same number of points.



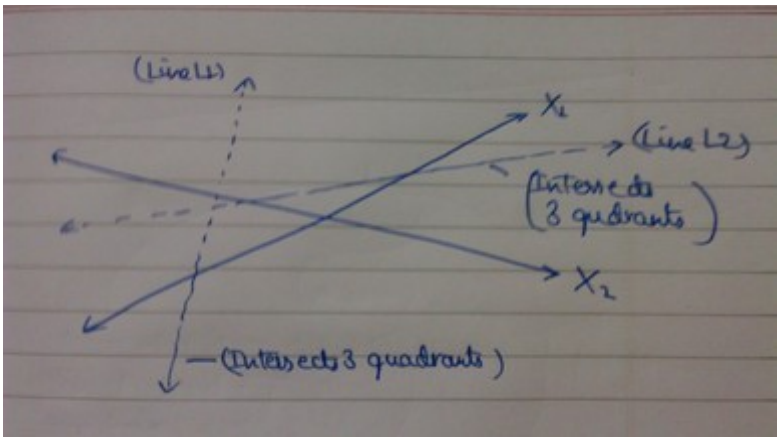
Now we consider each of the quadrants, again there exist two lines which divide this into four equal quadrants. We keep repeating this procedure till there is only one point.

The final data structure looks like the following,

So, the data structure we use contains sets of points which belong to a quadrant.

Algorithm:

Lets say we are given any line, L. Consider the two lines which divide the plane into four equal quadrants, i.e. X_1 and X_2. We now have four quadrants, and a line L.



So, we claim that L intersects at most 3 quadrants. This can be seen in the figure,

So the problem boils down to finding the number of points in these smaller problems.
Thus, at each iteration we can discard atleast one-fourth of the points.

Space Complexity Analysis:

We are stroring sets of points in our data structure.
The first level of points contains n points. The next level contains 4 smaller sub-problems, and each contains (n/4) points. Thus, it also contains n points.
The next level again contains a total of n points.
So, each level contains n points, and there are log n such levels. Thus the total space required is,
O(n*log n).

The space complexity is O(n*log n).

Time Complexity Analysis:

The recurrence relation for the number of operations is,
$T(n) = O(1) + 3*T(n/4)$
Which means, $T(n) = O(3\text{^}\log_4 n) = O(3^{(\log 4(n))})$

This can also be written as,
$T(n) = O(n^{\log 4(3)}) = O(n^{0.792})$

Thus the time taken is $O(n^{\log 4(3)}) = O(n^{0.792})$.

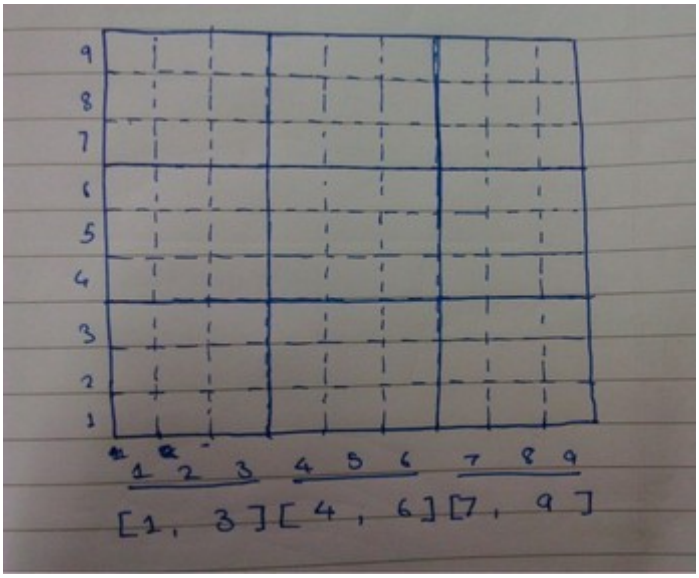
Simplex problem : First Approach

Introduction:

This was the first solution we were able to think of to answer a simplex query efficiently. We will quickly discuss the algorithms and data structures used.

Data Structures used:

Since we have n points, at best we can get n distinct x and y coordinates.
First we map all the x coordinates to the range [1,n] and do the same for the y coordinates.



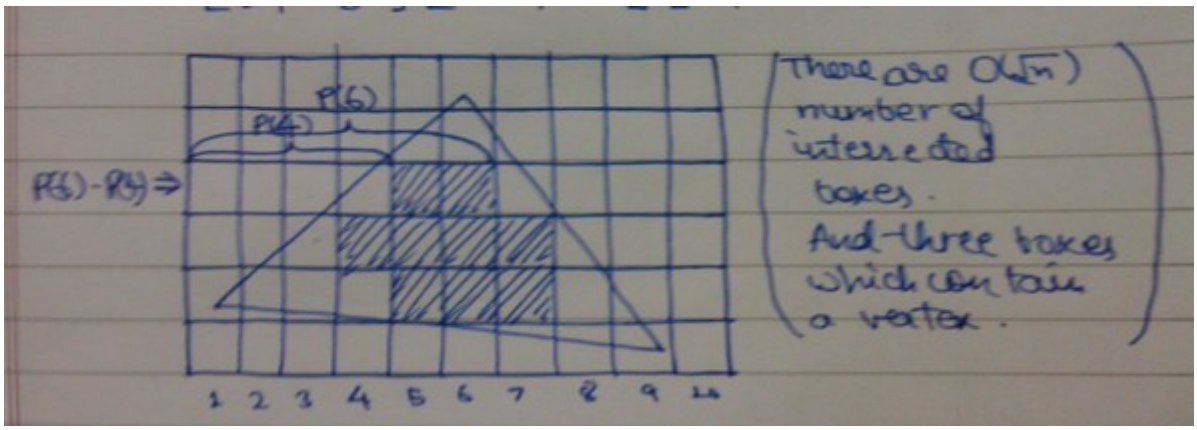
Now consider a grid of size n x n. We merge the interval [1,sqrt(n)], [sqrt(n)+1, 2*sqrt(n)] and so on. Now we have a grid of size sqrt(n) x sqrt(n).
Here in each box, we store the number of points lying inside it.

The figure describes the data structure.

Algorithm:

Consider a triangle, it consists of three line segments. If we put this triangle over our grid. Then the maximum number of boxes it can intersect is O(sqrt(n)).

The number of points in the boxes which are lying completely inside the triangle can be calculated usng the method in the figure,



Now, there will be atmost three boxes which will contain the vertex of the triangle. The number of valid points can be simply calculated by considering all the points in these boxes.

Time Complexity Analysis:

Notice, that the maximum number of points in a box can be sqrt(n).
First, we find the boxes which intersect the triangle in O(1). Next we calculate the number of points for the boxes completely inside the triangle, this takes O(sqrt(n)).

Now to calculate the number of points in the boxes which are intersected by the line, we consider this as a half plane query. Which takes $O(n^{0.792})$ for n points. But each box contains, sqrt(n) points in the worst case. So the time taken for this is $O(n^{0.5} * n^{(0.792/2)})$ which is $O(n^{0.896})$.

Finally, the valid points in the boxes in which the vertex lie, can be found out by just checking all the points in them. This takes O(sqrt(n)).

Thus the overall time complexity is $O(n^{0.896})$.

Simplex query : The Second Approach

Introduction:

This is an improved solution to solve the simplex problem. This is the main problem in the project, and took a considerable amount of effort. This solution involves an approach which is similar to the one used to solve half plane query. We will be discussing the data structure used and the related algorithms in this section.

Data Structures used:

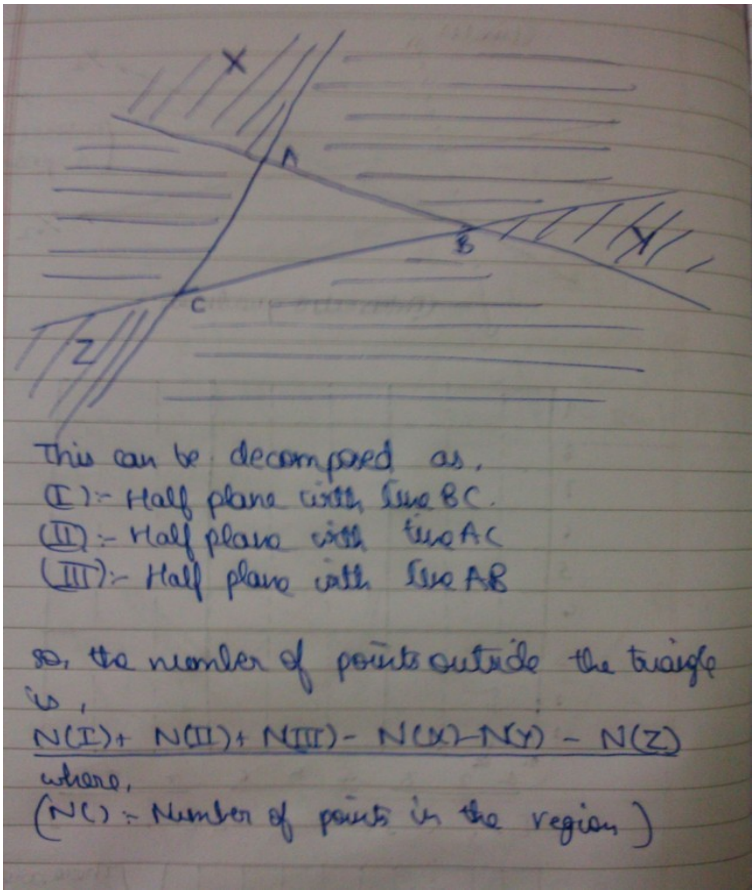
Originally we have n points. From the discussion in the last section, we know that there are two lines which divide the plane into four quadrants, such that all the quadrants contain the same number of points.

Now we consider each of the quadrants, again there exist two lines which divide this into four equal quadrants. We keep repeating this procedure till there is only one point.

The final data structure looks similar to the one discussed in the half plane problem.

So, the data structure we use contains sets of points which belong to a quadrant.

Algorithm:



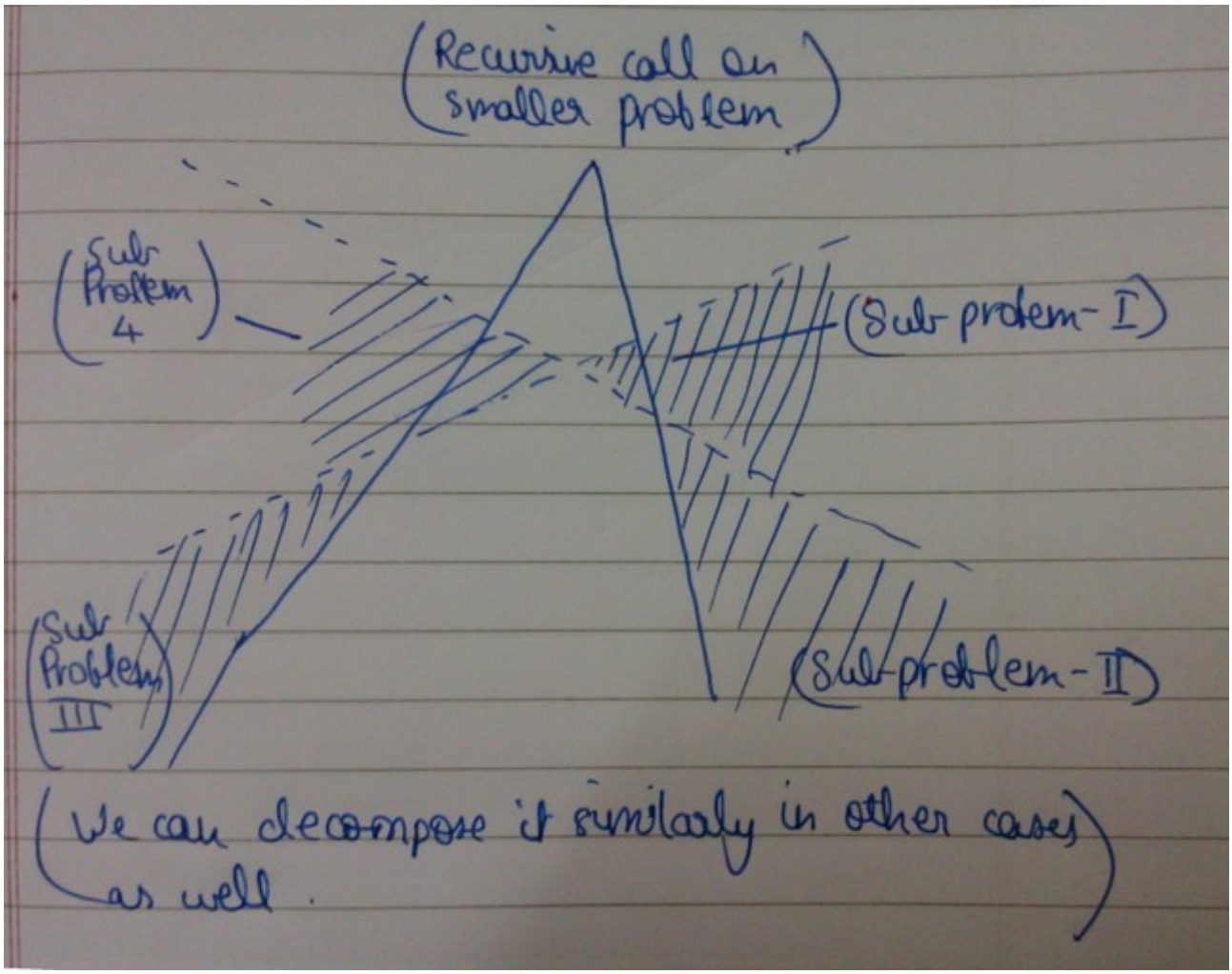
Lets say we are given any triangle, then it will be made up of three line segments. Consider the two lines which divide the plane into four equal quadrants, i.e. X_1 and X_2. We now have four quadrants, and a line L.

Now, to find the number of points inside the traingle, we try to find out the number of points outside the triangle. The way we decompose the problem can be seen in the figure.

Three of the sub-problems are simple half plane queries. We will try to develop an algorithm to find a solution to the V shaped problem.

We claim that the V shaped problem can be decomposed into three or four half plane queries and a smaller V shaped problem.

It can be done as shown in the figure,



Space Complexity Analysis:

The space complexity is O(n*log n). It was discussed while analysing the half plane query.

Time Complexity Analysis:

The recurrence relation for the number of operations in a V shaped query is,

$T(n) = O(n^{0.792}) + T(n/4)$

Which means, $T(n) = O(n^{0.792})$

So, to solve the simplex query, we make three half plane queries and three V shaped queries. This takes a total time of $O(n^{0.792})$.

Thus the time taken is $O(n^{\log_4(3)}) = O(n^{0.792})$.