COMPUTATIONAL GEOMETRY

Lec 20

DUALITY

What is a duality transformation?
If we look at any point in 2D, it has two coordinates

$$p: (p_x, p_y)$$

If you look at any line in 2D-plane, it can also be expressed using 2 parameters

$$l: y: mx + b$$

↳ Any line except the vertical line

Hence, line & point both can be defined using 2 parameters. Of course, the limitation of this approach is that we cannot define a vertical line. But we take care of that case by slightly rotating it such that it is no longer vertical.

Since both of them only have 2 parameters, we can use these to map line to a point & point to a line.

$p: (p_x, p_y)$ → we can transform this to $y = p_x x - p_y$ ⇑ Represented as p^*

Similarly, the line $l: y: mx + b$ can also be mapped to a point having coordinates as $(m, -b) \Rightarrow$ Represented by l^*

$p: (p_x, p_y)$ ⟷ $y = p_x a - p_y$ $\quad p^*$

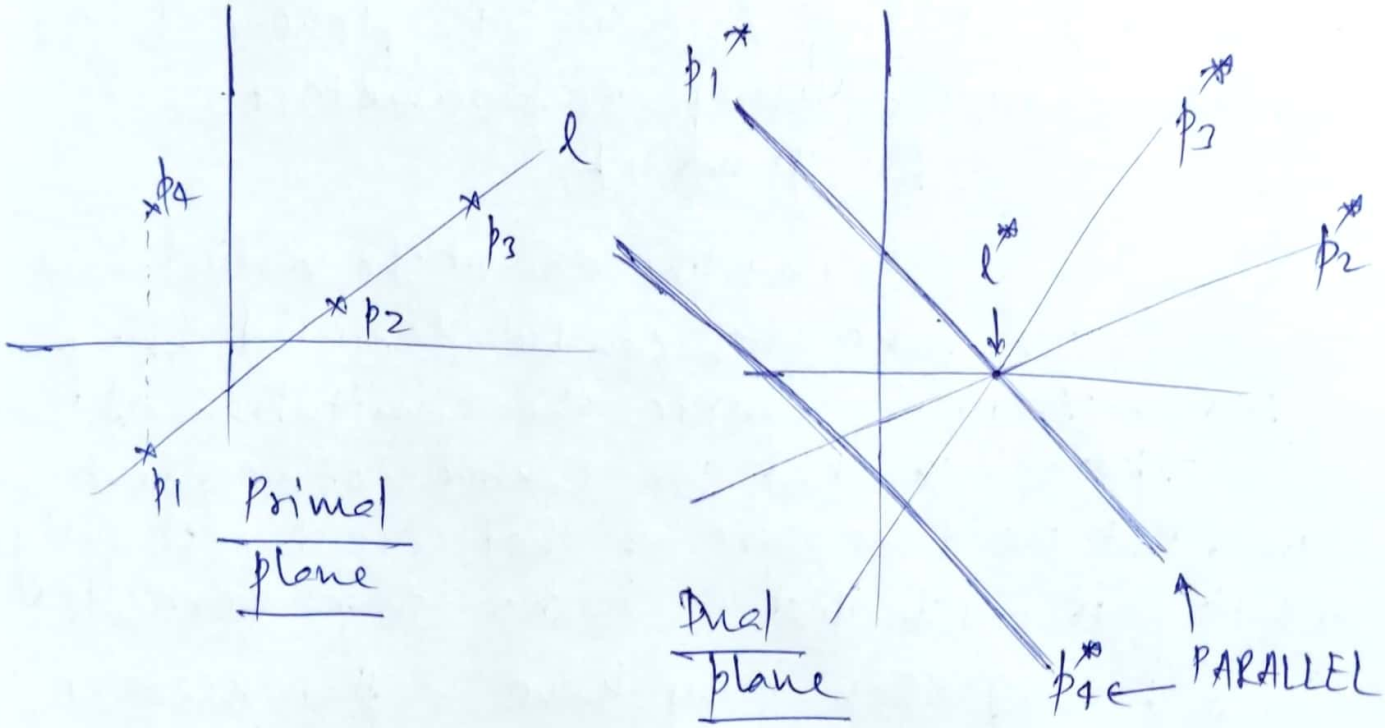
mappings

$l: y = mx + b$ ⟷ $(m, -b)$ $\quad l^*$

These are given in primal plane

These are corresponding objects in the dual plane.

This transformation has a very exciting property Let's take an example.



Several properties can be observed.

They are as follows:-

(point) (line)

(1) if $p_1 \in l$, $l^* \in p_1^*$

If p_1 is incident on line l, the property of incidence is preserved in the dual plane also, i.e., l^* is incident on p_1^*.

This is valid for points p_2 & p_3 also which indicates all of p_1^*, p_2^*, p_3^* will pass through l^*.
This is known as

INCIDENCE PRESERVING PROPERTY

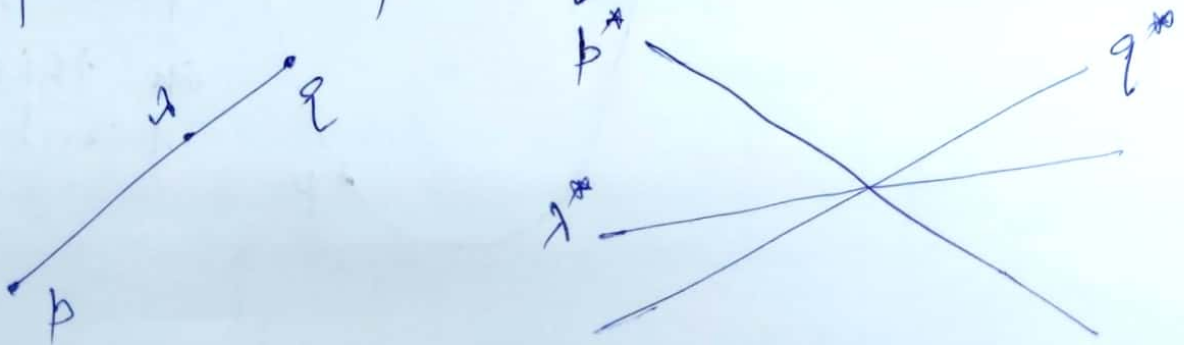
(2) It also preserves the order.
In the previous diagram,
p_4 is above l
In the dual plane,
l^* is above p_4^*

Point is above in primal plane $\Leftrightarrow$ Point is above in dual plane.

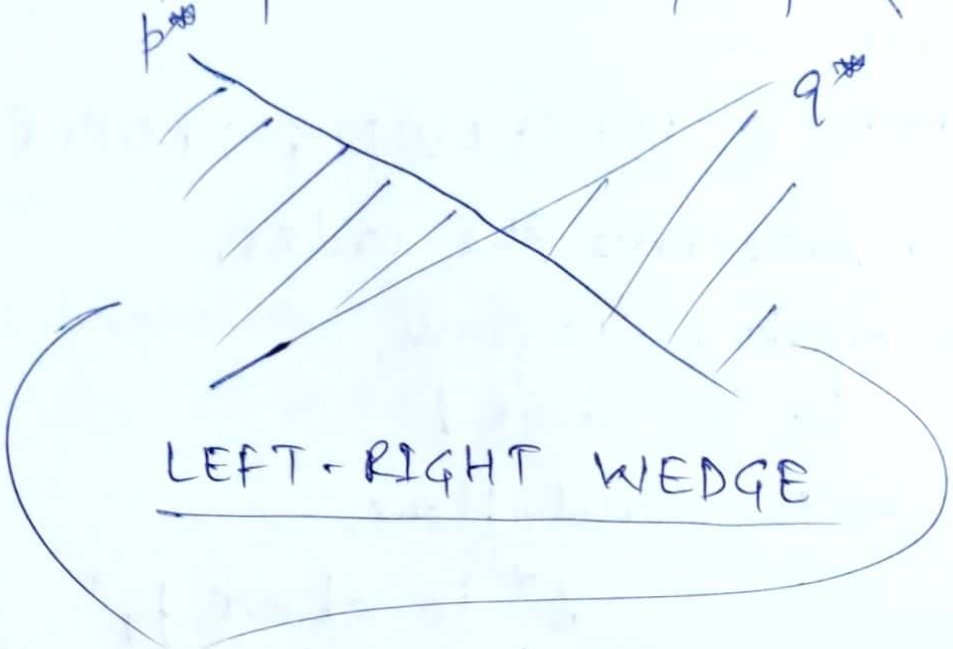
Point is below in ~~dual~~ primal plane $\Leftrightarrow$ Point is below in dual plane.

Hence, ORDER of point-line is preserved.

Using the definitions, we can also map some complex objects
Suppose we have a line segment having endpoints as p & q.



If we take the infinite points on the line segment then in the dual plane, they become the infinite lines b/w p^* & q^*.



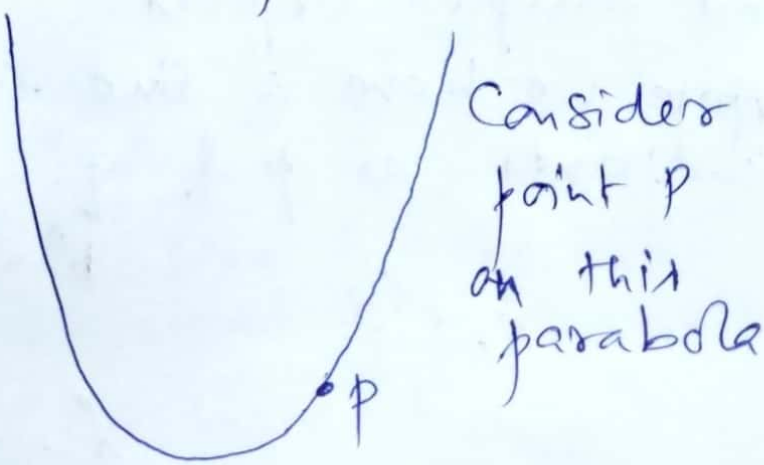
This area will be fully covered by infinite lines all passing through C
Hence, a line segment becomes a wedge in the DUAL plane,

We can also look at the geometric representation of this

This has a nice geometric property.

Suppose if we consider the parabola

$U: y = \frac{x^2}{2}$

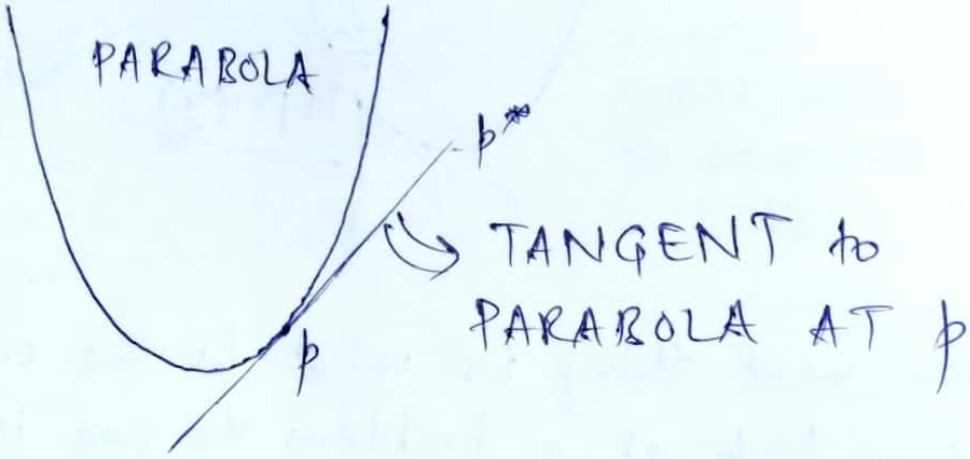


Consider point p on this parabola

$p : (p_x, p_y)$

$p^* : y : p_x x - p_y$

$$\frac{dy}{dx} = p_x$$



Slope of tangent of $y = \frac{x^2}{2}$

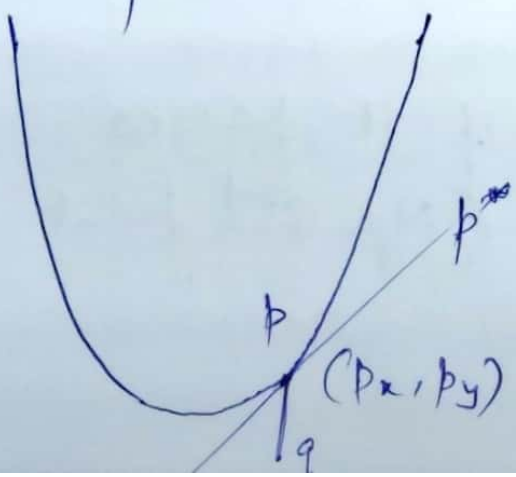
$$\frac{dy}{dx} = x$$

At p, slope is p_x. Hence, in the dual plane, p^* is a tangent to the parabola at p.

(IT PASSES THROUGH p)

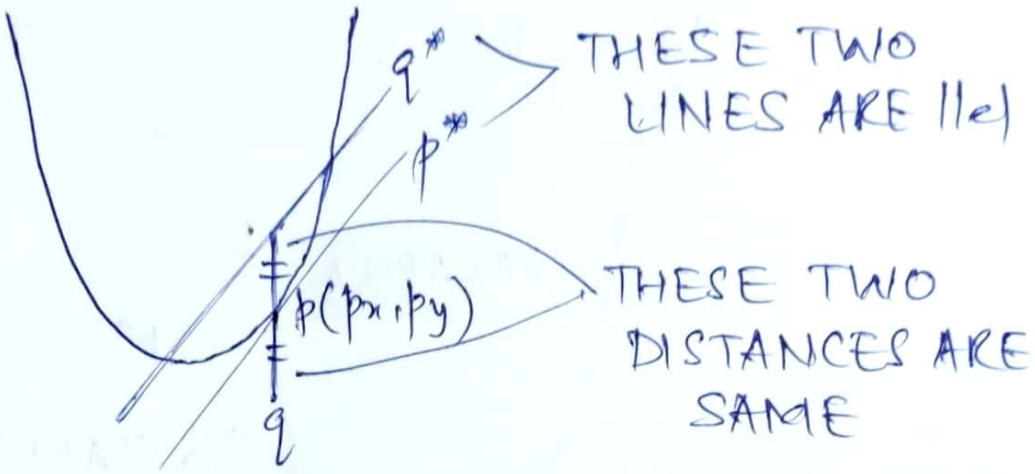
(PUT COORDINATES OF p in eq^n of p^*)

Let us take a point which is not a part of the parabola.



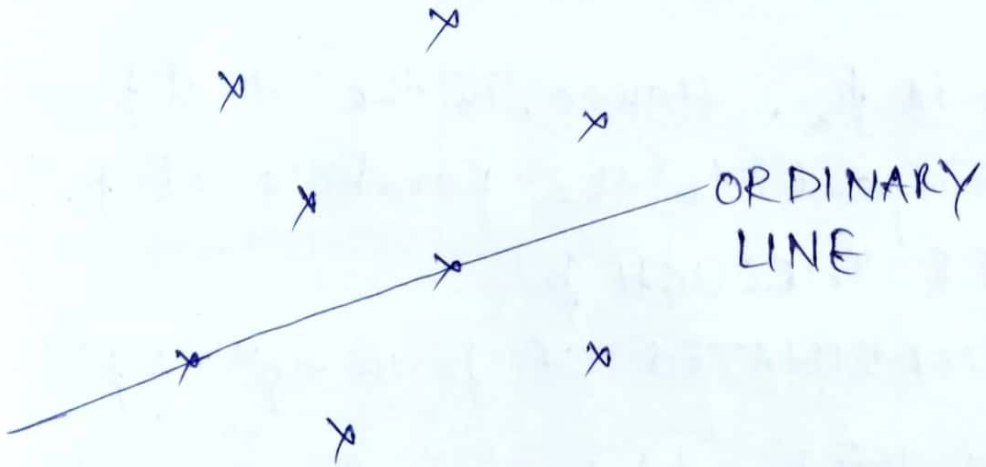
Let us take a point q vertically below point p

This point q becomes q^* in the dual plane with a property that



Now, next thing is what is the use of duality? Let us look at a problem to see its applications.

Suppose n points are given in a plane.



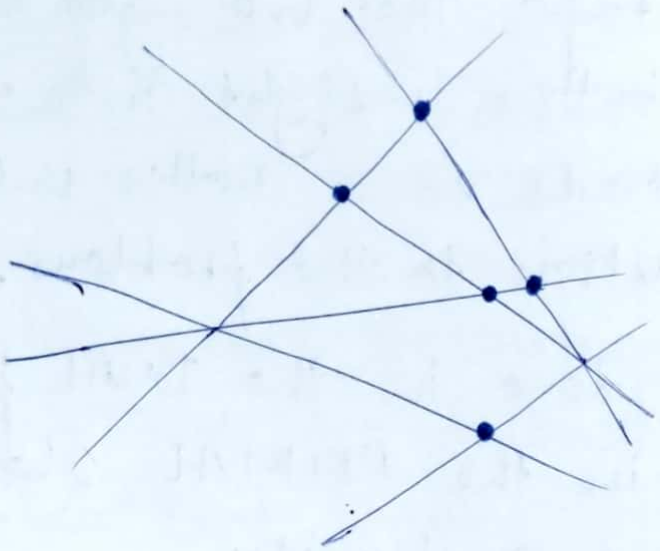
There are many lines passing through these points.

We call a line ordinary if it passes through only 2 points (No 3rd point lies on this line)

Suppose we are asked to find all ordinary lines, how can we find this out?

We can use Duality here,

In the dual plane, all the n points become lines



- POINTS CORRESPONDING TO ORDINARY LINES

In the dual plane, the ordinary lines became vertices where only two lines meet.

We can see some structure & property in the dual plane which we weren't able to see in the primal plane.

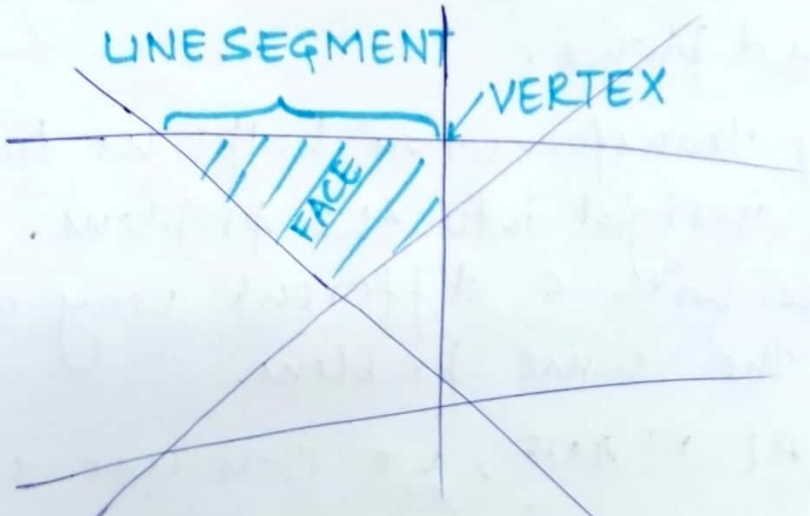
This duality transformation helps us to give some better insight into the problem. It provides us with a different way of looking at the same problem.

In the DUAL PLANE, we can use a SWEEP LINE algorithm & can count the number of ordinary lines in $O(n^2)$ TIME COMPLEXITY

In fact, the same algorithm can also be carried out in the primary plane with the same time complexity. So there wont be any difference as such.
The only advantage that we get is we achieve a better insight into the problem which leads to a better understanding & hence the solution to the problem.

Whatever can be done in the DUAL plane can also be done in the PRIMAL plane with the same time complexity.

Here, we have n lines in a plane.
This is what is called an
ARRANGEMENT OF LINES.



From computer science point of view, we want to store this arrangement in a data structure. What data structure is suitable for this arrangement?

By arrangement, we mean the line segments, the vertices & faces. Everything together is what is called arrangement of these lines.

For storing this information,
we use Doubly Connected Edge List (DCEL)

We have seen this for representing Voronoi diagram, Delaunay triangulation & many other 2-dimensional straight line embedded graphs.

Now we move onto the complexity of this arrangement.

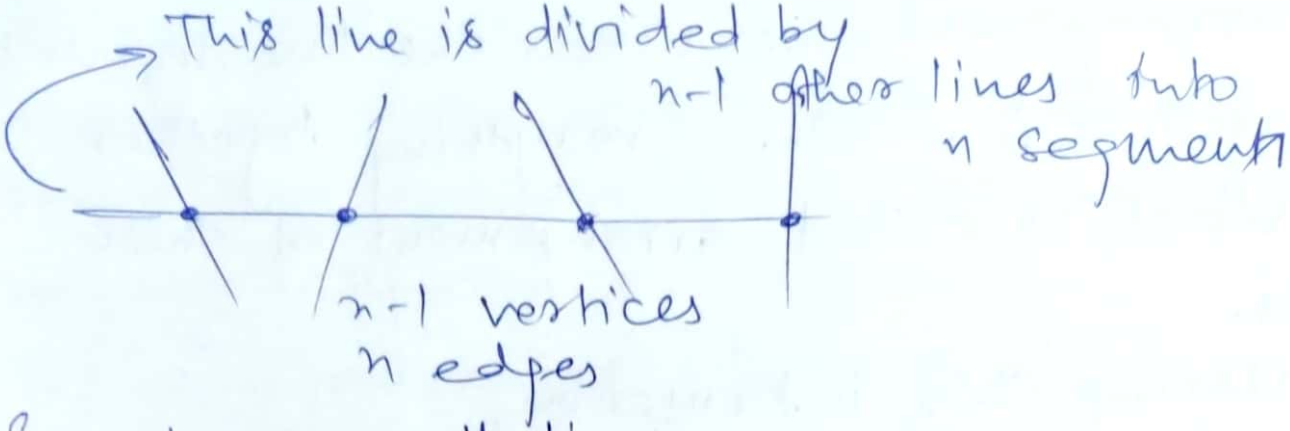
Complexity of arrangement means the total no. of vertices + edges + faces together. It is a measure of the number of objects in this Data structure.

If we estimate these crudely,

no. of vertices $(\#v) \leq \frac{n(n-1)}{2}$ ← Each vertex is at intersection point of two lines. Worst case occurs when every intersection point is unique.

no. of edges $(\#e) \leq n^2$

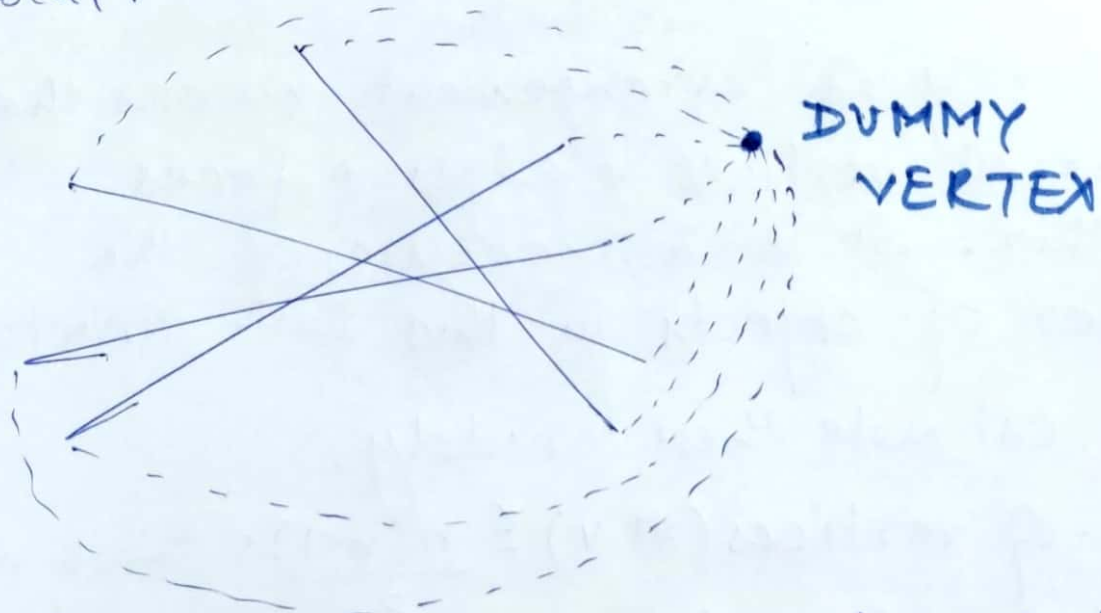
no. of faces $(\#f) \leq \frac{n^2}{2} + \frac{n}{2} + 1$



Summing over all lines,

$$\#e \leq n * n = n^2.$$

for faces, we will apply Euler's formula. But to be able to apply that, we need to introduce a dummy vertex & join the lines to that vertex. This will take care of multiple unbounded faces initially present.



this will ensure all faces are bounded except 1 face

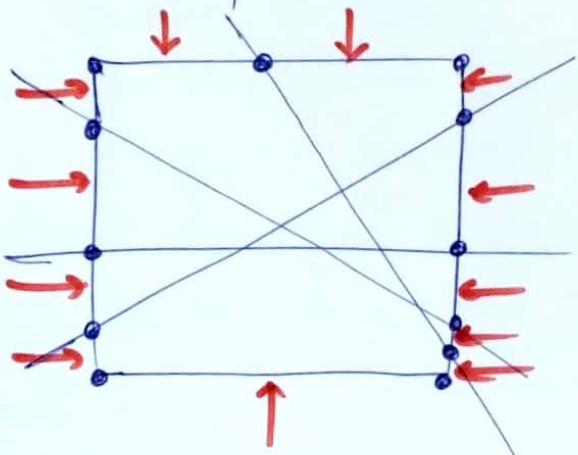
$$m_v - m_e + m_f = 2$$

$$\left(\frac{n(n-1)}{2} + 1\right) - (n^2) + m_f = 2$$

$$\Rightarrow \boxed{n_f = \frac{n^2}{2} + \frac{n}{2} + 1}$$

Now, the next question is how to compute this arrangement.

When we store the information regarding the arrangement in a DCEL, we cannot have multiple unbounded faces. Hence, we bound the arrangement in a rectangle & store its information.



These vertices & edges are also part of DCEL.

Hence, with a minor modification, we can construct the DCEL of this arrangement.

We can use a Sweep Line algorithm.

Here, we will discuss an Incremental algorithm which is different from what we have done earlier.

COMPUTATIONAL GEOMETRY

Lec 21

At the end of last class, we said that the DCEL for an arrangement can be constructed using a Sweep Line algorithm.

TIME COMPLEXITY = $O(n^2 \log n)$

n^2 intersection points are present & we have to maintain them in a priority queue where insertion & deletion take $O(\log n)$ time per insertion/deletion.

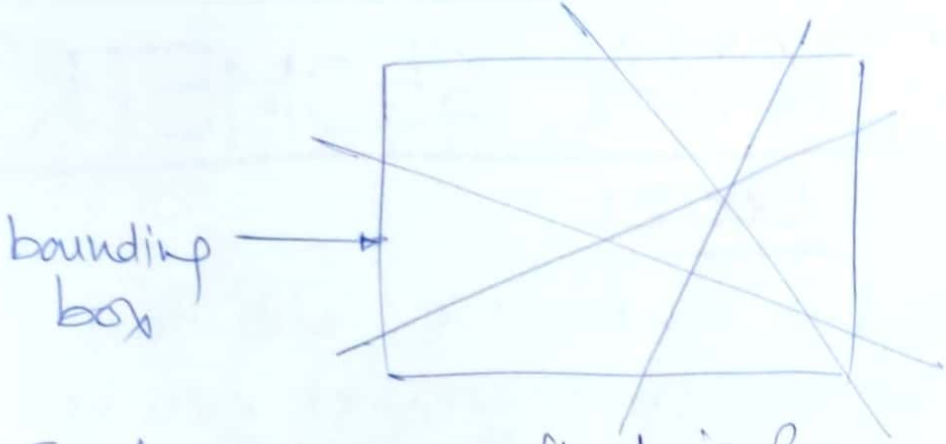
Hence, time complexity becomes $O(n^2 \log n)$.

Of course, this is not the optimal algorithm. This task can be done in $O(n^2)$ time.

We can do this by means of an Incremental algorithm.

INCREMENTAL ALGORITHM

Just before starting the algorithm, we need to find the boundary box where all the intersection point lies (intersection among the lines).



So how can we find it?

For this, we find out all intersection points in $O(n^2)$ & out of them, we find out leftmost, rightmost, topmost & bottommost points. These will be the extreme points. If we take a rectangle that is slightly bigger & contains all 4 points, it will contain all intersection points.

Hence, bounding box can be found in $O(n^2)$ time complexity.

Let us start with the problem again.

We have n lines

$$L = \{l_1, l_2, \cdots, l_n\}$$

These n lines in a plane create an arrangement denoted by $A(L)$.

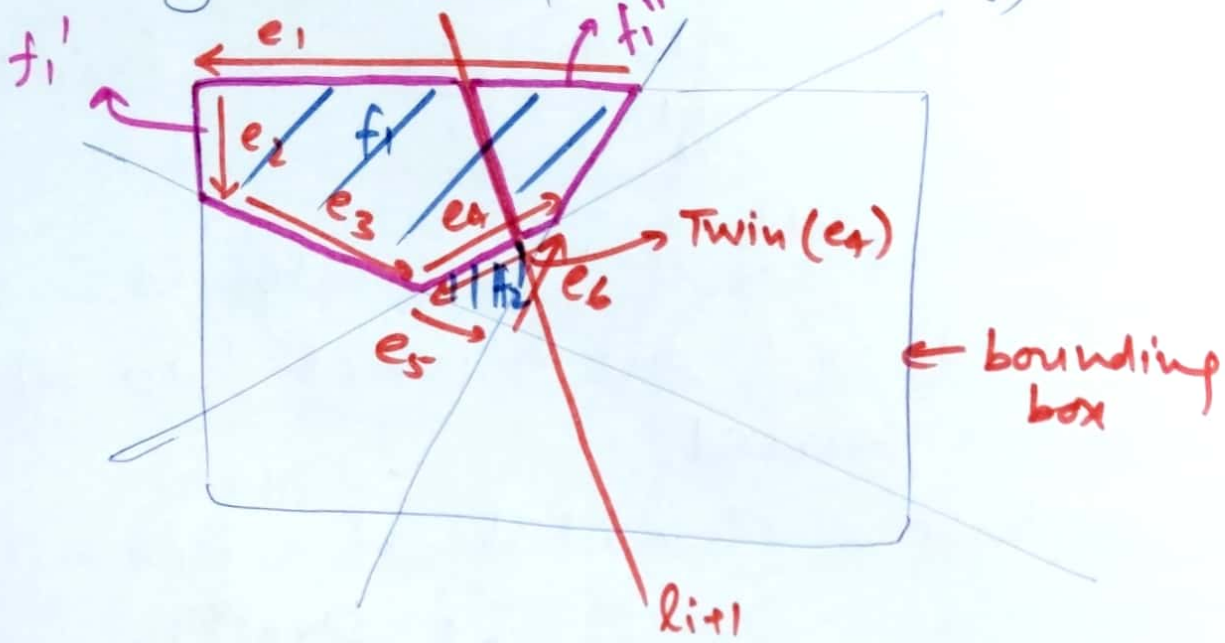
We are computing the subdivisions of the plane created by these lines & we want to store the subdivisions that are within the bounding box in a DCEL.

A(L) subdivision of the plane DCEL

So the main idea of incremental algorithm is that suppose we have an arrangement

$$A(\{l_1, l_2, \dots, l_i\})$$

& we want to insert the next line l_{i+1} then it gives $A(\{l_1, l_2, \dots, l_{i+1}\})$.



So what we do here is we traverse the line from top to bottom and detect the intersection points of this line with the bounding box & other lines.

We start with e_1

we get the first intersection point of line with f_1

Then we go to e_2, then e_3, then e_4.

Here we get the second intersection point of f_1 with line.

We use twin edge of e_4 & traverse over the edges of next face to get the second intersection point of line with f_2.

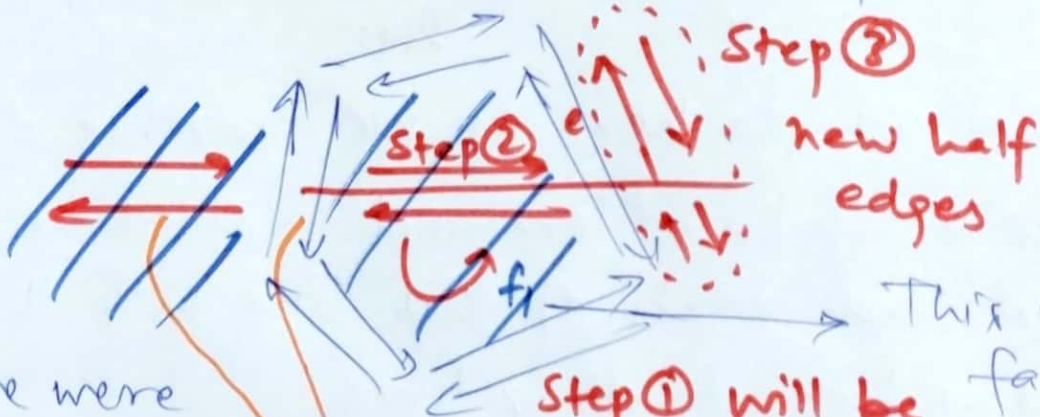
Similarly, we keep on continuing.

Now look at face f_1 to see what updates we need to do

f_1 is divided into two faces f_1' & f_1''

Two new vertices & 1 new edge is introduced in f_1 due to which two separate faces are formed.

Let us have a closer look at a face



we will traverse the edges of f_1 in CCW dir^n.

Edge e will be broken into 2 half edges. Similarly, its twin edge will also be broken into 2 half edges.

① 2 new faces
(or)
Modify existing face
+ 1 new face

② 2 new half edges

③ 2 half edges
update
create 2 new
half edges

Steps are shown in figure above

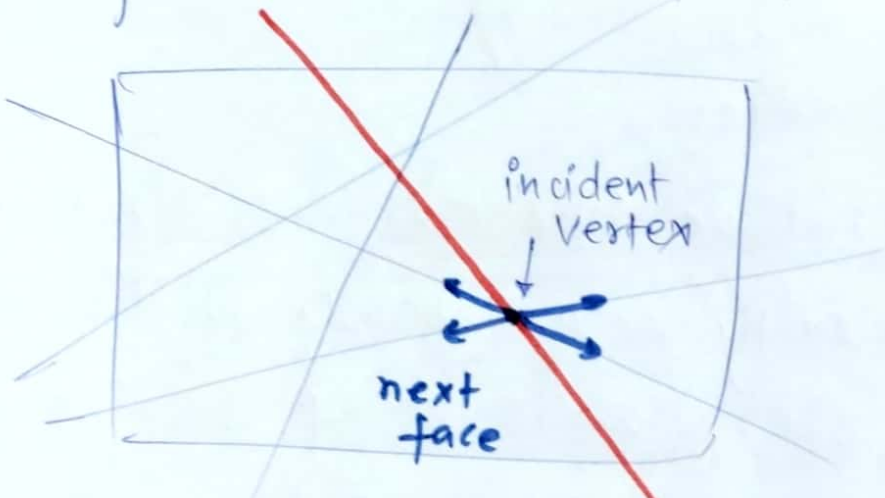
All the operations take constant time only.
Time complexity only depends on the no. of times an operation is being performed.
Hence, time complexity $\propto O$ (size of face)
to update face

(we traverse over all edges of face to find the second edge which intersects with the line)

Of course, while programming this, a lot of details have to be worked out.

At the top level we are discussing about algorithmic level.

There is also a possibility that the new line enters the face through a vertex. In that case, we can find the vertex on which this line is incident. Once we know that the line is leaving through a vertex, we can find out the face in which the line enters by traversing the edges incident on that vertex.



Time required to find out the face in which the line enters $\propto O$(no. of edges incident on vertex through which line leaves)

This happens when arrangement is not simple.

Simple arrangement

⇓

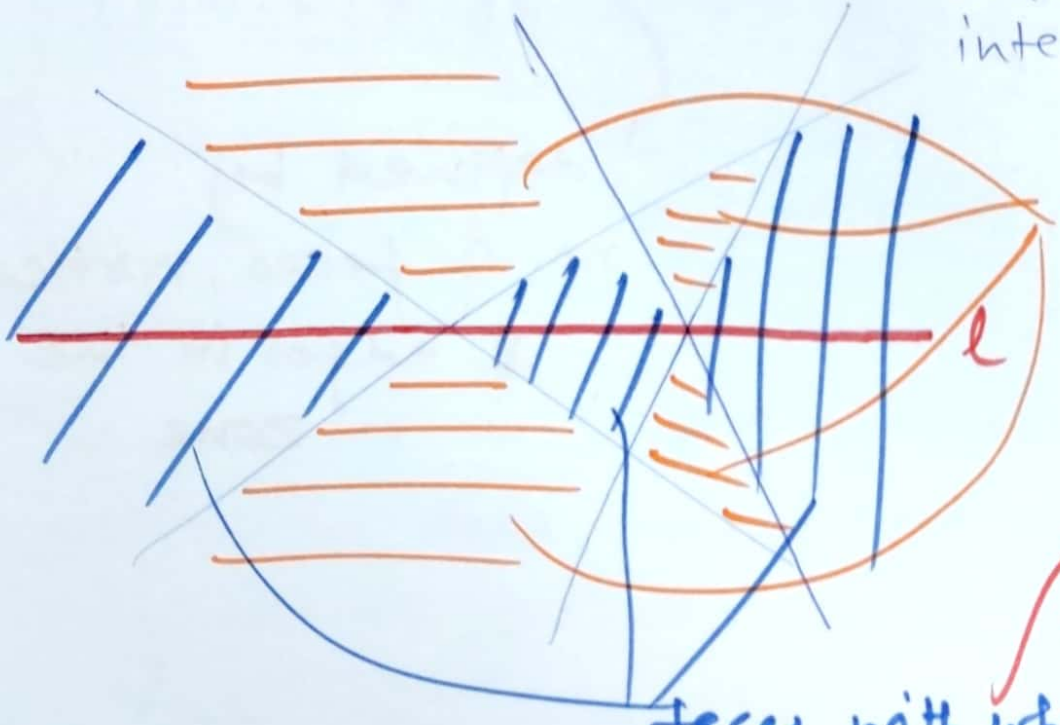
no more than 2 lines pass through same vertex.

Now, all these complexities need to be added for every face to calculate the complexity of adding a new line.

ZONE of a line l in $A(L)$

is set of faces of $A(L)$ whose closure intersects l } closed faces which are intersected by l

not open faces ↑



These are also included in closure

} combination of both is zone

faces with which l intersects

Let's suppose many lines pass through a given point. Since it doesn't pass through its adjacent faces, we do not count them. But in order to find the next face, we have to traverse over all edges incident on this vertex. This factor is not counted. Hence, to count this factor, we include all faces adjacent to this vertex as well.

This is why we take closure into account & not directly the faces.

Time complexity of adding l $\propto$ $\frac{\text{Complexity of zone}}{\text{of } l \text{ in } A(L)}$

defined by no. of faces, vertices & edges in the zone

Zone theorem

The complexity of the zone of a line in an arrangement of m lines is O(m). Here, we are talking about 2D plane only.

Proof:-

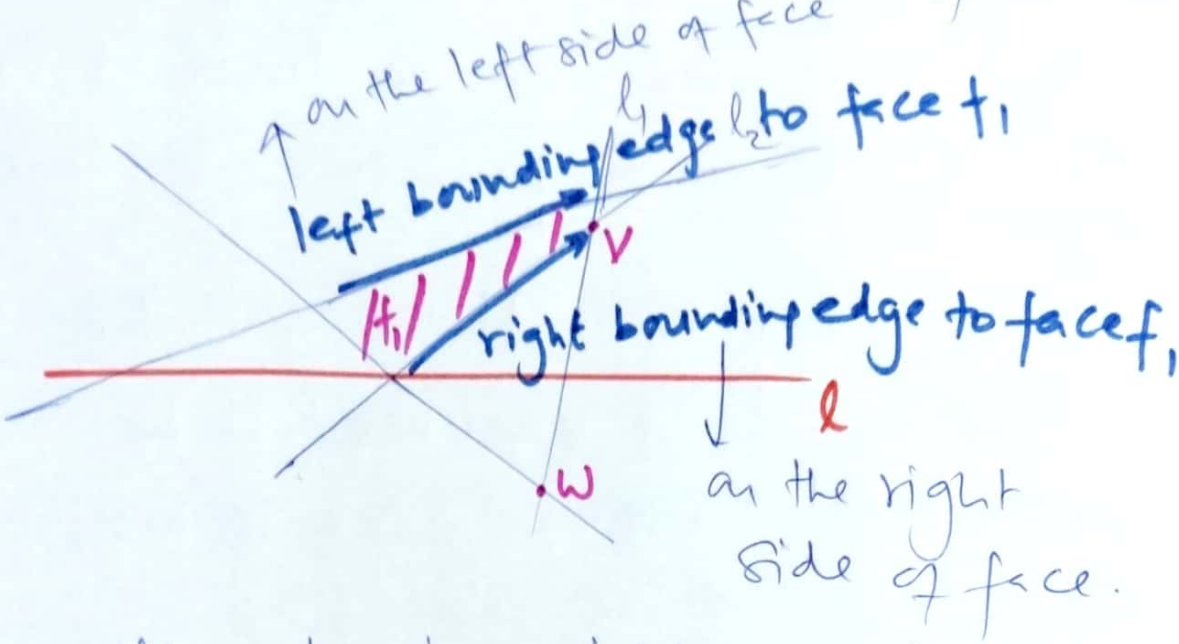
Take m lines in a plane & take a line l. l is assumed to be horizontal (for the sake of simplicity)

If already some horizontal line is present, it will be parallel to line l. Rotate that slightly in clockwise manner. Then intersection will occur b/w the two lines & hence, complexity of zone will increase.

The presence of parallel lines doesn't affect the zone complexity. Hence, they can be ignored.

Similarly, coincident lines also do not affect zone complexity, hence can be ignored.

Hence we can assume that no parallel or coincident lines to l are present.



$l_1 \longrightarrow$ line having rightmost intersection with l out of all lines

If we remove l_1 from L, then

$$|L - l_1| = m - 1$$

Conjecture:

number of left bounding edges in the zone of l $\leq 5m$

number of right bounding edges in the zone of l $\leq 5m$

total no. of bounding edges in the zone of l $\leq 10m$

TIME COMPLEXITY

Recurrence relation

$$T(n) = T(n-1) + O(n-1)$$

$$\Rightarrow \boxed{T(n) = O(n^2)}$$

COMPUTATIONAL GEOMETRY

Lec 22

INTERVAL TREE

All of use Google Maps. When we are using the app, the things around our current position are shown in the window.
The coordinates of the current location are obtained using GPS.
Even if Google Maps has information about entire country or entire district or entire world, it is only displaying a part of it which is relevant to us (which is around your current location).
Even though the database contains information about the entire world, the application only displays the relevant information (region around our current location).
This is called WINDOWING.
Given the current location, a certain window is created using it & the information pertaining to that window is extracted from the database.
This is known as WINDOWING QUERY (2D).
Windowing query can have multiple applications apart from Google Maps.

When you are designing a printed circuit board & we want to expand a particular area, we want to do modification or something else, again we are selecting a window & whatever circuit is ~~displayed~~ there within that window is displayed on the screen.

Similarly, when we use a flight simulator, it only displays the terrain around your location. Even though the flight simulator has information about a much larger terrain, it only displays the information about the terrain around current location.

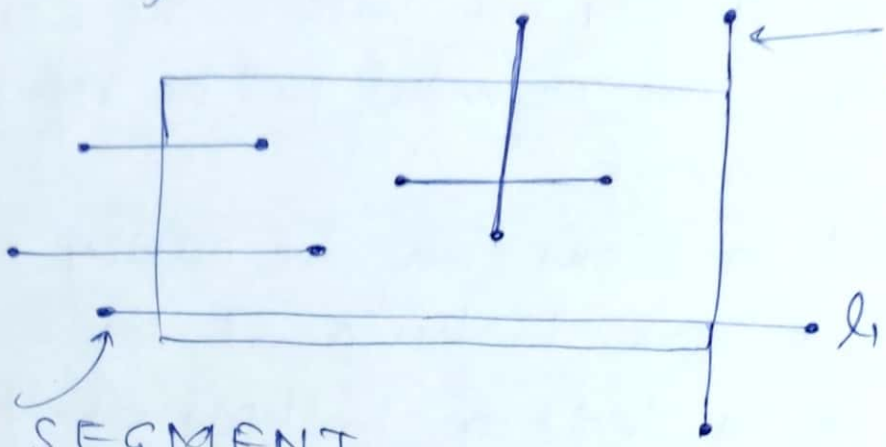
In this, we are querying the volume. This is called VIEWING VOLUME QUERY (3D).

When we look at Google Maps, there are various objects in current window. There are many lines but they aren't necessarily straight. They may have arbitrary dirn.

In order to simplify the representation, we take the orientation of line segments inside current window to be either horizontal or vertical. This is not a very unrealistic assumption because for eg: when we look at a PCB, all the circuit lines are either horizontal or vertical or there may be some lines at 45° angle.

So, fixed orientation isn't an unrealistic assumption. It is a realistic one.

So we have a window & a large number of line segments.



We have to identify all the line segments which are intersected by this window. This is something different than the RANGE SEARCH QUERY. We want to see every object that is present inside the window. This is WINDOWING QUERY.

So how can we solve this problem?

We can identify all the endpoints of the line segment that are within the window using the Range Tree data structure. Once we identify these points, we can report the corresponding line segments.

But this has 2 major problems:-

① There are many line segments whose both endpoints are outside the window but the line segment intersects with the window.

② If both endpoints are within the current window, the line segment will be reported twice which is incorrect.

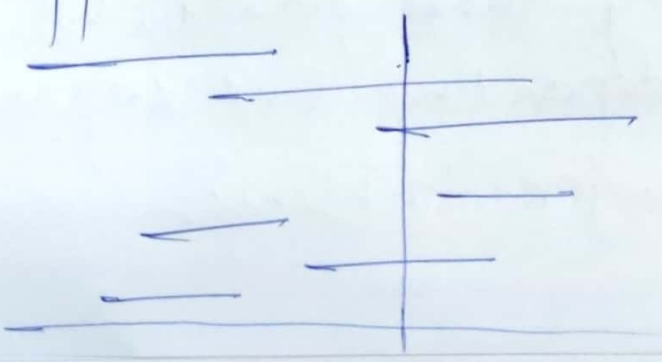
The second problem can be easily addressed. Whenever we are looking at an endpoint, check if it is left or bottom endpoint. Then only we report the line segment if both of its endpoints lie inside the window.

This ensures we report the segment only once.

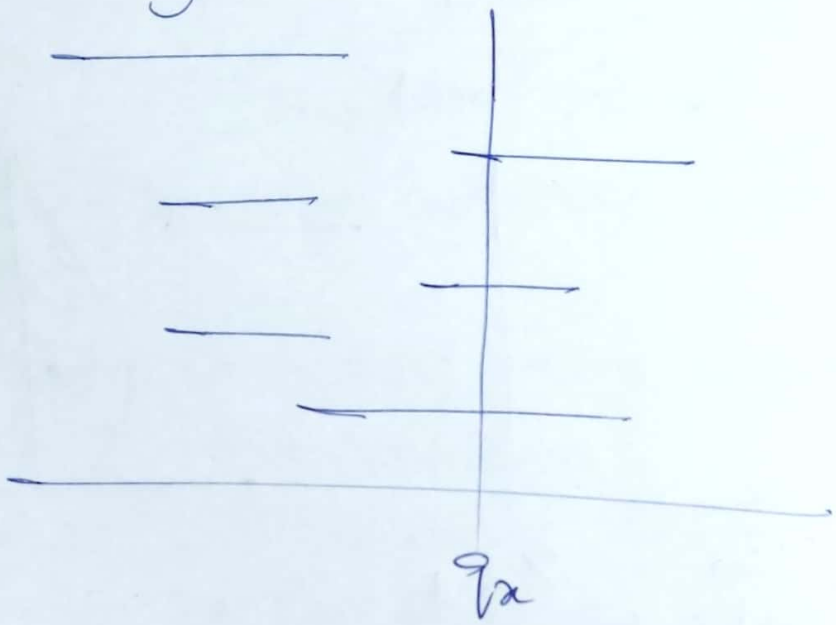
Let us take an example to see how do we tackle with the first problem.

Let us look at line l_1. It is intersected by left edge & right edge both. We will look at only one edge - left edge or right edge.

Let's suppose we look at right edge.



We have several horizontal line segments & a vertical line segment is given to us. We want to report all the line segments that are intersected by this particular segment. A similar procedure can also be done for the vertical line segments involving top edge & bottom edge.
We will simplify this further by replacing line segment by a line & we want to find out all line segments which are intersected by this vertical line.



Query line has x coordinate (vertical line)
Each line segment has x coordinate & y coordinate. y coordinate of the line segments do not play any role in this problem. Hence, it can be thought of as a 1D problem because only x coordinate is important.

So our problem is

Given a set of intervals

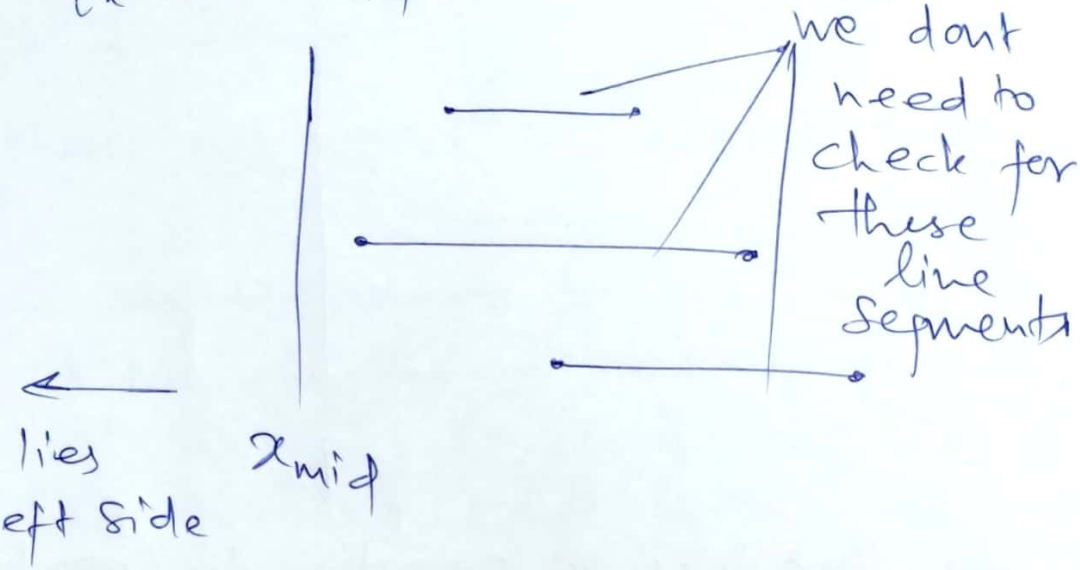
$$I = \{ [x_1 : x_1'], [x_2 : x_2'], \ldots [x_n : x_n'] \}$$

& we want to process these intervals such that given a vertical query line q_x report all intervals intersected by this vertical line.

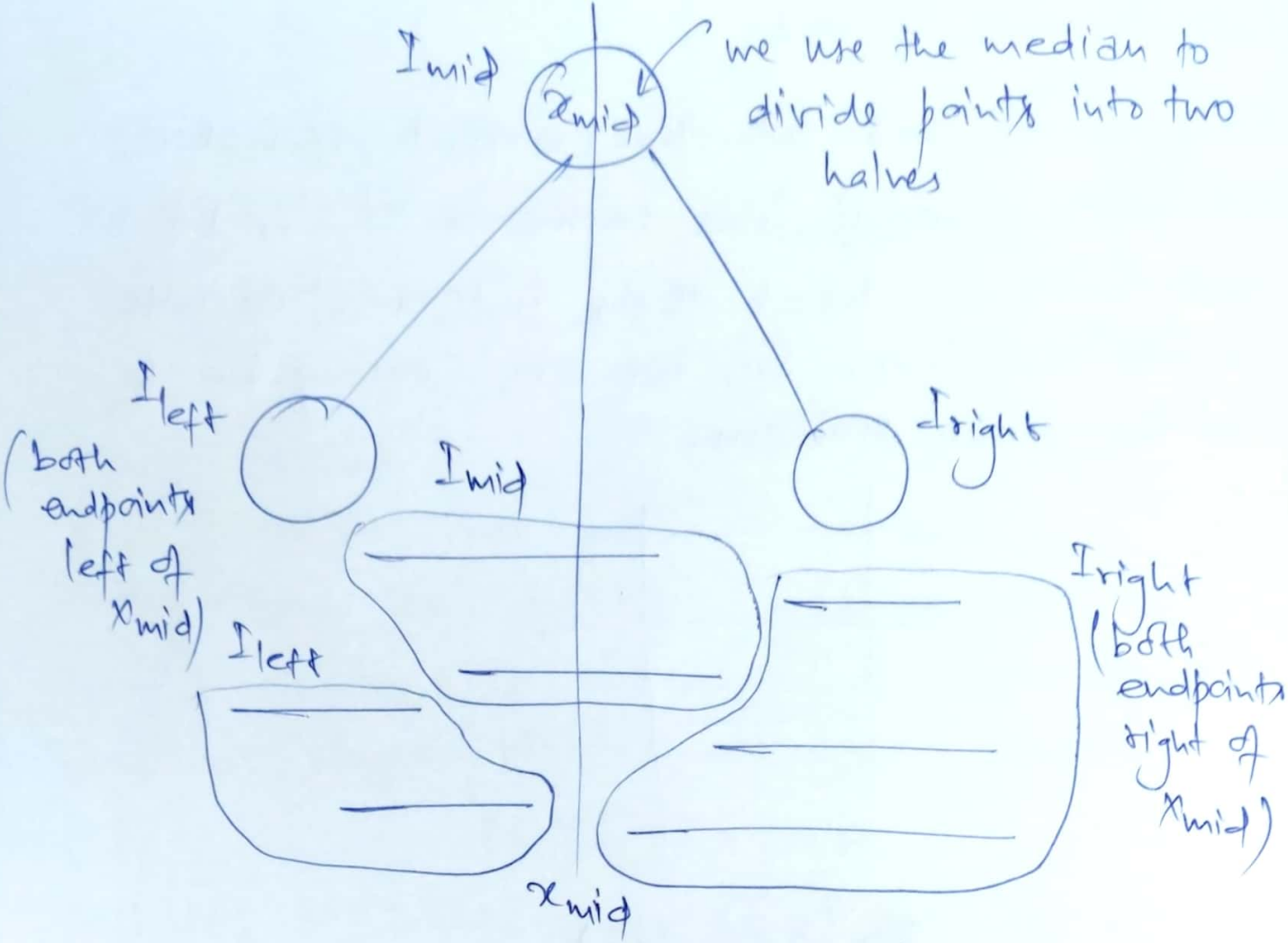
Lets consider the 2n endpoints of these intervals & take the median of these.

x_{mid}

Suppose $q_x \leq x_{mid}$



We need to only check line segment to the left of x_{mid} or intersecting x_{mid}. This leads to a data structure called INTERVAL TREE.



But there may be some intervals intersecting with x_{mid}.

One possibility is to keep these intervals in both left & right subtrees.

But doing so will result in another problem.

This may be repeated at the next level too.

Therefore, same interval may be stored many times.

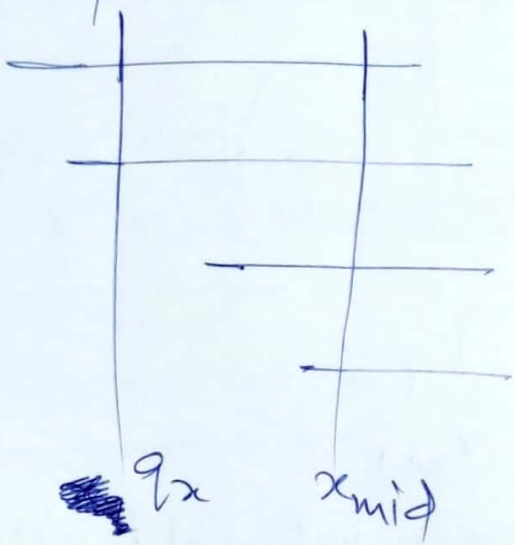
Space complexity will not be efficient.

So, we will store all such intervals in root itself (I_{mid}).

So, if $q_x \leq x_{mid}$, we can ignore the right subtree. We only have to check

left subtree & root.

But, if we look at the current scenario, we have several line segments at I_{mid} & we have to check their intersection with a vertical line. So, we are coming back to the same problem.



But here, we can see that all the line segments in I_{mid} have right endpoints to the right of x_{mid}. Hence, if we can store the left endpoints of these in increasing order, we can report the line segments in complexity $O(\text{size of output})$ by iterating over the list linearly.

Similarly, if query line is on right side, we store the intervals by right endpoint in decreasing order of x coordinate.

Query Time:

To report all line segments intersected by query line.

output size

$Q: O(\log n + k)$

height of Interval Tree

Every time we choose x_{mid} & hence, height of tree is $O(\log n)$ because subtree size is almost halved at every level.

$P: O(n \log n)$.

FINDING MEDIAN CAN BE DONE IN $O(n)$. by using k^{th} smallest element finding method. Then we divide these into I_{left}, I_{mid} & I_{right}. But in I_{mid}, we have to sort the list twice (once by increasing x coordinate of left end, second time by decreasing x coordinate of right end).

We can also start with two sorted arrays initially to avoid this.

But complexity still remains $O(n \log n)$.

I_{mid} preprocessing takes $|I_{mid}| \log(|I_{mid}|)$.

Summing this over all levels, it cannot exceed $O(n \log n)$. (To realize this,

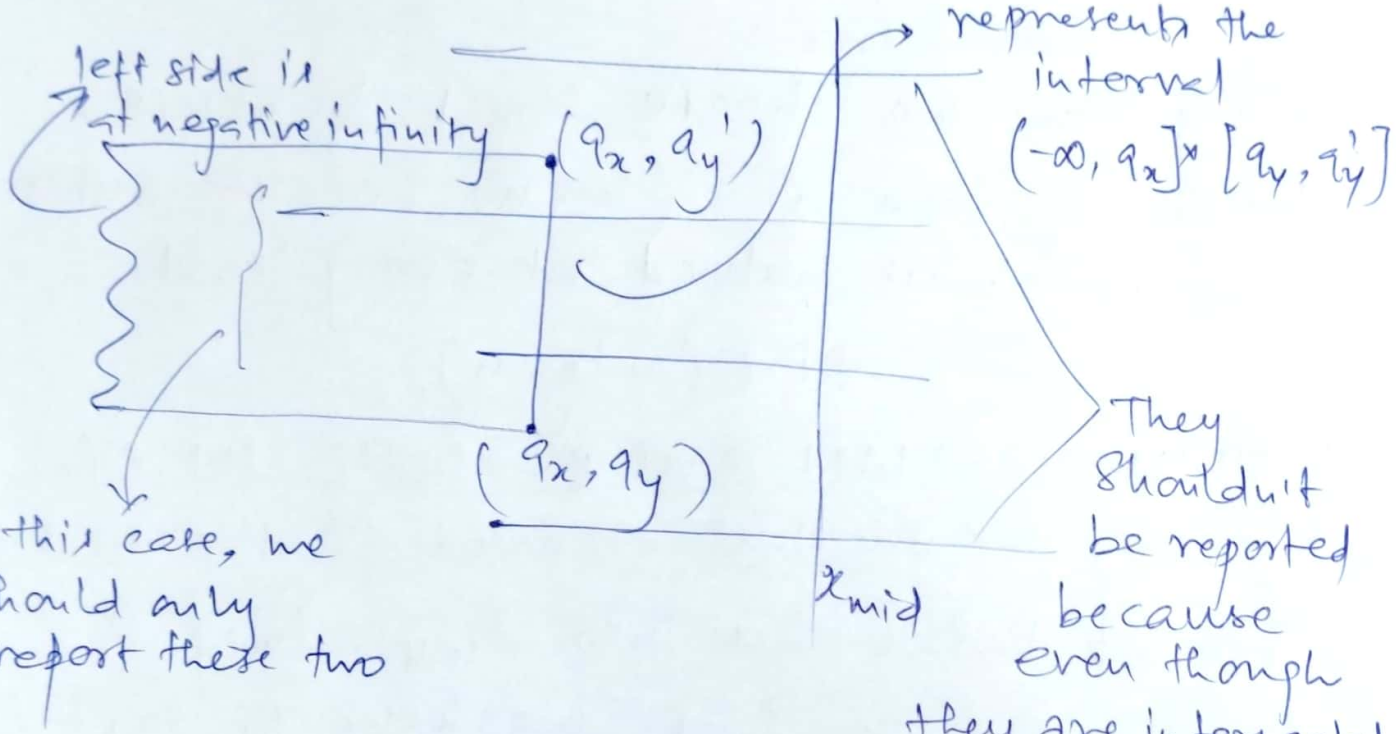
replace $\log(|I_{mid}|)$ by $\log n$).

S: $O(n)$

Every interval is stored only once.

Now we will go back to original query.

In the original query, we had a line segment & not a line.



In the previous case, we had stored a sorted order of these line segments by their left endpoints & right endpoints. Now, instead of this, we will use a Range Tree as the secondary structure to report these type of queries. Here, the primary data structure is the Interval Tree & I_{mid} is stored as Range

Tree to answer queries of the type

$$(-\infty, q_x] \times [q_y, q_{y'}]$$

In the secondary DS, we again have to search for the y-intervals to obtain the answer. This requires another $O(\log n)$ for searching.

Hence,

$$Q: O(\log^2 n + k)$$
$$P: O(n \log n)$$
$$S: O(n \log n)$$

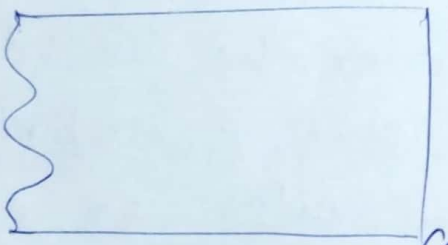
COMPUTATIONAL GEOMETRY

Lec 23

In the last lecture, we were looking at a query of the type

$$(-\infty : q_x] \times [q_y : q_y']$$

This is nothing but an unbounded rectangle (extends to negative infinity on the left)



We used Interval Tree as primary DS & Range Tree as secondary DS.

Today, we will look at another DS for the same problem. It is called PRIORITY SEARCH TREE.

PRIORITY SEARCH TREE

Before this, let us look at a 1D query

$$(-\infty, q_x]$$

We can use a sorted list of points or a height balanced BST to answer queries of the form $[q_x : q_x']$ in $O(\log n + k)$ time. We have already done that earlier.

If the query is of the form

$(-\infty : q_x]$

then what can we do?

Can we do better?

How should we preprocess the points so that we can answer the query efficiently?

Suppose that we have a sorted list of points, then can we answer the query in $O(1+k)$ TIME COMPLEXITY?

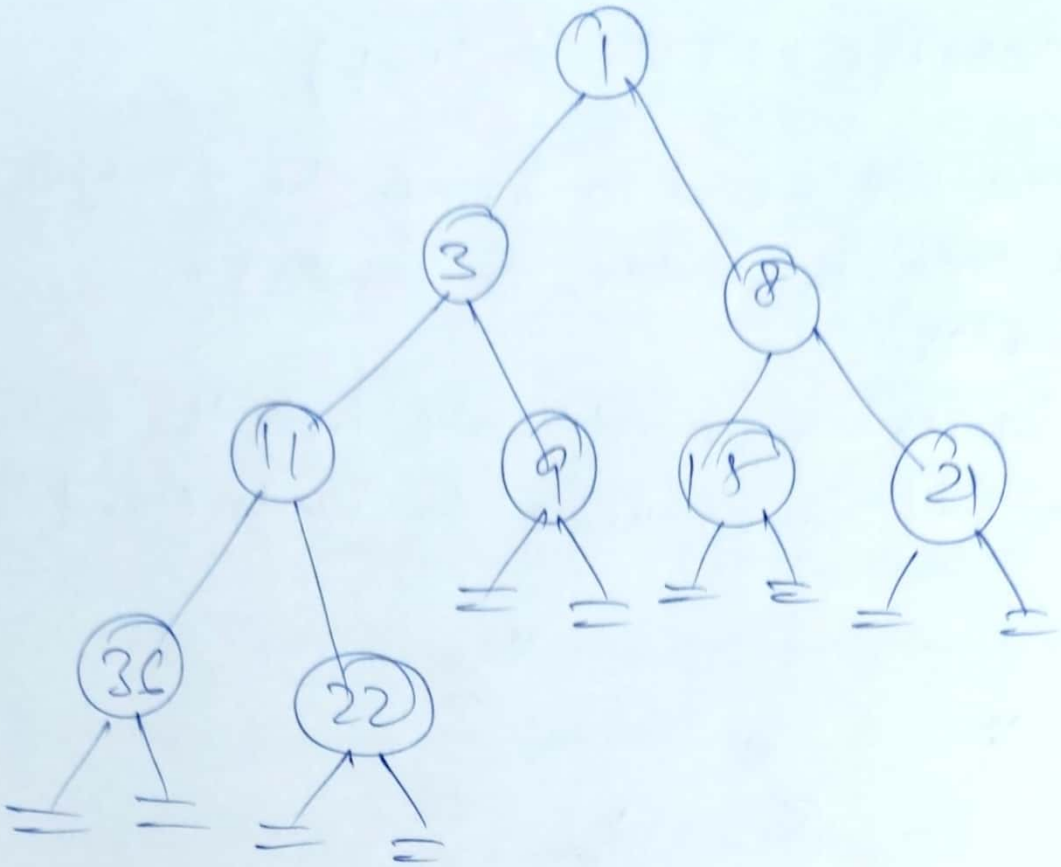
Yes, we can iterate over the sorted array starting from the left end & report every element until we encounter an element greater than q_x.

Hence, TIME COMPLEXITY is $\underbrace{O(1+k)}_{\text{better than } O(\log n + k)}$

We need some technique by which we can extend this idea into 2 dimensions.

Suppose we have a priority queue. Here, we use a min-heap for the priority queue.

How can we answer this query using a min-heap?



Let's take the query to be $(-\infty, 5]$.

We want to report all elements in this range in the min heap.

We look at root.

$1 \in (-\infty, 5]$. Hence,

output this and

check left & right subtree.

Left subtree of 1 contains 3.

output 3.

Check left & right subtree of 3,

left subtree of 3 contains 11 which doesn't lie in interval. Don't descent the subtree.

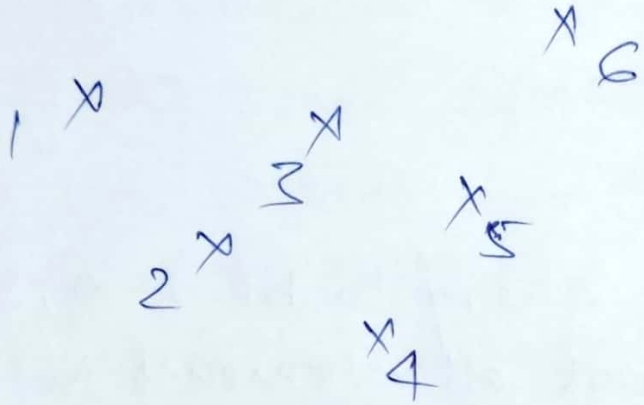
Right subtree of 3 contains 4. output 4.

Right subtree of 1 contains 8 which doesn't lie in interval. Don't descent the subtree

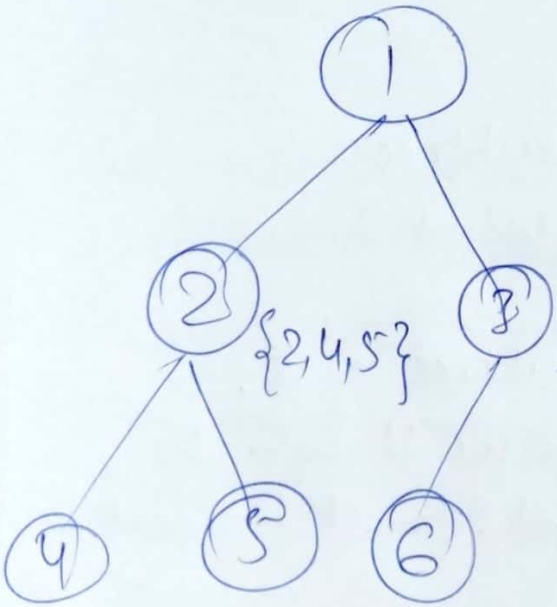
TIME COMPLEXITY = O(1+k)

Now, somehow we need to bring the y ranges also in the tree in order to answer 2D queries.

So, how to bring y coordinate into the picture? Let us look at an example to understand this.



Use x coordinate to build min heap. Now, which nodes should we keep in left subtree?



That is done on the basis of y coordinate.

Take remaining points.

Compute their median y coordinate & then use it to divide these points into 2 halves.

The value in node is chosen using min. x coordinate.

y coordinates are only used to divide remaining

points into subtrees.

This DS is called PRIORITY SEARCH TREE.

In the normal heap, once minimum is at the root, remaining elements can be present either in left or right subtrees. There is no restriction as such. We can put any point anywhere. But in 2D Priority Search Tree, we divide points into 2 halves using y coordinate.

Hence, this is combination of Priority Queue & Binary Search Tree. Hence the name Priority Search Tree.

It is a height balanced tree.

TIME COMPLEXITY FOR BUILDING THE TREE

P: $O(n \log n)$

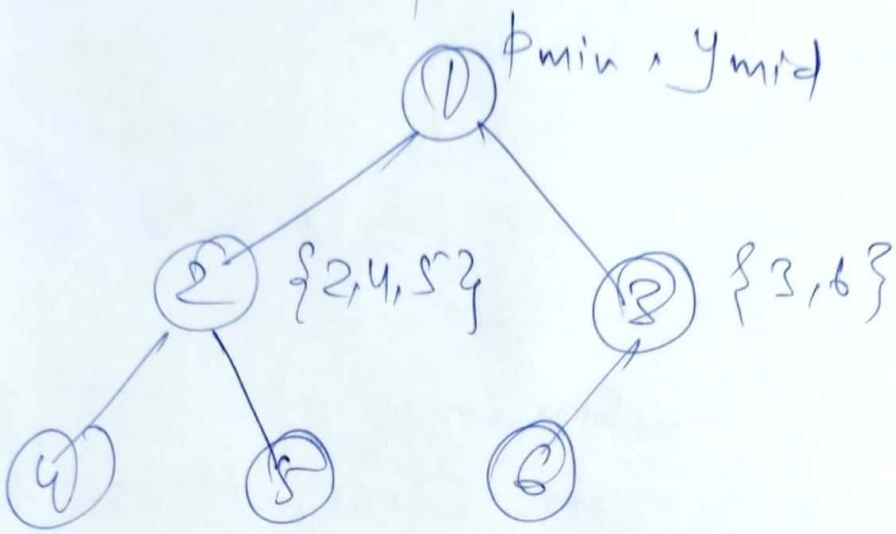
S: $O(n)$

Q: ? to answer query of type

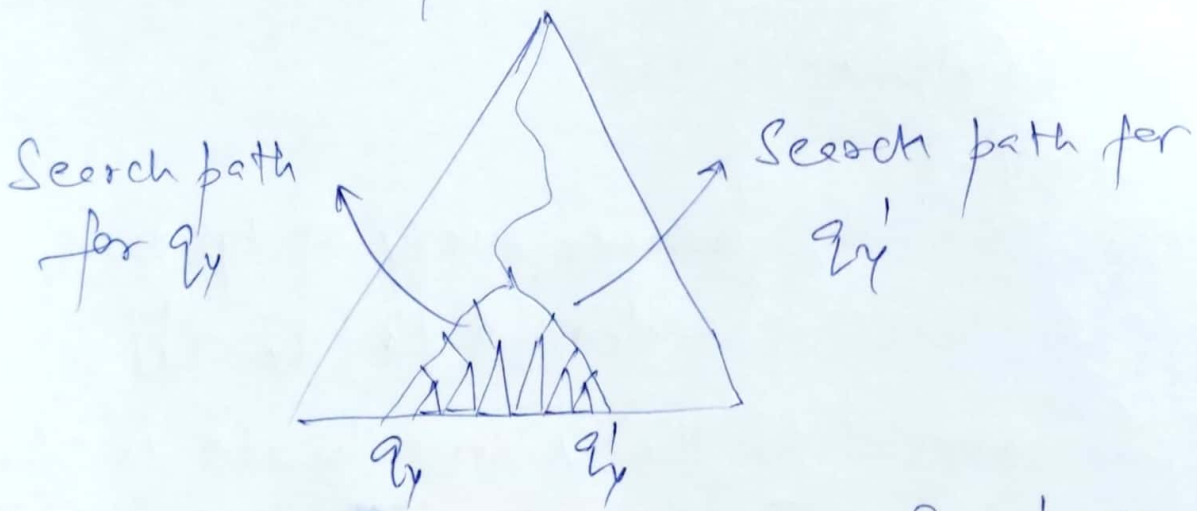
$(-\infty, q_x] \times [q_y, q'_y]$

We will search for the y coordinate in the Priority Search Tree. When we look at the y coordinate, we do not bother about the x coordinate. We just treat it as a

Simple BST & we will do the processing.
Node will store point with min. x coordinate & median of y coordinate (y_{mid}) used to divide remaining points into subtrees.



We can use these median values to search. It will be similar to searching in 1D Range Tree based on y coordinates.



We use y_{mid} to search for q_y & q_y'. That will take logarithmic time.
On the left search path, right subtrees will be hanging.
On the right search path, left subtrees will be hanging.

All the points that are there in the range $[q_y : q_{y'}]$ are candidates for output.

Among them, we only want points within the x range $(-\infty, q_x]$. Treat each subtree within the range as a priority queue or min heap & output the values in the range $(-\infty, q_x]$.

TIME COMPLEXITY

Searching for q_y & $q_{y'}$ takes $O(\log n)$ time.
Outputting the elements in the range takes $O(k)$ time.

Hence, time complexity is $O(\log n + k)$

Q: $O(\log n + k)$

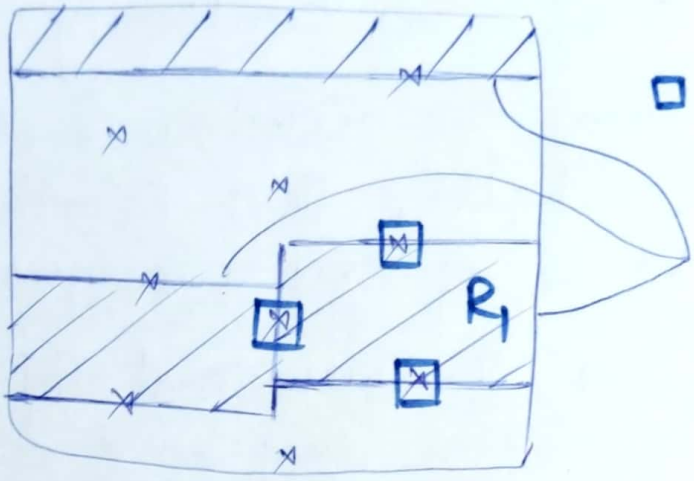
Now, we look at preprocessing time.
The preprocessing time of this algorithm is $O(n \log n)$.
Suppose the points are given in sorted order based on y coordinates.
Is it possible to construct the Priority Search Tree with preprocessing time lower than $O(n \log n)$?

Yes, in this case the Priority Search Tree can be built in $O(n)$ time complexity. We do it in a bottom up fashion.

COMPUTATIONAL GEOMETRY

Lec 24

Let us take a rectangular cloth in which there are point sized holes.



We want to cut out largest area rectangle without any hole. We will restrict the orientation of the rectangle to an axis parallel rectangle.

We might also ask what is the number of such rectangles.

Here, the problem is that we only have points in 2-dimensional plane & we are not able to see any structure among these points (so we are not able to do any search on this one.

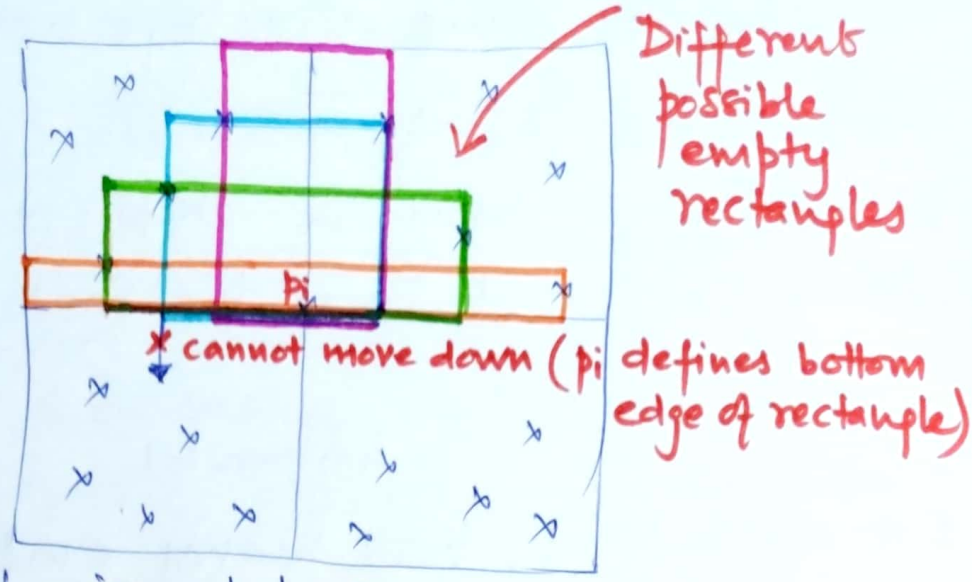
First of all, we should be able to enumerate all possible rectangles in some manner (some systematic order) & try to find out the largest area rectangle out of them.

This is the Brute Force method.
One possible brute force method is to choose 4 points & make a rectangle from them & then check if it is empty. If it is empty, find the area & keep track of the largest one. This is an $O(n^5)$ time complexity algorithm.

We do not know the number of empty rectangles & whether this $O(n^4)$ is a tight bound on the no. of empty rectangles or not. Since points are 0-dimensional objects, it is very hard to obtain any kind of structural property on the point of this problem.

Let us repose this problem slightly differently. If we look at any empty rectangle that cannot be extended further, each edge of these empty rectangles are passing through some vertex or coincide with an edge of the original rectangle.
Therefore, each edge is defined by a point or edge of the original rectangle.

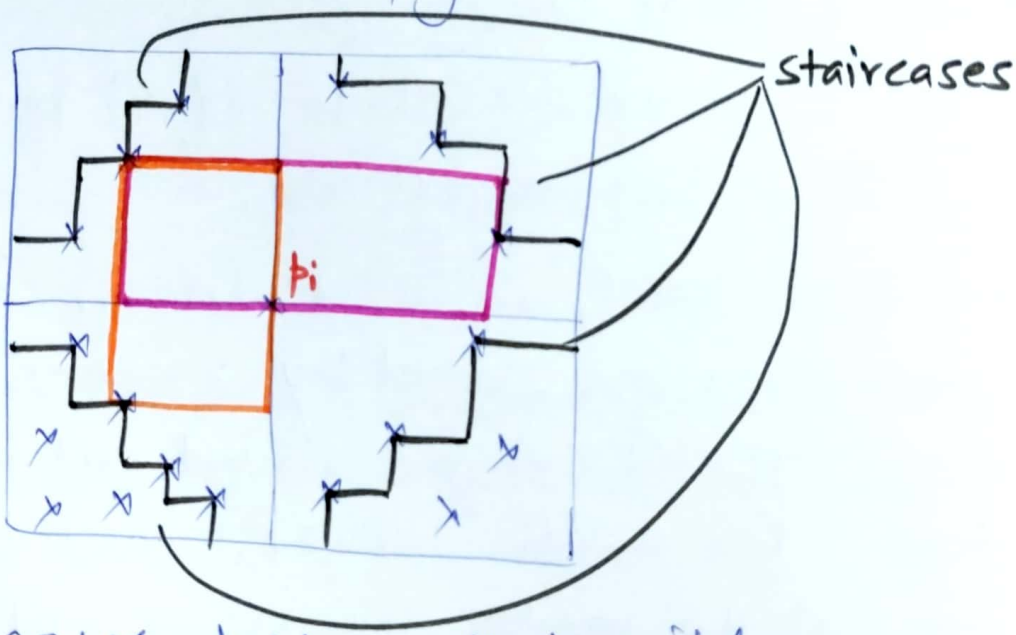
So we simplify this problem. We choose a point arbitrarily from the set of n points. We want to find out all the empty rectangles such that one edge is defined by this point



We draw a horizontal & vertical line passing through this point

Now, is it possible to find all empty rectangles such that one edge is defined by p_i

Let's suppose that edge is a horizontal edge. Some of the various possible rectangles are shown in figure.



These staircases define all possible empty rectangles with an edge defined by p_i

We can see some structure here.
Each corner in the staircase can be extended to form an empty rectangle.
Each corner defines two rectangles

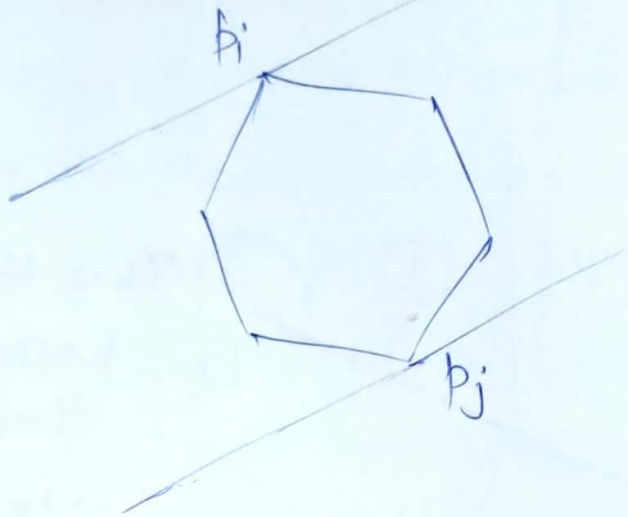
horizontal vertical

For each point, how many empty rectangles are possible?
Each corner defines two rectangles.
Hence, a maximum of $O(n)$ rectangles are possible for each point.
Hence, total no. of empty rectangles is $O(n^2)$.
Next question is
How to enumerate these $O(n^2)$ rectangles & find the largest one.
For each point, we find out the 4 staircases.
Once points are sorted by x-coordinate, the staircases can be found out in linear time. For each corner, find out the area of the rectangles associated with it & maintain the maximum area.
All of this can be done in a total of $O(n^2)$ time complexity.

Here, we are trying to enumerate all possible empty rectangles systematically & finding the largest one (area wise).

Here, we are considering only axis parallel rectangles. If we consider all possible orientations, then many rectangles are possible. Of course, their number is also bounded (by $O(n^3)$).

If we look at this convex polygon



We can draw two parallel tangents from p_i & p_j. This pair is called "antipodal pair".

Pairs of vertices of a convex polygon through which we can draw a parallel tangent.

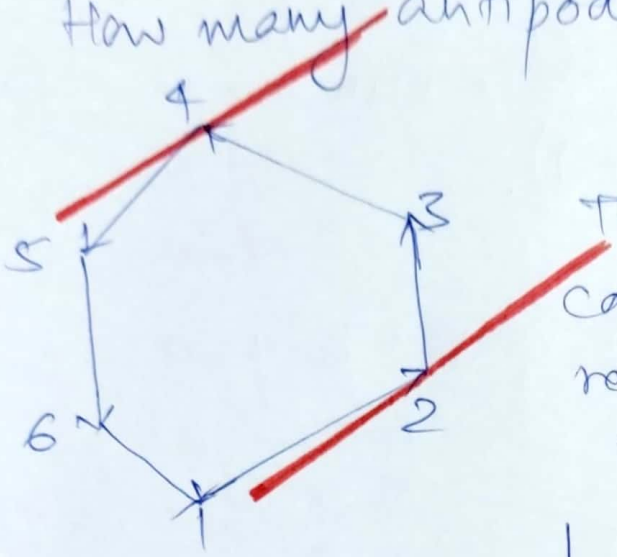
⇓⇓

ANTIPODAL PAIRS

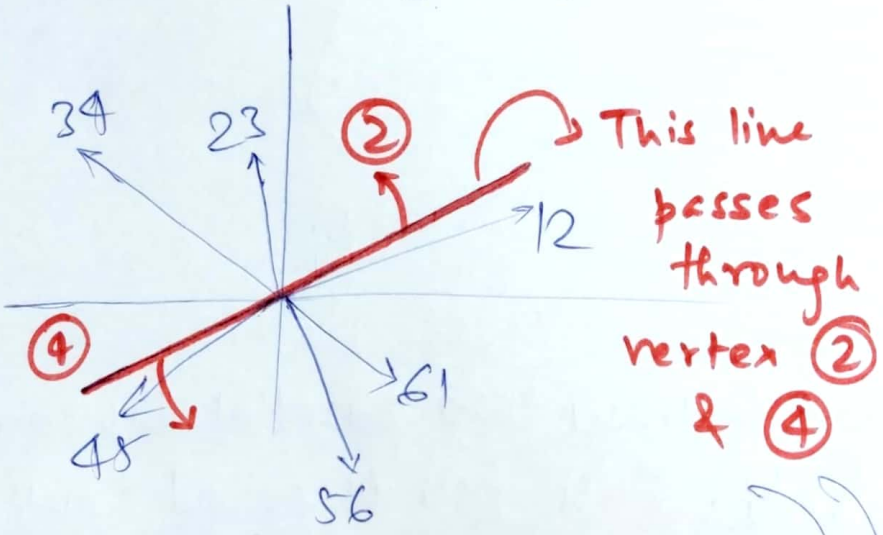
There are many antipodal pairs for a convex polygon. One vertex can be antipodal with many other vertices.

The question is

How many antipodal pairs is possible?



The edges of a convex polygon can be represented using a ray diagram



Now you take some line passing through the origin in the ray diagram.

we can draw a tangent parallel to this line through vertex ② & vertex ④

With each possible line passing through the origin, there is an associated antipodal pair. Now you rotate the line CCW. Whenever the line crosses a ray, we obtain another antipodal pair.

If we properly enumerate, we get a number which is $O(n)$.

We get more antipodal pairs when we consider regular polygons.

Can we find out a tighter bound than $O(n)$?

What is the maximum no. of antipodal pairs of any convex polygon of n vertices?

We place the vertices in such a way that antipodal pairs are maximized.

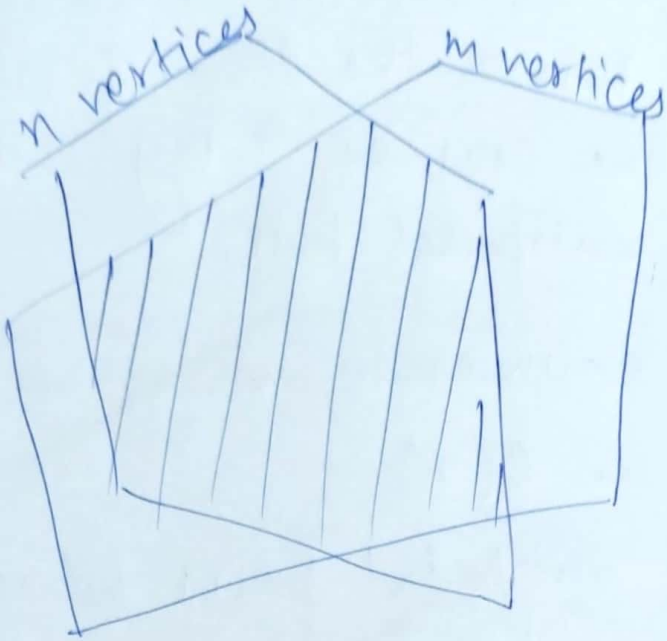
Let us look at another question.

Suppose we are given n-circles in a plane.

Our aim is to find all empty circles (m points are given. Circle is empty if it contains no point amongst these m points).

This can be obviously done in $O(nm)$ time complexity. But can we do better than this?

Suppose we have 2 convex polygons.



We want to find out their intersection.
How can we do so?

This problem is much easier.
Using a sweep line algorithm, this can be done in $O(n+m)$ time complexity.

Note :- No. of intersections b/w two convex polygons can be large ($\leq n+m$).