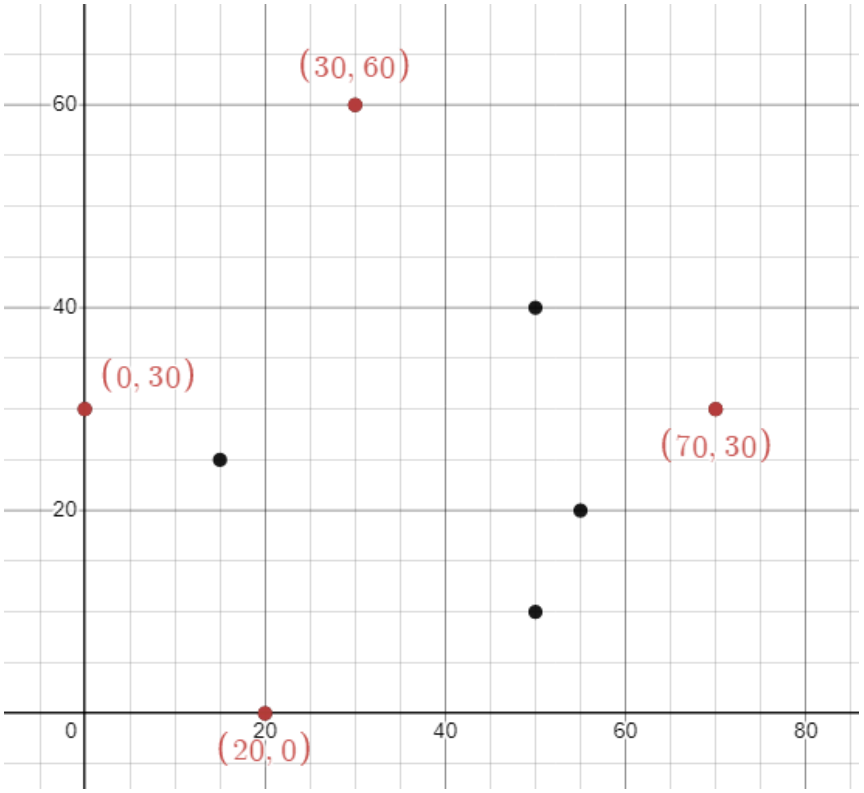
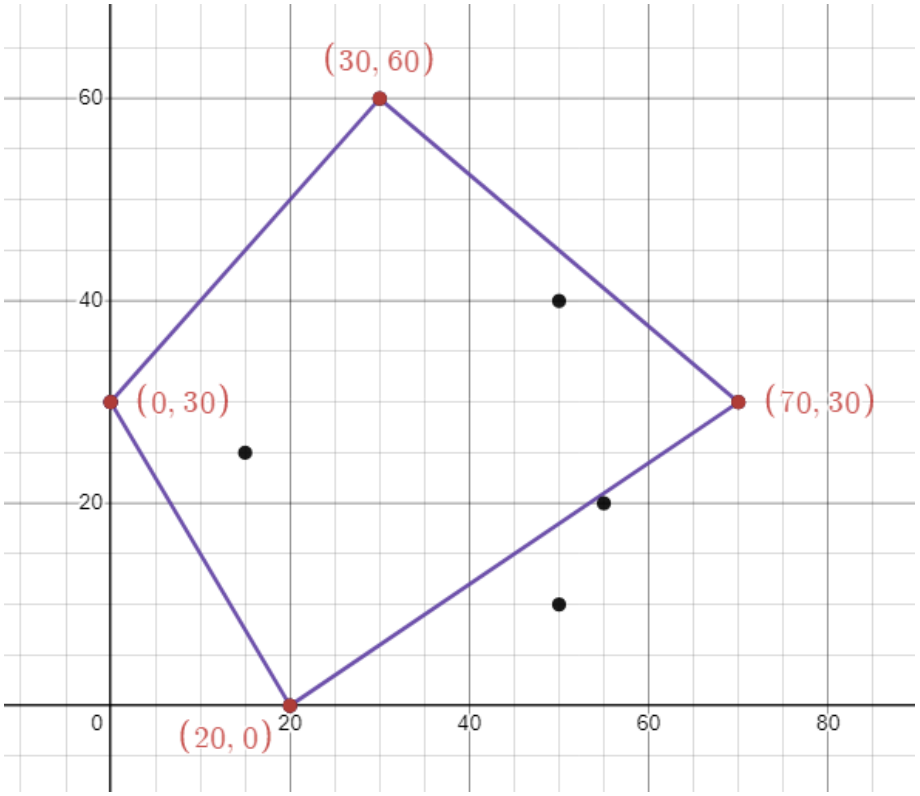
Q1)

a)

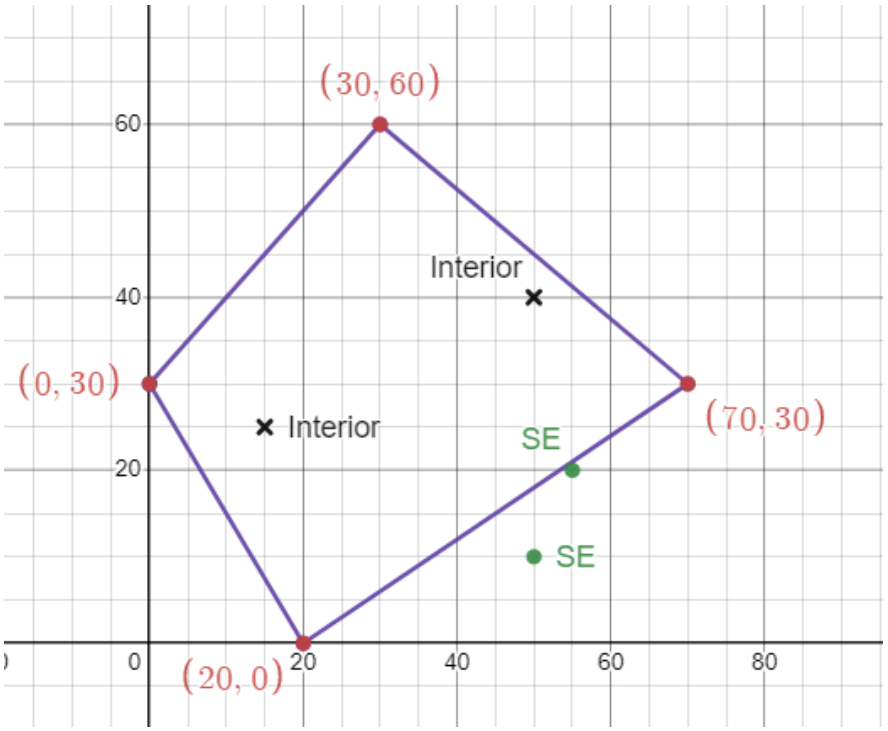
1. Find Extreme points in horizontal and vertical directions



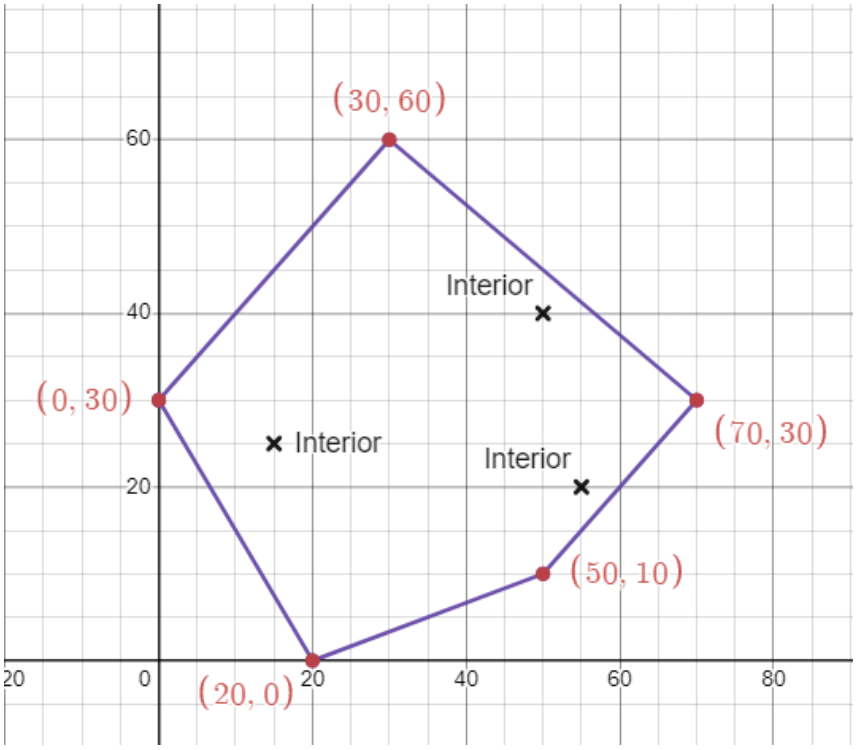
2. Compute the convex hull of these points



3. Test remaining points and classify them



4. Connect the endpoints of each edge with the extreme points from each region and update the convex hull



Final Convex Hull:

$$(0, 30)\ ,\ (30, 60)\ ,\ (70, 30)\ ,\ (50, 10)\ ,\ (20,\ 0)$$

b)

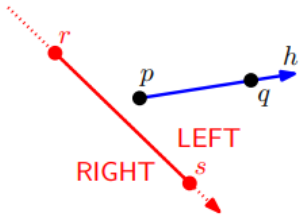
Since half the points got eliminated in each prune step then the runtime =

$$\theta(n\ log\ n)$$

Q2)

a)

1. T
2. F (مش شرط)



3. T (Lec 3-5, slide 7)

B)

1. $O(n^2)$
2. $\leq 2n - 5$ (lec Vornoi, slide 20)
3. $\leq 3n - 6$ (lec Vornoi, slide 20)

Q3)

a)

Algorithm FINDINTERSECTIONS(S)
Input. A set S of line segments in the plane.
Output. The set of intersection points among the segments in S.
1. **for** each pair of line segments $e_i, e_j \in S$
2. **do if** e_i and e_j intersect
3. **then** report their intersection point

b)

The issue is that we need the algorithm to be **output sensitive** too, that is the complexity is affected by the number of intersections in the data too while the brute force algorithm is **only affected by the input** .

C)

No, Since the number of intersections k is close to the number of segments n so the brute force algorithm becomes more efficient

Q4)

a)

Algorithm FINDINTERSECTIONS(S)
Input. A set S of line segments in the plane.
Output. The intersection points of the segments in S, with for each intersection point the segments that contain it.

1. Initialize an empty event queue Q. Insert the segment endpoints into Q; when an upper endpoint is inserted, the corresponding segment should be stored with it
2. Initialize an empty status structure T
3. **while** Q is not empty
4. **do** Determine next event point p in Q and delete it
5. HANDLEEVENTPOINT(p)

(lec Line Segment Intersection, slide 41)

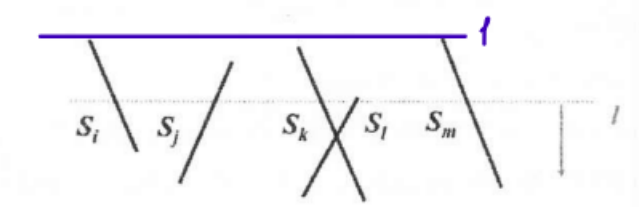
$HandleEventPoint(p)$:

If the event is an upper endpoint event, and s is the line segment that starts at p: 1. Search with p in T, and insert s 2. If s intersects its left neighbor in T, then determine the intersection point and insert in Q 3. If s intersects its right neighbor in T, then determine the intersection point and insert in Q	If the event is a lower endpoint event, and s is the line segment that ends at p: 1. Search with p in T, and delete s 2. Let s_l and s_r be the left and right neighbors of s in T (before deletion). If they intersect *below the sweep line*, then insert their intersection point as an event in Q	If the event is an intersection point event where s and s' intersect at p: 1. Exchange s and s' in T 2. If s' and its new left neighbor in T intersect below the sweep line, then insert this intersection point in Q 3. If s and its new right neighbor in T intersect below the sweep line, then insert this intersection point in Q 4. Report the intersection point

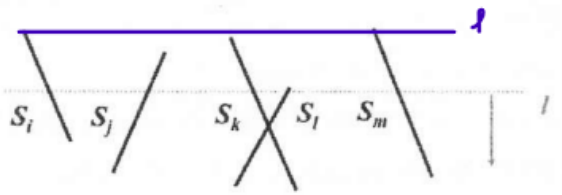
b)

Notes:

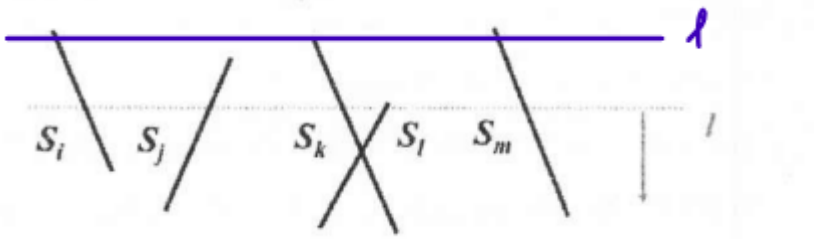
- Current Order = status = T
- Event queue: ترتيب النقط اللي هنقابلهم بس احنا مش كاتبينه بما انه ظاهر فالرسمة



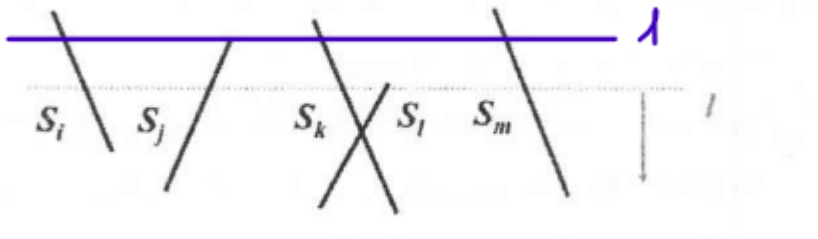
- Insert S_m to event queue , Current order : S_m



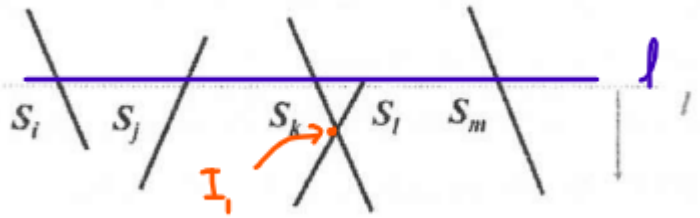
- Insert S_i to event queue, Current order; S_i , S_m <- ترتيبهم من الشمال لليمين



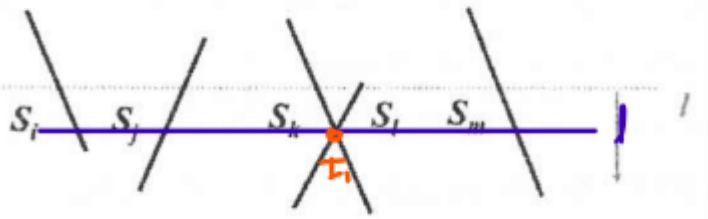
- Insert S_k to event queue, Current order: S_i , S_k , S_m
  - Check if S_k intersects with S_i or S_m -> no intersection



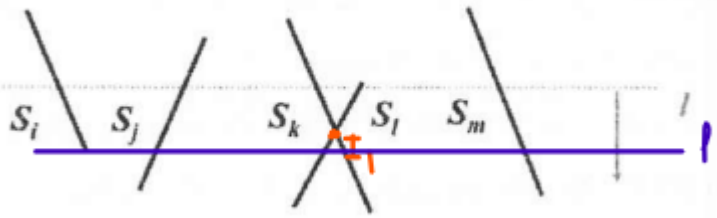
- Insert S_j to event queue, Current order: S_i, S_j, S_k, S_m
  - Check if S_j intersects S_i or S_j -> no intersection



- Insert S_l to event queue, Current order : S_i , S_j , S_k , S_l , S_m
    - Check if S_l intersects S_k or S_m -> Intersects S_k at I_1
    - Add I_1 to event queue

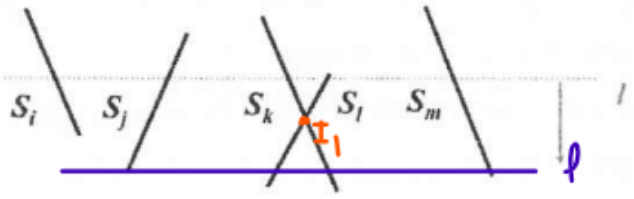


- **Intersection Event:** Swap S_k and S_l in T, Current Order: S_i , S_j , S_l , S_k , S_m
    - Check if S_k and S_l intersect with their new neighbours:
        - S_l and S_j -> no intersection
        - S_k and S_m -> no intersection
    - Save intersection point I_1

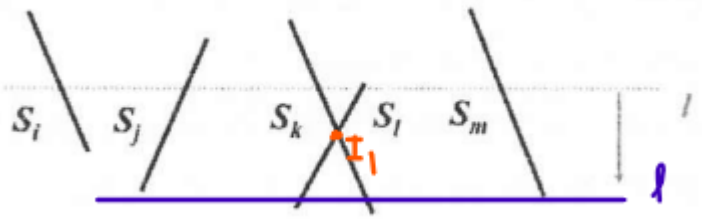


- Remove S_i from T , Current order : S_j , S_l , S_k , S_m
    - Check if new neighbours intersect -> no new neighbours

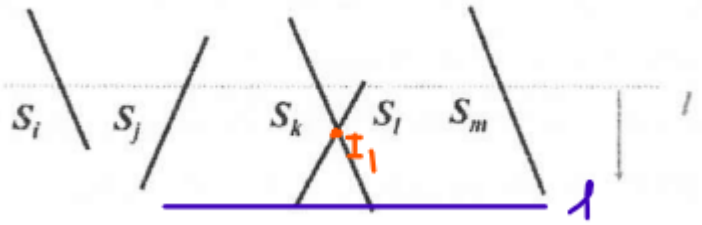
(كانت هتفرق لو كنا شيلنا واحد من الخطوط اللي فالنص عشان كان ترتيبهم هيفرق
فا هنحتاج نشوف لو الخطوط اللي بقت جمب بعضها جديد بتقطع بعض قدام)



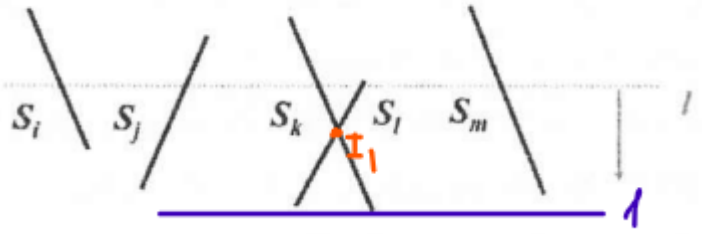
- Remove S_j from T , Current order : S_l , S_k , S_m
    - Check if new neighbours intersect -> no new neighbours



- Remove S_m from T , Current order : S_l , S_k
    - Check if new neighbours intersect -> no new neighbours



- Remove S_l from T , Current order : S_k
    - Check if new neighbours intersect -> no new neighbours

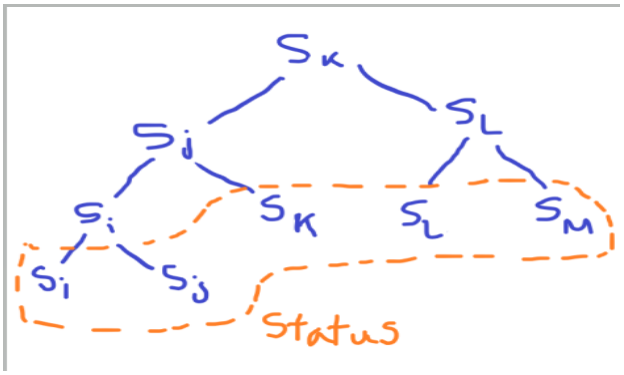


- Remove S_k from T , Current order :
- Terminate Algorithm

Found Intersection points : I_1

c)

Data structure : balanced binary search tree,
The status of the **line in figure** after handling event:



Q5)

a)

1. F , Necessary (lec Voronoi, slide11)
2. T (lec Voronoi, slide 15)

b)

1.

we get, for $n \geq 3$

$$|v| \leq 2n-5$$
$$|e| \leq 3n-6$$

2.

Beach Line properties

- Voronoi edges are traced by the break points as the sweep line moves down.
  - Emergence of a new break point(s) (from formation of a new arc or a fusion of two existing break points) identifies a new edge
- Voronoi vertices are identified when two break points meet (fuse).
  - Decimation of an old arc identifies new vertex

(lec Voronoi, slide 32)

3.

Data Structures

- Current state of the Voronoi diagram
  - Doubly linked list of half-edge, vertex, cell records
- Current state of the beach line
  - Keep track of break points
  - Keep track of arcs currently on beach line
- Current state of the sweep line
  - Priority event queue sorted on decreasing y-coordinate

(lec Voronoi, slide 33)