CS 301 – Algorithms

Computational Geometry

Hüsnü Yenigün

Algorithms for “geometric problems”

- Inputs: geometric object (points, lines, polygons)
- Problems :
  - Queries about the input objects
    - do two given line segments intersect?
    - given a point, does it fall into a given polygon?
    - etc...
  - Production of new objects
    - construct a convex hull from a given set of points

We will study 2 dimensional problems only, but in general 3D or n dimensional geometric problems can be solved computationally.

Representing Geometric Objects

- A *point* in 2D space: $p = (x, y)$ where $x, y \in \mathbb{R}$
- Given 3 points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ and $p_3 = (x_3, y_3)$, p_3 is a convex combination of p_1 and p_2 iff $\exists 0 \leq \alpha \leq 1$ such that:

  (i) $x_3 = \alpha x_1 + (1 - \alpha) x_2$

  (ii) $y_3 = \alpha y_1 + (1 - \alpha) y_2$
- If p_3 is a convex combination of p_1 and p_2, we also write $p_3 = \alpha p_1 + (1 - \alpha) p_2$
- Intuitively, p_3 is convex combination of p_1 and p_2 iff p_3 in on the same line as and between p_1 and p_2.

Lines and Line Segments

- Given two points p_1 and p_2, the *line segment* $\overline{p_1p_2}$ is set of convex combinations of p_1 and p_2.
- p_1 and p_2 are said to be the *end points* of the line segment $\overline{p_1p_2}$.
- If the ordering is important, then we use the notation $\overrightarrow{p_1p_2}$ to denote the *directed line segment* from p_1 to p_2.
- If $p_1 = (0, 0)$ *–the origin–*, then we treat the directed segment $\overrightarrow{p_1p_2}$ as the *vector* $\overrightarrow{p_2}$

Some Geometric Problems

(I) Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, is $\overrightarrow{p_0p_1}$ clockwise from $\overrightarrow{p_0p_2}$ with respect to the common point p_0?

(II) Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_1p_2}$, when we travel first from p_0 to p_1 and then from p_1 to p_2, do we make a left or right turn at p_1?

(III) Given two segments $\overline{p_0p_1}$ and $\overline{p_2p_3}$, do these two line segments intersect?

- All these problems can be solved in $O(\ldots)$ time !!!
- We will solve these problems by using only : $+, -, *$ and comparison (avoiding division and trigonometric functions, which are computationally expensive)

A Note on Trivial Approaches

Given two line segments $\overline{p_0p_1}$ and $\overline{p_2p_3}$, do they intersect?

- Find the equations

$$\begin{aligned} y = & \quad m_1x + b_1 \\ y = & \quad m_2x + b_2 \end{aligned}$$

- Find the intersection point

$$p_4 = \left(\frac{b_1 - b_2}{m_2 - m_1}, \frac{m_2b_1 - m_1b_2}{m_2 - m_1} \right)$$

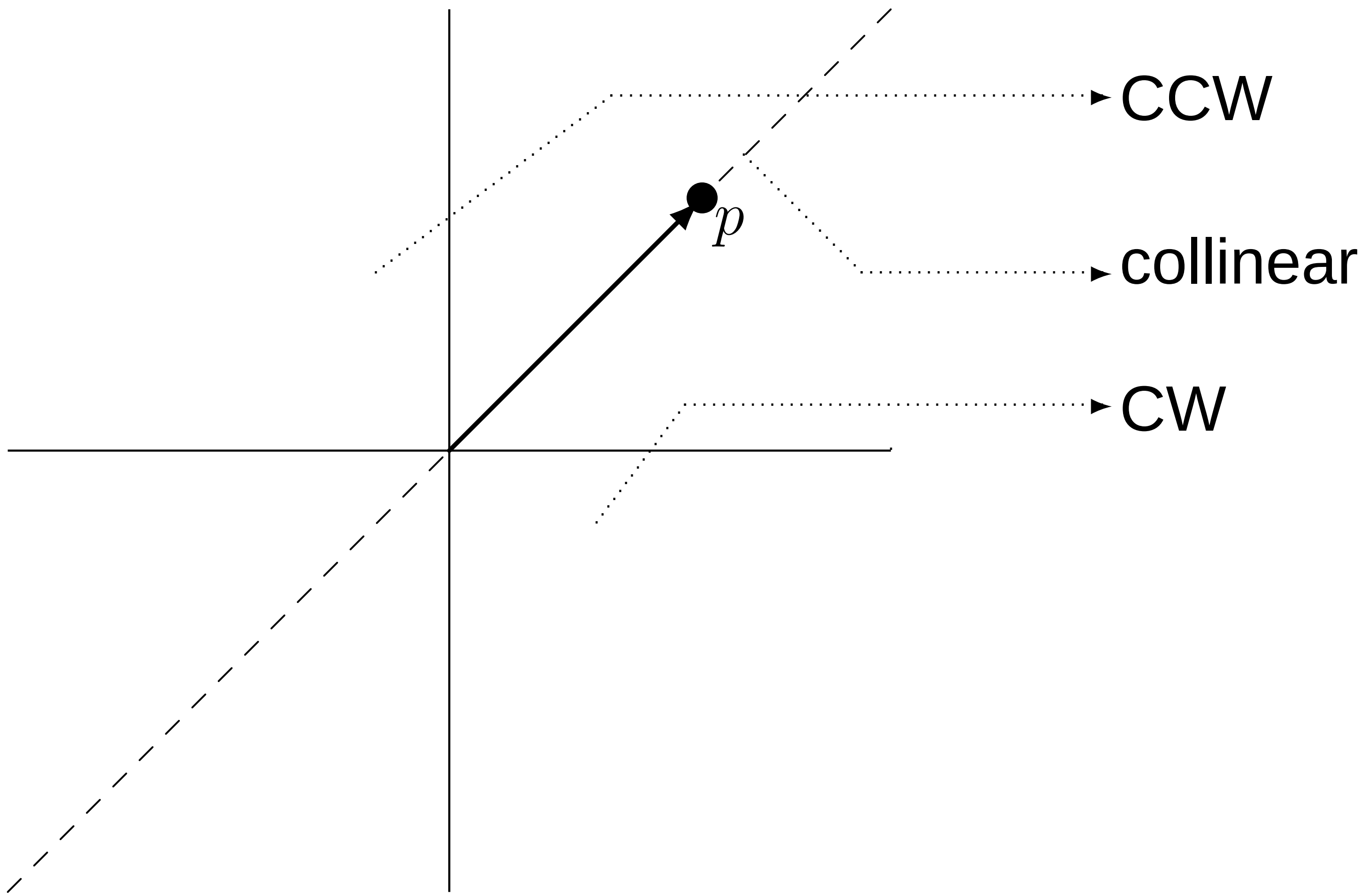
- Check if p_4 is a convex conbination of p_0 and p_1
- Check if p_4 is a convex conbination of p_2 and p_3

A Note on Trivial Approaches(cont'd)

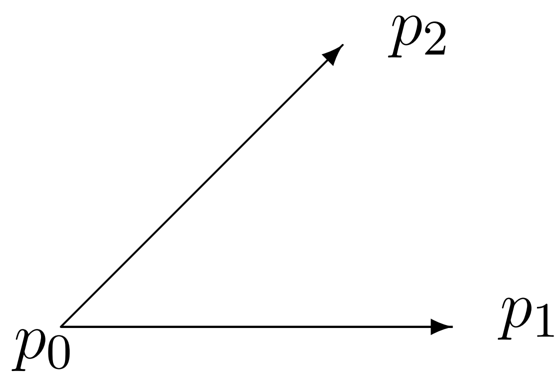
- This method requires division – expensive
- Very sensitive to the slopes of $\overline{p_0p_1}$ and $\overline{p_2p_3}$, i.e. to the value $|m_2 - m_1|$, as it may get quite close to 0.
- The slope of a non-horizontal line segment can be computed as 0 due to the precision limit of the computers.

Problem I

Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, is $\overrightarrow{p_0p_1}$ clockwise from $\overrightarrow{p_0p_2}$ with respect to the common point p_0?

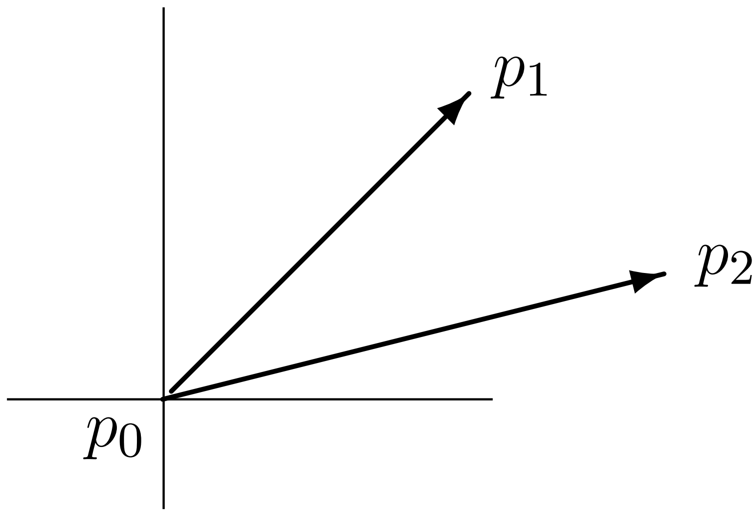


CW & CCW Segments



- $\overrightarrow{p_0p_1}$ is CW from $\overrightarrow{p_0p_2}$ wrt p_0
- $\overrightarrow{p_0p_2}$ is CCW from $\overrightarrow{p_0p_1}$ wrt p_0

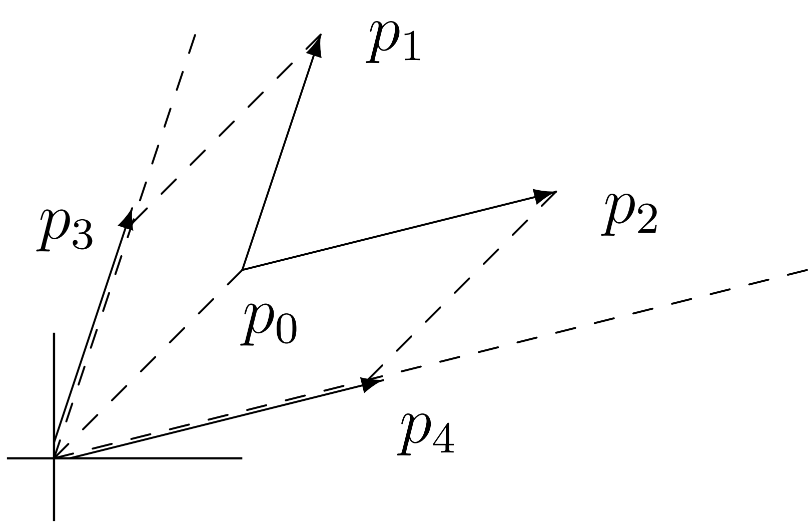
Special Case : p_0 is the origin



$$\overrightarrow{p_1} \times \overrightarrow{p_2} = \det \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} = x_1 y_2 - x_2 y_1$$

- $\overrightarrow{p_1} \times \overrightarrow{p_2} > 0$ implies $\overrightarrow{p_1}$ is CW from $\overrightarrow{p_2}$ wrt $(0, 0)$
- $\overrightarrow{p_1} \times \overrightarrow{p_2} < 0$ implies $\overrightarrow{p_1}$ is CCW from $\overrightarrow{p_2}$ wrt $(0, 0)$
- $\overrightarrow{p_1} \times \overrightarrow{p_2} = 0$ implies $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$ are collinear

General Case



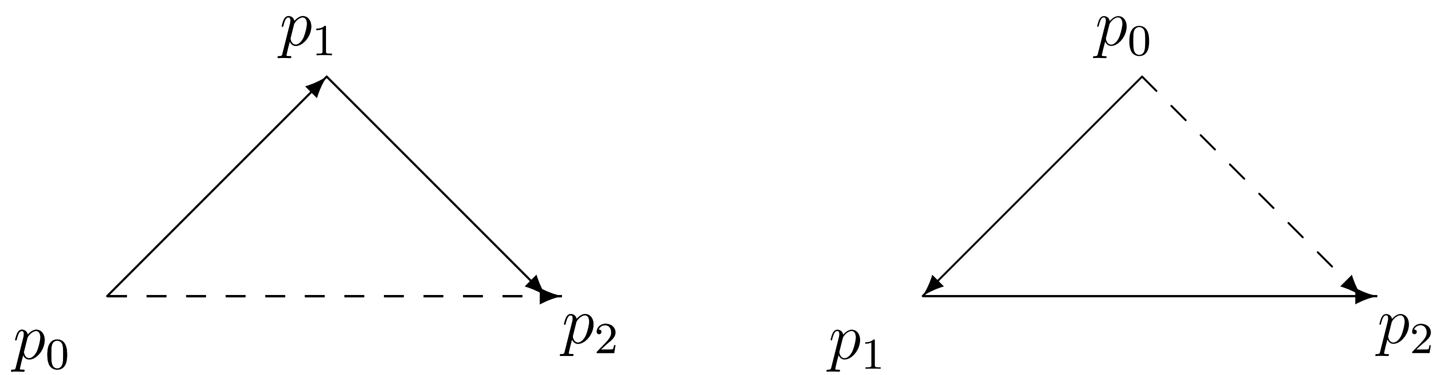
$$\overrightarrow{p_3} = \overrightarrow{p_0 p_1} - \overrightarrow{p_0}$$
$$\overrightarrow{p_4} = \overrightarrow{p_0 p_2} - \overrightarrow{p_0}$$

- $\overrightarrow{p_0 p_1}$ is CW (CCW) from $\overrightarrow{p_0 p_2}$ wrt p_0, iff $\overrightarrow{p_3}$ is CW (CCW) from $\overrightarrow{p_4}$ wrt $(0, 0)$

- $$\begin{aligned} \overrightarrow{p_3} \times \overrightarrow{p_4} &= (\overrightarrow{p_0 p_1} - \overrightarrow{p_0}) \times (\overrightarrow{p_0 p_2} - \overrightarrow{p_0}) \\ &= \det \begin{bmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{bmatrix} \\ &= (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) \end{aligned}$$

Problem II

Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_1p_2}$, when we travel first from p_0 to p_1 and then from p_1 to p_2, do we make a left or right turn at p_1?

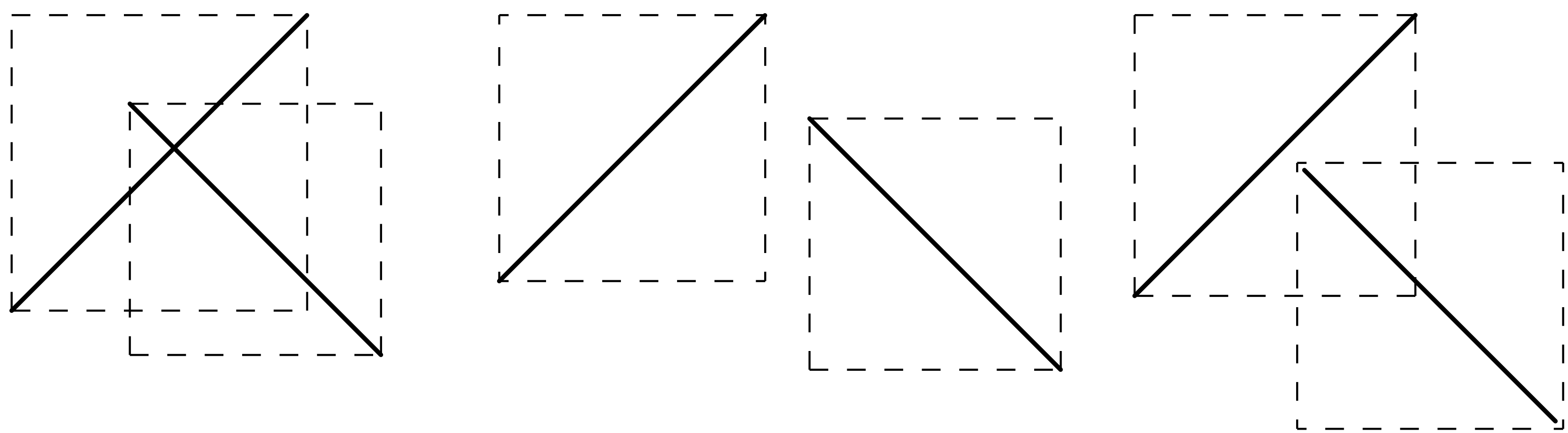


- "right turn" iff $\overrightarrow{p_0p_2}$ is CW from $\overrightarrow{p_0p_1}$ wrt p_0.
- "left turn" iff $\overrightarrow{p_0p_2}$ is CCW from $\overrightarrow{p_0p_1}$ wrt p_0.
- check the sign of $(\overrightarrow{p_0p_1} - \overrightarrow{p_0}) \times (\overrightarrow{p_0p_2} - \overrightarrow{p_0})$

Problem III

Given two segments $\overline{p_0p_1}$ and $\overline{p_2p_3}$, do these two line segments intersect?
The problem is solved using a two stage decision process:

- Stage 1: Quick rejection – tests a necessary condition – If 2 line segments intersect, then their bounding boxes also intersect



– Not a sufficient condition though!!!

Rectangles & Bounding Boxes

- A *rectangle* is represented by a pair of nodes $(\hat{p_1}, \hat{p_2})$, where $\hat{p_1}$ is the lower left corner, and $\hat{p_2}$ is the upper right corner of the rectangle.
- Let $(\hat{p_1}, \hat{p_2})$ (where $\hat{p_1} = (\hat{x_1}, \hat{y_1})$, $\hat{p_2} = (\hat{x_2}, \hat{y_2})$) be a rectangle, and $\overline{p_1 p_2}$ (where $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$) be a line segment. $(\hat{p_1}, \hat{p_2})$ is the *bounding box* of $\overline{p_1 p_2}$ iff

  (i) $\hat{x_1} = \min\{x_1, x_2\}$

  (ii) $\hat{x_2} = \max\{x_1, x_2\}$

  (iii) $\hat{y_1} = \min\{y_1, y_2\}$

  (vi) $\hat{y_2} = \max\{y_1, y_2\}$

Intersection of Rectangles

Given four points
$\hat{p_1} = (\hat{x_1}, \hat{y_1})$
$\hat{p_2} = (\hat{x_2}, \hat{y_2})$
$\hat{p_3} = (\hat{x_3}, \hat{y_3})$
$\hat{p_4} = (\hat{x_4}, \hat{y_4})$
the rectangles $(\hat{p_1}, \hat{p_2})$ and $(\hat{p_3}, \hat{p_4})$ intersect iff

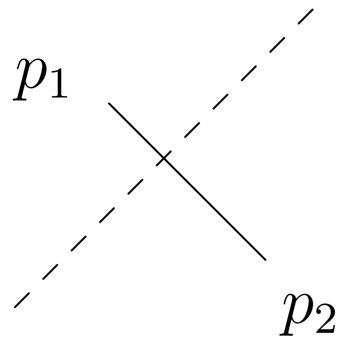
$$(\hat{x_2} \geq \hat{x_3}) \wedge (\hat{x_1} \leq \hat{x_4}) \wedge (\hat{y_2} \geq \hat{y_3}) \wedge (\hat{y_1} \leq \hat{y_4})$$

Back to Problem III

- Find the bounding boxes of the line segments : $O(1)$
- Check if the bounding boxes (which are rectangles) intersect : $O(1)$
- If not $\Rightarrow$ the line segments do not intersect
- If yes, go to the next stage

Segment–Line Crossing

- A line segment $\overline{p_1 p_2}$ *crosses* a line if p_1 appears on one side and p_2 appears on the other side of the line.



- If p_1 or p_2 is on the line, then we also say $\overline{p_1 p_2}$ crosses the line

Full Solution to Problem III

Two line segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersects iff

(i) they pass the quick rejection phase

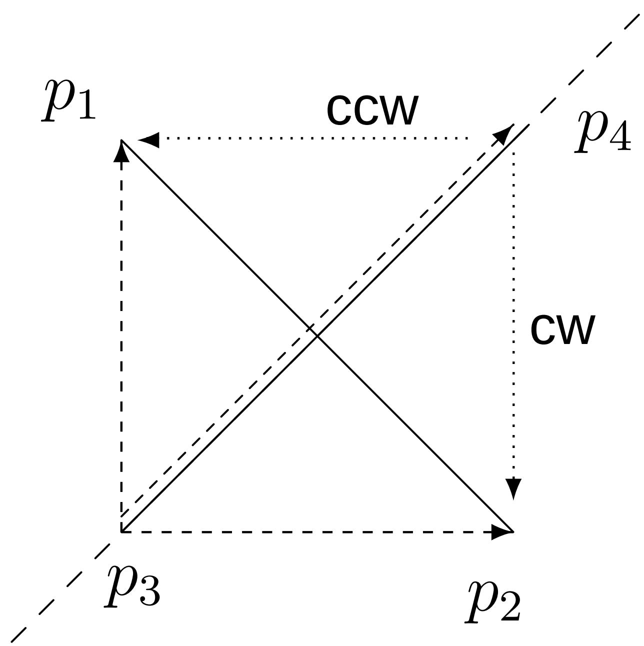
(ii) $\overline{p_1 p_2}$ crosses the line on which $\overline{p_3 p_4}$ resides

(iii) $\overline{p_3 p_4}$ crosses the line on which $\overline{p_1 p_2}$ resides

$\overline{p_1p_2}$ crosses the line of $\overline{p_3p_4}$: Intuitively

Note that, $\overline{p_1p_2}$ crosses the line on which $\overline{p_3p_4}$ resides iff either

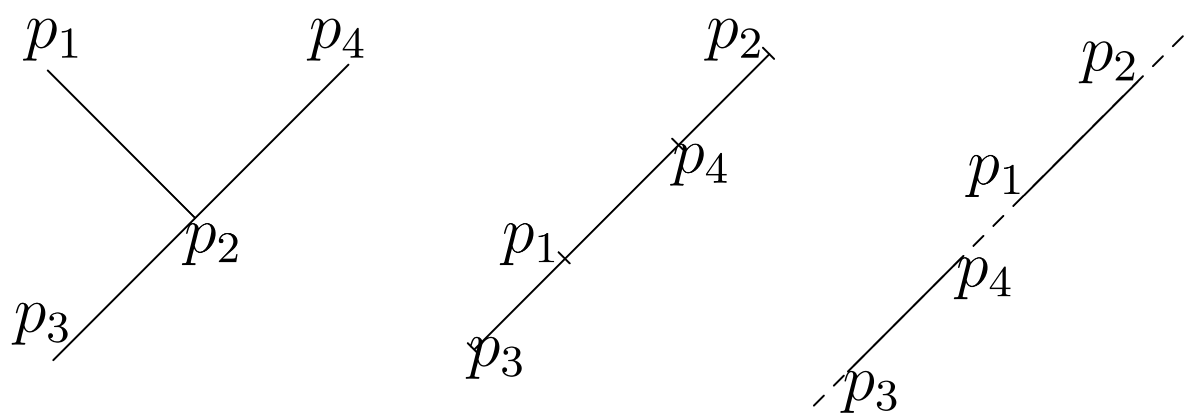
- $\overrightarrow{p_3p_1}$ is CW from $\overrightarrow{p_3p_4}$ wrt p_3 and $\overrightarrow{p_3p_2}$ is CCW from $\overrightarrow{p_3p_4}$ wrt p_3; or
- $\overrightarrow{p_3p_1}$ is CCW from $\overrightarrow{p_3p_4}$ wrt p_3 and $\overrightarrow{p_3p_2}$ is CW from $\overrightarrow{p_3p_4}$ wrt p_3



$\overline{p_2}$ crosses the line of $\overline{p_3p_4}$: Computational

$\overline{p_1p_2}$ crosses the line of $\overline{p_3p_4}$ iff

- $\overrightarrow{p_3p_1} \times \overrightarrow{p_3p_4}$ and $\overrightarrow{p_3p_2} \times \overrightarrow{p_3p_4}$ have either opposite signs, or at least one of them is 0.
- Note that the boundary conditions are also considered:



(a) (b) (c)

- In cases (a) & (b), $\overrightarrow{p_3p_1} \times \overrightarrow{p_3p_4}$ or $\overrightarrow{p_3p_2} \times \overrightarrow{p_3p_4}$ is 0.
- In case (c), both are 0 but quick rejection stage will filter out this case.

The Complete Algorithm

TwoSegmentsIntersect($\overrightarrow{p_1p_2}, \overrightarrow{p_3p_4}$) {

if (bounding box of $\overrightarrow{p_1p_2}$ and $\overrightarrow{p_3p_4}$ do not intersect) **then**

return false;

else

return(DoesSegmentCrossLineOf($\overrightarrow{p_1p_2}, \overrightarrow{p_3p_4}$) $\wedge$

DoesSegmentCrossLineOf($\overrightarrow{p_3p_4}, \overrightarrow{p_1p_2}$));

end if

}

DoesSegmentCrossLineOf($\overrightarrow{q_1q_2}, \overrightarrow{q_3q_4}$) {

// returns true iff $\overrightarrow{q_1q_2}$ crosses the line on which $\overrightarrow{q_3q_4}$ resides

if $((\overrightarrow{q_3q_1} \times \overrightarrow{q_3q_4} = 0) \vee (\overrightarrow{q_3q_2} \times \overrightarrow{q_3q_4} = 0))$ **then**

return true;

else

return ($(\overrightarrow{q_3q_1} \times \overrightarrow{q_3q_4}) * (\overrightarrow{q_3q_2} \times \overrightarrow{q_3q_4}) < 0)$;

end if

}

Problem: n Segment Intersection

Given a set of n segments, does there exist any to segments that intersect?

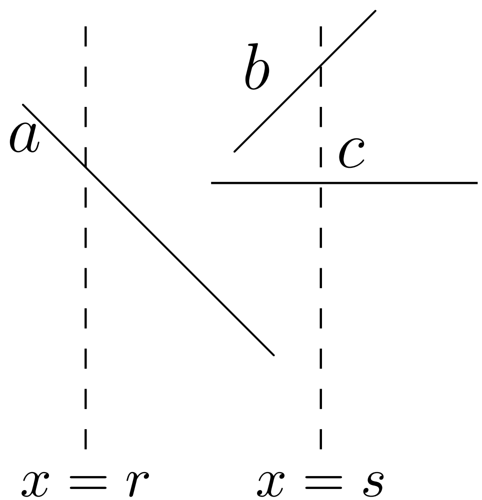
- Trivial approach: an $O(n^2)$ algorithm
  - Consider $\frac{n(n-1)}{2}$ possible pairs of segments one–by–one
  - Check if $\exists$ any pair of segments that intersect
- Can we find a faster algorithm?
- Note that we do not need to find the intersection point, but just need to find if $\exists$ two such segments
- Two assumptions:

(A1) None of the segments are vertical

(A2) No three segments intersect at a single point

Sweeping Technique

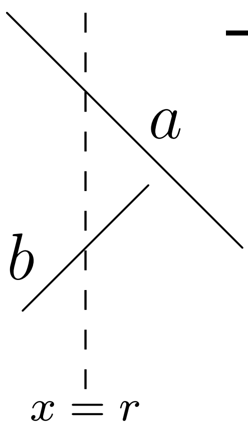
- A general technique used in many computational geometry algorithms
- An imaginary “sweep line” passes through the given set of objects to impose an order on the objects



- By (A1), a (vertical) sweep line intersects with a segment at most at one point
- We can order the segments that intersect with a sweep line wrt the $y-$ $-$coordinates of their intersection points with the sweep line

Ordering Segments on a Sweep Line

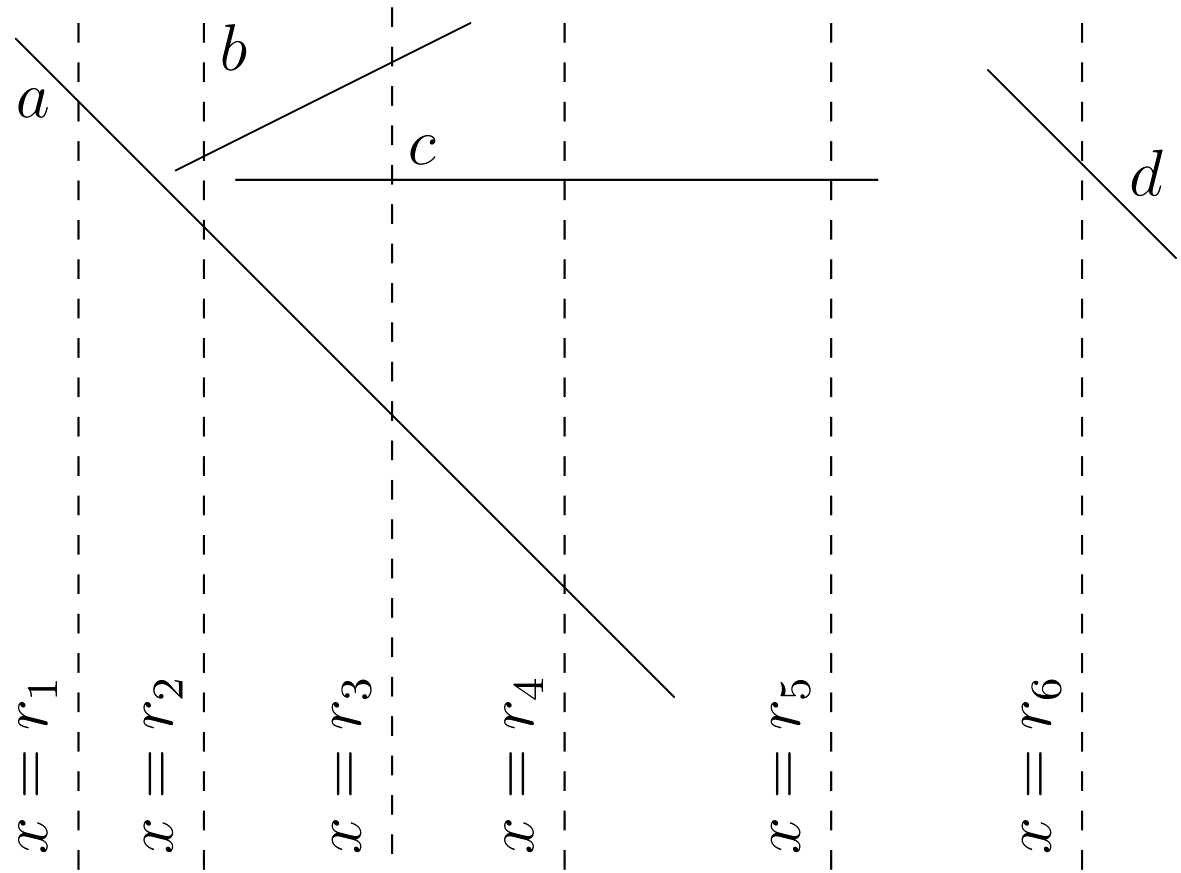
- Consider two non–intersecting segments a and b and a sweep line $x = r$
- a and b are *comparable* on $x = r$ if $x = r$ intersects with both a and b



– a is *above* b on $x = r$ (denoted by $a >_r b$) iff

- a and b are comparable on $x = r$
- The y coordinate of the intersection of a with $x = r$ is greater than the y coordinate of the intersection of b with $x = r$

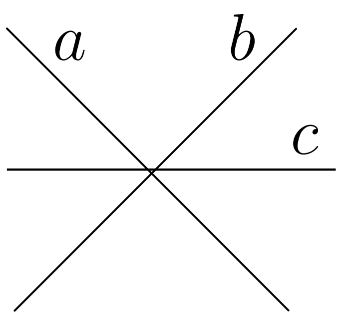
An Example



- $b >_{r_2} a$
- $b >_{r_3} c >_{r_3} a$
- $c >_{r_4} a$

A Useful Property of Orderings

- If two line segments a and b intersect, then $\exists$ a sweep line $x = r$ for which $a >_r b$ and $\not\exists c$ such that $a >_r c >_r b$
- In other words, if a and b intersect, then for some sweep line, they should be adjacent to each other wrt to the ordering imposed by that sweep line.

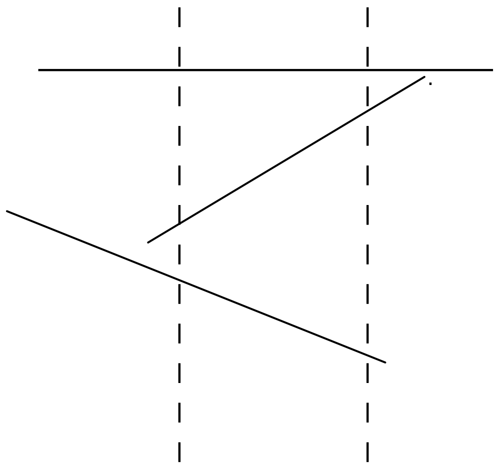


- Not exactly true: consider a and b in the figure on the left
- However, such cases (3 segments intersecting at a single point) are eliminated by the assumption A2

Reducing Number of Pairs to Check

- The useful property of the orderings can be used to reduce the numbers of segment pairs to be considered for the intersection (rather than checking all $\frac{n(n-1)}{2}$ possible pairs)
- Only the segment pairs that appear adjacent to each other on some sweep line can have a chance to intersect
- There are infinitely many sweep lines, which ones should we consider?
- Well, there are infinitely many sweep lines that induce the same ordering!!!
- As we will see, only a finite number of sweep lines can induce different orderings, hence it is sufficient to consider only them.

Finding the Useful Sweep Lines



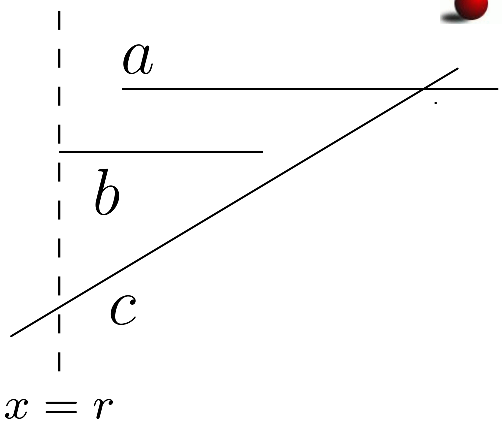
- Note that for any sweep line between $x = r$ and $x = s$, we will have the same ordering.
- The ordering only changes when our sweep line goes over an end point of some segment

- n segments

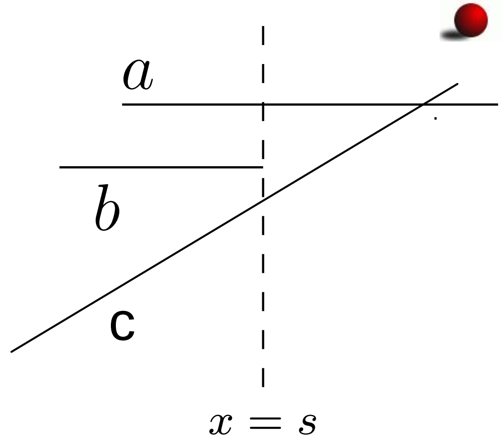
  $\Rightarrow 2n$ end points

  $\Rightarrow 2n$ sweep lines to consider

Pairs of Segments to Consider



- The sweep line goes over the left end point of a segment b: b will be included in the ordering starting from that point on. b should be checked for intersection with the segment above and below (e.g. check intersection of b with c for the figure on the left)



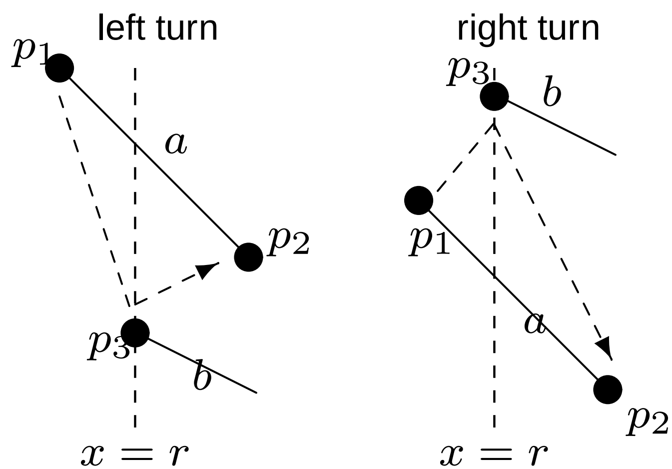
- The sweep line goes over the right end point of a segment b: b will be dropped out from the ordering. The segments above and below b in the old ordering will become adjacent to each other in the new ordering (e.g. a and c for the figure on the left)

Towards a Concrete Algorithm

- Need to store an ordered set of segments
- Use an RB–tree T to keep an ordered, dynamic set of segments
- Replace key comparisons with cross products
- When passing over the left end point of a segment s:
  $Insert(T, s) \longrightarrow O(\lg n)$
- When passing over the right end point of a segment s:
  $Delete(T, s) \longrightarrow O(\lg n)$
- To find the segment s' above a segment s:
  $s' = Successor(T, s) \longrightarrow O(\lg n)$
- To find the segment s' below a segment s:
  $s' = Predecessor(T, s) \longrightarrow O(\lg n)$

Using $\times$ for Ordering Segments

Note that, when we $Insert(T, b)$, we have to compare b with the other segments that are alredy in T.



- Let a be a segment already in T
- Let p_1 be a's left end point
- Let p_2 be a's right end point
- Let p_3 be b's left end point

- $a >_r b$ iff we turn left while following the vectors $\overrightarrow{p_1p_3}$ and then $\overrightarrow{p_3p_2}$
- $b >_r a$ iff we turn right while following the vectors $\overrightarrow{p_1p_3}$ and then $\overrightarrow{p_3p_2}$

The Algorithm

Any_Segment_Intersect (a set of n segments) {

$T = \emptyset$; $P =$ left to right sorted list end points;
for all end points $p \in P$ **do**
    if (p is the left endpoint of a segment s) **then**
        $Insert(T, s)$;
        if (($s' = Above(T, s)$) $\wedge$ (s' not NIL) $\wedge$ (s and s' intersects)) $\vee$
            (($s' = Below(T, s)$) $\wedge$ (s' not NIL) $\wedge$ (s and s' intersect s)) **then**
            return (true);
        end if
    else
        let $s' = Above(T, s)$ and $s'' = Below(T, s)$;
        $Delete(T, s)$;
        if (s' not NIL) $\wedge$ (s'' not NIL) $\wedge$ (s' and s'' intersects) **then**
            return(true);
        end if
    end if
end for
return(false);

}

Running Time

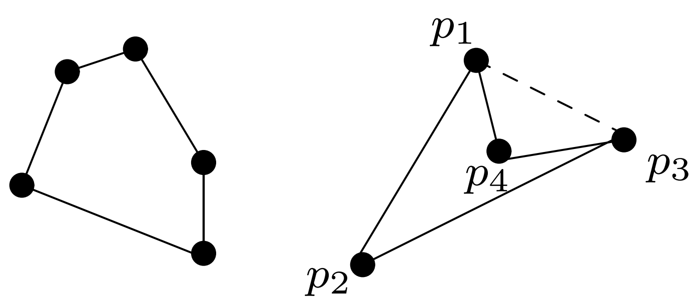
- Line 1: $O(n \lg n)$
- Line 4: $O(\lg n)$
- Line 5: $O(\lg n)$
- Line 9: $O(\lg n)$
- Line 10: $O(\lg n)$
- the for loop: $O(n \lg n)$ ($2n$ end points $\Rightarrow$ iterates $O(n)$ times)
- entire algorithm : $O(n \lg n)$

Problem: Finding the Convex Hull

- Given a set of points Q, find the smallest convex hull $CH(Q)$ such that $\forall p \in Q$, p is either inside or on the boundary of $CH(Q)$.
- A polygon with vertices $p_1, p_2, \ldots, p_n$ is *convex* iff $\forall 0 \le \alpha_1, \alpha_2, \ldots, \alpha_n \le 1$ such that $\sum_{1 \le i \le n} \alpha_i = 1$:

$$\alpha_1 p_1 + \alpha_2 p_2 + \cdots + \alpha_n p_n$$

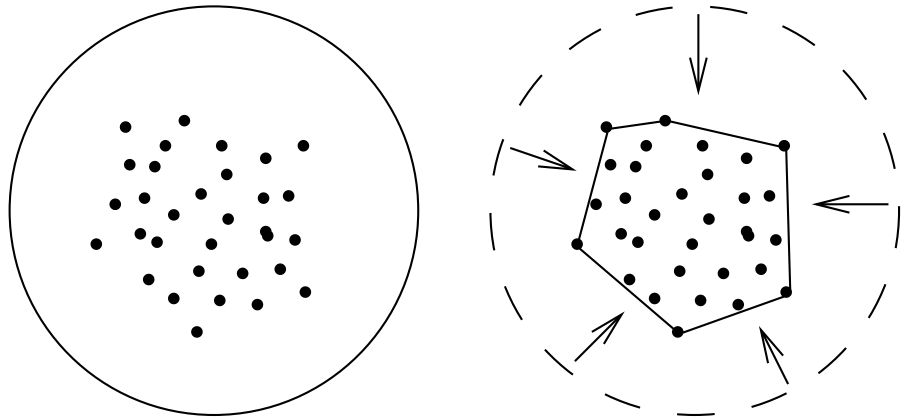
is either inside or on the boundary of the polygon.



(a) (b)

- (a) is convex
- (b) is not convex since $0.5p_1 + 0p_2 + 0.5p_3 + 0p_4$ is outside the polygon

Convex Hull of n Points



- Assume the points are pins sticking out from a board
- Hold a rubber band wide enough so that all the pins are indise the band and let the band go
- The shape the rubber band takes is the convex hull of the points
- The band only touches the pins corresponding to the vertices of the convex hull

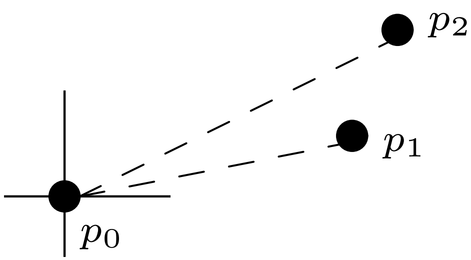
First Observations

- The point with the minimum y coordinate (let's call it p_0) will be a vertex of $CH(Q)$.
- If there are more than one points with the same minimum y coordinate, take the left most one as p_0.
- Similarly the points with the maximum y coordinate, minimum x coordinate and maximum x coordiate will all be the vertices of $CH(Q)$.
- Since we know p_0 will be a vertex of $CH(Q)$, we can start from p_0 and try to find the other vertices of $CH(Q)$.
- We will use the sweeping technique again.
- However, this time sweeping will be rotational.

Graham's Approach

- Rotational sweep is used in order to put an order on the points
- Builds a stack S of points
  - $TOP(S)$: returns top element of S (without popping it)
  - $NEXT2TOP(S)$: returns second top element of S (without popping it)
- Each point is pushed onto S exactly once
- Points that are not vertices of $CH(Q)$ are eventually popped from S
- At the end, S contains the vertices of $CH(Q)$

Sorting wrt Polar Angles



- In order to sort points wrt to their polar angles, we do not need to compute the angles
- Suppose p_0 is the origin
- We are given two points p_1 and p_2 in the same quadrant
- The polar angle (measured ccw) of a point p_1 from the x axis is less than that of a point p_2 iff: we make a right turn while travelling along the directed segments $\overrightarrow{p_1p_0}$ and $\overrightarrow{p_0p_2}$.
- If p_1 and p_2 are in different quadrants, than a simple comparison of their x and y coordinates can be used to determine their order wrt to their angle

Graham's Algorithm

Graham (Q : set of points) {

p_0 : the point with min y coordinate (if there are more than one, take the left most one)

$< p_1, p_2, \ldots p_m >$: the remaining points sorted wrt their angles (if there are more than one points having the same angle, take the one farthest away from p_0)

$S = empty$;

$Push(S, p_0)$; $Push(S, p_1)$; $Push(S, p_2)$;

for ($i = 3$; $i \leq m$; i++) **do**

    while $<NEXT2TOP(S):TOP(S):p_i>$ form a non–left turn **do**

        $POP(S)$;

    end while

    $PUSH(S, p_i)$;

end for

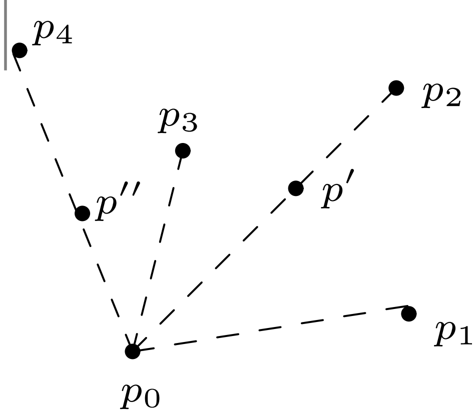
return the points in S;

}

Analysis of Graham's Algorithm

- Line 1: $\Theta(n)$
- Line 2: $O(n \lg n)$
- Line 4–9 : aggregate analysis $\longrightarrow O(n)$
- entire algorithm : $O(n \lg n)$

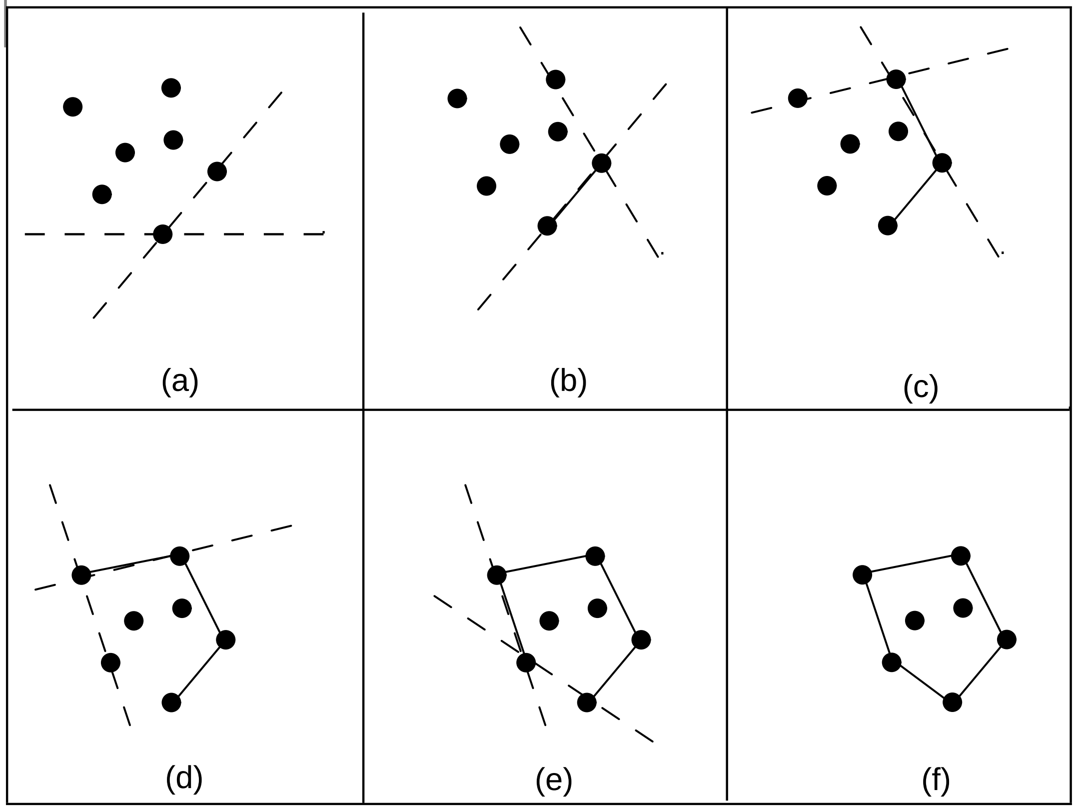
Example Run for Graham's Algorithm



- First the points are ordered wrt their polar angle from the x axis (from the origin p_0)
- Note that p_2 and p' have the same angle. We omit p' since it is closer to p_0.
- Same argument applies to p_4 and p''.

- Initially stack contains $p_0 : p_1 : p_2$, and we need consider the points p_3 and p_4 by the for loop at line 5
- Since $< p_1 : p_2 : p_3 >$ is a left turn, p_3 is pushed onto the stack, and stack becomes $p_0 : p_1 : p_2 : p_3$.
- Since $< p_2 : p_3 : p_4 >$ is a non–left turn p_3 is popped out
- Since $< p_1 : p_2 : p_4 >$ is a left turn, p_4 is pushed onto the stack, and stack becomes $p_0 : p_1 : p_2 : p_4$.

Jarvis's March



- Rotational sweep is used
- The idea of "gift wrapping"
- An $O(nh)$ algorithm, where h is the number of vertices of $CH(Q)$
- Asymptotically faster than Graham's algorithm when $h = o(\lg n)$.

Problem: Finding the Closest Pair of Points

Given a set of points Q with $|Q| \geq 2$, find the closest pair of points.

- For two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- Brute–force method:
  calculate for each pair of points $\longrightarrow O(n^2)$
- We'll see an $O(n \lg n)$ algorithm

The Algorithm

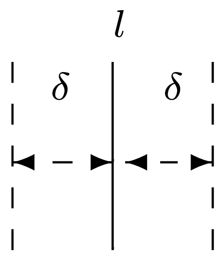
```
Find_Closest_Pair ( Q : set of points ) {

  1:  X = Q sorted wrt x coordinate of the points;
  2:  Y = Q sorted wrt y coordinate of the points;
  3:  Find_Closest_Pair_Sorted ( Q, X, Y );
}
```

The Subalgorithm

Find_Closest_Pair_Sorted (Q, X, Y) {

if $|Q| \leq 3$ **then**
    apply brute–force method;
end if
find a vertical line l that divides Q into Q_L and Q_R s.t. $|Q_L| = \left\lceil \frac{|Q|}{2} \right\rceil$
δ_L = Find_Closest_Pair_Sorted (Q_L, $X|_{Q_L}$, $Y|_{Q_L}$);
δ_R = Find_Closest_Pair_Sorted (Q_R, $X|_{Q_R}$, $Y|_{Q_R}$);
δ = $\min\{\delta_L, \delta_R\}$;
Y' = project Y on $\longrightarrow$



for all $p \in Y'$ **do**
    let $< p_1, p_2, \ldots, p_7 >$ be the next 7 points in Y'
    for ($i = 1$;$i \leq 7$;i++) **do**
        if $d(p, p_i) < \delta$ **then**
            $\delta = d(p, p_i)$;
        end if
    end for
end for
return δ;

}

Analysis of the Algorithm

In the main algorithm :

- Line 1: sorting $\rightarrow O(n \lg n)$
- Line 2: sorting $\rightarrow O(n \lg n)$

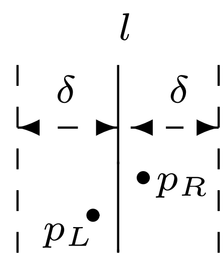
In the subalgorithm :

- Line 4: Finding median in a sorted set $\rightarrow O(1)$ (divide)
- Line 5 & 6: Conquer steps (and some divide steps due to projections)
- Remaining part : combine steps
- for loop at line 9 iterates $O(n)$ time
- for loop at line 11 iterates $O(1)$ time

Entire algorithm runs in $O(n \lg n)$ time

Explanation for the Combine Steps

- By the recursive calls, we know in both left and right part, the points are at least δ units away from each other
- However, a point $p_L \in P_L$ and a point $p_R \in P_R$ may be closer than δ units.



- For this to happen
  - p_L must be at most δ units away from the line l.
  - p_R must be at most δ units away from the line l.
  - y coordinates of p_L are p_R can differ at most δ units.
- Therefore if there exists such p_L and p_R they must be in a rectangle of $2\delta \times \delta$ centered on the line l.

Explanation for the Combine Steps (cont'd)

- Consider a rectangle with size $2\delta \times \delta$ centered on the line l. What is the maximum number of points that can fall into such a rectangle?
- The left half of the rectangle (which has size $\delta \times \delta$) can only contain points from P_L.
- Since the points in P_L are at least δ units away from each other, there can be at most 4 points on the left half (one point at each corner of the $\delta \times \delta$ rectangle).
- The same argument applies to the right half as well.
- Therefore, there can be at most 8 points in a rectangle of size $2\delta \times \delta$.

Explanation for the Combine Steps (cont'd)

- Hence, while considering a point p in Y', which is guaranteed to be at most δ units away from the line l, only the next 7 points can be in the same $2\delta \times \delta$ rectangle as p, due to the fact that Y' is sorted.