Advanced Algorithms

Lecture 6

Computational Geometry Algorithms:

Sweeping techniques

Bin Yang

byang@cs.aau.dk

Self-study 1

- Only 2 hand-ins
- Solutions have been uploaded to Moodle already. Check the solutions by yourselves!

ILO of Lecture 6

- Computational Geometry: sweeping techniques
  - to understand how the basic geometric operations (such as determining how two line segments are oriented and whether they intersect) are performed;
  - to understand the basic idea of the sweeping algorithm design technique;
  - to understand and be able to analyze the sweeping-line algorithm to determine whether any pair of line segments intersect
  - to understand and be able to analyze the Graham's scan algorithm for identifying convex hulls.

Agenda

- Computational geometry
- Basic geometric operations
- Sweeping techniques
- Graham's scan

Computational geometry

- Computational geometry studies algorithms for solving geometric problems.
- Algorithmic basis for many scientific and engineering disciplines:
  - Geographic Information Systems (GIS)
  - Robotics
  - Computer graphics
  - Computer vision
  - Computer Aided Design/Manufacturing (CAD/CAM),
  - Very-large-scale integration (VLSI) design.

Computational geometry problems

- Input: a description of a set of geometric objects.
  - A set of points
  - A set of line segments
  - Vertices of a polygon
- Output:
  - a response to a query about the objects
    - E.g., whether any of the lines intersect
  - a new geometric object,
    - E.g., convex hull of the set of points

- We will deal with *points* and *line segments* in **2D** space.

Agenda

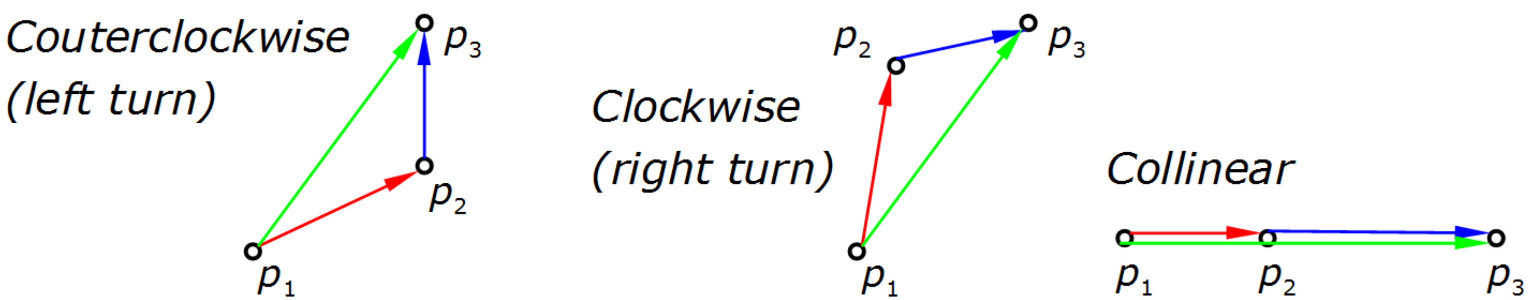
- Computational geometry
- Basic geometric operations
- Sweeping techniques
- Graham's scan

Line-segment properties

- What is the line-segment $\overline{p_1 p_2}$ between $p_1=(x_1, y_1)$ and $p_2=(x_2, y_2)$?
  - It contains any point p_3 that is on the line passing through p_1 and p_2 and is on or between p_1 and p_2 on the line.
  - The set of ***convex combinations*** of $p_1=(x_1, y_1)$ and $p_2=(x_2, y_2)$.
    - $p_3 = \alpha\, p_1 + (1- \alpha\,)\, p_2$ where $0 \leq \alpha \leq 1$
  - $p_1=(0, 0)$ and $p_2=(10, 10)$
    - $\alpha = 0$, p_2.
    - $\alpha = 1$, p_1.
    - $\alpha = 0.5$, $(5, 5)$
    - $\alpha = 0.02$, $(9.8, 9.8)$
  - We call p_1 and p_2 as the endpoints of the line-segment $\overline{p_1 p_2}$.
- Directed line-segment $\overrightarrow{p_1 p_2}$ from p_1 to p_2.

Basic operation

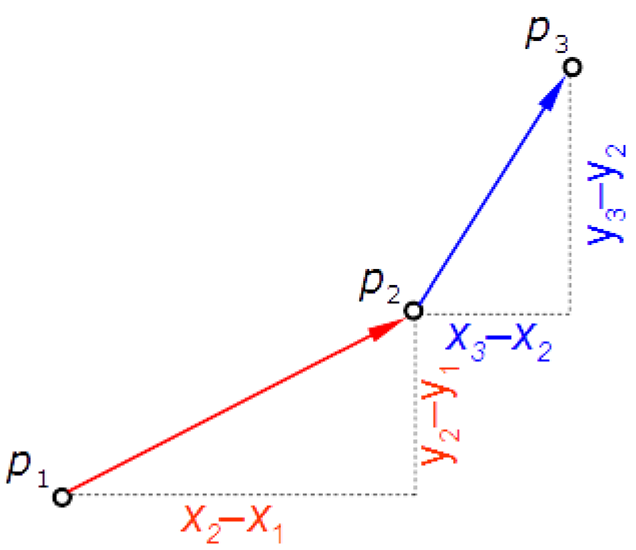
- How to find "orientation" of two line segments?
  - Three points: $p_1(x_1, y_1)$, $p_2(x_2, y_2)$, $p_3(x_3, y_3)$
  - Is segment $\overrightarrow{\mathrm{p_1p_3}}$ **clockwise** or **counterclockwise** from $\overrightarrow{\mathrm{p_1p_2}}$?
  - Going from segment $\overline{\mathrm{p_1p_2}}$ to segment $\overline{\mathrm{p_2p_3}}$, do we make a **right** or a **left** turn?



  - For simplicity, use p_1p_2 to denote $\overline{p_1p_2}$.

Computing the orientation

- Compute the slopes of the two line-segments:
  - Slope of segment p_1p_2: $a = (y_2 - y_1)/(x_2 - x_1)$
  - Slope of segment p_2p_3: $b = (y_3 - y_2)/(x_3 - x_2)$
- How do you compute the orientation then?
  - When $a \geq 0$ and $b \geq 0$
    - counterclockwise (left turn): $a<b$
    - clockwise (right turn): $a>b$
    - collinear (no turn): $a=b$
- p1(0,0), p2(2,1), p3(3, 3)
  - p1p2: (1-0)/(2-0)=0.5
  - p2p3: (3-1)/(3-2)=2
  - 0.5<2, thus p1p2 left turn to p2p3.
  - p3p2: (1-3)/(2-3)=2
  - p2p1: (0-1)/(0-2)=0.5
  - 2>0.5, thus p3p2 right turn to p2p1.



Problem of using slopes

- When computing slopes, we need the division operation.
- When segments are nearly parallel, this method is very sensitive to the precision of the division operation on real computers.
- p1(0,0), p2(2,1), p3(4.0000001, 2.0000001)
  - p1p2: (1-0)/(2-0)=0.5
  - p2p3: (2.0000001-1)/(4.0000001-2)=1.0000001/2.0000001
    - 0.5, collinear
    - 0,50000001, left turn.

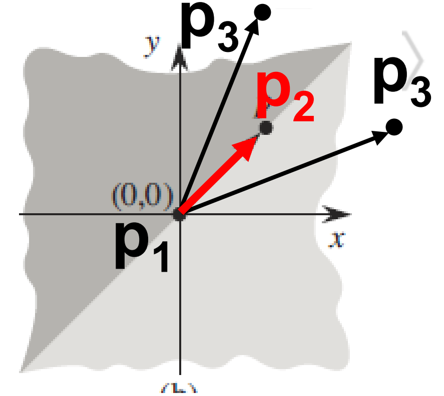
- If a method avoids division, it is much more accurate.

Method without division: cross product

- Finding orientation without division to avoid numerical problems on different computers.
- Whether p_1p_3 is clockwise or counter-clockwise from $\boldsymbol{p_1p_2}$.
- Cross product
  - $(p_3-p_1)\times(p_2-p_1)=(x3-x1)(y2-y1)-(x2-x1)(y3-y1)$
- Or determinant of the following matrix
  - $\begin{pmatrix} x3-x1 & x2-x1 \\ y3-y1 & y2-y1 \end{pmatrix}$
- Positive – p_1p_3 is clockwise from $\boldsymbol{p_1p_2}$
- Negative – p_1p_3 is counterclockwise from $\boldsymbol{p_1p_2}$
- Zero – collinear

Example

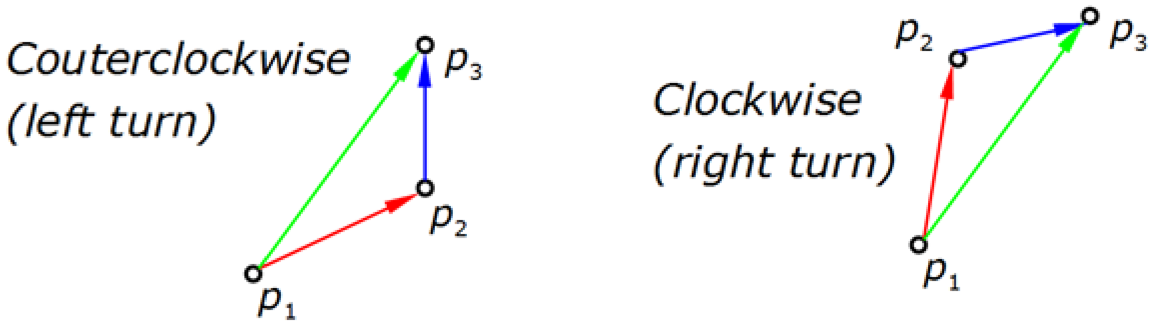
$$\begin{pmatrix} x3 - x1 & x2 - x1 \\ y3 - y1 & y2 - y1 \end{pmatrix}$$



- Assume p_1=(0, 0) p_2=(1,1)
- If p_3 is in the lightly shaded region, p_1p_3 is clockwise from p_1p_2
  - E.g., p_3=(2, 1), we have $\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$=2-1=1
- If p_3 is in the darkly shaded region, p_1p_3 is counter-clockwise from p_1p_2
  - E.g., p_3=(1, 2), we have $\begin{pmatrix} 1 & 1 \\ 2 & 1 \end{pmatrix}$=1-2=-1

Determine left/right turn

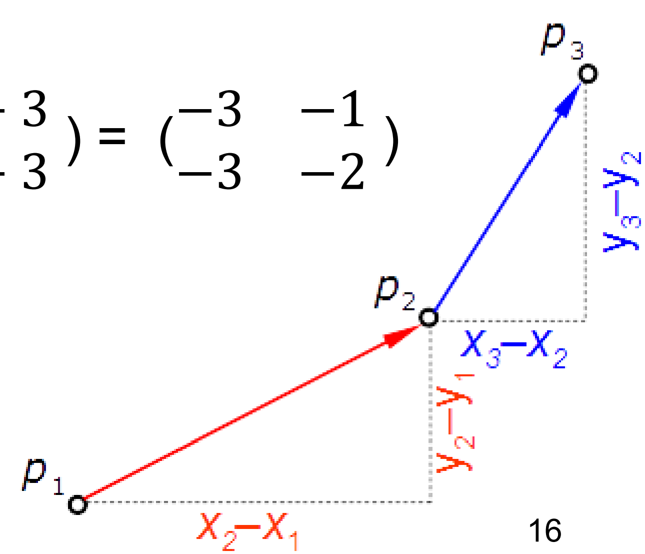
- Determine whether two consecutive segments p_1p_2 and p_2p_3 turn left or right at p_2.



- Segment p_1p_3 is clockwise or counterclockwise relative to segment $\boldsymbol{p_1p_2}$
  - Counterclockwise: left turn. Clockwise: right turn.

Examples

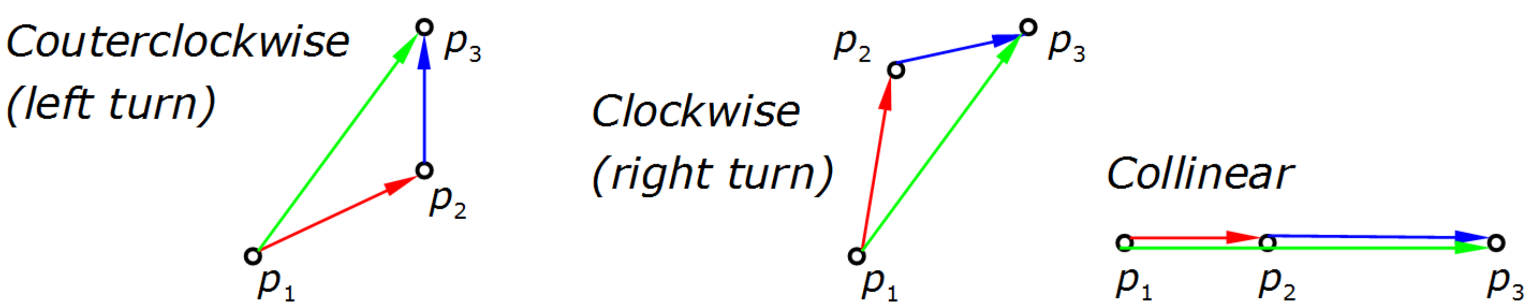
- p1(0,0), p2(2,1), p3(3, 3)
- From p1p2 and p2p3, left or right turn?
  - Determine whether p1p3 is clockwise/counter-clockwise from ***p1p2***.
  - p1p3, ***p1p2***: (p3-p1) × (p2-p1)=$\begin{pmatrix} 3-0 & 2-0 \\ 3-0 & 1-0 \end{pmatrix}$ = $\begin{pmatrix} 3 & 2 \\ 3 & 1 \end{pmatrix}$=3-6=-3, counter-clockwise, thus left turn.
- From p3p2 to p2p1, left or right turn?
  - Determine whether p3p1 is clockwise/counter-clockwise from ***p3p2***.
  - p3p1, ***p3p2***: (p1-p3) × (p2-p3)= $\begin{pmatrix} 0-3 & 2-3 \\ 0-3 & 1-3 \end{pmatrix}$ = $\begin{pmatrix} -3 & -1 \\ -3 & -2 \end{pmatrix}$ =6-3=3, clockwise, thus right turn.



A quick summary

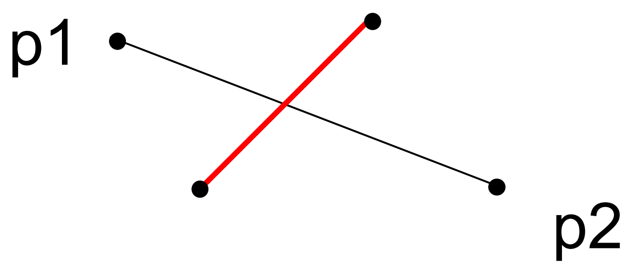
- Q1: Whether p_1p_3 is clockwise/counter-clockwise from p_1p_2?
- Q2: From p_1p_2 to p_2p_3, do you need to turn right/left at p_2?
- Compute the cross product
  - $(p_3-p_1)\times(p_2-p_1)=(x3-x1)(y2-y1)-(x2-x1)(y3-y1)$
  - $= \begin{pmatrix} x3-x1 & x2-x1 \\ y3-y1 & y2-y1 \end{pmatrix}$

Cross product	Q1	Q2
Negative	Counterclockwise	Left turn
Positive	Clockwise	Right turn
Zero	Collinear	Go straight



Whether two line segments intersect

- A segment p1p2 ***straddles*** a line if point p1 lies on one side of the line but p2 lies on the other side.



  - Along the line, one needs to turn different directions to go to p1 and p2.
- Two line segments intersect if and only if either of the following two conditions holds:
  - Each segment straddles the line containing the other.
  - An endpoint of one segment lies on the other segment.

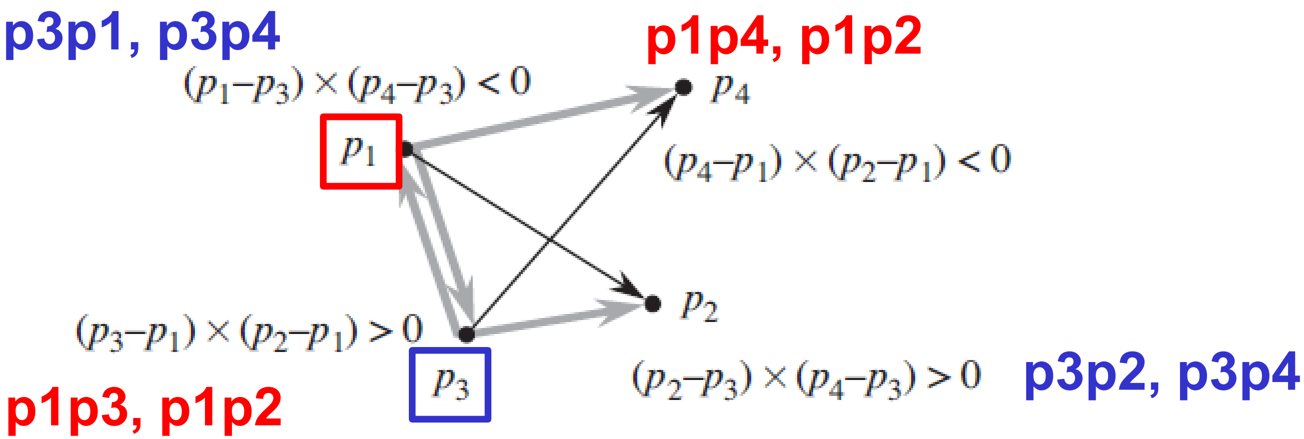
Pseudo code

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
d1 = DIRECTION(p3, p4, p1)
d2 = DIRECTION(p3, p4, p2)
d3 = DIRECTION(p1, p2, p3)
d4 = DIRECTION(p1, p2, p4)
if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
        return TRUE
elseif d1 == 0 and ON-SEGMENT(p3, p4, p1)
        return TRUE
elseif d2 == 0 and ON-SEGMENT(p3, p4, p2)
        return TRUE
elseif d3 == 0 and ON-SEGMENT(p1, p2, p3)
        return TRUE
elseif d4 == 0 and ON-SEGMENT(p1, p2, p4)
        return TRUE
else return FALSE
```

Check if each segment straddles the line containing the other.
If p1p2 straddles p3p4 and p3p4 straddles p1p2

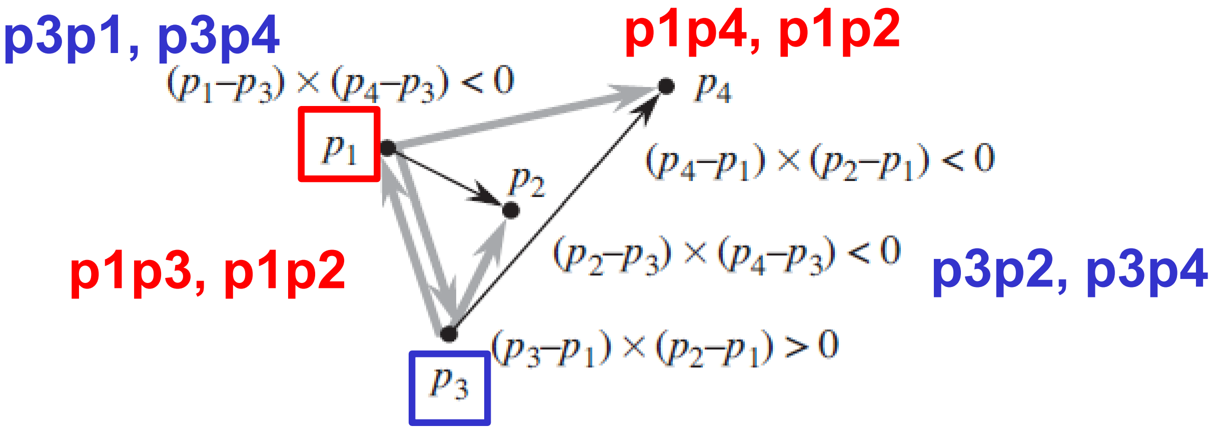
An endpoint of one segment lies on the other segment.

Example



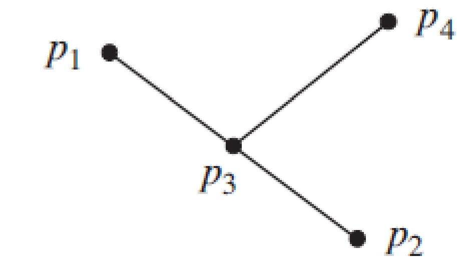
- Check if p1p2 straddles p3p4
  - The turn from p3p4 to p4p1 vs. the turn from p3p4 to p4p2.
  - p3p1, p3p4: (p1-p3) × (p4-p3) < 0
  - p3p2, p3p4: (p2-p3) × (p4-p3) > 0, so yes.
- Then, check if p3p4 straddles p1p2
  - The turn from p1p2 to p2p3 vs. the turn from p1p2 to p2p4.
  - p1p3, p1p2: (p3-p1) × (p2-p1) >0
  - p1p4, p1p2: (p4-p1) × (p2-p1) <0, so yes.
- Yes, they intersect.

Example 2

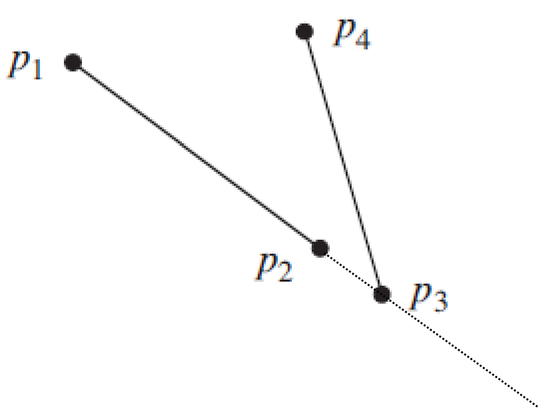


- Check if p1p2 straddles p3p4
  - The turn from p3p4 to p4p1 vs. the turn from p3p4 to p4p2.
  - p3p1, p3p4: (p1-p3) × (p4-p3) < 0
  - p3p2, p3p4: (p2-p3) × (p4-p3) < 0, so no.
- Then, check if p3p4 straddles p1p2
  - The turn from p1p2 to p2p3 vs. the turn from p1p2 to p2p4.
  - p1p3, p1p2: (p3-p1) × (p2-p1) >0
  - p1p4, p1p2: (p4-p1) × (p2-p1) <0, so yes.
- No, they do not intersect.

Example 3



- p1p2, p2p3: collinear.
- p3 is on segment p1p2.
- So they intersect.



- p1p2, p2p3: collinear.
- But p3 is not on segment p1p2.
- So they do not intersect.

Mini-quiz

- What is the time complexity of the algorithm that tests if two segments intersect?

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
d1 = DIRECTION(p3, p4, p1)
d2 = DIRECTION(p3, p4, p2)
d3 = DIRECTION(p1, p2, p3)
d4 = DIRECTION(p1, p2, p4)
if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
    return TRUE
elseif d1 == 0 and ON-SEGMENT(p3, p4, p1)
    return TRUE
elseif d2 == 0 and ON-SEGMENT(p3, p4, p2)
    return TRUE
elseif d3 == 0 and ON-SEGMENT(p1, p2, p3)
    return TRUE
elseif d4 == 0 and ON-SEGMENT(p1, p2, p4)
    return TRUE
else return FALSE
```

Constant time.

A quick summary

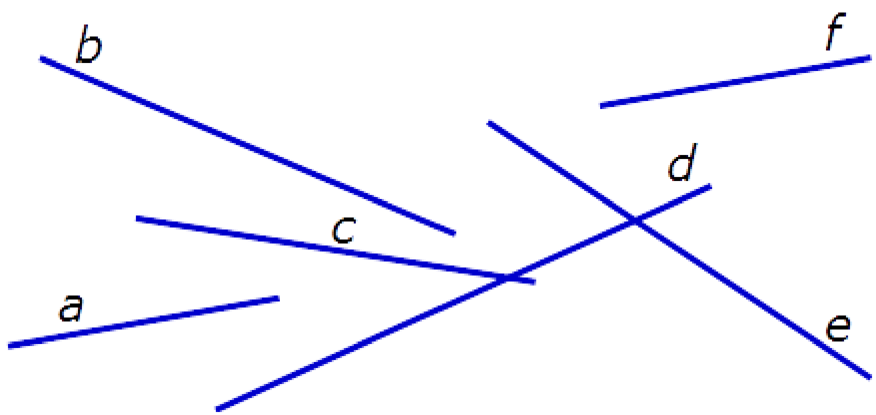
- Cross product is a fundamental operation in computational geometry.
- Checking whether two segments intersect is based on cross products.
- The complexity of checking whether two segments intersect is constant.

Agenda

- Computational geometry
- Basic geometric operations
- Sweeping techniques
- Graham's scan

Intersections in a set of line segments

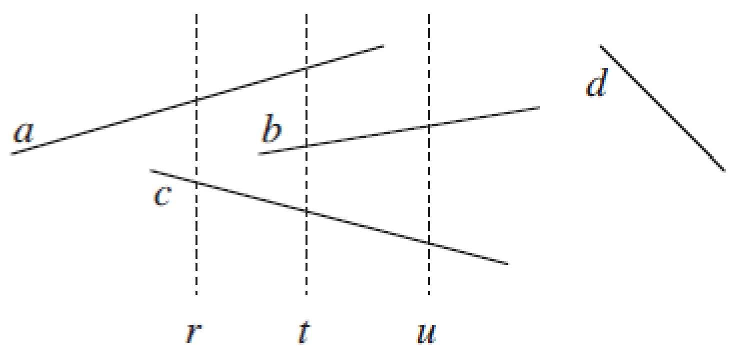
- Given a set of *n* line segments, determine whether any two line segments intersect.
  - Note: not asking to report all intersections, but just true or false.
  - *What would be the brute force algorithm and what is its worst-case complexity in terms of n, i.e., the number of line segments?*



  - *We will see a O(nlgn) algorithm using the sweeping technique.*

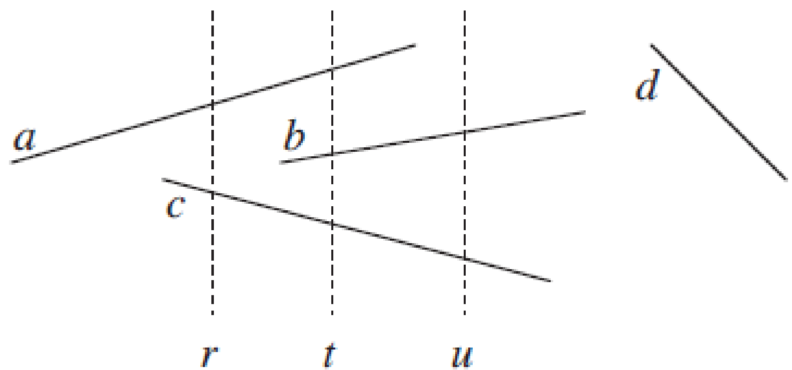
Sweeping

- Image a vertical sweep line passes through the given set of geometric objects, usually from left to right.
- Sweeping provides a method for ***ordering*** geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of the relationships among them.



Ordering segments

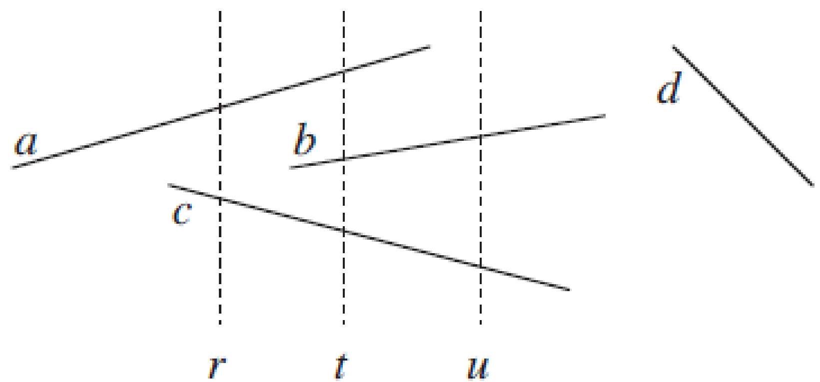
- Assume that in the given set of line segments, we do not have vertical line segments.
- We order the given segments that intersects a vertical sweep line according to the y-coordinates of the points of intersection.



- Given two segments s1 and s2. They are comparable at x if the vertical sweep line with x-coordinate being x intersects both of them.
  - E.g., Segments a and c are comparable at r.

Ordering segments

- We say that s1 is above s2 at x, denoted as s1$\geq_x$ s2.
  - if s1's y coordinate of the intersection is higher than that of s2's.
  - or if s1 and s2 intersect at the sweep line.



  - At r: a $\geq_r$ c
  - At t: a $\geq_t$ b, a $\geq_t$ c , b $\geq_t$ c
  - At u: b $\geq_u$ c

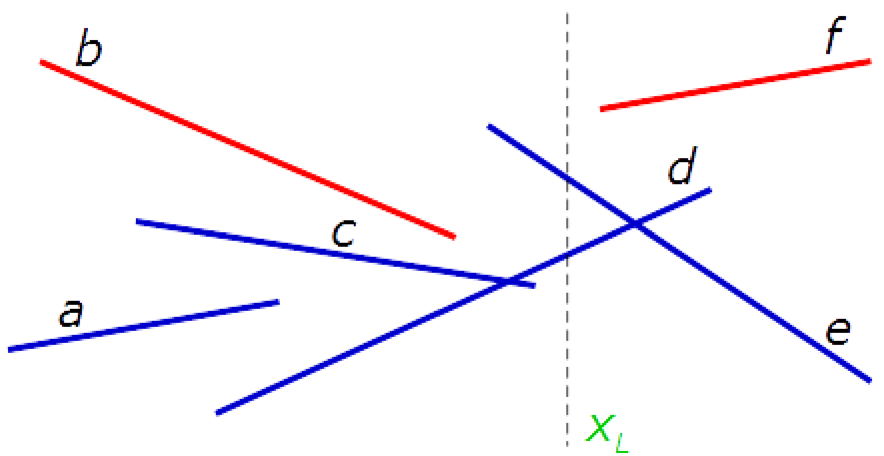
Moving the sweep line

- Sweeping algorithms typically manage two sets of data:
- **Sweep-line status:** the relationships among the objects that the sweep line intersects.
- **Event-point schedule:** is a sequence of points where updates to the sweep-line status are required.

- Let's see two algorithms using the sweeping techniques which both are able to identify whether any two line segments intersect.

Algorithm 1

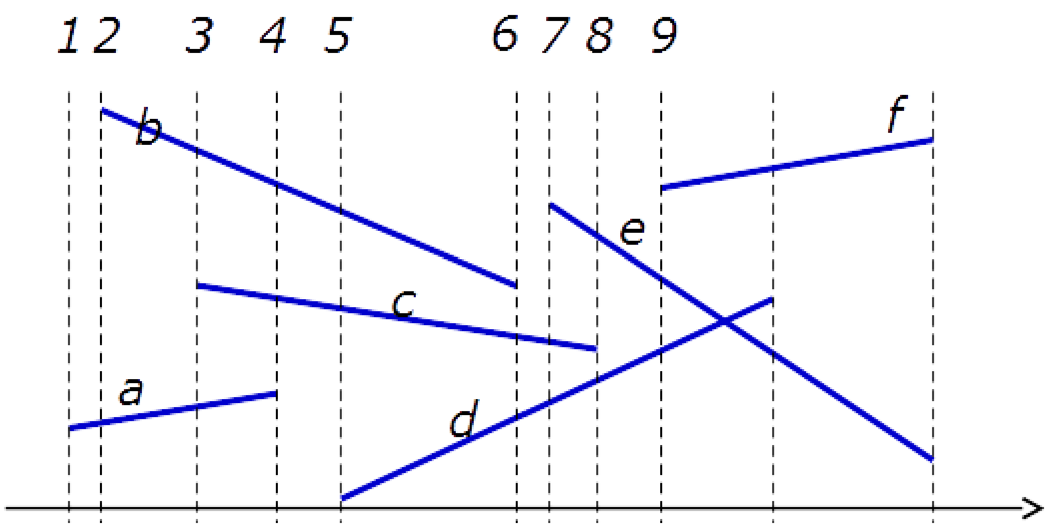
- Observations:
  - *Two segments definitely* ***do not*** *intersect if their projections to the x axis do not intersect.*
  - In other words: *If segments intersect, there is some x_L such that a vertical line* at *x_L intersects both segments.*



  - *b and f cannot intersect since their projects to x axis do not intersect.*

Algorithm 1

- **Event-point schedule:**
  - Each segment's end points are event points.
  - Order them from left to right.
- **Sweep-line status:**
  - At an event point, update the status of the sweep line and perform intersection tests.
  - Left end point: a new segment is added to the status and it needs to be checked against ***all*** the existing segments in the status
  - Right end point: the corresponding segment is deleted from the status.



1. {a}
2. {a, b}
3. {a, b, c}
4. {b, c}
5. {b, c, d}
6. {c, d}
7. {c, d, e}

d and e intersect, stop

Mini quiz

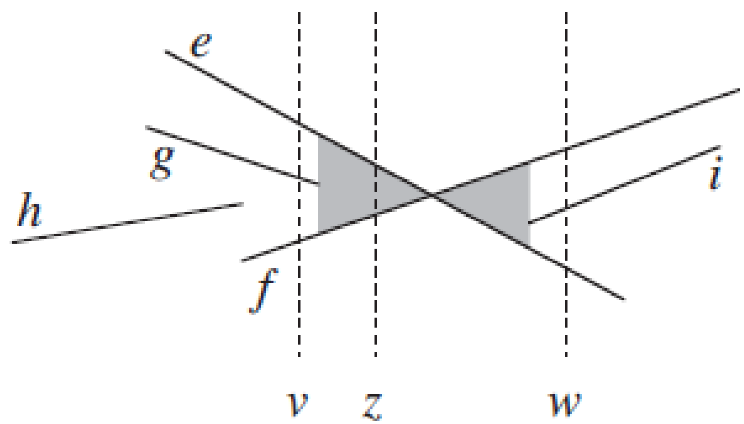
- What is the worst case example?
- What is the worst case complexity?
- Is it better than brute-force?

- $O(n^2)$

- Why we can remove a segment from the status when we see its right end point?
- Why do we insert a segment into the status when we see its left end point?

Algorithm 2

- More useful observations:
  - For a specific position of the sweep line, there is an **order of** segments in the y-axis;
  - If two segments intersect, there is a position of the sweep-line such that the two segments are **adjacent** in this order;
  - Order does not change in-between event points until the first intersection point.



  - We do not need to check ***all*** segments in the sweep-line status, but only the ***adjacent*** ones (which are ***at most 2 segments***).

Algorithm 2

- Sweep-line status data structure:
  - Operations:
    - Insert(T, s): insert segment s into the status T.
    - Delete(T, s): delete segment s from the status T.
    - Above(T, s): return the segment immediately above s in T, predecessor.
    - Below(T, s): return the segment immediately below s in T, successor.
  - Balanced binary search tree T (e.g., red-black tree)
    - All operations can be done in O(lgn).

Algorithm 2

Event-point schedule

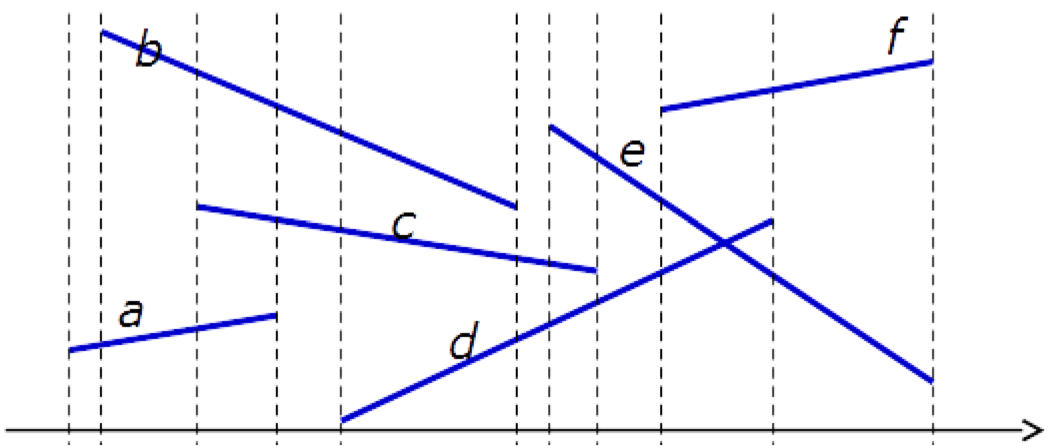
```
ANY-SEGMENTS-INTERSECT(S)
T = Ø
sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
for each point p in the sorted list of endpoints
        if p is the left endpoint of a segment s
                INSERT(T, s)
                if (ABOVE(T, s) exists and intersects s)
                        or (BELOW(T, s) exists and intersects s)
                        return TRUE
        if p is the right endpoint of a segment s
                if both ABOVE(T, s) and BELOW(T, s) exist
                        and ABOVE(T, s) intersects BELOW(T, s)
                        return TRUE
                DELETE(T, s)
return FALSE
```

A new segment comes in. Check it with its predecessor and its successor.

An old segment comes out. Check its predecessor and its successor.

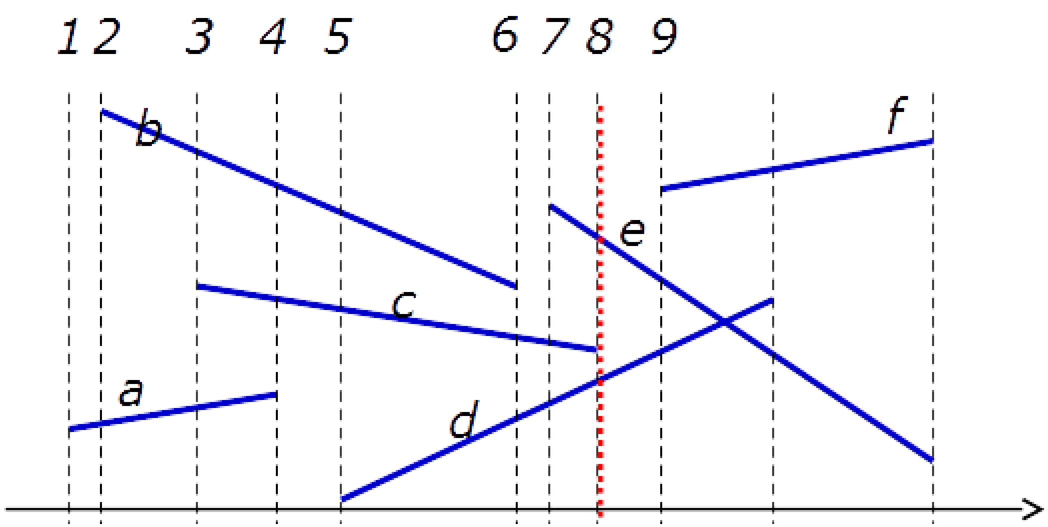
Mini quiz (also on Moodle)

- How many intersection tests do we need to do on the following set of segments?
- At which event an intersection is discovered?
  - Use this format: segment.l or segment.r
    - E.g.: a.l, a.r, b.l, b.r



Mini quiz

- How many intersection tests do we need to do on the following set of segments?
- At which event an intersection is discovered?
  - Use this format: segment.l or segment.r
    - E.g.: a.l, a.r, b.l, b.r



1. <a>, 0 check.
2. <b, a>, 1 check: ba
3. <b, c, a>, 2 checks: bc, ca
4. <b, c>, 0 check.
5. <b, c, d>, 1 check: cd
6. <c, d>, 0 check.
7. <e, c, d>, 1 check: ec
8. <e, d> , 1 check: ed, found!

6 checks.

c.r

Algorithm 2 Complexity

```
ANY-SEGMENTS-INTERSECT(S)
T = ∅
sort the endpoints of the segments in S from left to right,
      breaking ties by putting left endpoints before right endpoints
      and breaking further ties by putting points with lower
      y-coordinates first
for each point p in the sorted list of endpoints
      if p is the left endpoint of a segment s
            INSERT(T, s)
            if (ABOVE(T, s) exists and intersects s)
                  or (BELOW(T, s) exists and intersects s)
                  return TRUE
      if p is the right endpoint of a segment s
            if both ABOVE(T, s) and BELOW(T, s) exist
                  and ABOVE(T, s) intersects BELOW(T, s)
                  return TRUE
            DELETE(T, s)
return FALSE
```

Event-point schedule. Sorting O(nlgn)

For loop iterates 2n times.

Each of the insert, delete, above, below operations takes O(lgn).

Intersection test takes constant time.

In total, O(nlgn).

Sweeping technique principles

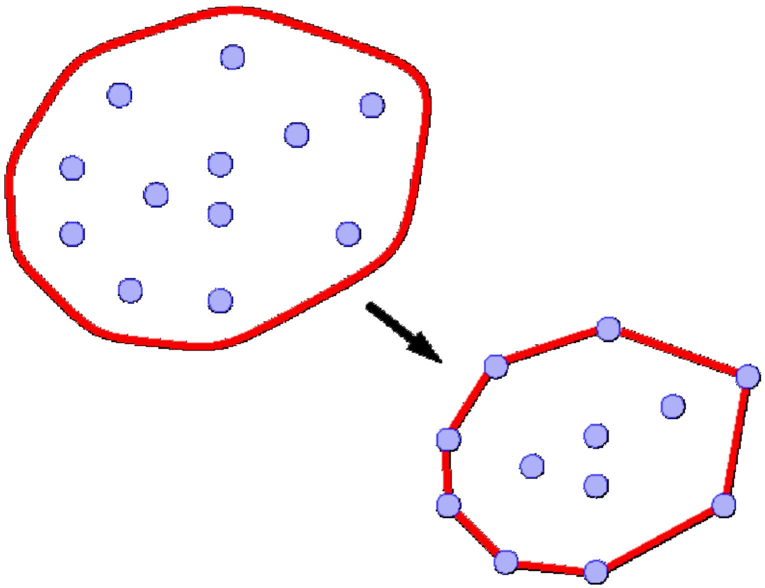
- Define events and their order.
  - If all the events can be determined in advance – *sort* the events.
  - Otherwise, use a *priority queue* to manage the events.
- Determine which operations have to be performed with the sweep-line status at each event point.
  - Left endpoint: add a new segment into the status.
  - Right endpoint: delete the corresponding new segment from the status.
- Choose a data-structure for the sweep-line status to efficiently support those operations.
  - A balanced binary tree for efficient predecessor and successor operations.

Agenda

- Computational geometry
- Basic geometric operations
- Sweeping techniques
- Graham's scan

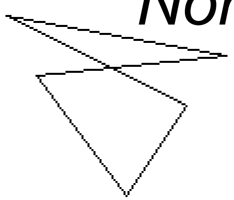
Finding the Convex Hull

- Let S be a set of n points in the plane. Compute the convex hull of these points.
- Intuition :
  - Each point in S is a nail sticking out from a board.
  - The convex hull is a tight rubber band that surrounds all the nails.

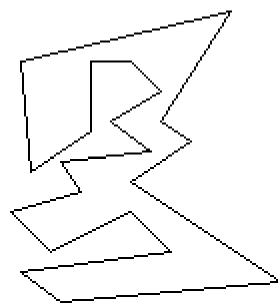


Convex hull

- Formal definition: the convex hull of S is the smallest convex polygon that contains all the points of S.
- *A polygon P is said to be* ***convex*** *if* :
  - *P* is *simple* (boundaries do not intersect in the middle but only at endpoints);
  - And, for any two points *p* and *q* on the boundary of *P* , segment *pq* lies entirely inside *P*



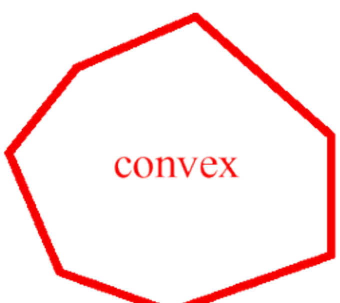
Non-simple



Simple

non convex

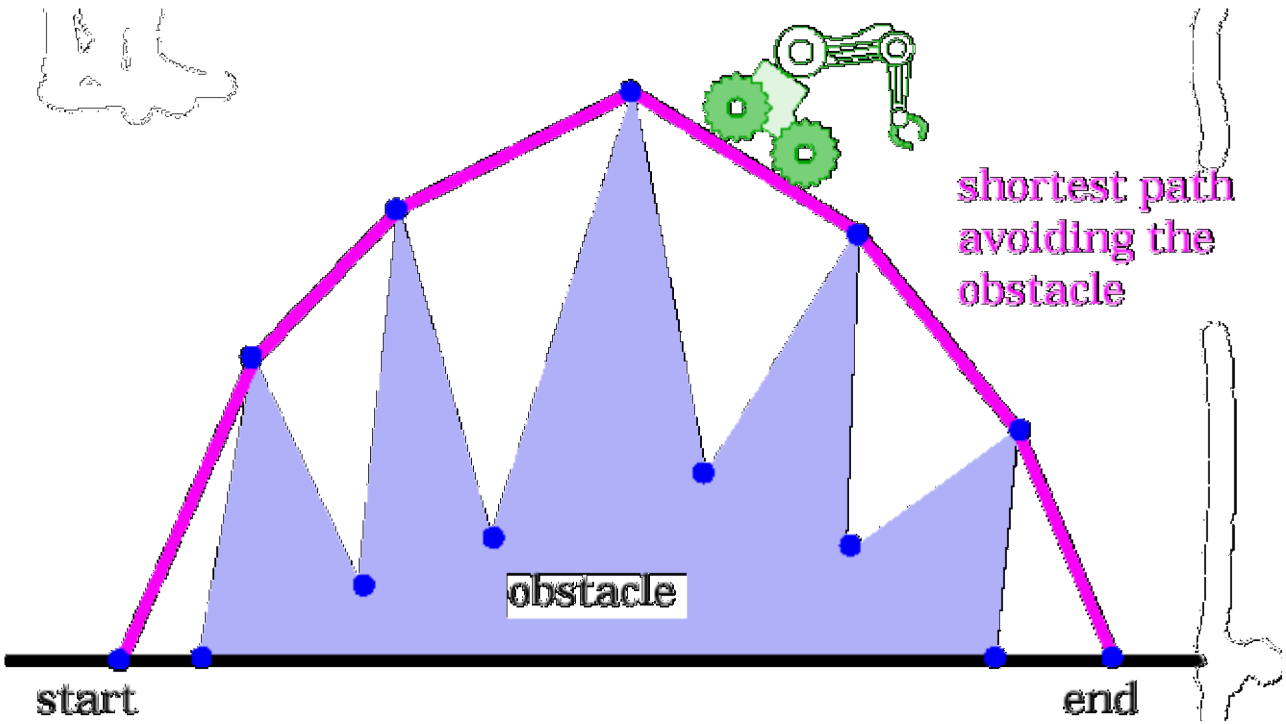
simple



simple

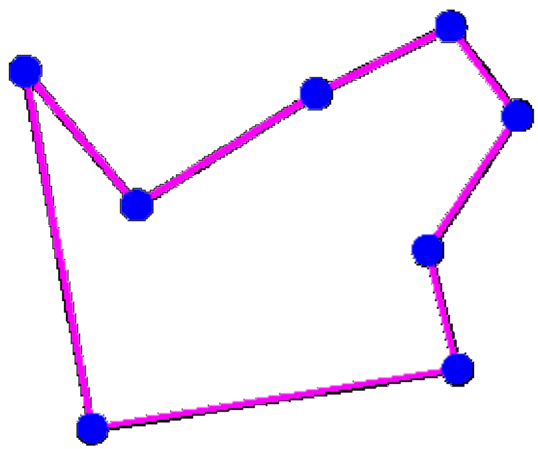
Robot motion planning

- In motion planning for robots, sometimes there is a need to compute convex hulls.



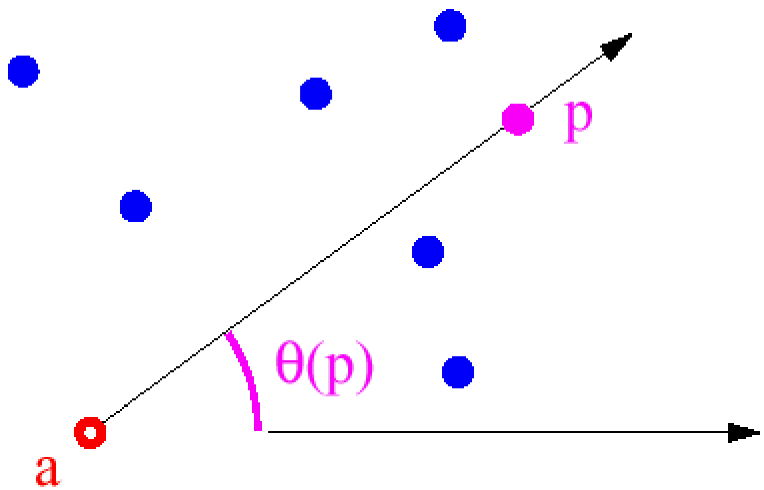
Graham scan

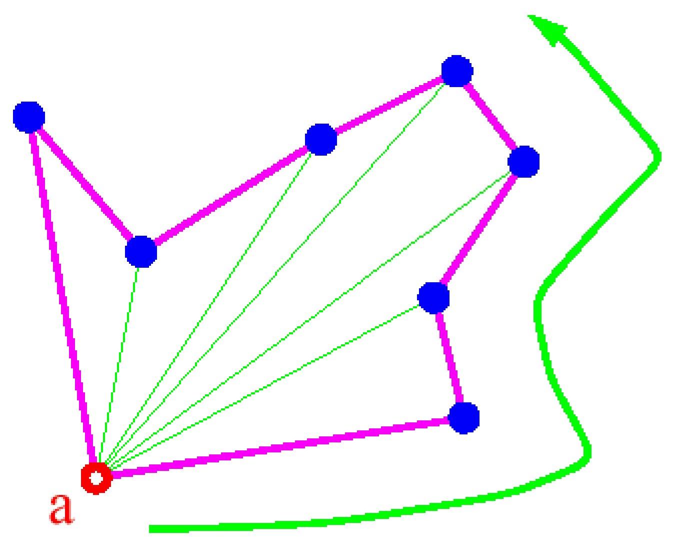
- *Phase 1:* Solve the problem of finding the simple (non-crossing) closed path visiting all points



Finding non-crossing path

- *How do we find such a non-crossing path:*
  - Pick the point ***a*** as the anchor point, where ***a*** has the minimum y-coordinate, or the leftmost such point in case of a tie.
  - For each point p, we have an angle θ (p) of the segment ap with respect to the *x*-axis (i.e., the polar-angle)
  - Traversing the points by **increasing angle** yields a simple closed path.

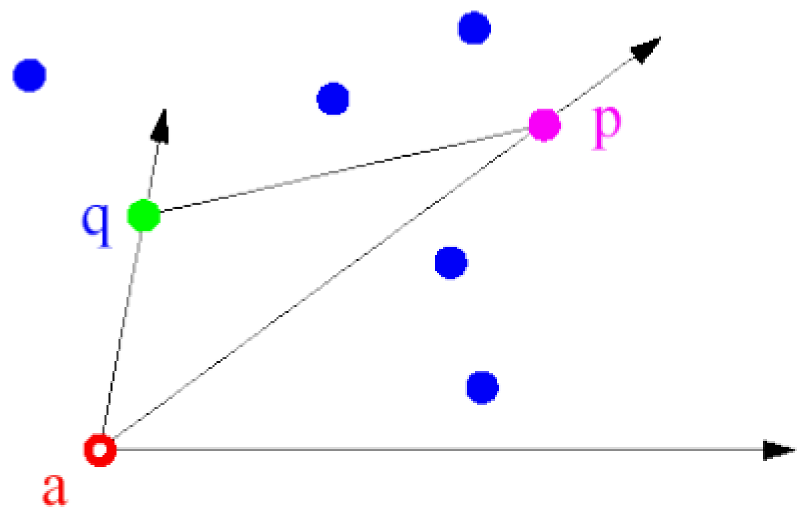




Sorting by angle

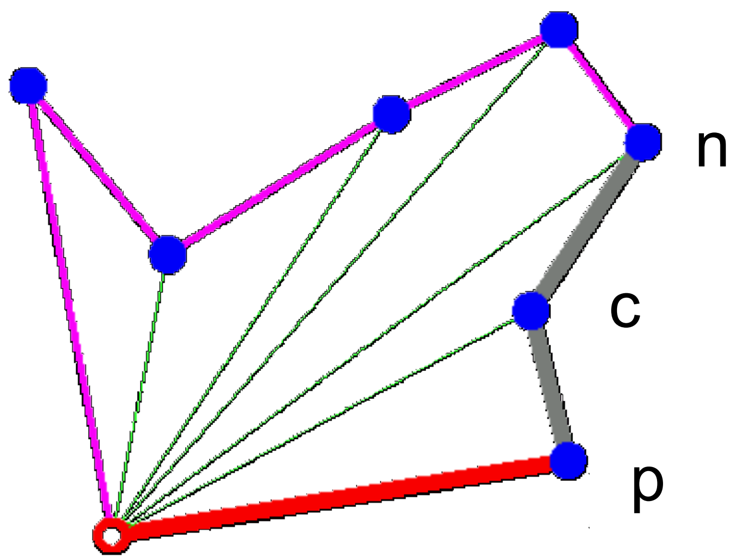
- *How do we sort by increasing angle*?
  - *Observation*: We do not need to compute the actual angle.
  - We just need to compare them for sorting

$\theta(p) < \theta(q)$
$\Leftrightarrow$ orientation(a,p,q) =
counterclockwise



Rotational sweeping

- *Phase 2 of Graham Scan:* ***Rotational sweeping***
- The anchor point and the first point in the polar-angle order have to be in the hull.
- Traverse the remaining points in the sorted order:
  - We denote the current point c, its previous point p, and its next point n.
  - If from segment pc to cn, we need to make a left turn, include c.
  - If not, discard *c* and consider its previous point as a new c.



Pseudo code

Phase 1: sorting, O(nlgn)

```
GRAHAM-SCAN(Q)
let p0 be the point in Q with the minimum y-coordinate,
        or the leftmost such point in case of a tie
let ⟨p1, p2, ..., pm⟩ be the remaining points in Q,
        sorted by polar angle in counterclockwise order around p0
        (if more than one point has the same angle, remove all but
        the one that is farthest from p0)
let S be an empty stack
PUSH(p0, S)
PUSH(p1, S)
PUSH(p2, S)
for i = 3 to m
        while the angle formed by points NEXT-TO-TOP(S), TOP(S),
                and pi makes a nonleft turn
            POP(S)
        PUSH(pi, S)
return S
```

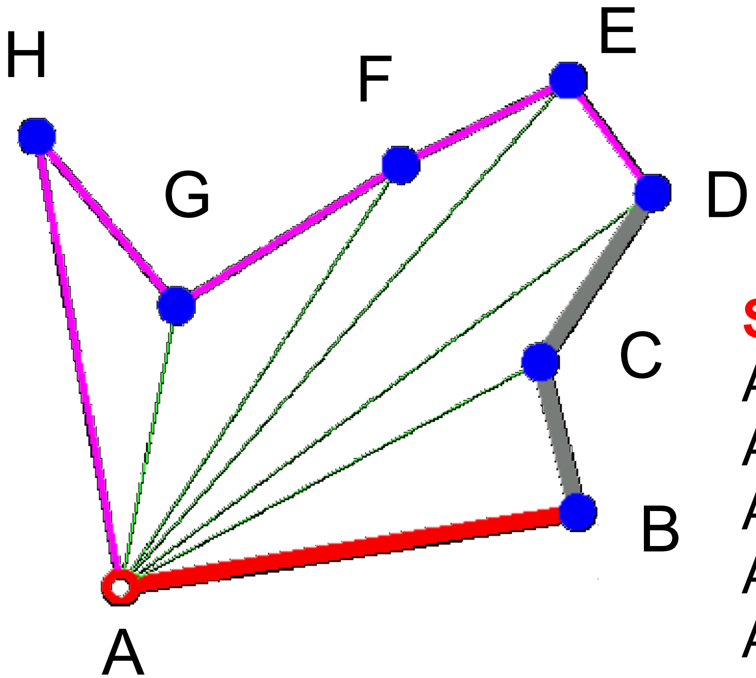
Previous point p

Current point c

Next point n

Phase 2:
Each point is inserted into and removed from the stack at most once. O(n)

In total: O(nlgn)



Stack	**Test**
A, B, C	B, C, D, right turn, pop
A, B	A, B, D, left turn, push(D)
A, B, D	B, D, E, left turn, push(E)
A, B, D, E	D, E, F, left turn, push(F)
A, B, D, E, F	E, F, G, left turn, push(G)
A, B, D, E, F, G	F, G, H, right turn, pop
A, B, D, E, F	E, F, H, right turn, pop
A, B, D, E	D, E, H, left turn, push(H)
A, B, D, E, H	done

A, B, D, E, H are the vertices on the convex hull.

ILO of Lecture 6

- Computational Geometry: sweeping techniques
  - to understand how the basic geometric operations (such as determining how two line segments are oriented and whether they intersect) are performed;
  - to understand the basic idea of the sweeping algorithm design technique;
  - to understand and be able to analyze the Graham's scan and the sweeping-line algorithm to determine whether any pair of line segments intersect.

Next lecture

- Computational Geometry Algorithms: Divide and Conquer