Algorithms and Satisfiability

Lecture 3
Computational Geometry Algorithms: Sweeping

DAT6 spring 2023

Simonas Šaltenis

AALBORG UNIVERSITY
DENMARK

Computational geometry

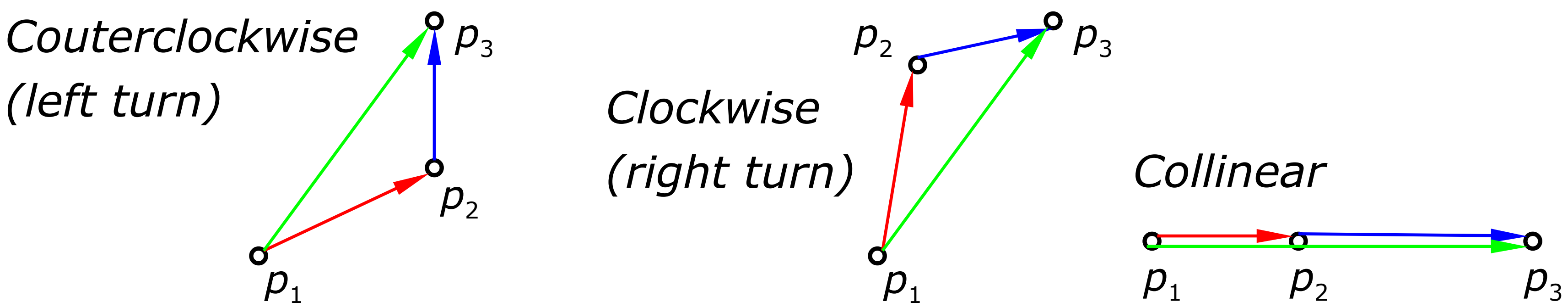
- Main goals of the lecture:
  - *to understand how the **basic geometric operations** are performed;*
  - *to understand the basic idea of the **sweeping** algorithm design **technique**;*
  - *to understand the concept of **output-sensitive algorithms**;*
  - *to understand and be able to analyze **Graham's scan, Jarvis's march,** and the sweeping-line algorithm to determine whether any pair of line segments intersect.*

Computational geometry

- *Computational geometry*:
  - Algorithmic basis for many scientific and engineering disciplines:
    - Geographic Information Systems (GIS)
    - Robotics
    - Computer graphics
    - Computer vision
    - Computer Aided Design/Manufacturing (CAD/CAM),
    - VLSI design, etc.
  - The term first appeared in the 70’s.
  - We will deal with *points* and *line segments* in 2D space.

Basic problems: Orientation

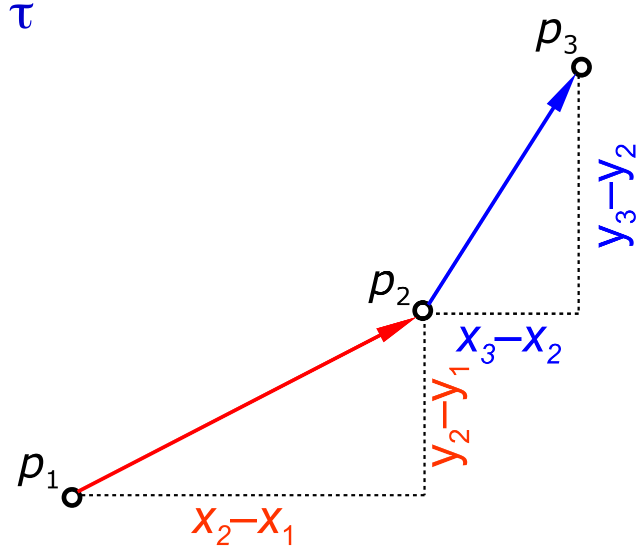
- *How to find "orientation" of two line segments?*
  - Three points: $p_1(x_1, y_1)$, $p_2(x_2, y_2)$, $p_3(x_3, y_3)$
  - *Is segment* (p_1, p_3) ***clockwise*** *or* ***counterclockwise*** *from* (p_1, p_2)?
  - Equivalent to: *Going from segment* (p_1, p_2) to (p_2, p_3) *do we make a* ***right***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



Computing the orientation

- *Orientation the standard way:*
  - slope of segment (p_1, p_2): $\sigma = (y_2-y_1)/(x_2-x_1)$
  - slope of segment (p_2, p_3): $\tau = (y_3-y_2)/(x_3-x_2)$

- How do you compute then the orientation?
  - counterclockwise (left turn): $\sigma < \tau$
  - clockwise (right turn): $\sigma > \tau$
  - collinear (no turn): $\sigma = \tau$



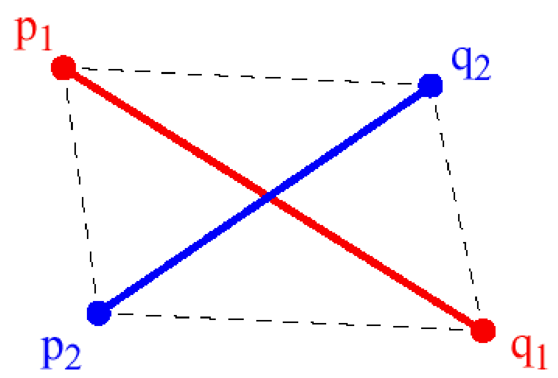
Cross product

- *Finding orientation without division (to avoid numerical problems)*
  - $(y_2-y_1)(x_3-x_2) - (y_3-y_2)(x_2-x_1) = ?$
    - Positive – clockwise / right turn
    - Negative – counterclockwise / left turn
    - Zero – collinear
  - This is (almost) a ***cross product*** of two vectors

$$(x_2-x_1, y_2-y_1)\times(x_3-x_2, y_3-y_2)=\det\begin{pmatrix} x_2-x_1 & x_3-x_2 \\ y_2-y_1 & y_3-y_2 \end{pmatrix}$$

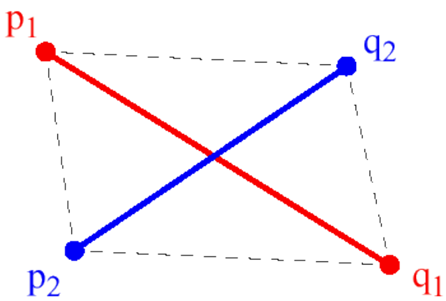
Intersection of two segments

- *How do we test whether two line segments intersect?*
  - *What would be the standard way?*
  - *What are the problems?*



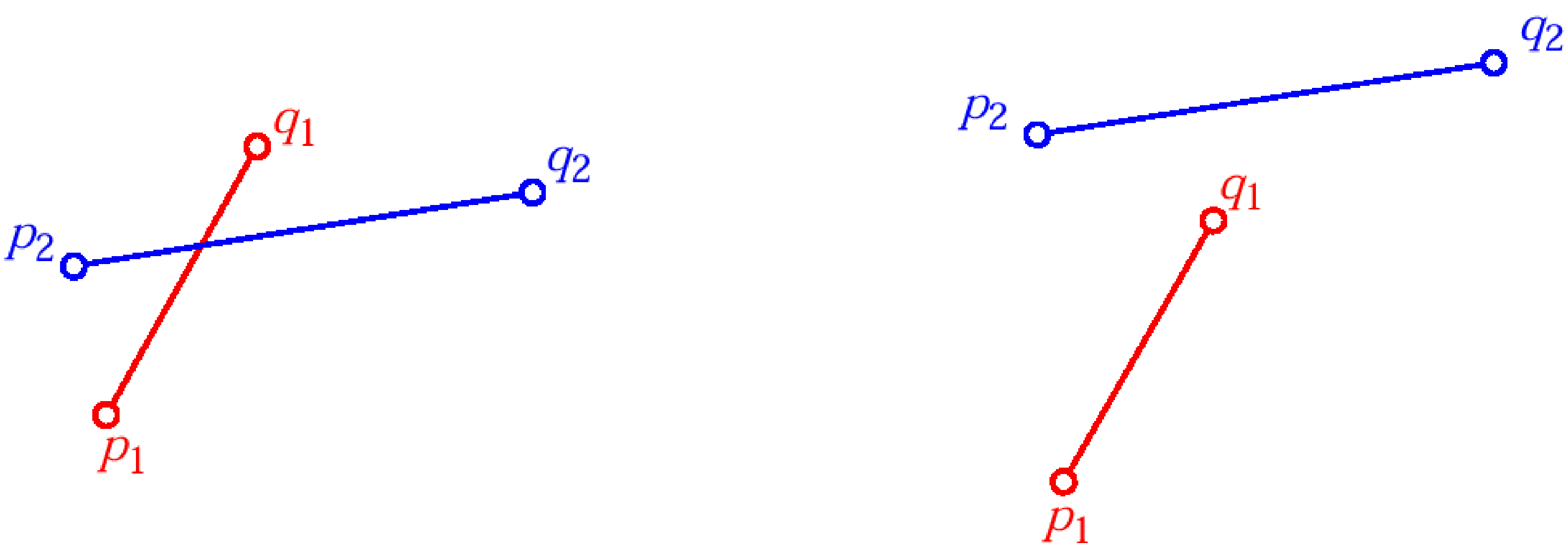
Intersection and orientation

- *We can use just cross products to check for intersection!*
  - Two segments (p_1,q_1) and (p_2,q_2) intersect *if and only if* one of the two is satisfied:
  - General case:
    - (p_1,q_1,p_2) and (p_1,q_1,q_2) have different orientations **and**
    - (p_2,q_2,p_1) and (p_2,q_2,q_1) have different orientations
  - Special case
    - (p_1,q_1,p_2), (p_1,q_1,q_2), (p_2,q_2,p_1), and (p_2,q_2,q_1) are all collinear **and**
    - the x-projections of (p_1,q_1) and (p_2,q_2) intersect
    - the y-projections of (p_1,q_1) and (p_2,q_2) intersect



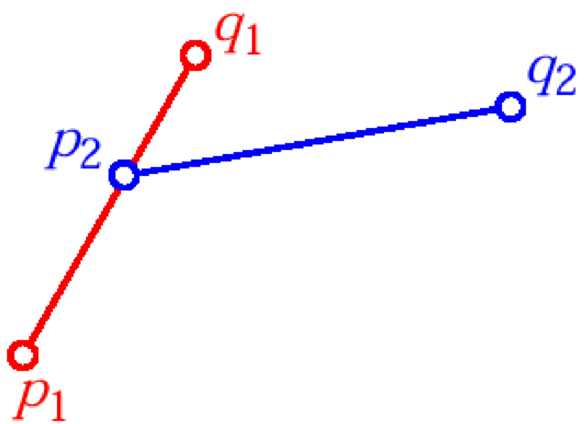
Orientation examples

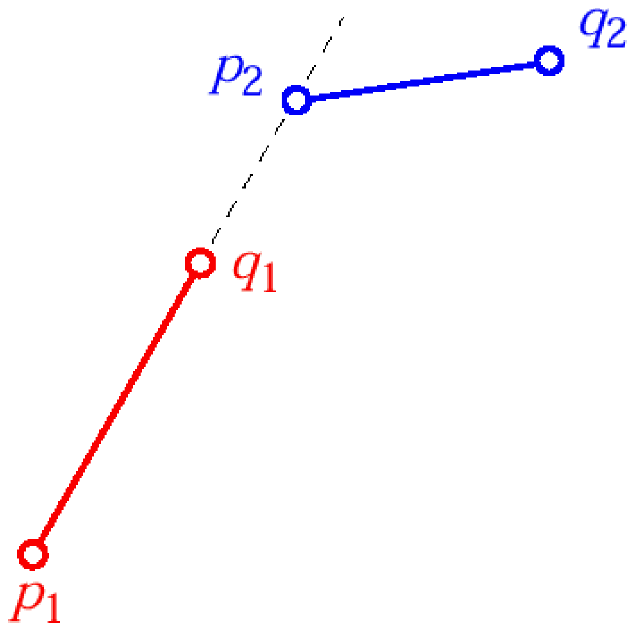
- General case:
  - (p_1,q_1,p_2) and (p_1,q_1,q_2) have different orientations **and**
  - (p_2,q_2,p_1) and (p_2,q_2,q_1) have different orientations



Orientation examples (2)

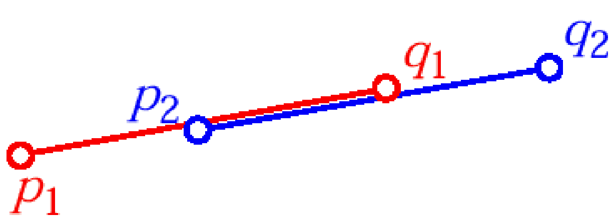
- General case:
  - (p_1,q_1,p_2) and (p_1,q_1,q_2) have different orientations **and**
  - (p_2,q_2,p_1) and (p_2,q_2,q_1) have different orientations

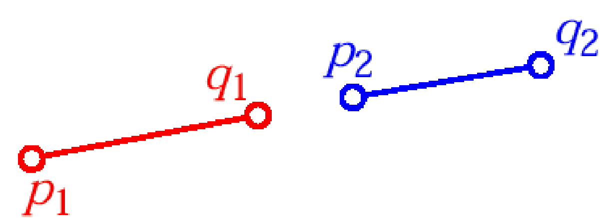




Orientation examples (3)

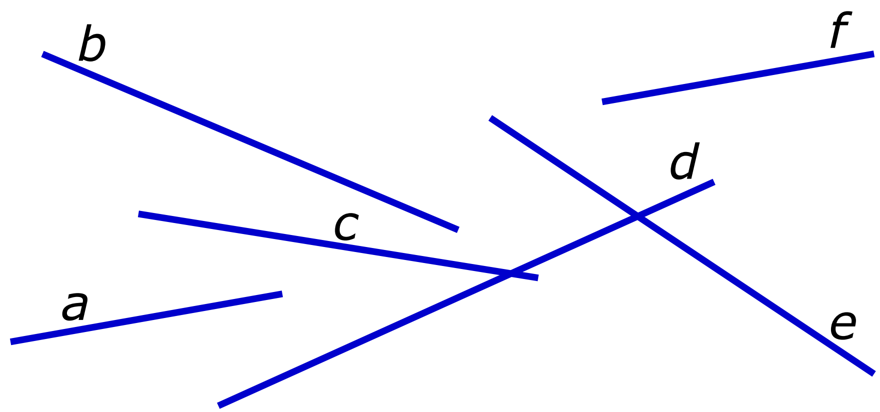
- Special case
  - (p_1,q_1,p_2), (p_1,q_1,q_2), (p_2,q_2,p_1), and (p_2,q_2,q_1) are all collinear **and**
  - the x-projections of (p_1,q_1) and (p_2,q_2) intersect
  - the y-projections of (p_1,q_1) and (p_2,q_2) intersect





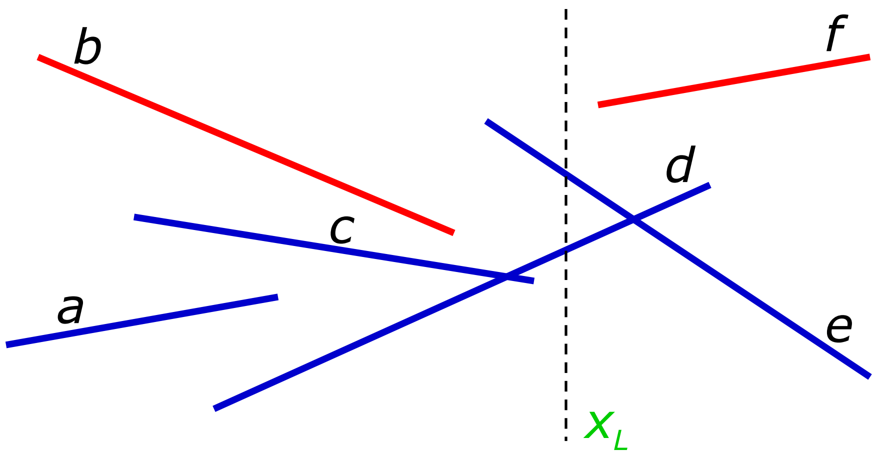
Determining intersections

- *Given a set of n segments, determine whether any two line segments intersect*
  - Note: not asking to report all intersections, just true or false.
  - Usefull as a building block: e.g., check intersection of arbitrary polygons.
  - *What would be the brute force algorithm and what is its worst-case complexity?*



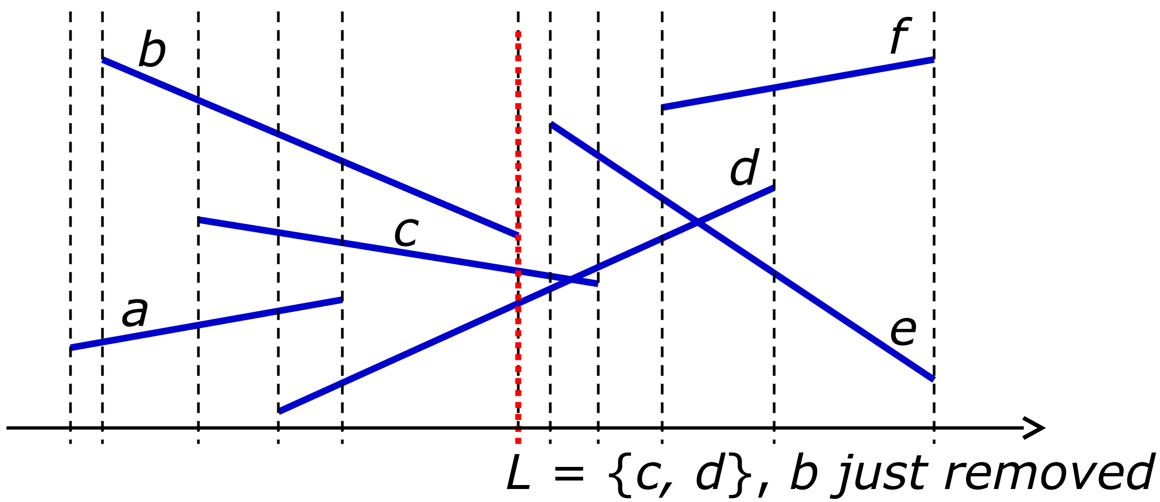
Observations

- *Helpful observation*:
  - *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 line $x = x_L$ intersects both segments*



Sweeping technique

- *A powerful algorithm design technique:* ***sweeping***.
  - Two sets of data are maintained:
    - ***sweep-line status***: the set of segments intersecting the sweep line *L*
    - ***event-point schedule***: where updates to *L* are required

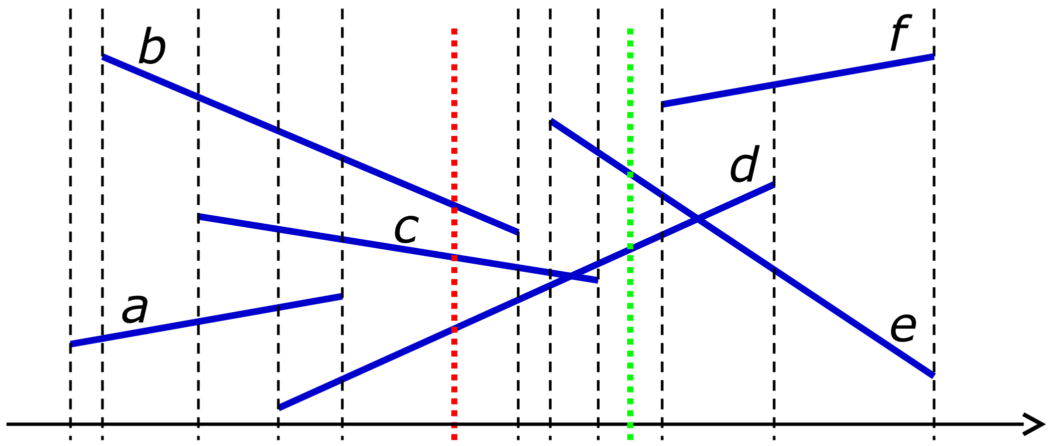


Plane-sweeping algorithm

- Skeleton of the algorithm:
  - Each segment end point is an event point
  - 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 of L and it is tested for intersection against the other segments in the status
    - right end point: it is deleted from the status of L
- Analysis:
  - *What is the worst-case complexity?*
  - *Worst-case example?*

Improving the algorithm

- More useful observations:
  - *For a specific position of the sweep line, there is* ***an order*** *of segments in the y-axis;*
  - *If segments intersect – there is a position of the sweep-line such that two segments are* ***adjacent*** *in this order;*
  - *Order does not change in-between event points;*
    - True only to the left of the leftmost intersection point. If the algorithm does not return before, this intersection is detected!
  - *Main idea: check only all* ***new pairs of neighbors*** *in the order!*



Sweep-line status DS

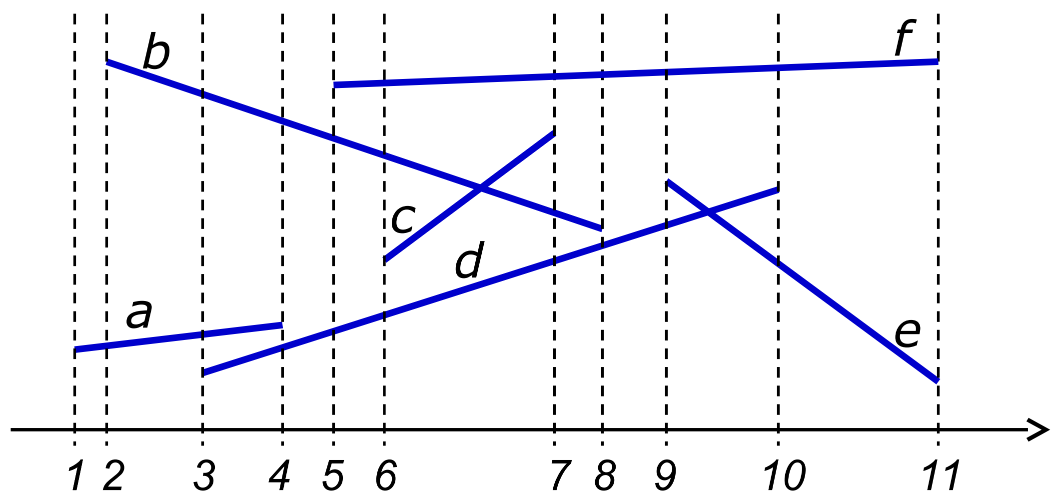
- *Sweep-line status data structure:*
  - Operations:
    - Insert
    - Delete
    - Below (Predecessor)
    - Above (Successor)
  - Balanced binary search tree *T* (e.g., Red-Black)
    - The bottom-to-top order of segments on the line *L* ⇔ the left-to-right order of in-order traversal of *T*
  - *How do you do comparison?*

Algorithm

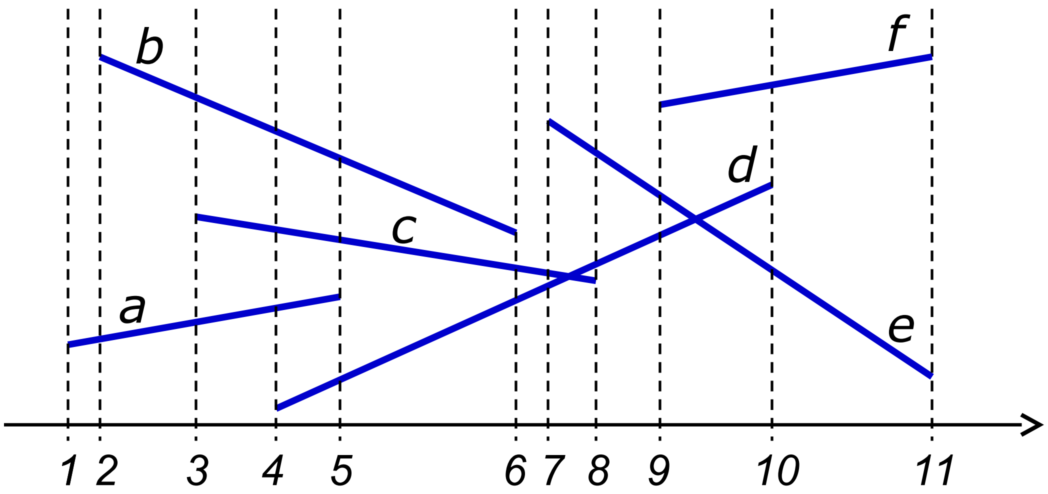
```
AnySegmentsIntersect(S)
T ← ∅
sort the left and right endpoints of the segments in
  S from left  to right, breaking ties by putting left
  endpoints first
for each point p in the sorted list of endpoints do
    if p is the left endpoint of a segment s then
        Insert(T,s)
        if (Above(T,s) exists and intersects s) or
           (Below(T,s) exists and intersects s) then
            return TRUE
    if p is the right endpoint of a segment s then
        if both Above(T,s) and Below(T,s) exist and
           Above(T,s) intersects Below(T,s) then
            return TRUE
        Delete(T,s)
return FALSE
```

Let’s run it

```
AnySegmentsIntersect(S)
T ← ⊘
sort the left and right endpoints of the segments in S from
left  to right, breaking ties by putting left endpoints
first
for each point p in the sorted list of endpoints do
   if p is the left endpoint of a segment s then
      Insert(T,s)
       if (Above(T,s) exists and intersects s) or
      (Below(T,s) exists and intersects s) then
           return TRUE
   if p is the right endpoint of a segment s then
       if both Above(T,s) and Below(T,s) exist and
          Above(T,s) intersects Below(T,s) then
           return TRUE
         Delete(T,s)
return FALSE
```



Example



- *Which intersection checks are done in each step?*
- *At which event an intersection is discovered?*
- *Go to Socrative and write in your answer:*
  - *E.g., "3, 4" means 3 intersection checks, at the 4th event.*
- *What if sweeping is from right to left?*

Analysis, Special cases

- Running time:
  - Sorting the segment endpoints: $\Theta(n \lg n)$
  - The loop is executed once for every end point ($2n$) taking each time $\Theta(\lg n)$ (at most three red-black tree operations)
  - The total running time is $\Theta(n \lg n)$

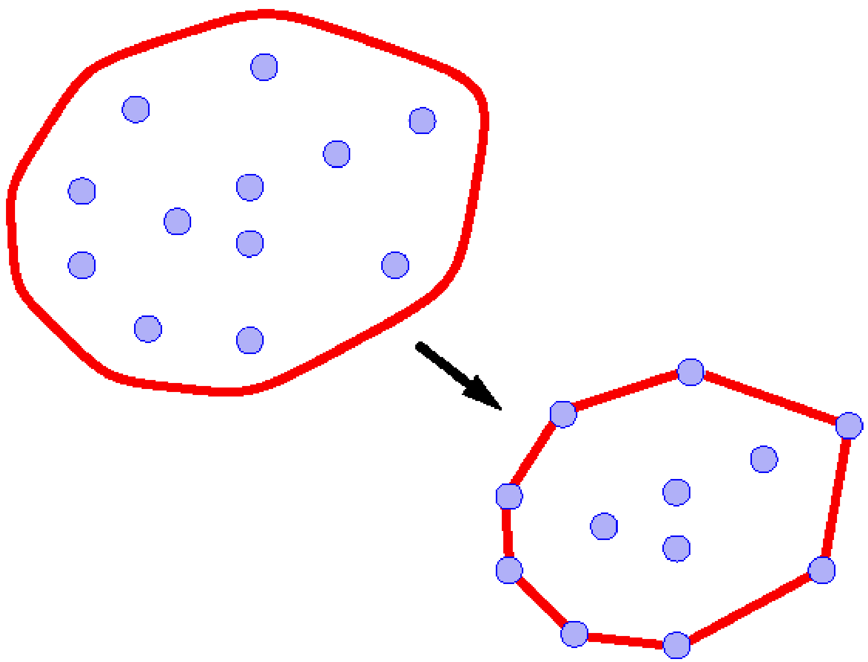
- Special cases (correctness not obvious, have to prove separately):
  - More than two segments intersect at one point
    - Can be shown to work just fine
  - There are some vertical segments
    - Can be proven to work correctly if bottom endpoints are treated as left endpoint (processed first in the event sequence)

Sweeping technique principles

- *Principles of the sweeping technique*:
  - Define events and their order
  - If all the events can be determined in advance – *sort* the events
  - Else use a *priority queue* to manage the events
  - See which operations have to be performed with the sweep-line status at each event point
  - Choose a data-structure for the sweep-line status to efficiently support those operations

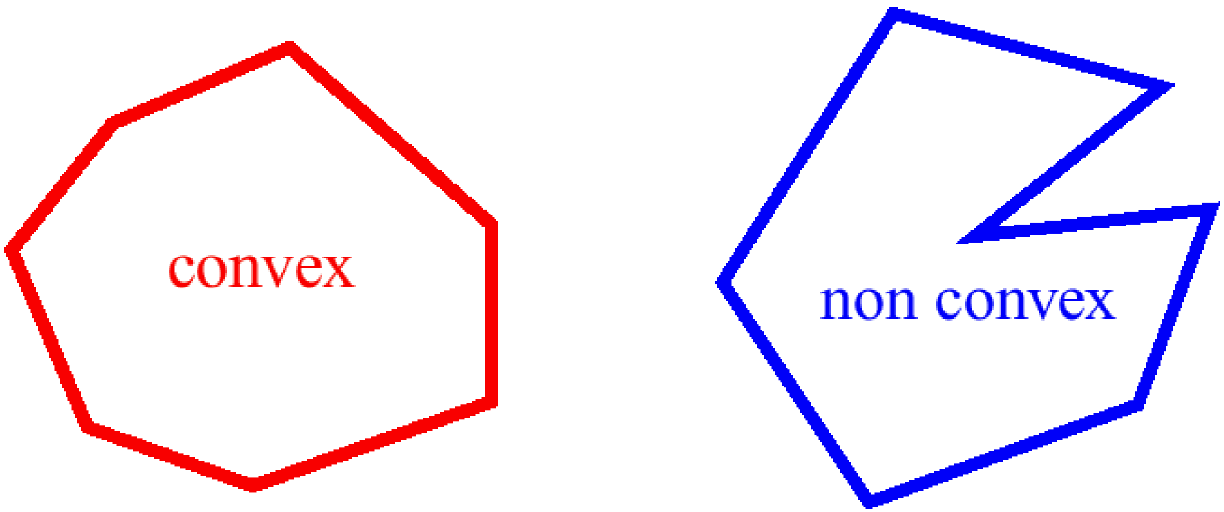
Convex hull problem

- *Convex hull problem:*
  - Let *S* be a set of *n* points in the plane. Compute the convex hull of these points.
  - *Intuition*: rubber band stretched around the pegs
  - *Formal definition:* the ***convex hull*** of *S* is the smallest convex polygon that contains all the points of *S*



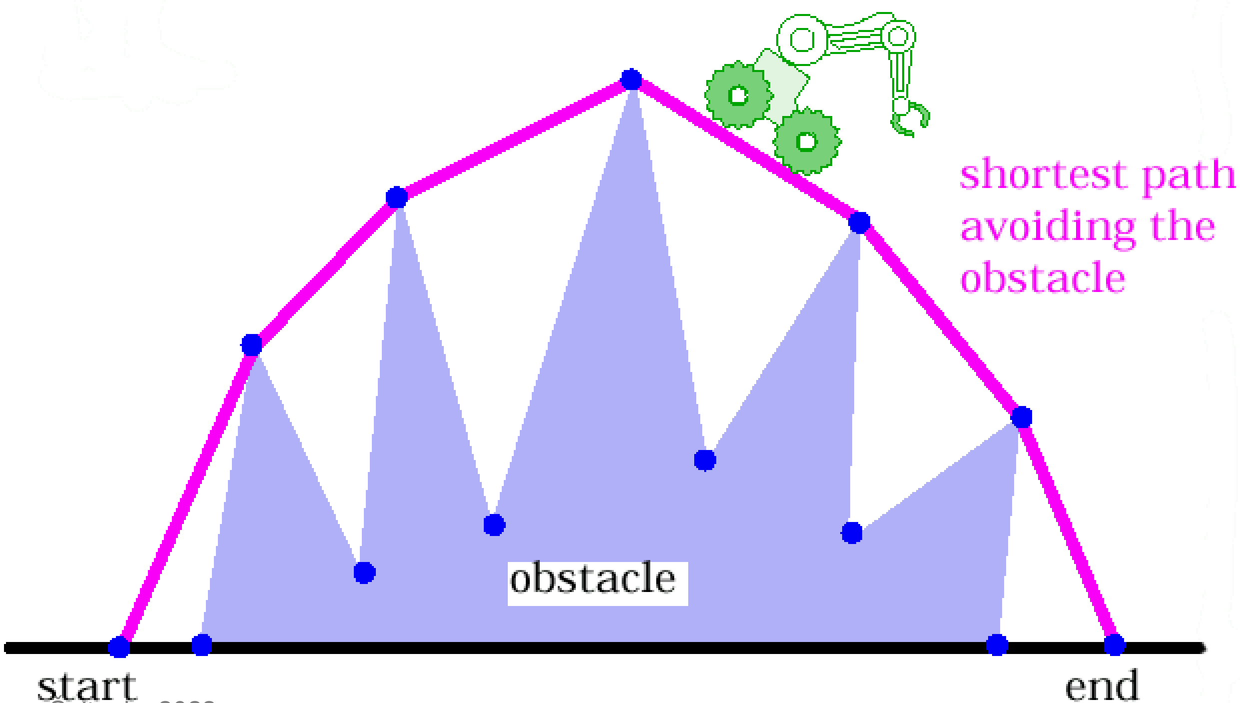
What is convex

- *A polygon P is said to be* ***convex*** *if*:
  - *P* is *simple* (non-intersecting); and
  - for any two points *p* and *q* on the boundary of *P*, segment (*p,q*) lies entirely inside *P*



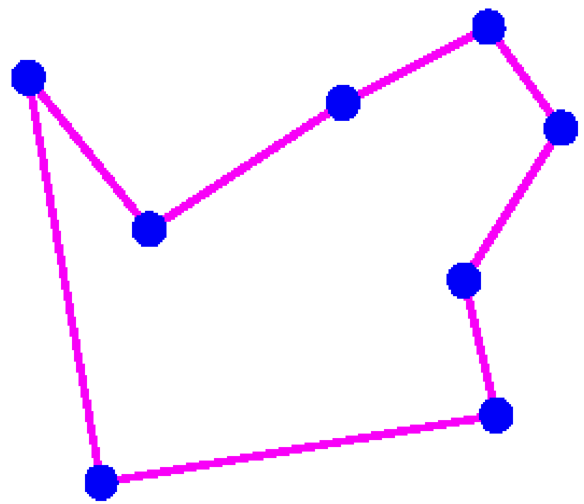
Many applications, many algs

- In motion planning for robots, often there is a need to compute convex hulls.
- Convex hulls are often useful as a starting point in comp. geometry
  - For example, *furthest-pair problem.*



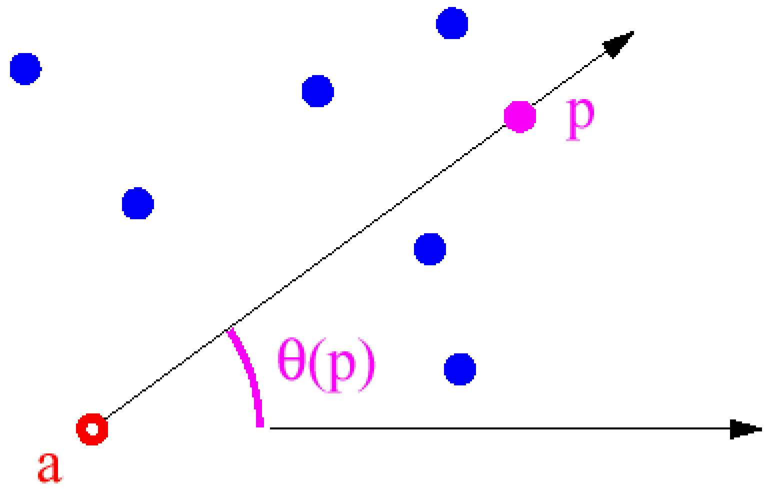
Graham Scan

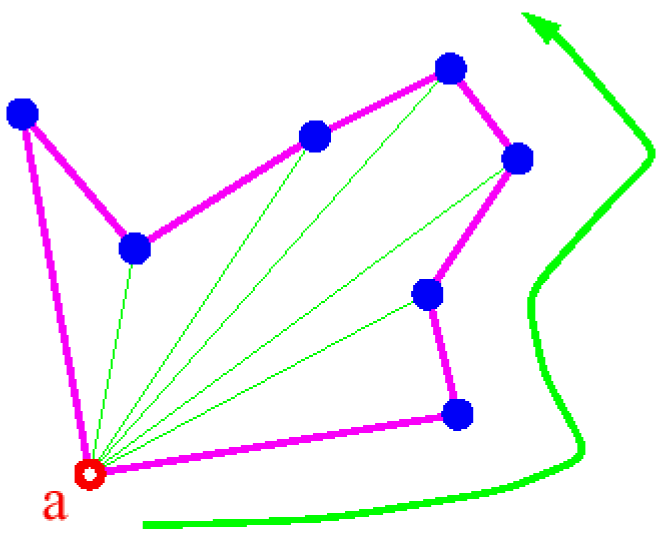
- ***Graham Scan** algorithm.*
  - *Phase 1:* Solve the problem of finding the simple (non-crossing) closed path visiting all points



Finding non-crossing path

- *How do we find such a non-crossing path:*
  - Pick the bottommost point *a* as the anchor point
  - For each point p, compute the angle θ(p) of the segment (a,p) with respect to the *x*-axis.
  - Traversing the points by **increasing angle** yields a simple closed path

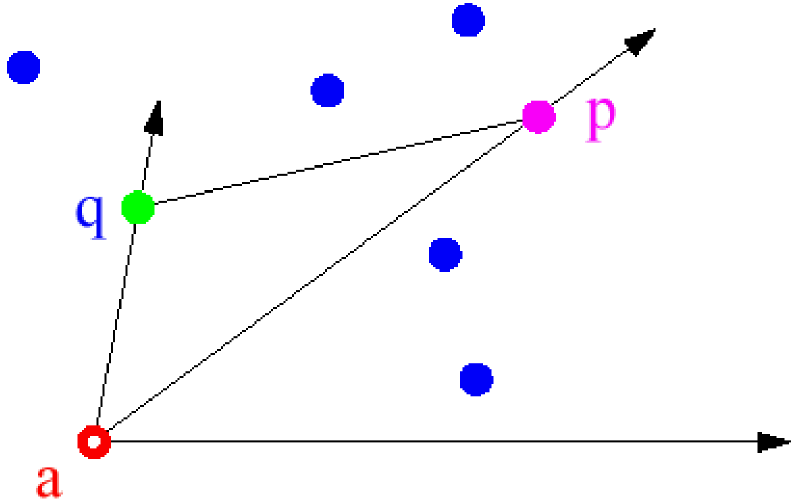




Sorting by angle

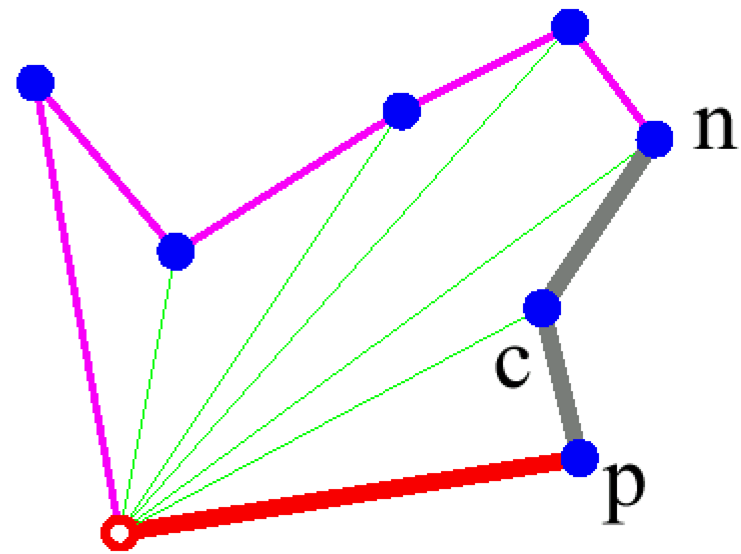
- *How do we sort by increasing angle*?
  - *Observation*: We do not need to compute the actual angle!
  - We just need to compare them for sorting

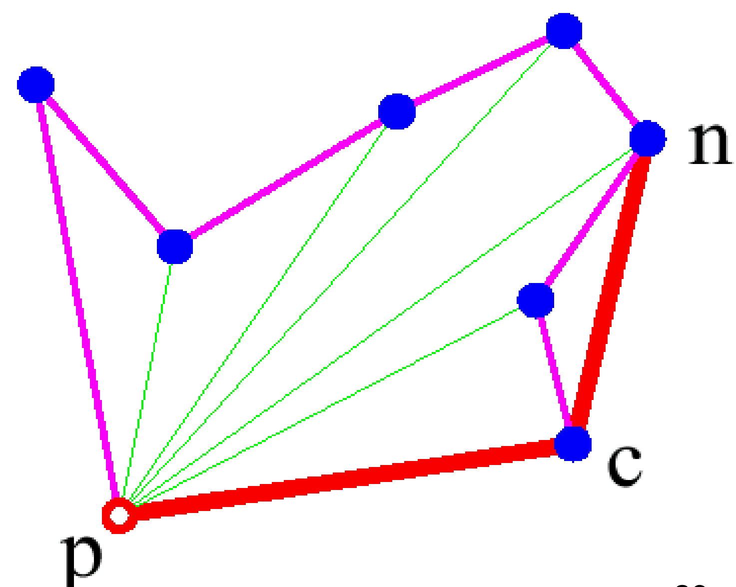
θ(*p*) < θ(*q*)
⇔ orientation(a,p,q) =
counterclockwise



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 points in the sorted order:
    - Before including the next point *n* check if the new added segment makes a left turn
    - If not, keep discarding the previous point (*c*) until a left turn is made





Implementation

- *Implementation:*
  - *Stack to store the vertices of the convex hull*

```
GrahamScan(S)
let p0 be the bottommost point in S
sort p1, p2, …, pn according to the angle of p0pi.
s.push(p0)
s.push(p1)
s.push(p2)
for i ← 3 to n
   while (s.nexttop, s.top, pi) is a non-left turn do
      s.pop
   s.push(pi)
return s
```

Analysis

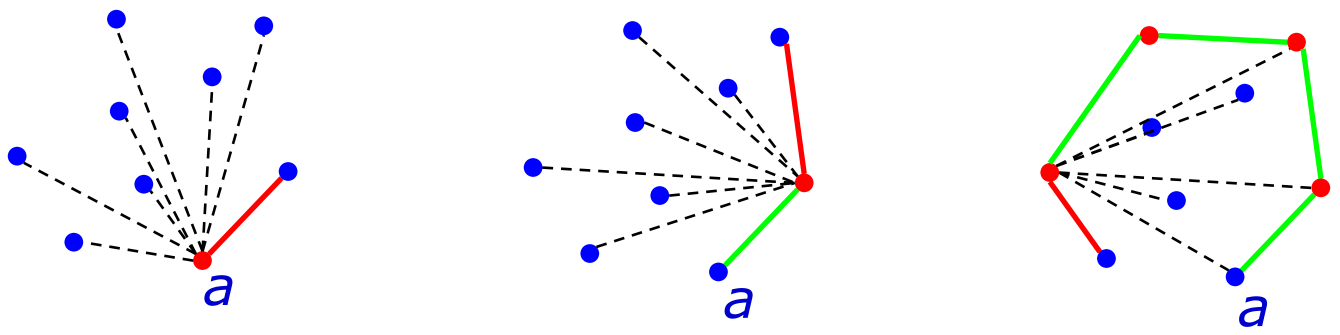
- *Analysis*:
  - Phase 1: $\Theta(n \log n)$
    - the anchor point is found: $\Theta(n)$
    - points are sorted by the angle around the anchor
  - Phase 2: $\Theta(n)$
    - each point is pushed into the stack once
    - each point is removed from the stack at most once
  - Total time complexity $\Theta(n \log n)$

Size of the output

- *In computational geometry, the size of an algorithm's **output** may differ/depend on the **input**.*
  - Line-intersection problem vs. convex-hull problem
  - *Observation*: Graham's scan running time depends only on the size of the **input** – it is independent of the size of the **output**

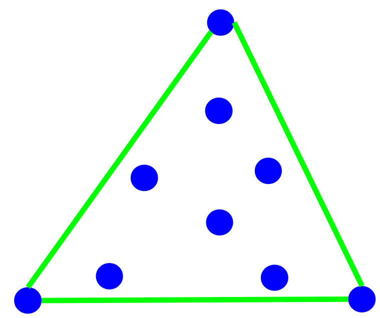
Gift wrapping

- *Would be nice to have an algorithm that runs fast if the convex hull is small*
  - Idea: ***gift wrapping*** (a.k.a **Jarvis's march**)
    - 1. Start with the lowest point *a.*
    - 2. The **next** point in the convex hull has to be in the clockwise direction with respect to all the remaining points looking from the **current** point on the convex hull
    - 3. Repeat 2. until *a* is reached. Include *a* in the convex hull



Jarvis's march

- *How many cross products are computed for this example?*



- The running time of Jarvis's march:
  - Find lowest point: $O(n)$
  - For each vertex in the convex hull: n–2 cross-product computations
  - Total: $\boldsymbol{O(nh)}$, where h is the number of vertices in the convex hull

Output-sensitive algorithms

- ***Output-sensitive** algorithm: its running time depends on the size of the output.*
  - *When should we use Jarvi's march instead of the Graham's scan*?
  - The asymptotically optimal output-sensitive algorithm of Kirkpatrick and Seidel runs in $O(n \lg h)$