# *Advanced Algorithm Design and Analysis (Lecture 9)*

SW5 fall 2004

*Simonas Šaltenis*
*E1-215b*
*simas@cs.aau.dk*

# Computational geometry

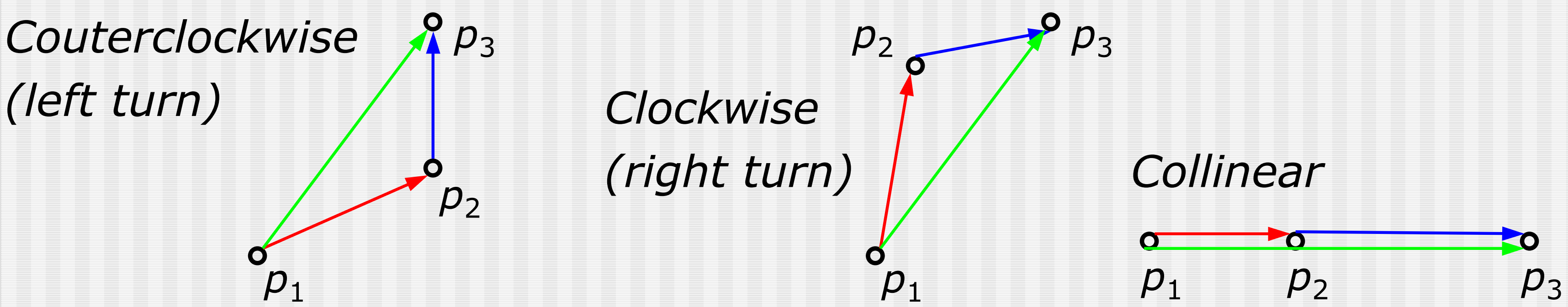
- Main goals of the lecture:
  - *to understand how the **basic geometric operations** are performed;*
  - *to understand the basic idea of the **sweeping** algorithm design **technique**;*
  - *to understand and be able to analyze the **Graham's scan** and the sweeping-line algorithm to determine whether 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

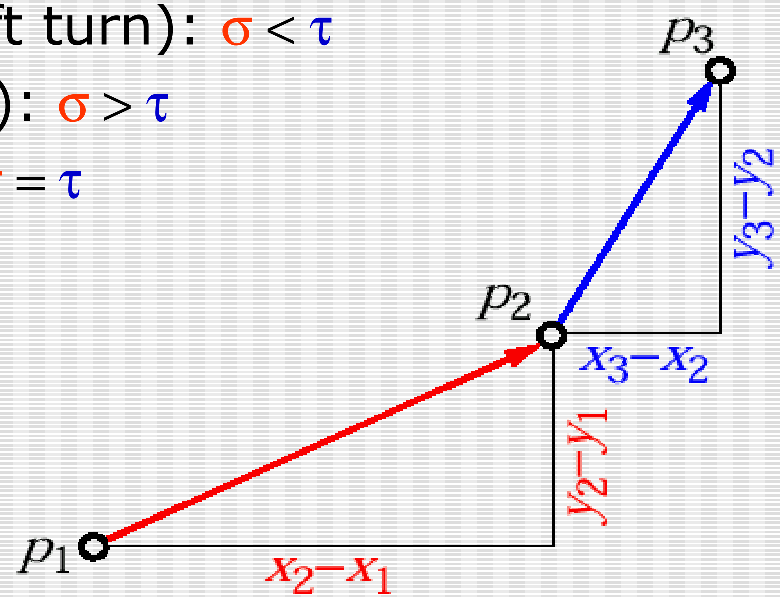
- *How to find "orientation" of two line segments?*
  - Three points: $p_1(x_1, y_1)$, $p_2(x_2, y_2)$, $p_3(x_3, y_3)$
  - *Is segment* ($p_1$, $p_3$) ***clockwise*** *or* ***counterclockwise*** *from* ($p_1$, $p_2$)?
  - Equivalent to: *Going from segment* ($p_1$, $p_2$) to ($p_2$, $p_3$) *do we make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



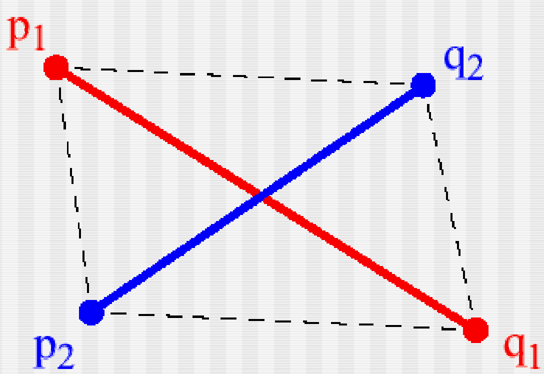
# Cross product

- *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
    - Negative – counterclockwise
    - Zero – collinear
  - This is (almost) a ***cross product*** of two vectors

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

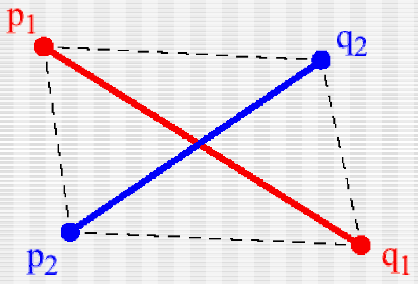
# Intersection of two segments

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



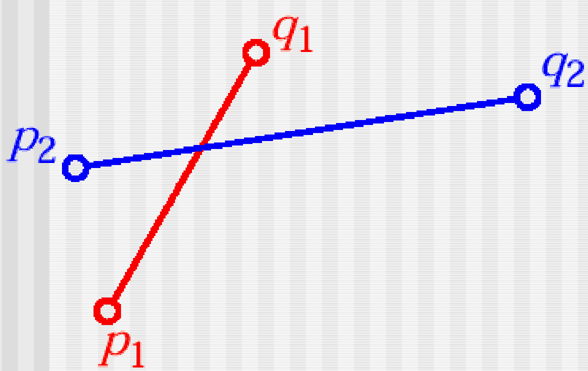
# Intersection and orientation

- *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

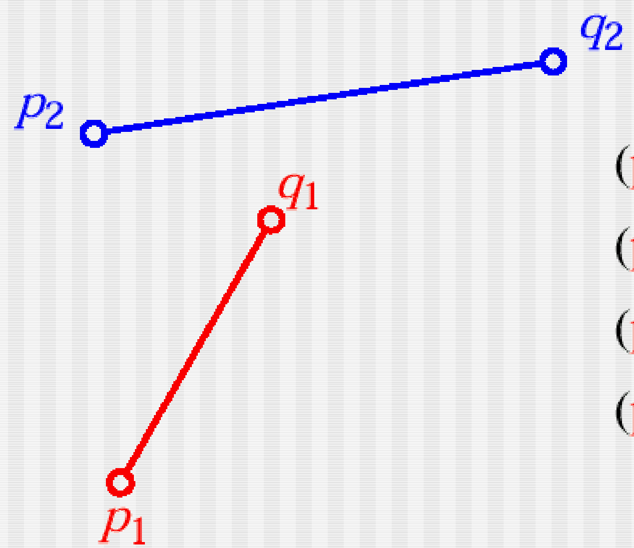



($p_1$,$q_1$,$p_2$)

($p_1$,$q_1$,$q_2$)

($p_2$,$q_2$,$p_1$)

($p_2$,$q_2$,$q_1$)



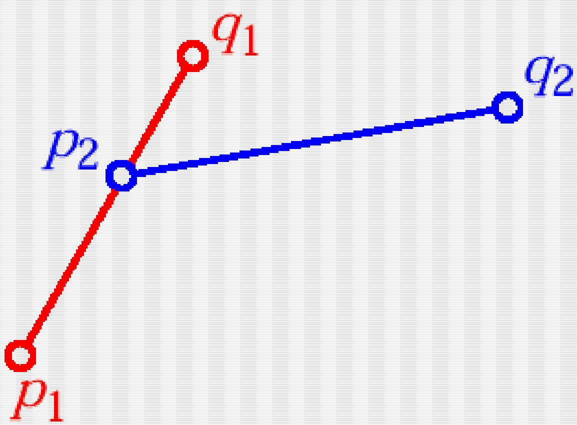

($p_1$,$q_1$,$p_2$)

($p_1$,$q_1$,$q_2$)

($p_2$,$q_2$,$p_1$)

($p_2$,$q_2$,$q_1$)

# Orientation Examples (2)

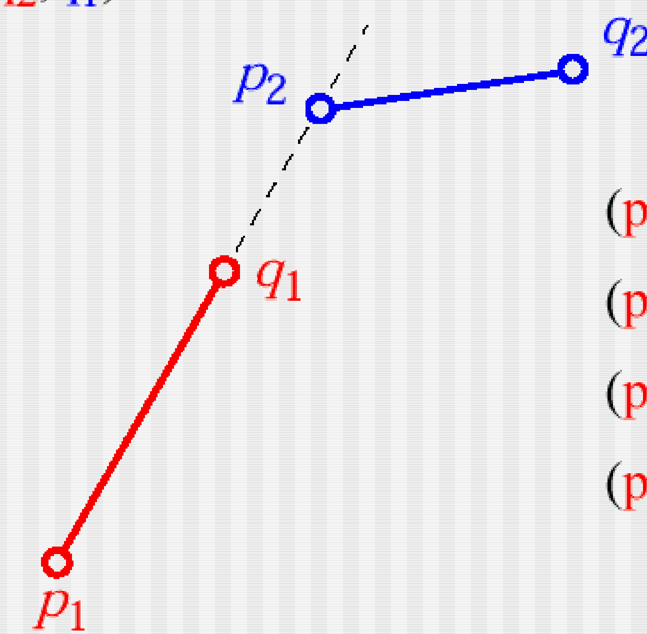



$(p_1, q_1, p_2)$

$(p_1, q_1, q_2)$

$(p_2, q_2, p_1)$

$(p_2, q_2, q_1)$




$(p_1, q_1, p_2)$

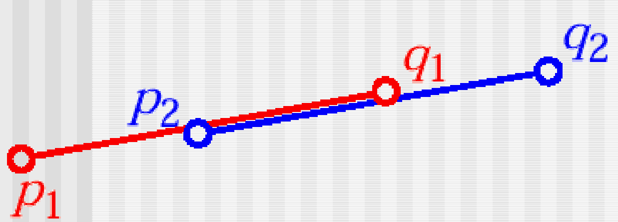
$(p_1, q_1, q_2)$

$(p_2, q_2, p_1)$

$(p_2, q_2, q_1)$

# 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

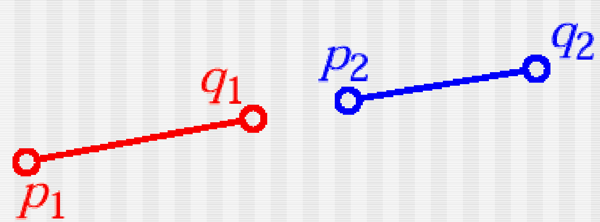



$(p_1,q_1,p_2)$

$(p_1,q_1,q_2)$

$(p_2,q_2,p_1)$

$(p_2,q_2,q_1)$




$(p_1,q_1,p_2)$

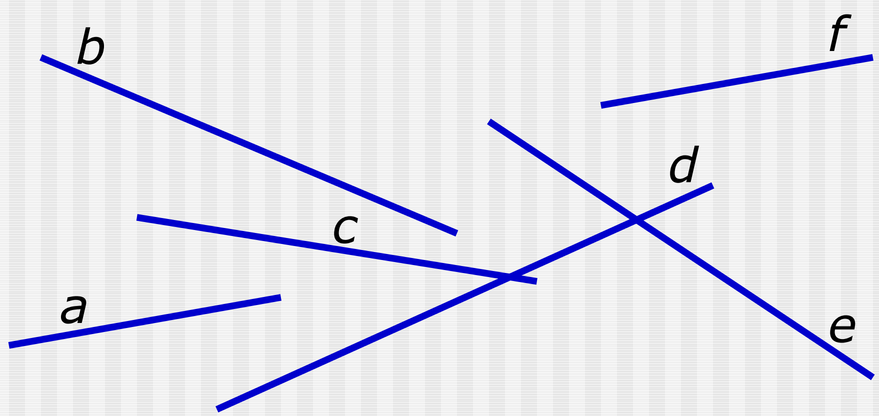
$(p_1,q_1,q_2)$

$(p_2,q_2,p_1)$

$(p_2,q_2,q_1)$

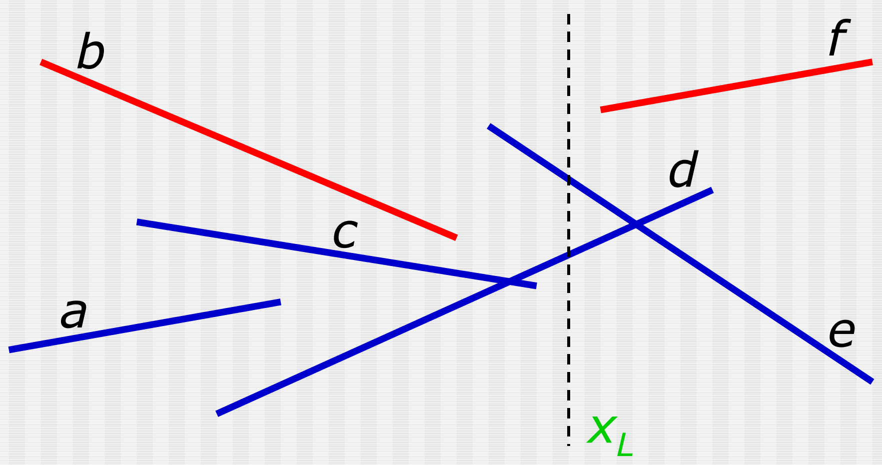
# Determining Intersections

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



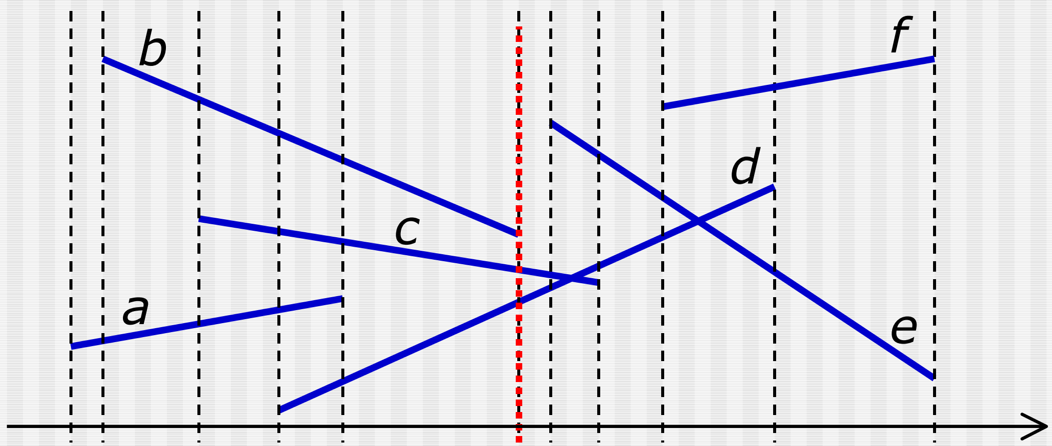
# Observations

- *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



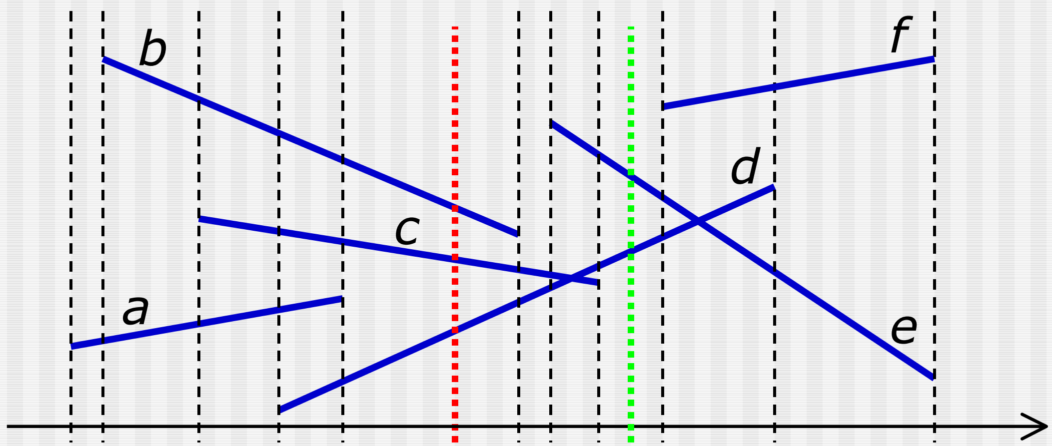

*L* = {*c*, *d*}, *b just removed*

# *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's tested against the rest
    - right end point: it's deleted from the status of *L*
- Analysis:
  - *What is the worst-case comlexity*?
  - *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*



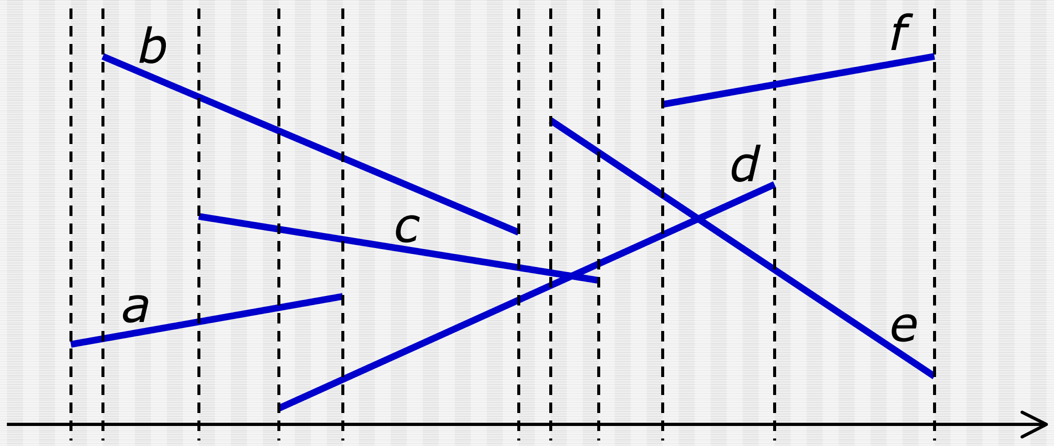
# Sweep-line status DS

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

# Pseudo Code

```
AnySegmentsIntersect(S)
T ← ∅
sort the left and right end points of the segments
  in S from left  to right, breaking ties by putting
  left end points first
for each point p in the sorted list of end points do
   if p is the left end point 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 end point 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 comparisons are done in each step?*
- *At which event the intersection is discovered? What if sweeping is from right to left?*

# Analysis, Assumptions

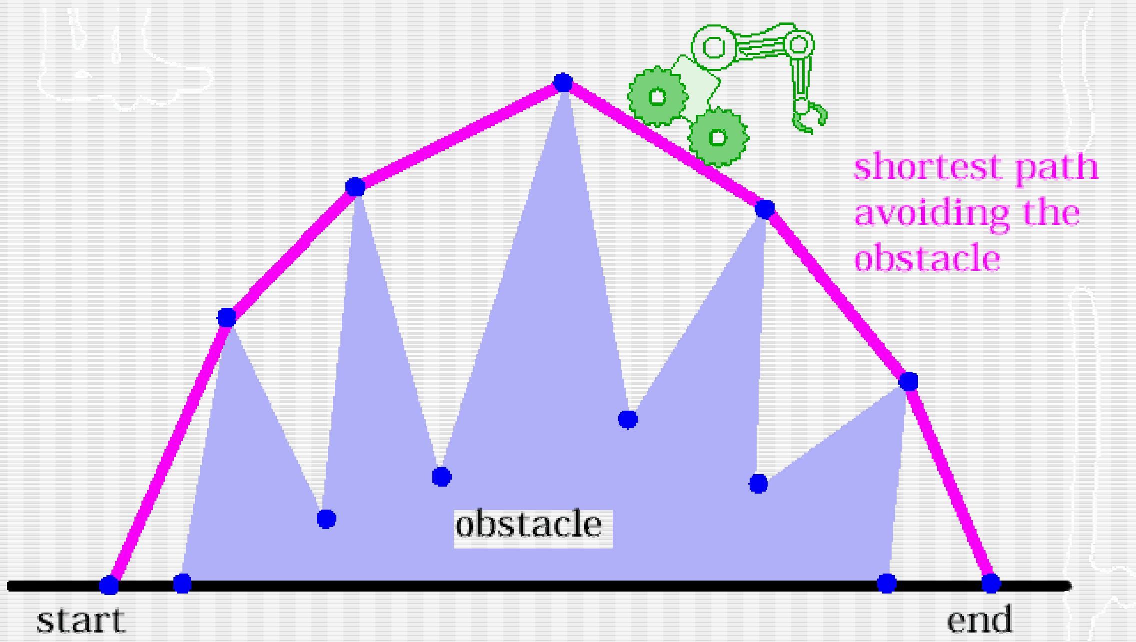
- *Running time*:
  - Sorting the segments: O($n$ log $n$)
  - The loop is executed once for every end point (2$n$) taking each time O(log $n$) (e.g., red-black tree operation)
  - The total running time is O($n$ log $n$)
- *Simplifying assumptions*:
  - At most two segments intersect at one point
  - No vertical segments

# Sweeping technique principles

- *Principles of sweeping technique*:
  - Define events and their order
  - If all the events can be determined in advance – sort the events
  - Else use the *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

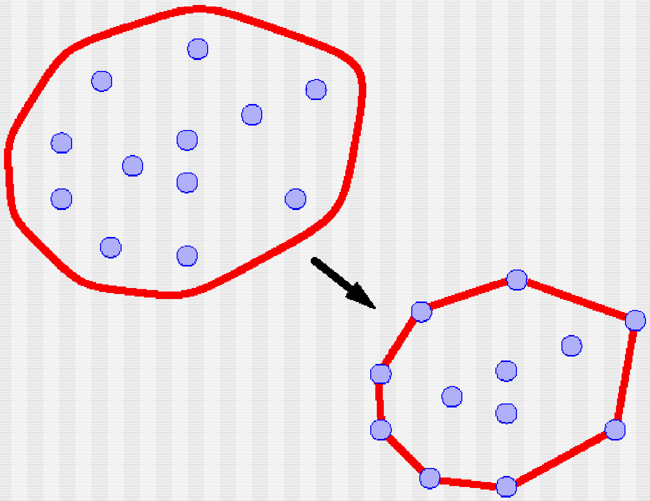
# Robot motion planning

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



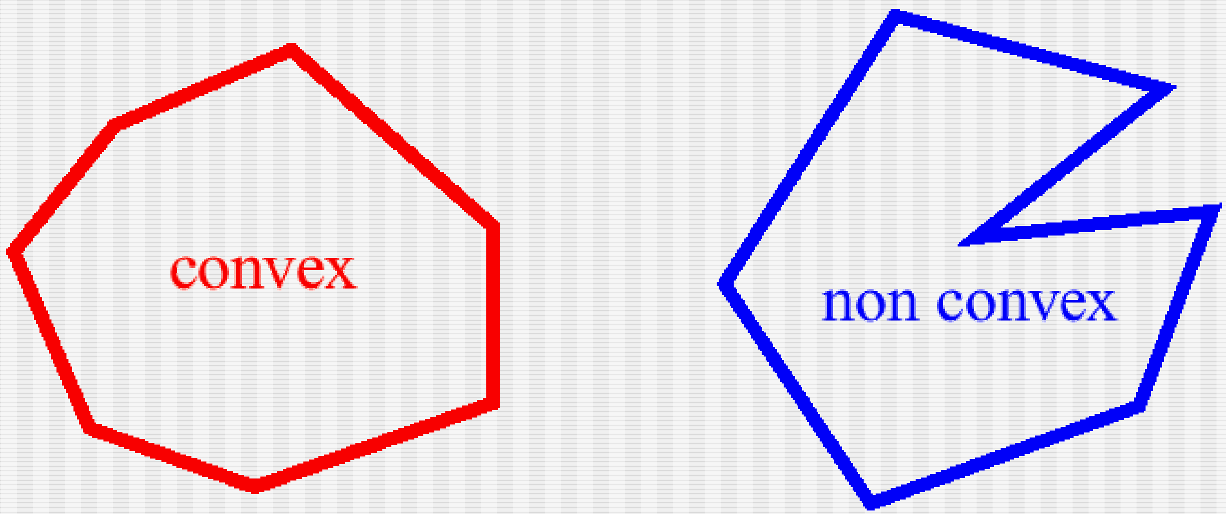
# Convex hull problem

- *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 $S$



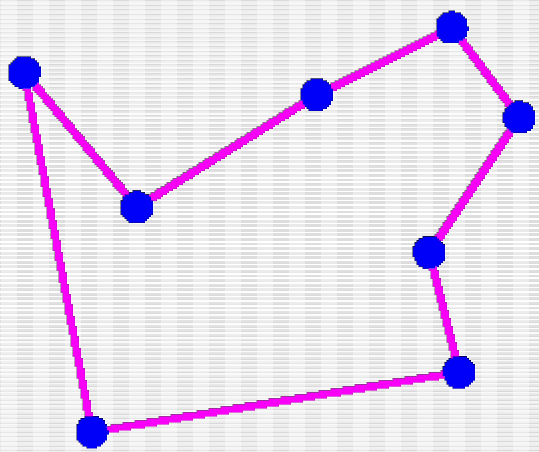
# What is convex

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



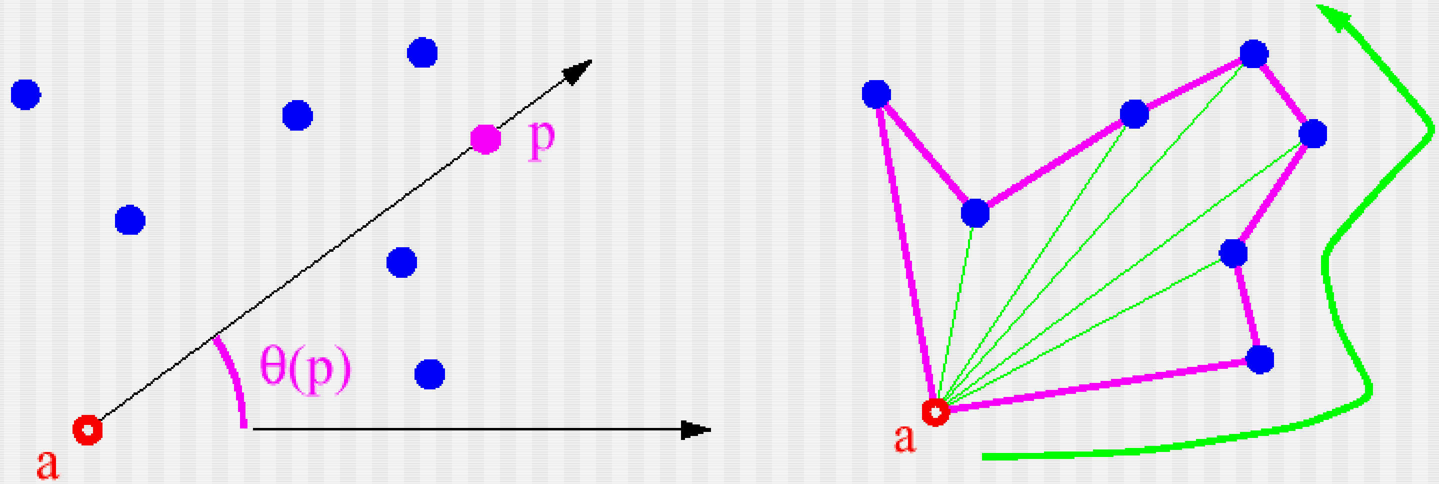
# Graham Scan

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



# Finding non-crossing path

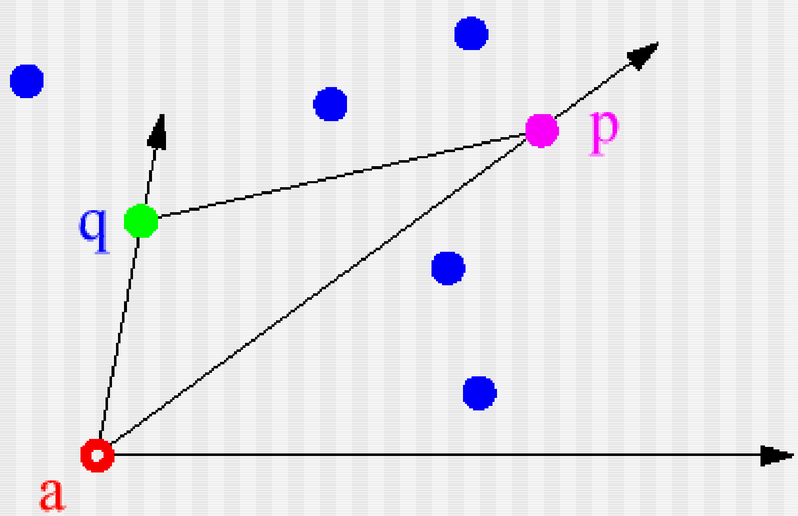
- *How do we find such a non-crossing path:*
  - Pick the bottommost point *a* as the anchor point
  - For each point p, compute the angle θ(p) of the segment (a,p) with respect to the *x*-axis.
  - Traversing the points by **increasing angle** yields a simple closed path



# Sorting by angle

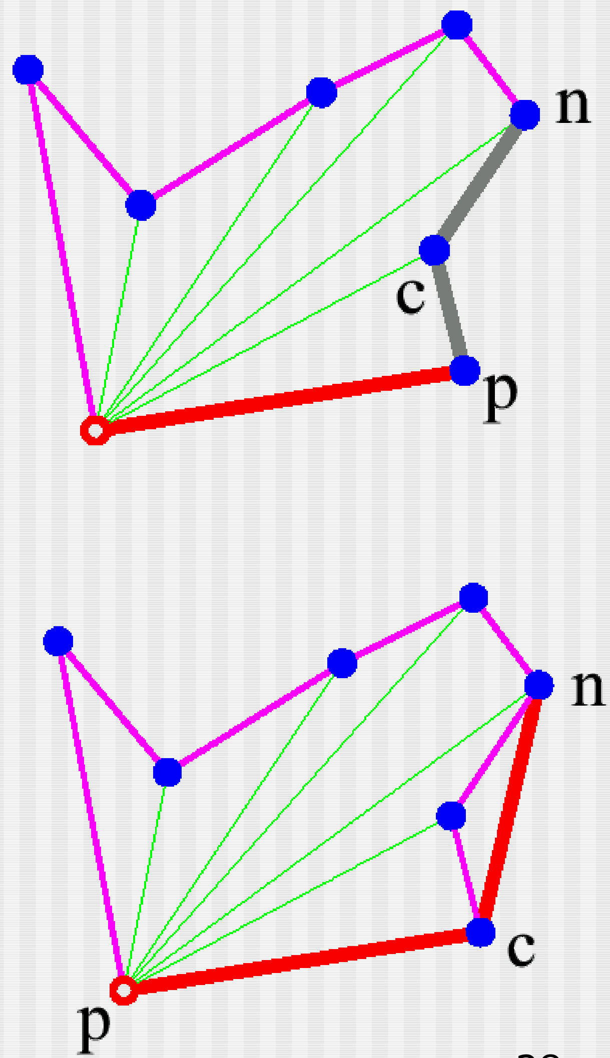
- *How do we sort by increasing angle*?
  - *Observation*: We do not need to compute the actual angle!
  - We just need to compare them for sorting

$\theta(p) < \theta(q) \Leftrightarrow$
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 points in the sorted order:
    - Before including the next point *n* check if the new added segment makes a right turn
    - If not, keep discarding the previous point (*c*) until the right turn is made



# Implementation and analysis

- *Implementation:*
  - *Stack to store vertices of the convex hull*
- *Analysis*:
  - Phase 1: O($n$ log $n$)
    - points are sorted by angle around the anchor
  - Phase 2: O($n$)
    - each point is pushed into the stack once
    - each point is removed from the stack at most once
  - Total time complexity O($n$ log $n$)