

# Algorithms and Data Structures Lecture XIV

---



# This Lecture

---

- Introduction to computational geometry
  - Algorithms
    - Points and lines in 2D space
    - *Line sweep* technique
    - Closest pair of points using divide-and-conquer
  - A peek at data structures for multidimensional range searching




# Computational Geometry

---

- The term first appeared in the 70's
- Originally referred to computational aspects of solid/geometric modeling
- Later as the field of algorithm design and analysis of discrete geometry
- Algorithmic bases for many scientific & engineering disciplines
  - GIS, robotics, computer graphics, computer vision, CAD/CAM, VLSI design, etc.

# Geometric Objects in Plane

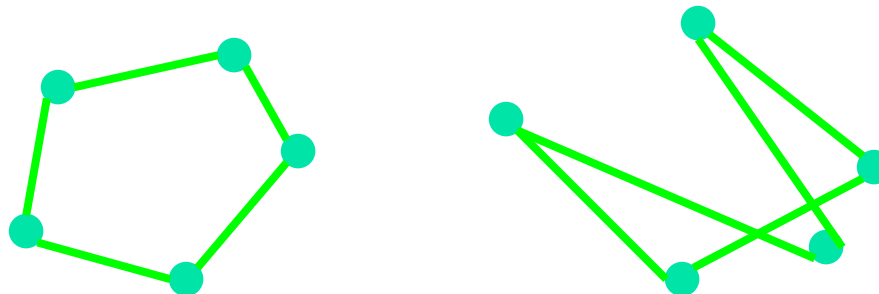
- **Point:** defined by a pair of coordinates  $(x,y)$
- **Segment:** portion of a straight line between two points – their *convex combination*



A diagram showing a horizontal line segment in red. The left endpoint is labeled 'A' and the right endpoint is labeled 'B'. Both endpoints are marked with small red circles.

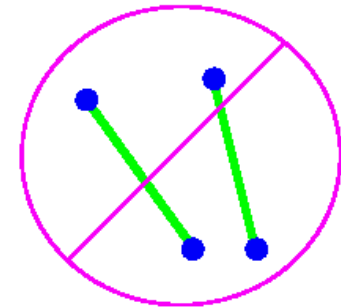
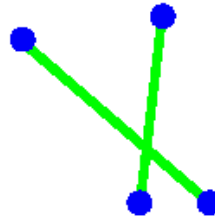
$$\begin{cases} x = \alpha x_A + (1 - \alpha)x_B \\ y = \alpha y_A + (1 - \alpha)y_B \end{cases}$$

- **Polygon:** a circular sequence of points (vertices) and segments (edges) between them

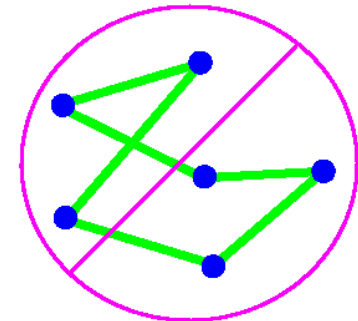
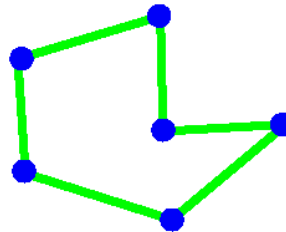


# Some Geometric Problems

- **Segment intersection:** Given two segments, do they intersect?

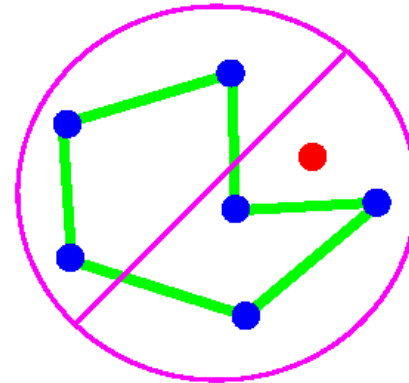
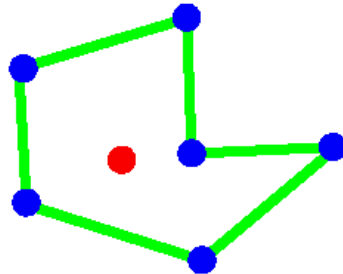


- **Simple closed path:** Given a set of points, find a non-intersecting polygon with vertices on the points



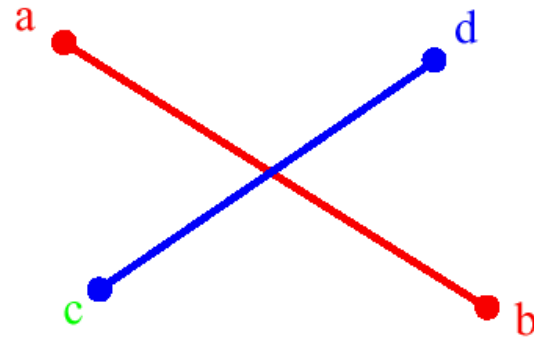
# Some Geometric Problems (2)

- **Inclusion in polygon:** Is a point inside or outside a polygon?



# Segment Intersection

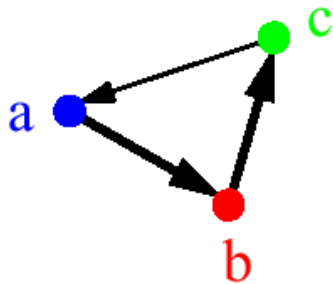
- Test whether segments  $(a,b)$  and  $(c,d)$  intersect.  
*How do we do it?*



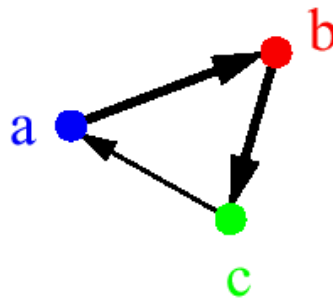
- We could start by writing down the **equations of the lines through the segments**, then test whether the lines intersect, then ...
- An alternative (and simpler) approach is based in the notion of **orientation of an ordered triplet of points** in the plane

# Orientation in the Plane

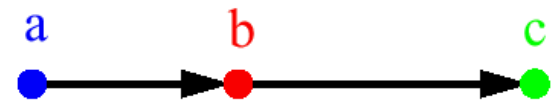
- The orientation of an ordered triplet of points in the plane can be
  - counterclockwise (left turn)
  - clockwise (right turn)
  - collinear (no turn)



counterclockwise  
(left turn)



clockwise  
(right turn)

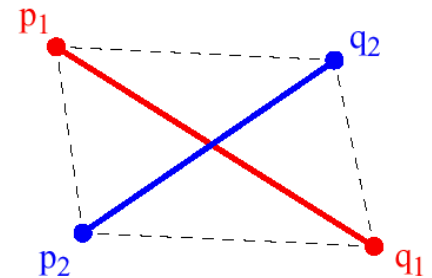


collinear  
(no turn)



# Intersection and Orientation

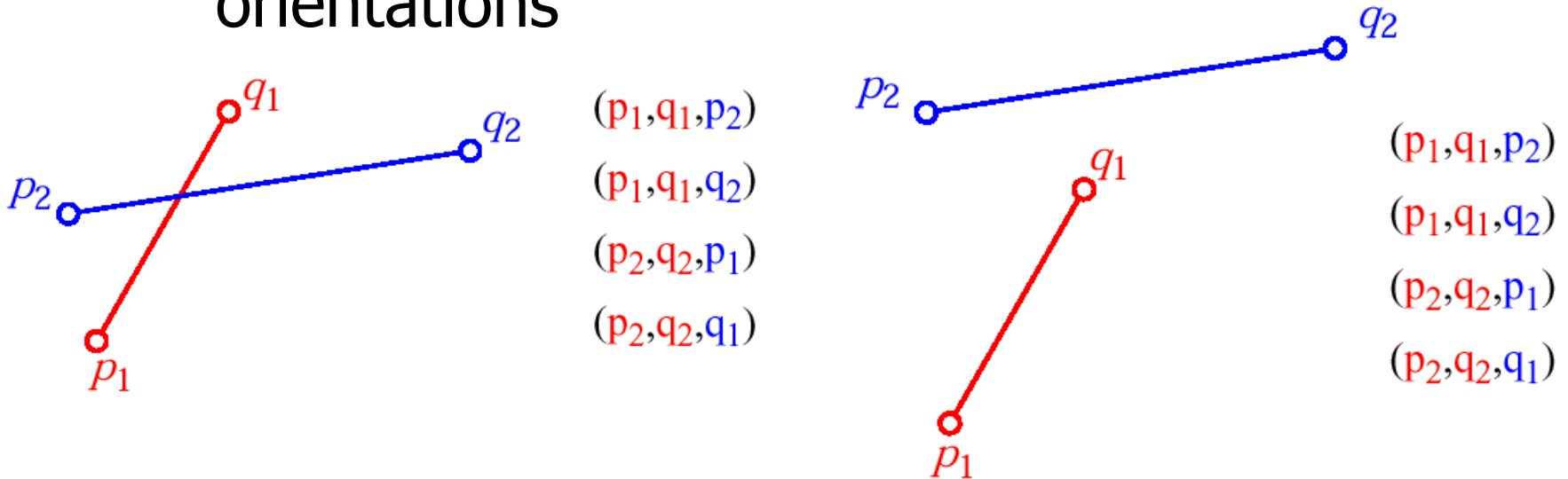
- Two segments  $(p_1, q_1)$  and  $(p_2, q_2)$  intersect if and only if one of the following two conditions is verified
- General case:
  - $(p_1, q_1, p_2)$  and  $(p_1, q_1, q_2)$  have different orientations **and**
  - $(p_2, q_2, p_1)$  and  $(p_2, q_2, q_1)$  have different orientations
- Special 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



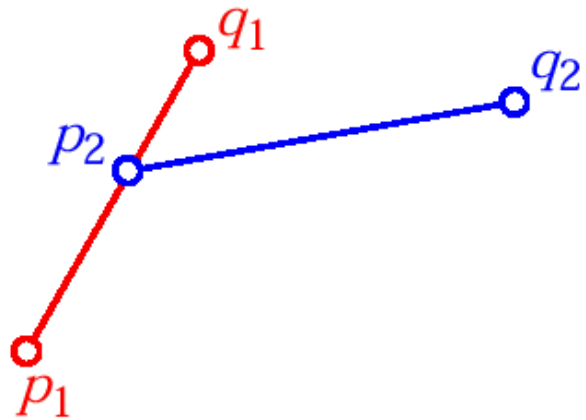
# Orientation Examples

- 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)

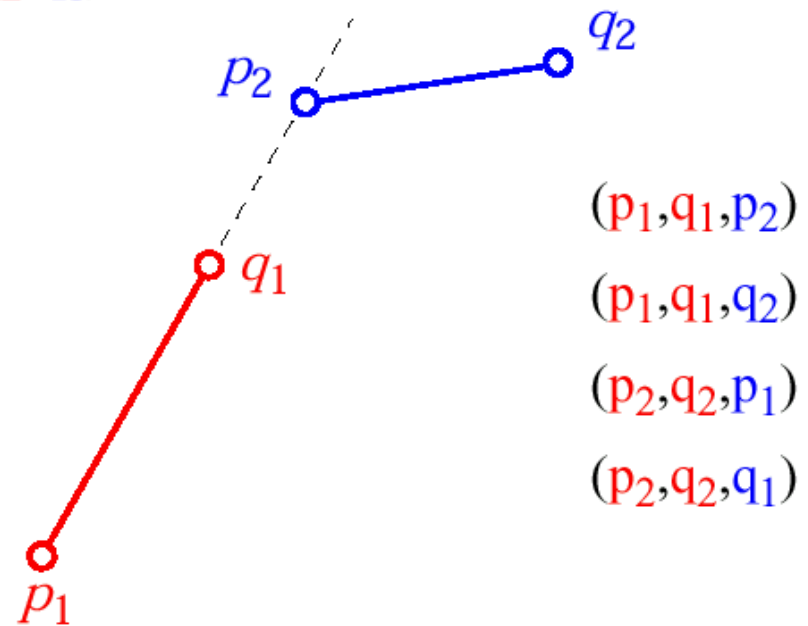


$(p_1, q_1, p_2)$

$(p_1, q_1, q_2)$

$(p_2, q_2, p_1)$

$(p_2, q_2, q_1)$



$(p_1, q_1, p_2)$

$(p_1, q_1, q_2)$

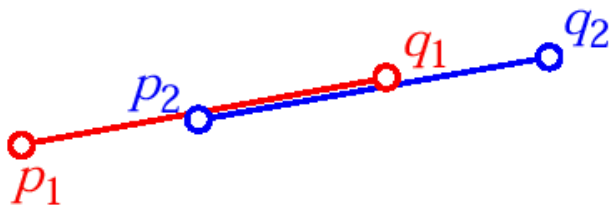
$(p_2, q_2, p_1)$

$(p_2, q_2, q_1)$

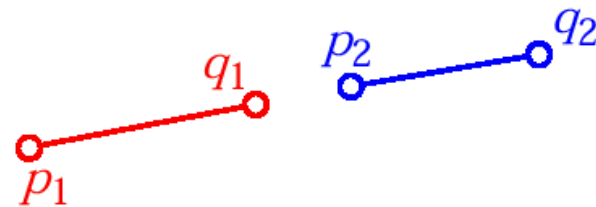
# Orientation Examples (3)

- 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



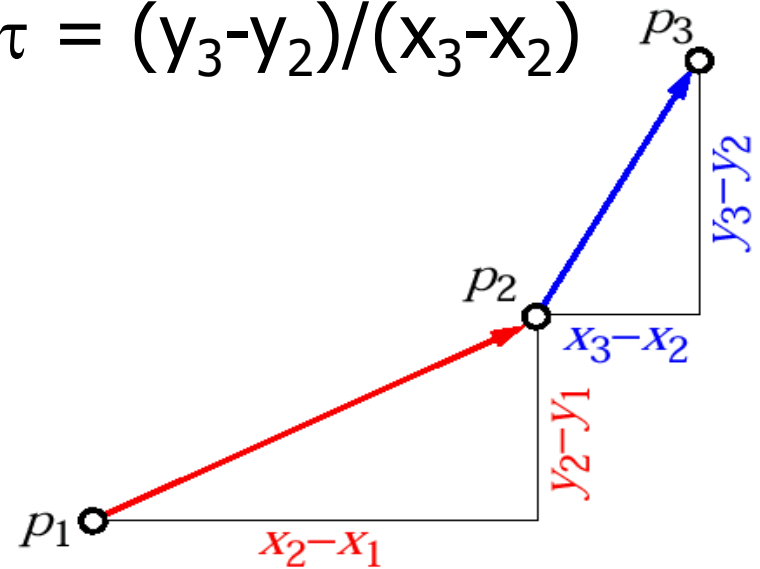
- $(p_1, q_1, p_2)$
- $(p_1, q_1, q_2)$
- $(p_2, q_2, p_1)$
- $(p_2, q_2, q_1)$



- $(p_1, q_1, p_2)$
- $(p_1, q_1, q_2)$
- $(p_2, q_2, p_1)$
- $(p_2, q_2, q_1)$

# Computing the Orientation

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



- Orientation test
  - counterclockwise (left turn):  $\sigma < \tau$
  - clockwise (right turn):  $\sigma > \tau$
  - collinear (no turn):  $\sigma = \tau$



# Computing the Orientation (2)

- The orientation depends on whether the following expression is positive, negative, or null

$$(y_2 - y_1)(x_3 - x_2) - (y_3 - y_2)(x_2 - x_1) = ?$$

- This is a *cross product* of two vectors

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



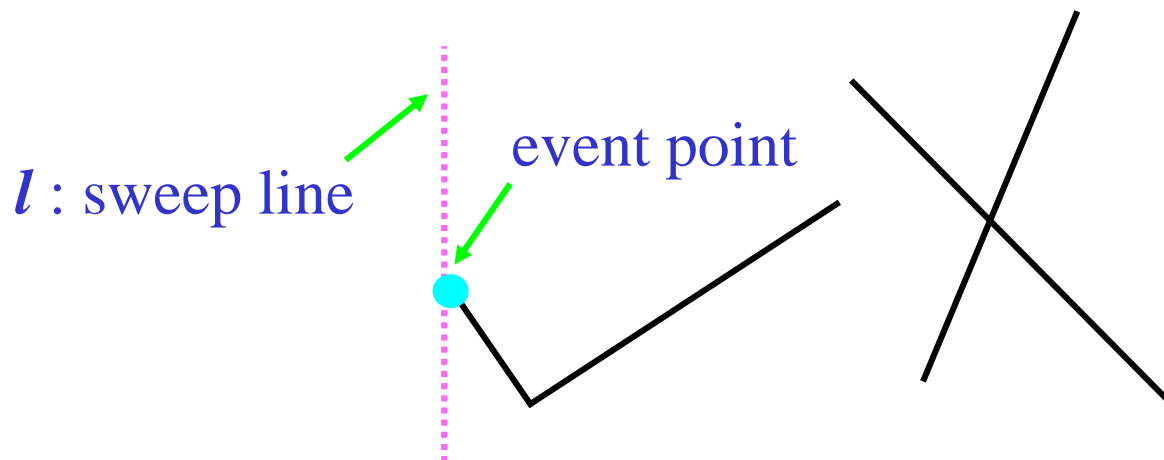
# Determining Intersections

---

- Given a set of  $n$  segments, we want to determine whether any two line segments intersect
- A **brute force approach** would take  $O(n^2)$  time (testing each with every other segment for intersection)

# Determining Intersections (2)

- It can be done faster by using a powerful comp. geometry technique, called ***plane-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

---

- Each segment end point is an event point
- At an event point, update the status of the sweep line & perform intersection tests
  - left end point: a new segment is added to the status of / and it's tested against the rest
  - right end point: it's deleted from the status of /
- Only testing pairs of segments for which there is a horizontal line that intersects both segments
- This might be not good enough. It may still be inefficient,  $O(n^2)$  for some cases

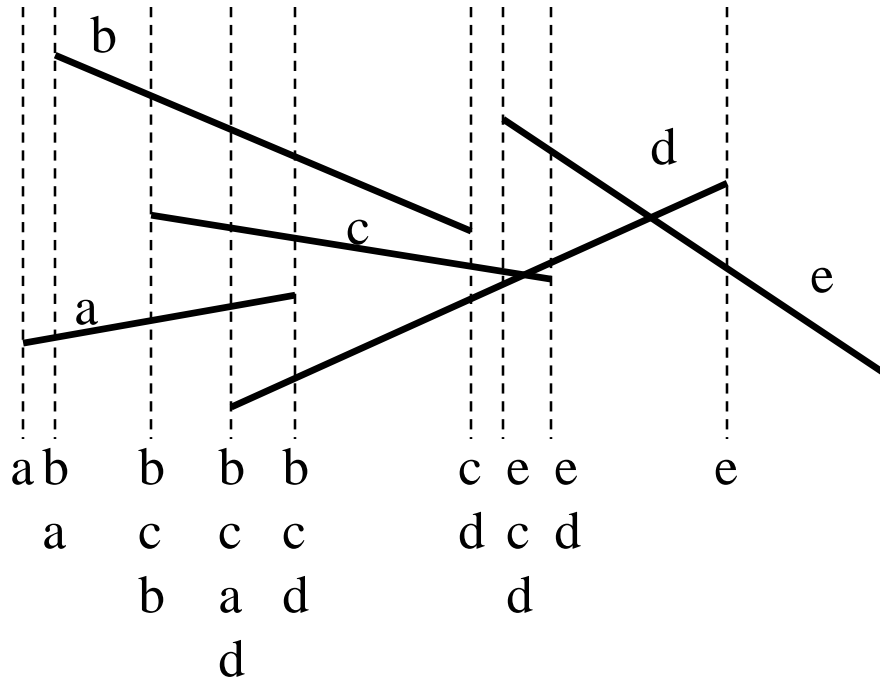


# Plane Sweep Algorithm (2)

---

- To include the idea of being close in the horizontal direction, only test segments that are **adjacent in the vertical direction**
- Only test a segment with the **ones above and below** (predecessor and successor, *rings a bell...*)
  - the "above-below" relationship among segments does not change unless there is an intersection
- New "status": *ordered* sequence of segments

# Plane Sweeping Algorithm (3)





# Pseudo Code

---

**AnySegmentsIntersect** (S)

01  $T \leftarrow \emptyset$

02 sort the end points of the segments in S from left to right, breaking ties by putting points with lower y-coords first

03 **for** each point p in the sorted list of end points **do**

04     **if** p is the left end point of a segment s **then**

05         Insert(T,s)

06             **if** (Above(T,s) exists and intersects s) or  
               (Below(T,s) exists and intersects s) **then**

07                 **return** TRUE

08     **if** p is the right end point of a segment s **then**

09         **if** both Above(T,s) and Below(T,s) exist and  
           Above(T,s) intersects Below(T,s) **then**

10             **return** TRUE

11         Delete(T,s)

12 **return** FALSE

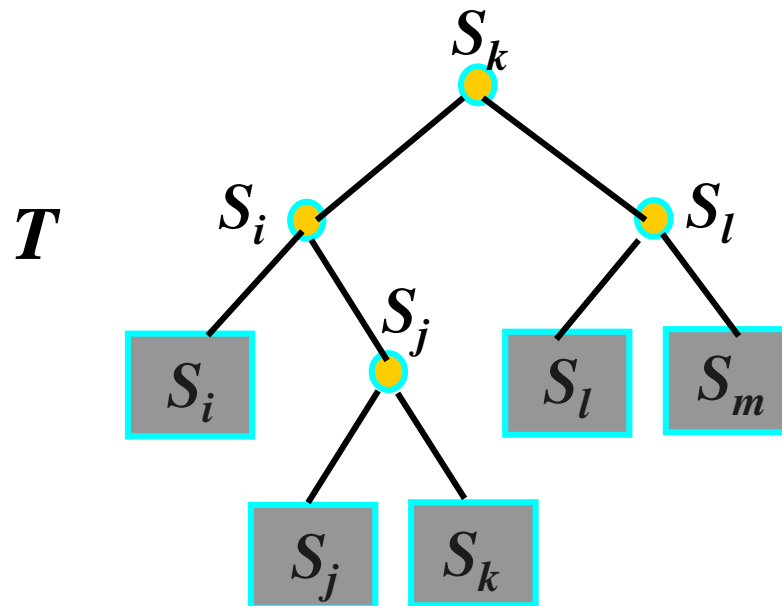
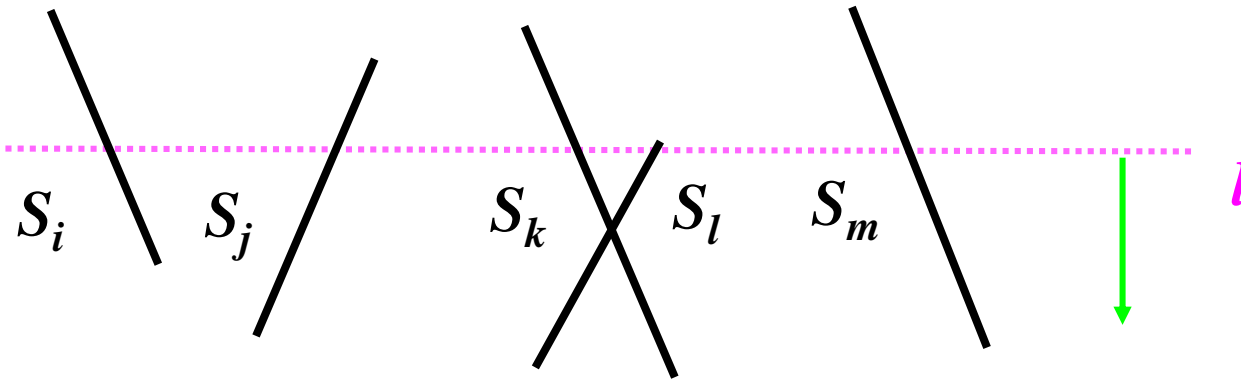


# Implementing Event Queue

---

- Define an order on event points
- Store the event points in a priority queue
  - both insertion or deletion takes  $O(\log n)$  time, where  $n$  is the number of events, and fetching minimum –  $O(1)$
  - For our algorithm sorted array is enough, because the set of events does not change.
- Maintain the status of  $/$  using a binary search tree  $T$ 
  - the up-to-down order of segments on the line  $/ \iff$  the left-to-right order of leaves in  $T$
  - segments in internal nodes guide search
  - each update and search takes  $O(\log n)$

# Status and Structure





# Running Time

---

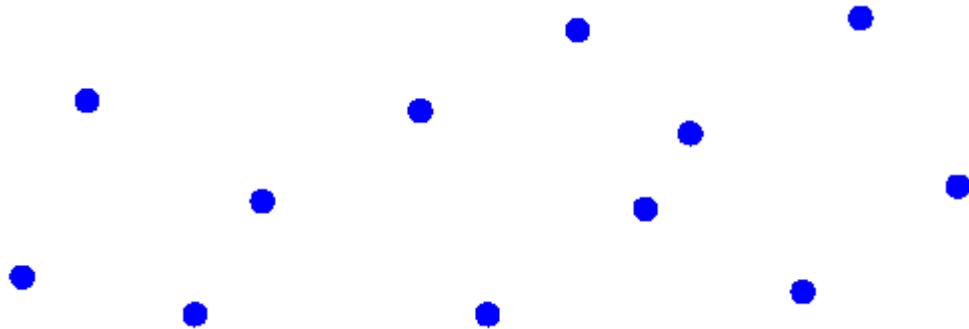
- Sorting the segments takes  $O(n \log n)$  time
- The loop is executed once for every end point ( $2n$ ) taking each time  $O(\log n)$  time (e.g., red-black tree operation)
- The total running time is  $O(n \log n)$



# Closest Pair

---

- Given a set  $P$  of  $N$  points, find  $p, q \in P$ , such that the distance  $d(p, q)$  is minimum
- Algorithms for determining the closest pair:
  - Brute Force  $O(n^2)$
  - Divide and Conquer  $O(n \log n)$
  - Sweep-Line  $O(n \log n)$







# Brute Force

---

- Compute all the distances  $d(p, q)$  and select the minimum distance
- Running time  **$O(n^2)$**

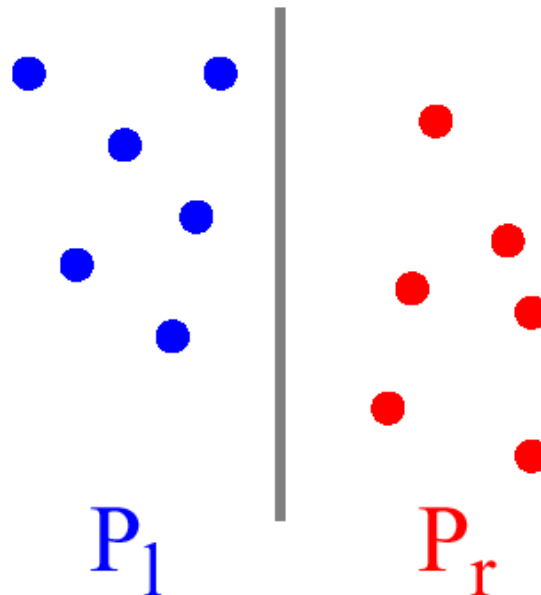
●  $(x_1, y_1)$   
 $p_1$

$(x_2, y_2)$   
●  
 $p_2$

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

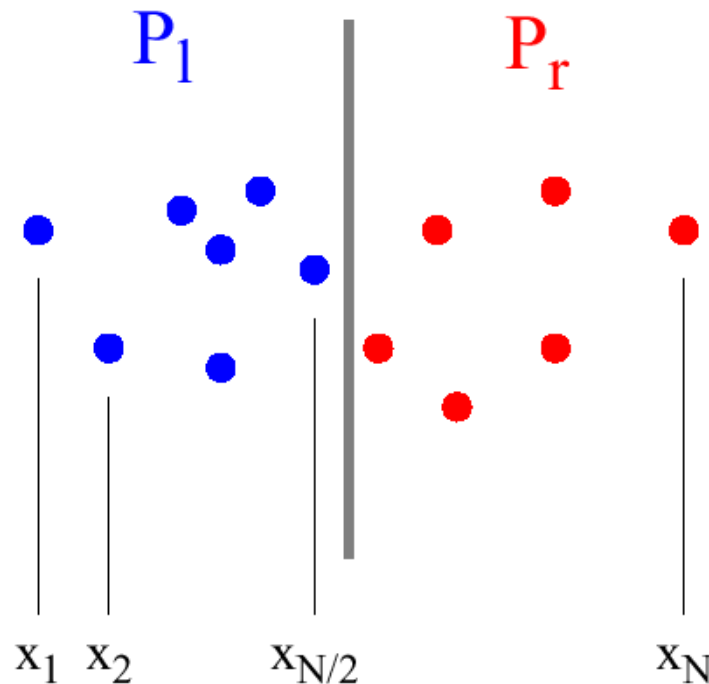
# Divide and Conquer

- **Sort points** on the x-coordinate and **divide them in half**
- Closest pair is either in one of the halves or has a member in each half



# Divide and Conquer (2)

- **Phase 1:** Sort the points by their x-coordinate
- $p_1 p_2 \dots p_{n/2} \dots p_{n/2+1} \dots p_n$





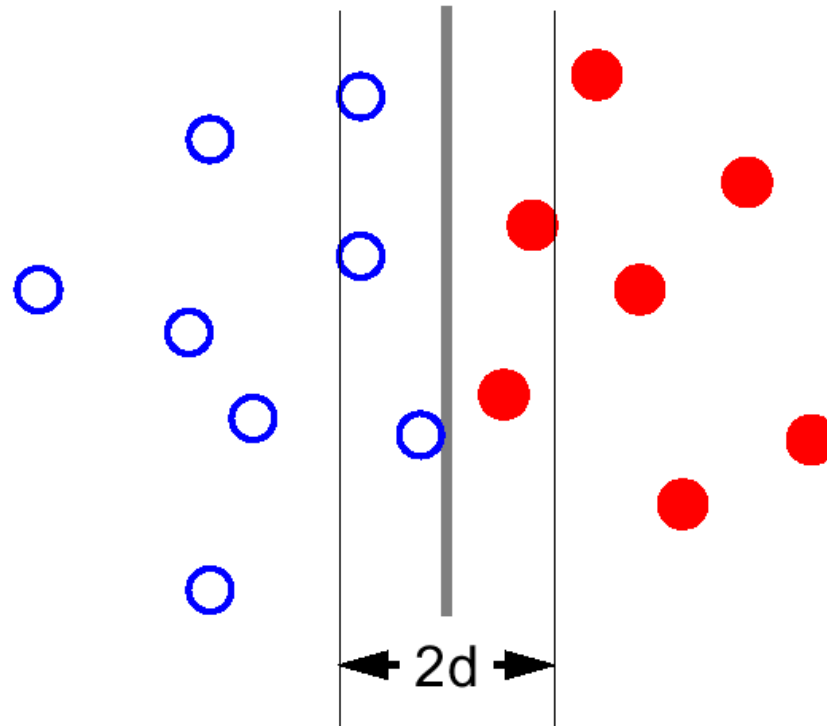
# Divide and Conquer (3)

---

- **Phase 2:** Recursively compute closest pairs and minimum distances,
  - $d_l, d_r$  in
  - $P_l = \{p_1, p_2, \dots, p_{n/2}\}$
  - $P_r = \{p_{n/2+1}, \dots, p_n\}$
- Find the closest pair and closest distance in central strip of width  $2d$ , where
  - $d = \min(d_l, d_r)$
- in other words...

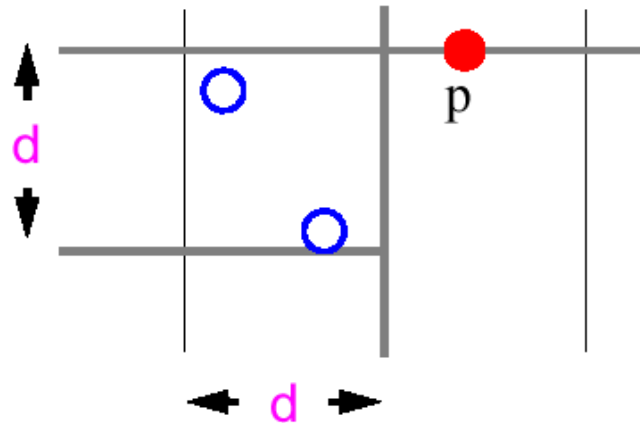
# Divide and Conquer (4)

- Find the closest (○, ●) pair in a strip of width  $2d$ , knowing that no (○, ○) or (●, ●) pair is closer than  $d$



# Combining Two Solutions

- For each point  $p$  in the strip, check distances  $d(p, q)$ , where  $p$  and  $q$  are of different colors and:  
$$y(p) - d \leq y(q) \leq y(p)$$



- There are no more than four such points!



# Running Time

- Sorting by the y-coordinate at each conquering step yields the following recurrence

$$\begin{aligned}T(n) &= 2T(n/2) + n \log n \\T(1) &= 1\end{aligned}$$

$$O(n \log^2 n)$$

$$\begin{aligned}T(n) &= 2T(n/2) + n \log n \\&= 4T(n/4) + 2(n/2) \log(n/2) + n \log n \\&= 4T(n/4) + n(\log n - 1) + n \log n \\&\dots \\&= 2^k T(n/2^k) + n(\log n + (\log n - 1) + \dots + (\log n - k + 1)) \\&\dots \\&\text{stop when } n/2^k = 1; k = \log n \\&= n + n(1 + 2 + 3 + \dots + \log n) \\&= n + n((\log n + 1) \log n) / 2\end{aligned}$$



# Improved Algorithm

---

- The idea: **Sort** all the points by y-coordinate **once**
- Before recursive calls, **partition the sorted list** into two sorted sublists for the left and right halves





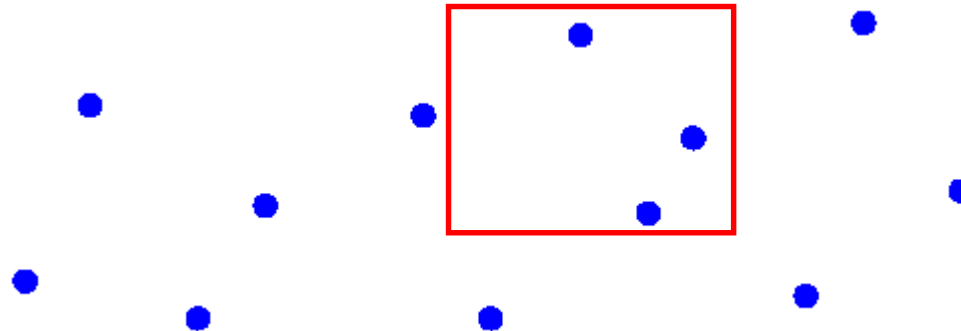
# Running Time

---

- Phase 1:
  - Sort by x and y coordinate:
  - $O(n \log n)$
- Phase 2:
  - Partition:  $O(n)$
  - Recur:  $2T(n/2)$
  - Combine:  $O(n)$
- $T(n) = 2T(n/2) + n = O(n \log n)$
- **Total Time:  $O(n \log n)$**

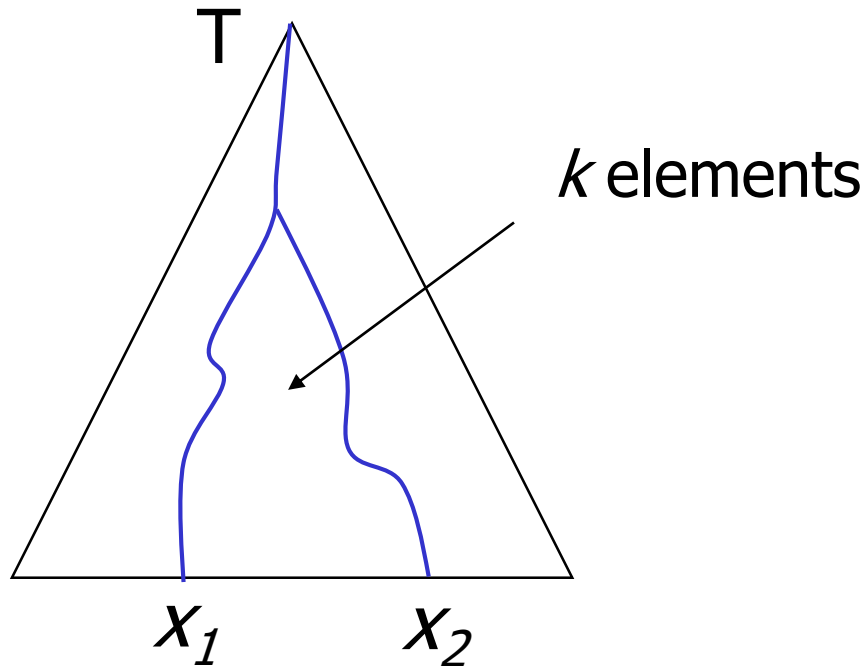
# Multidimensional DS

- Multidimensional Data Structures are used to answer queries on the set of multidimensional points
  - Range query ( $[x_1, x_2], [y_1, y_2]$ ): find all points  $(x, y)$  such that  $x_1 < x < x_2$  and  $y_1 < y < y_2$



# 1D Range Searching

- Find  $x$ , such that  $x_1 < x < x_2$ 
  - Can be done with balanced binary search trees (e.g., red-black trees) in  $O(\log n + k)$  time, where  $k$  is the size of the answer





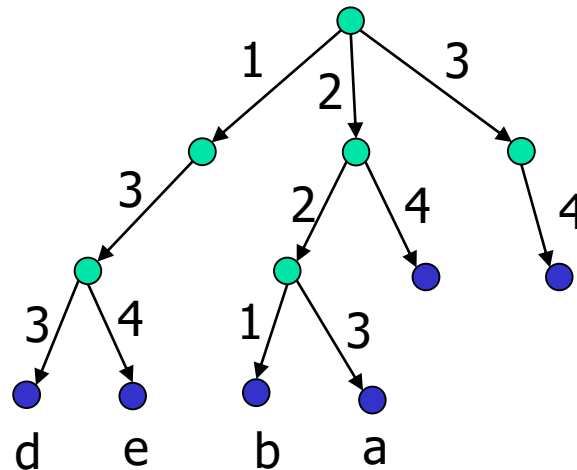
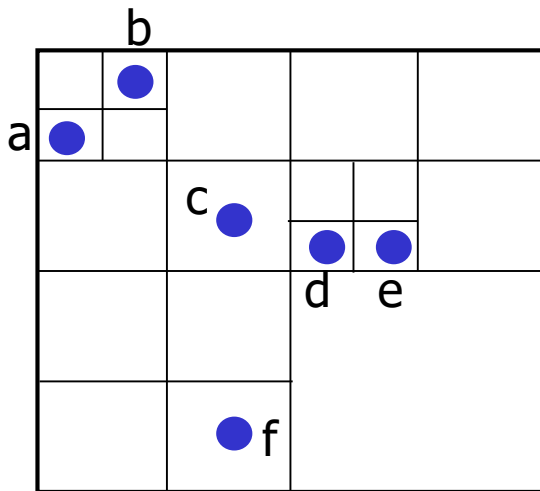
# Range Trees

---

- Build a binary tree  $T$  on  $x$ -coordinates
- With each node  $v$ , associate a binary tree  $T_v$  that stores all the points in the subtree of  $T$  rooted at  $v$ . Organize  $T_v$  on  $y$ -coordinates.
- Space:  $O(n \log n)$
- Range query:  $O(\log^2 n + k)$

# Quadtrees

- Quadtrees – partition tree
  - Linear space
  - Good average query performance





# Next lecture

---

- Amortized Analysis