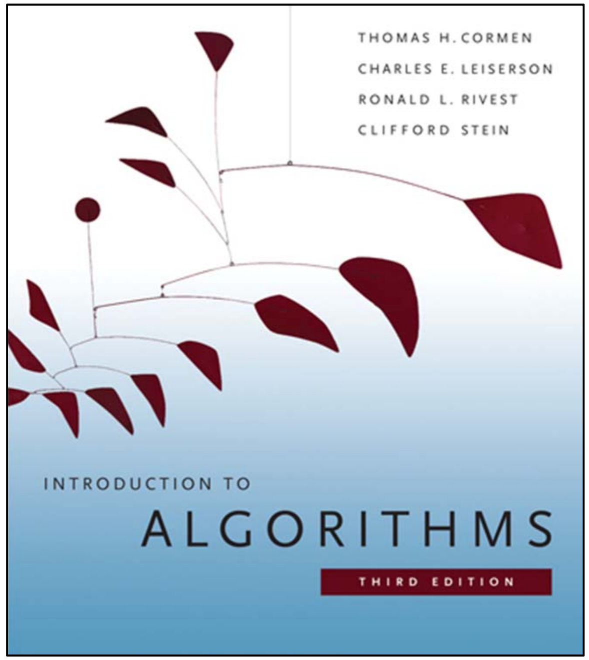
6.006

Introduction to Algorithms



Lecture 24: Geometry

Prof. Erik Demaine

Today

- Computational geometry
- Line-segment intersection
  - Sweep-line technique
- Closest pair of points
  - Divide & conquer strikes back!

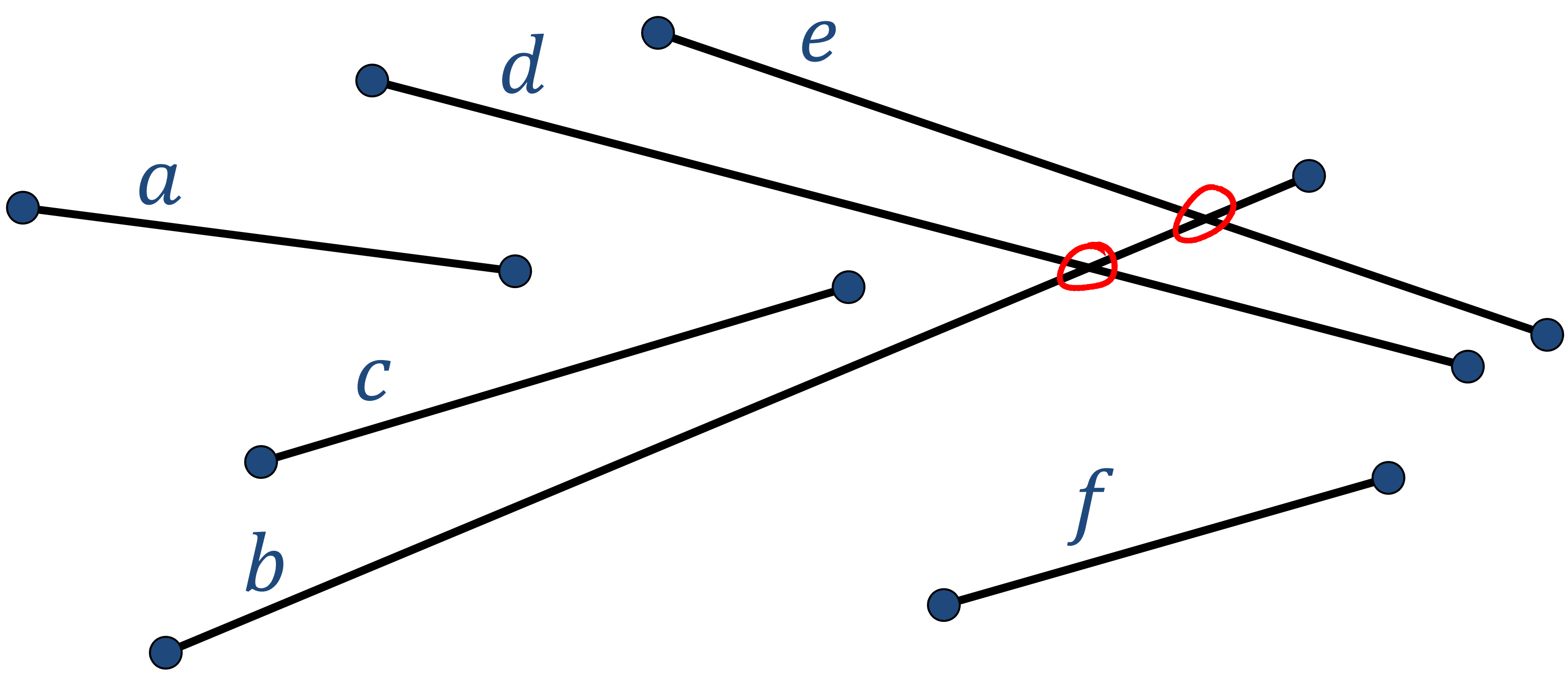
Motivation: Collision Detection

Motivation: Collision Detection

"GTA 4 Carmageddon!" by dot12321
http://www.youtube.com/watch?v=4-620xx7yTo

Line-Segment Intersection

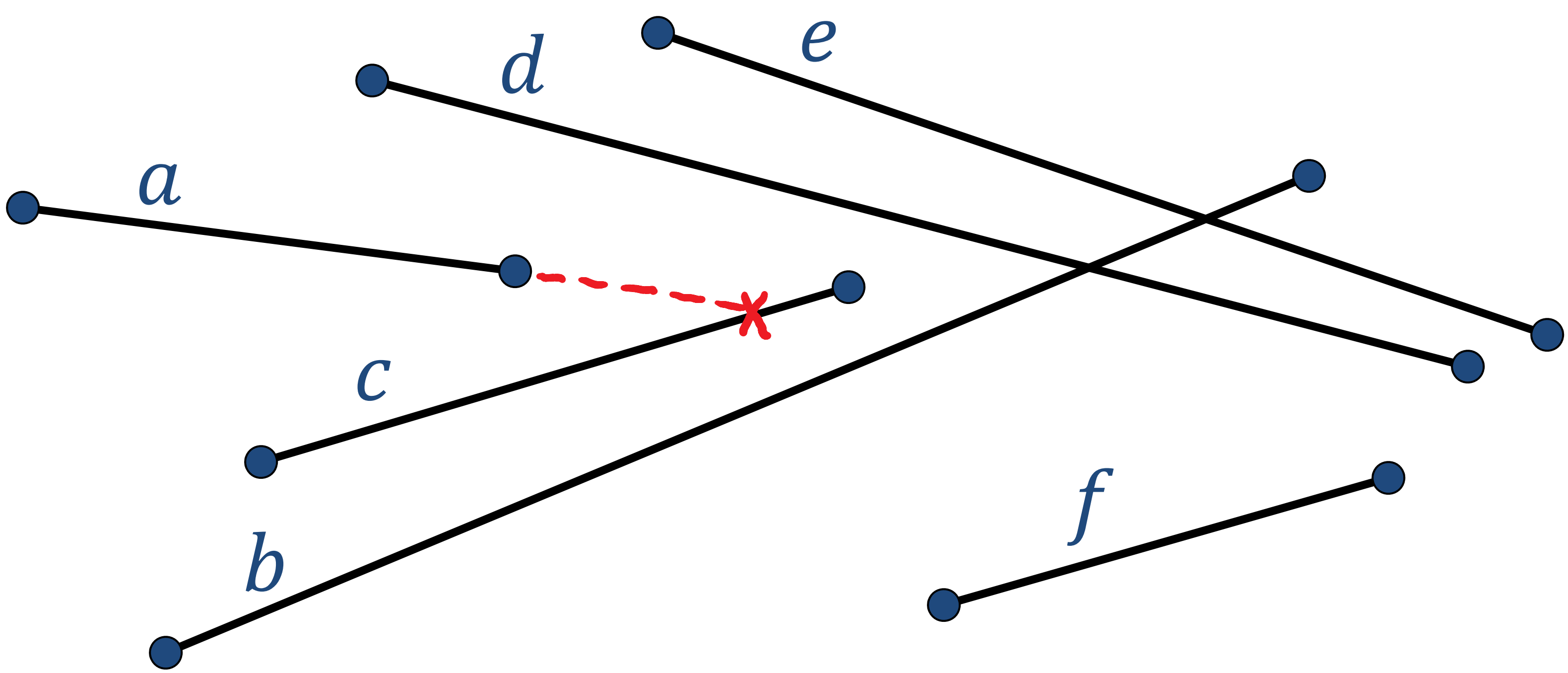
- Input: n line segments in 2D
- Goal: Find the k intersections



Obvious Algorithm

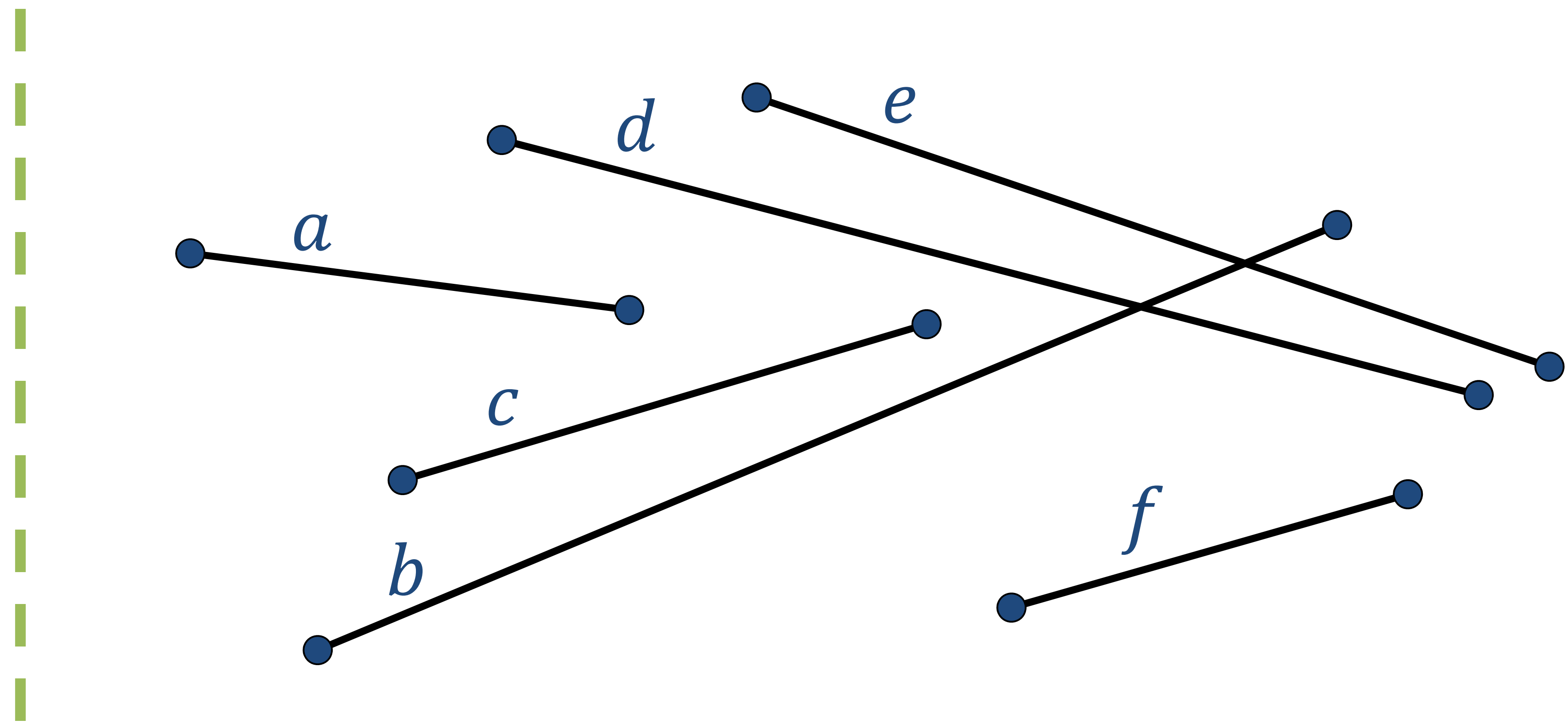
for every pair (ℓ, ℓ') of line segments:
    check for intersection

$\Big\}\, O(n^2)$

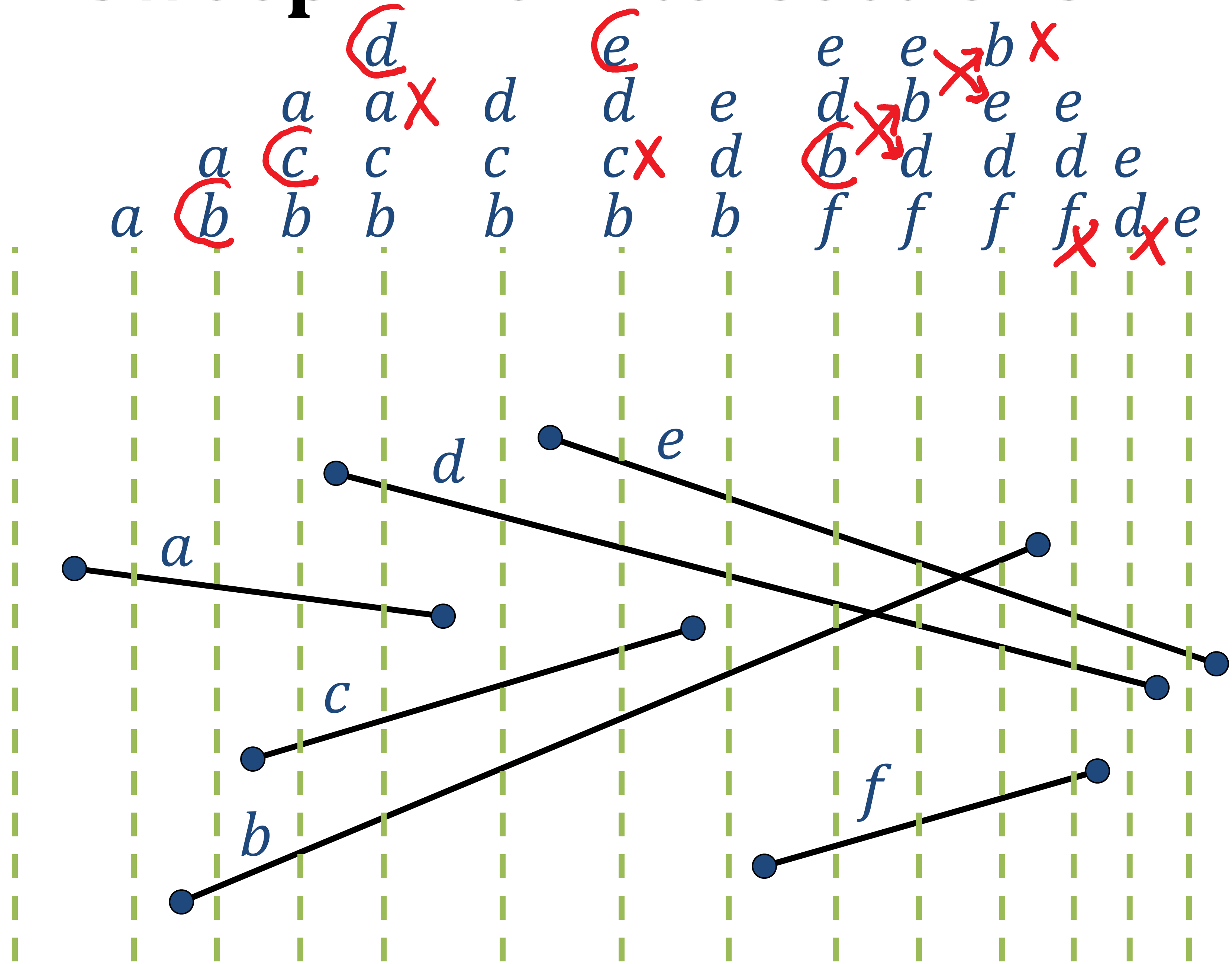


Sweep-Line Technique

- <u>Idea:</u> Sweep a vertical line from left to right
- Maintain intersection of line with input

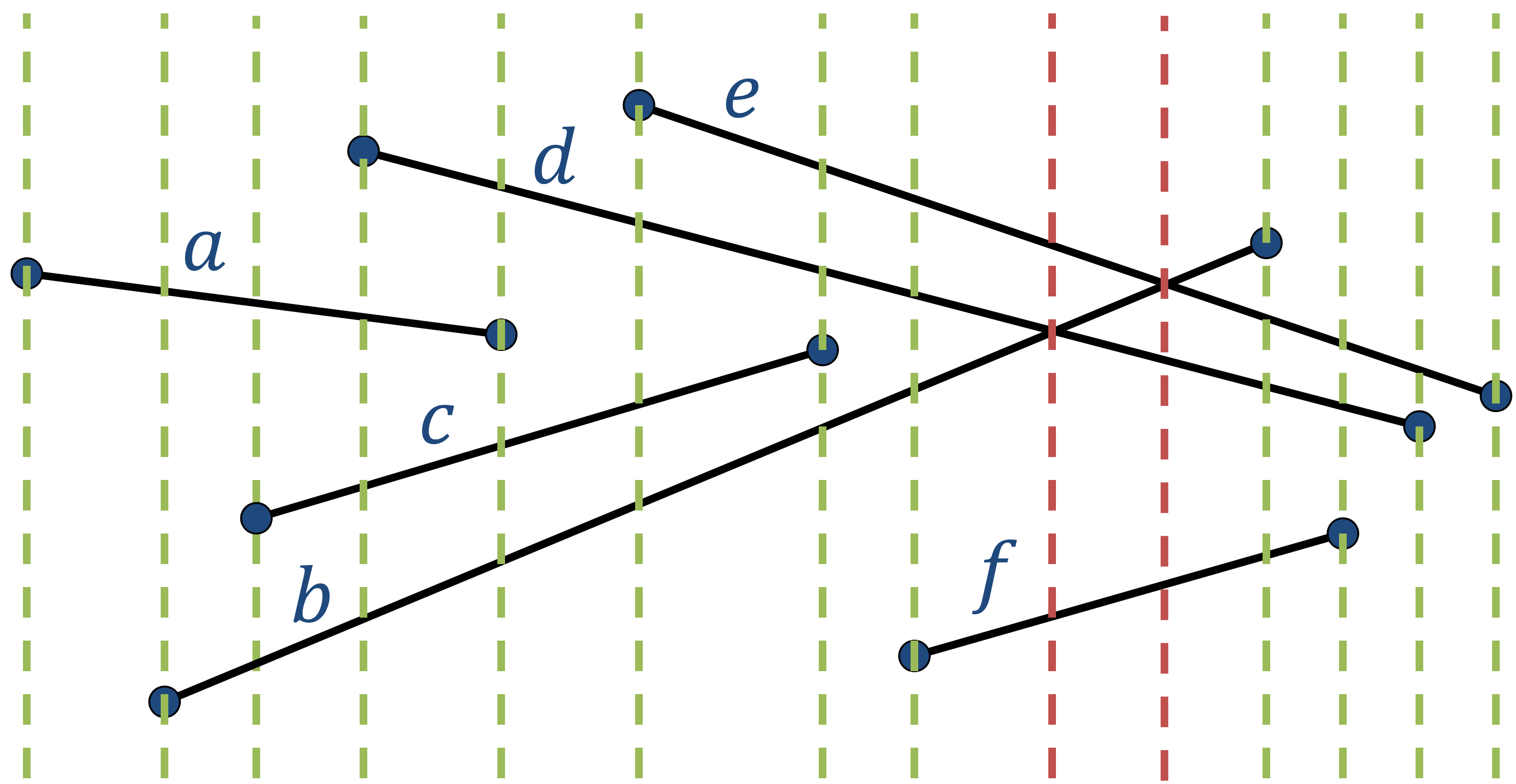


Sweep-Line Intersections

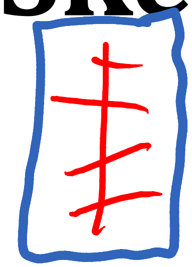


Sweep-Line Events

- Discretize continuous motion of sweep line
- ***Events*** when intersection changes
  - Segment endpoints
  - Intersections



Sweep-Line Algorithm Sketch

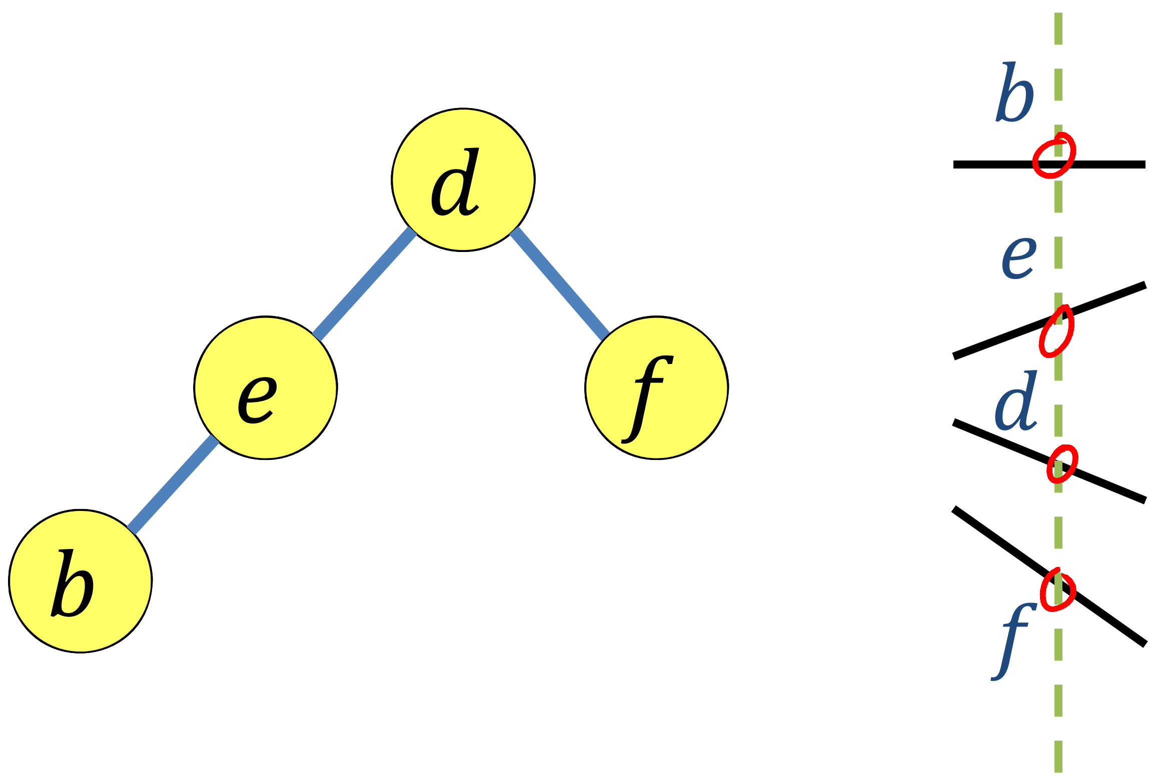


- Maintain sweep-line intersection
- Maintain **priority queue** of (possible) event ***times*** ($= x$ coordinates of sweep line)
- Until queue is empty:
  - Delete minimum event time t from priority queue
  - Update sweep-line intersection from $< t$ to $> t$
  - Update possible event times in priority queue

"Discrete-event simulation"

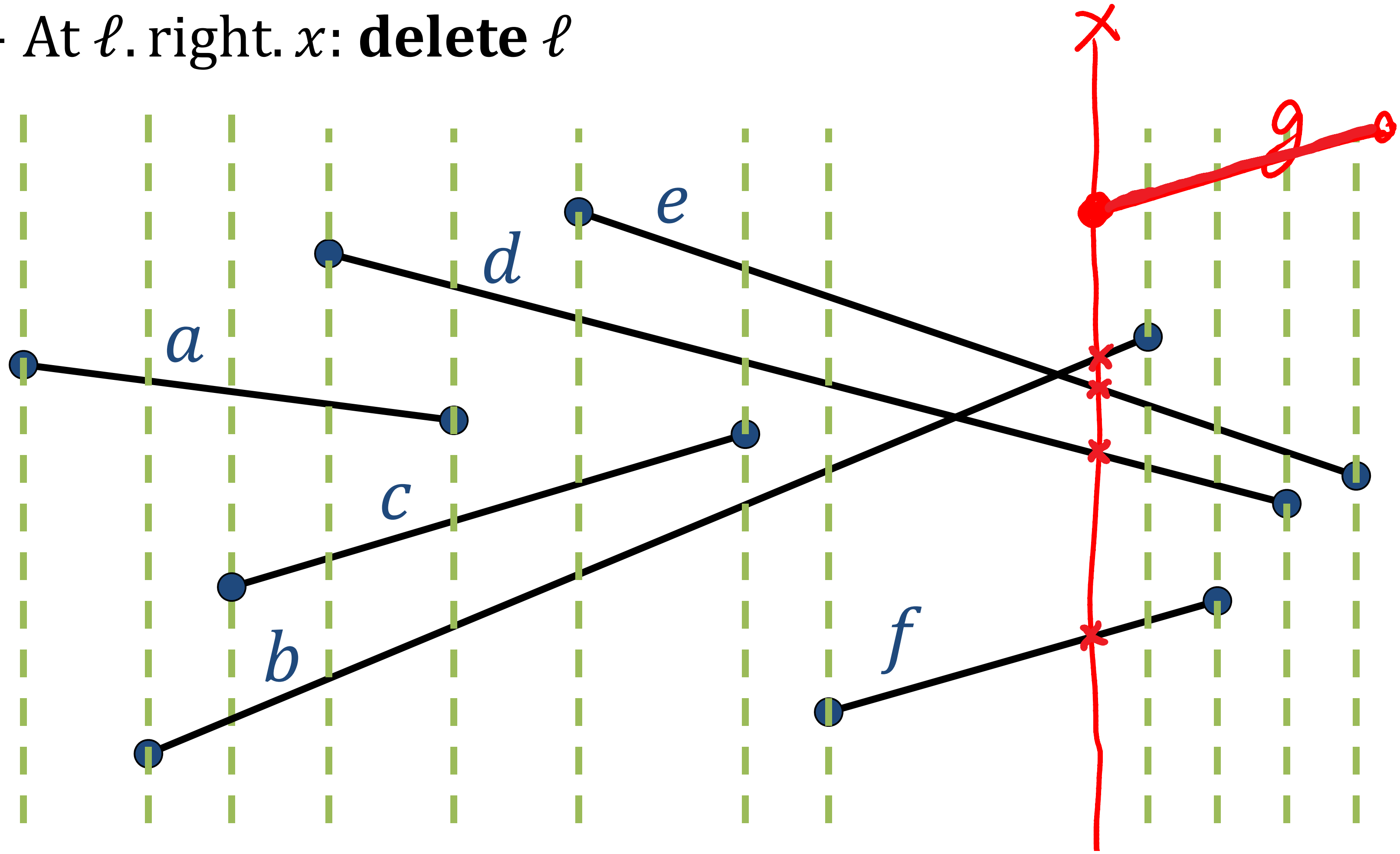
Intersection Data Structure

- Balanced binary search tree (e.g., AVL tree) to store sorted (y) order of intersections



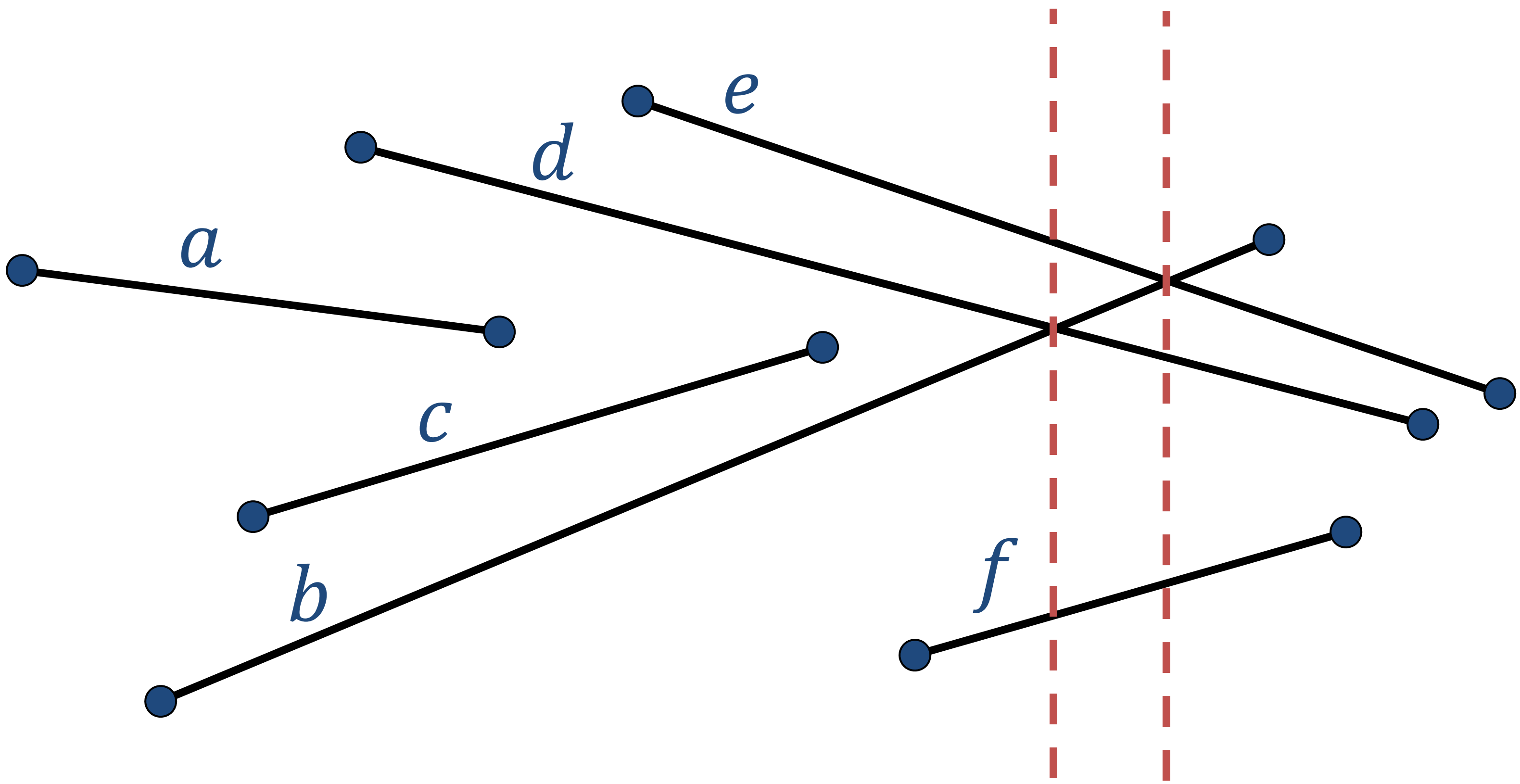
Endpoint Events

- For each line segment $\ell = (\ell.\text{left}, \ell.\text{right})$:
  - At $\ell.\text{left}.x$: **insert** ℓ (binary search for neighbors then)
  - At $\ell.\text{right}.x$: **delete** ℓ



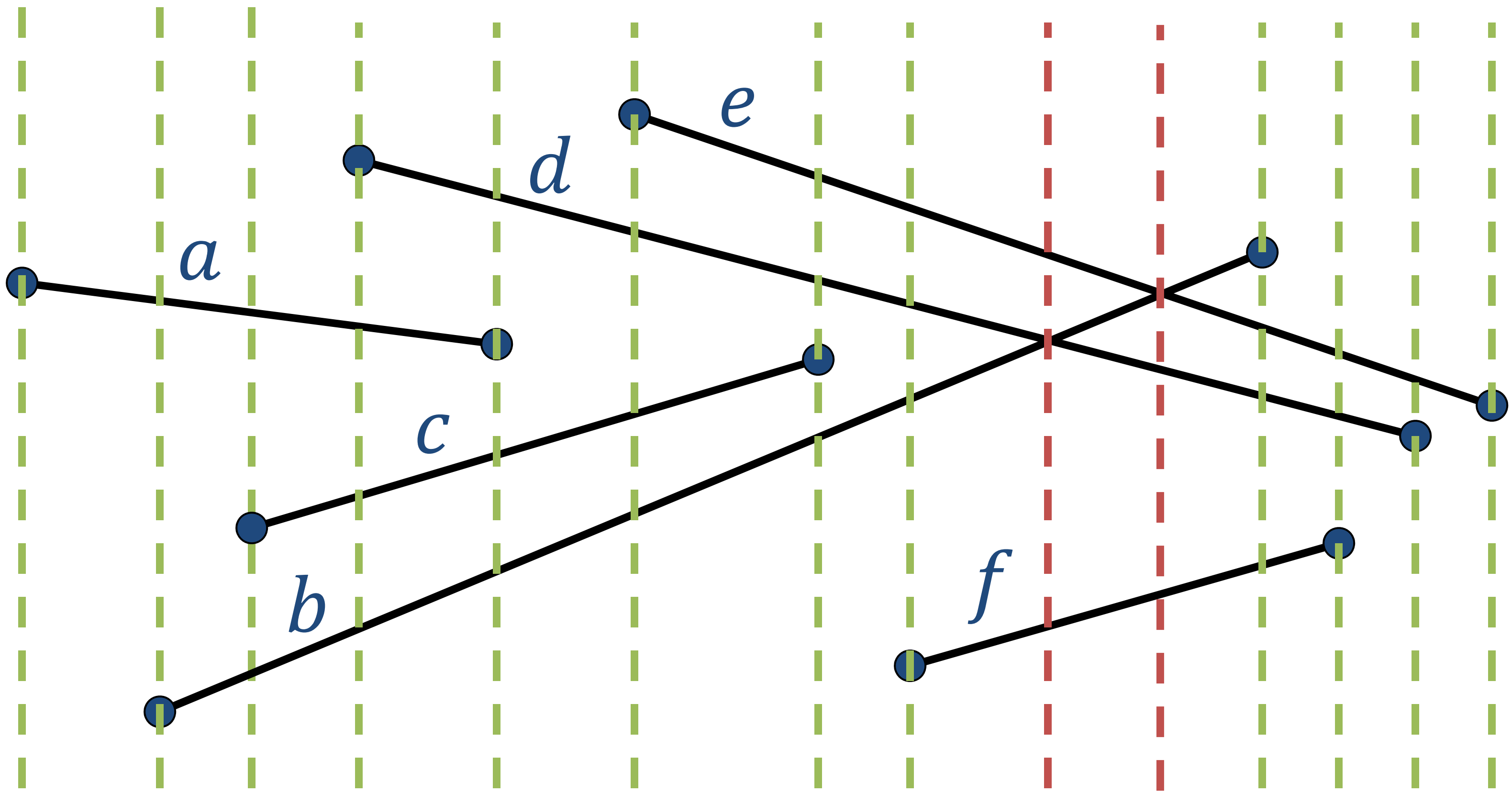
Intersection Events?

- How to know when have intersection events?
- This is the whole problem!



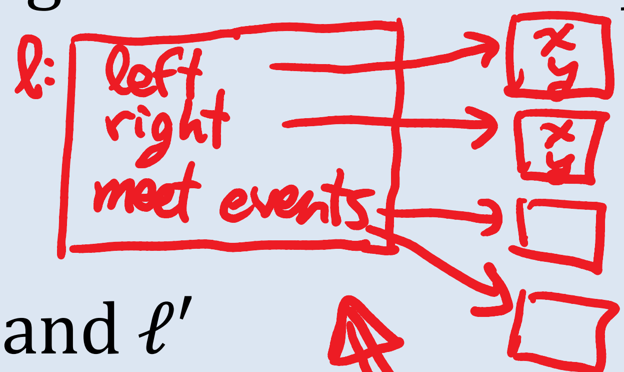
Intersection Events

- Compute when neighboring segments would intersect, if nothing changes meanwhile
- If such an event is next, then it really happens



Sweep-Line Algorithm

T = empty AVL tree
Q = Build−Heap($\{\ell.\text{left}.x, \ell.\text{right}.x$ for segment $\ell\}$)
while Q is not empty:
    event = Q.delete−min()
    if event is $\ell.\text{left}.x$:
        insert ℓ into T (binary searching with $x = \ell.\text{left}.x$)
    elif event is $\ell.\text{right}.x$:
        delete ℓ from T
    elif event is meet(ℓ, ℓ'):
        report intersection between ℓ and ℓ'
        swap contents of nodes for ℓ and ℓ' in T
    for each node ℓ changed or neighboring changed in T:
$O(1)$ delete all meet events involving ℓ from Q
        add meet(ℓ, T.pred(ℓ)) & meet(ℓ, T.succ(ℓ)) to Q



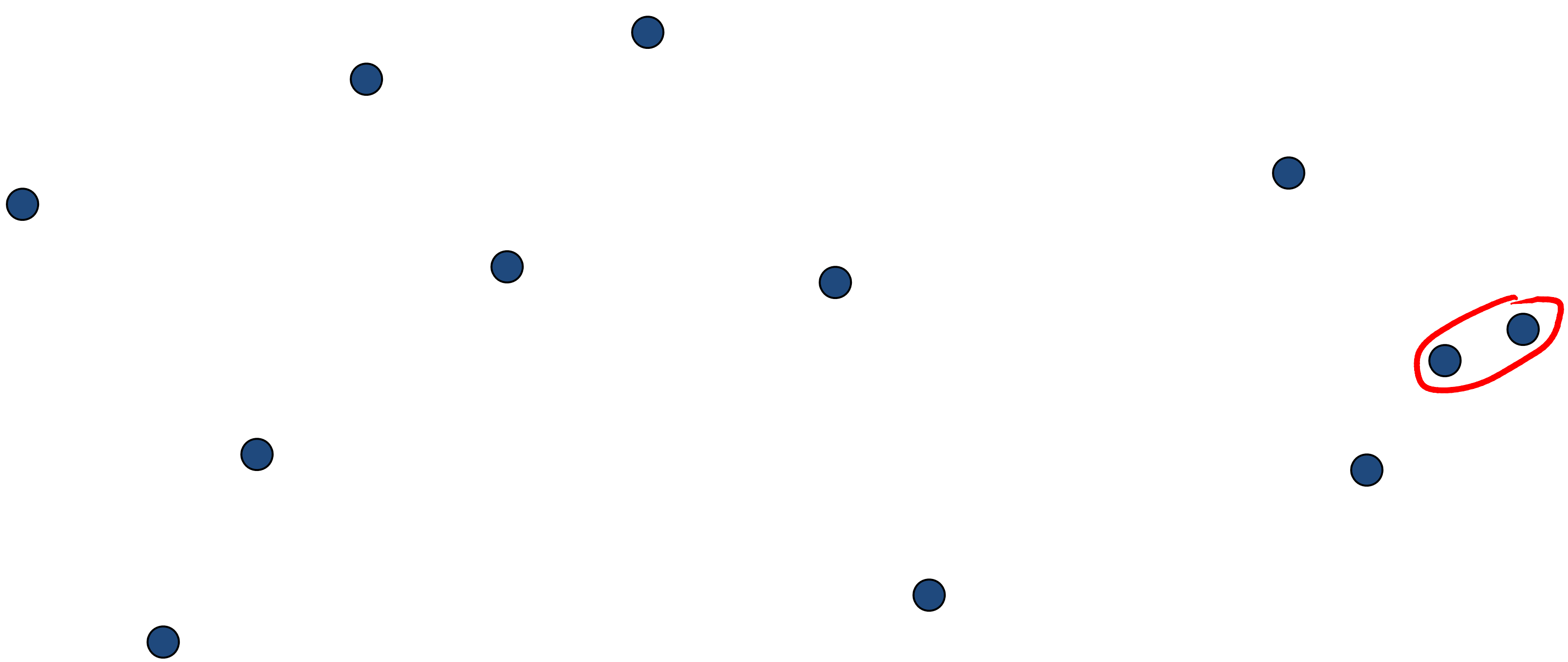
Sweep-Line Analysis

- Running time = $O(\text{\# events} \cdot \lg(\text{\# events}))$
- Number of endpoint events = $2n$
- Number of meet events =
  number k of intersections = size of output
- Running time = $O((n+k)\lg n)$

  $\lg(n+k)$
  $\leq \lg(n+n^2)$
  $= O(\lg n)$
- ***Output sensitive algorithm:***
  running time depends on size of output
- Best algorithm runs in $O(n \lg n + k)$ time

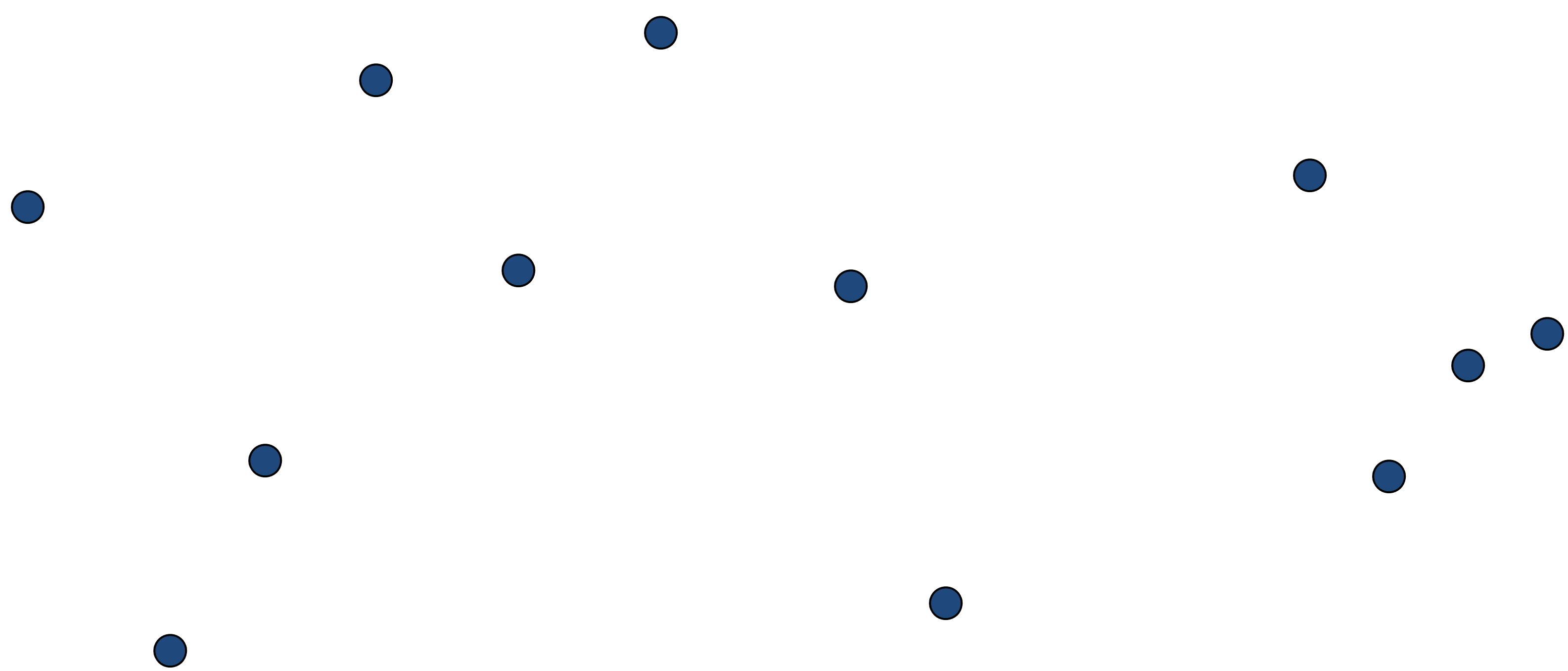
Closest Pair of Points

- Input: n points in 2D
- Goal: find two closest points



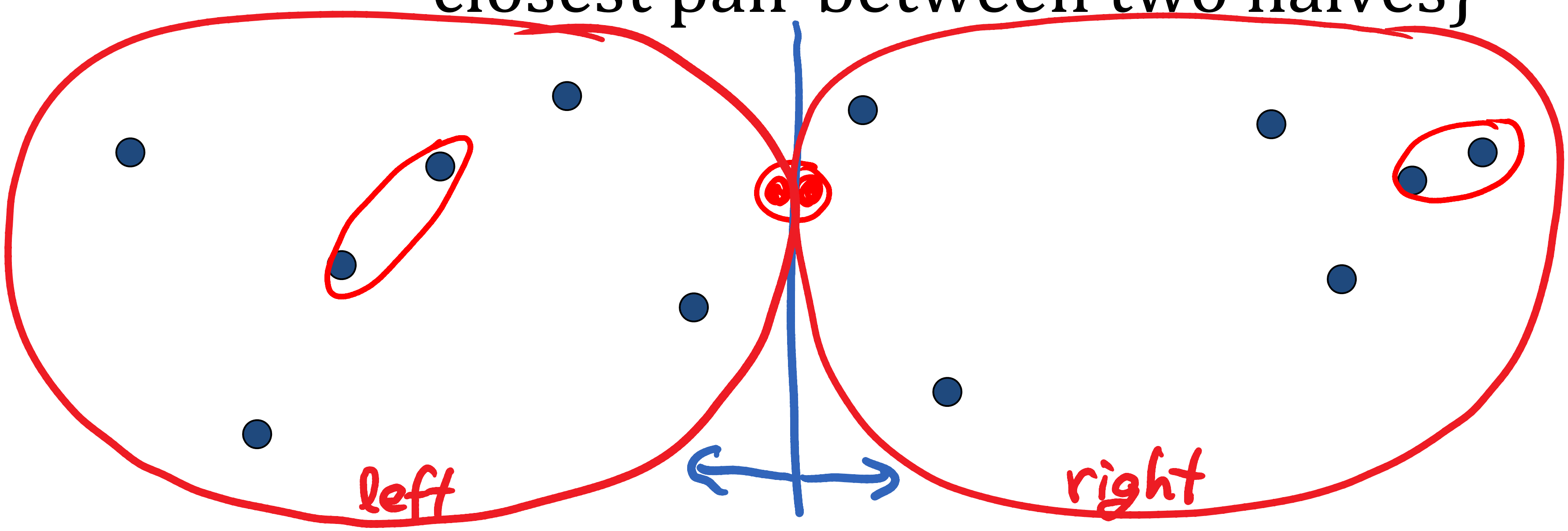
Obvious Algorithm

$$\left.\begin{array}{l}\min(\text{distance}(p, q)\\ \qquad\qquad \text{for every pair } (p, q) \text{ of points})\end{array}\right\} O(n^2)$$



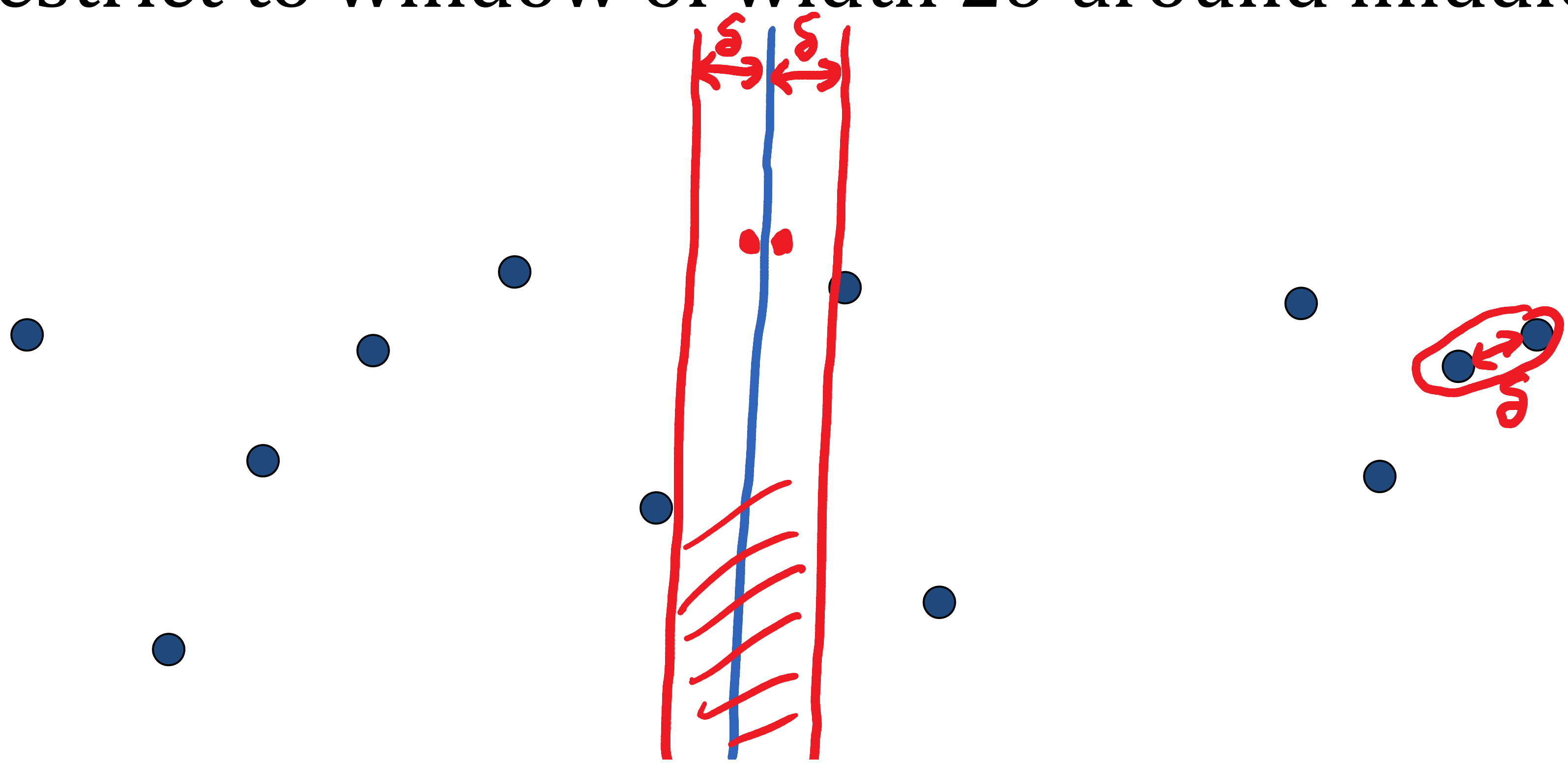
Divide-and-Conquer Idea

- Initially sort points by x coordinate
- Split points into left half and right half
- Recurse on each half: find closest pair
- return min{closest pair in each half,
  closest pair between two halves}



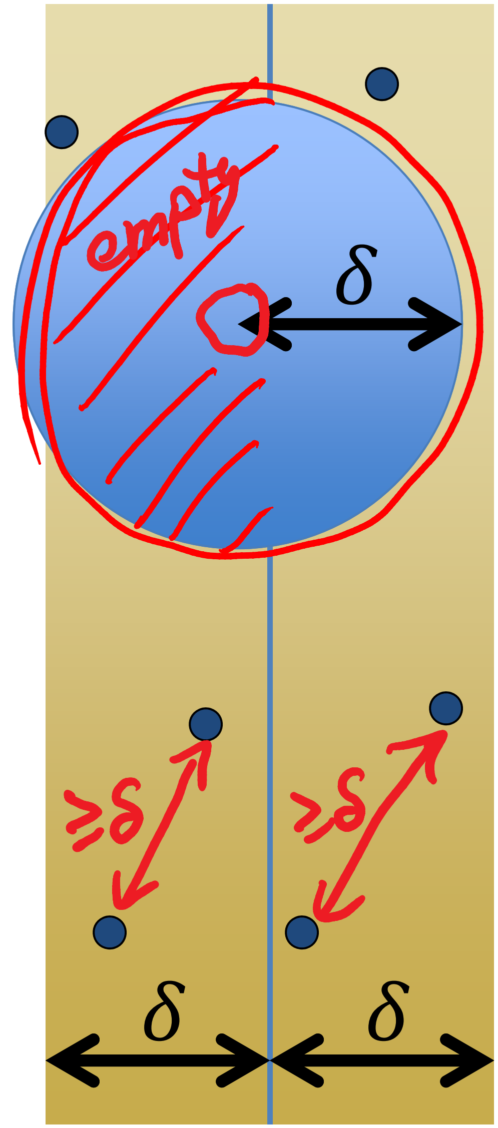
Closest Pair Between Two Halves?

- Let δ = min{closest pair in each half}
- Only interested in pairs with distance $< \delta$
- Restrict to window of width 2δ around middle

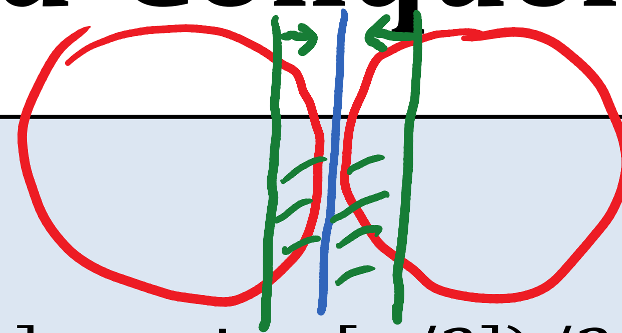


Closest Pair Between Two Halves

- For each left point, interested in points on right within distance δ
- Points on right side can't be within δ of each other
- So at most three right points to consider for each left point
  - Ditto for each right point
- Can compute in $O(n)$ time by merging two sorted arrays



Divide-and-Conquer



presort points by x

def closest−pair(points):

  middle $=$ (points$[n/2-1]$ $+$ points$[n/2]$)$/2$ $O(1)$

  δ $=$ min{closest−pair(points$[:n/2]$),
      closest−pair(points$[n/2:]$)} $2T(\frac{n}{2})$

  sort points[points.succ(middle $-\,\delta$)$: n/2$] by y
  sort points$[n/2:$ points.pred(middle $+\,\delta$)] by y $O(n \lg n)$

  merge and find closest pair between two lists $O(n)$

  return min{δ, closest distance from merge}

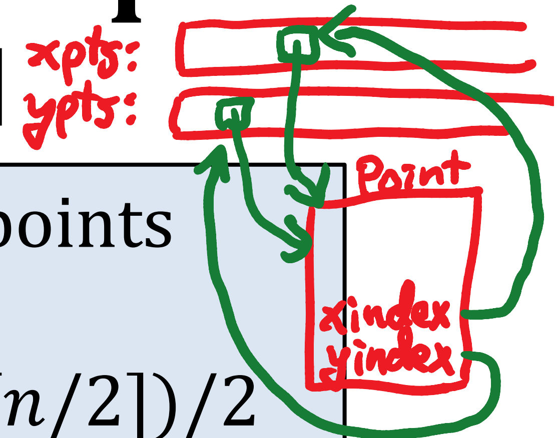
$$n \lg n$$

$$\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}\lg\frac{n}{2} = n\lg\frac{n}{2} = \boxed{n \lg n} - n$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \lg n)$$
$$= O(n \lg^2 n)$$

Faster Divide-and-Conquer

[Shamos & Hoey 1975]



presort points by x and y, and cross link points

def closest–pair(xpoints, ypoints):

  middle = (xpoints[$n/2 - 1$] + xpoints[$n/2$])/2

  δ = min{closest–pair(xpoints[: $n/2$], →ypoints),

        closest–pair(xpoints[$n/2$:], →ypoints)} $2T\left(\frac{n}{2}\right)$

  xpoints[xpoints.succ(middle − δ): $n/2$]

  xpoints[$n/2$: xpoints.pred(middle + δ)] $O(n)$

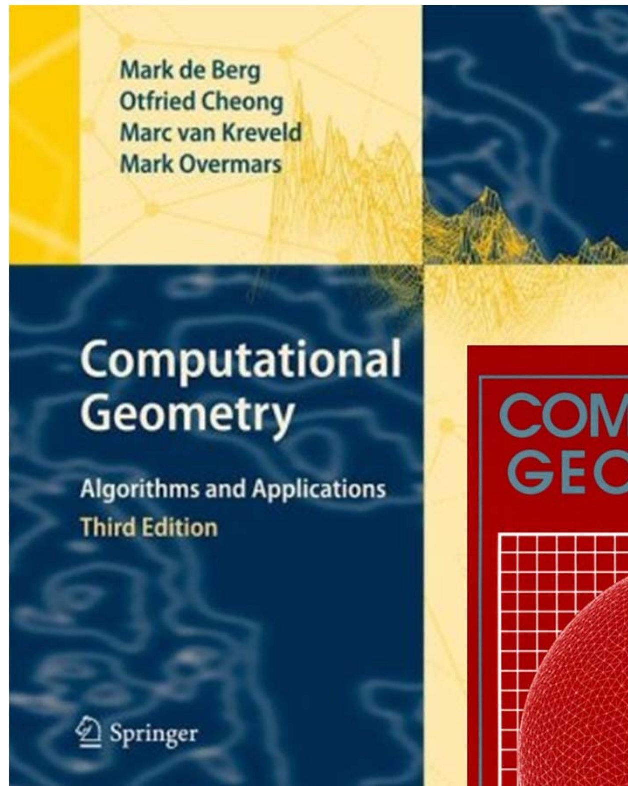
  map to ypoints & find closest pair between lists $O(n)$

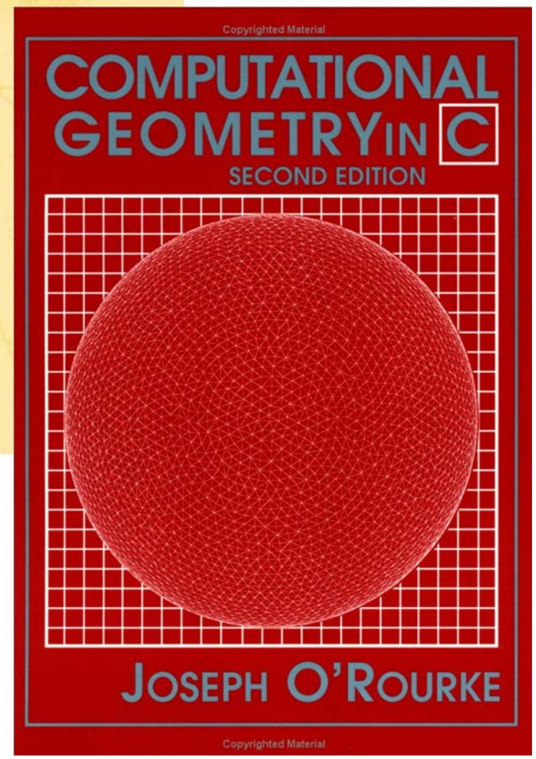
  return min{δ, closest distance from merge}

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$
$$= O(n \lg n)$$

Other Computational Geometry Problems

- Convex hull
- Voronoi diagram
- Triangulation
- Point location
- Range searching
- Motion planning
- ...





6.850: Geometric Computing