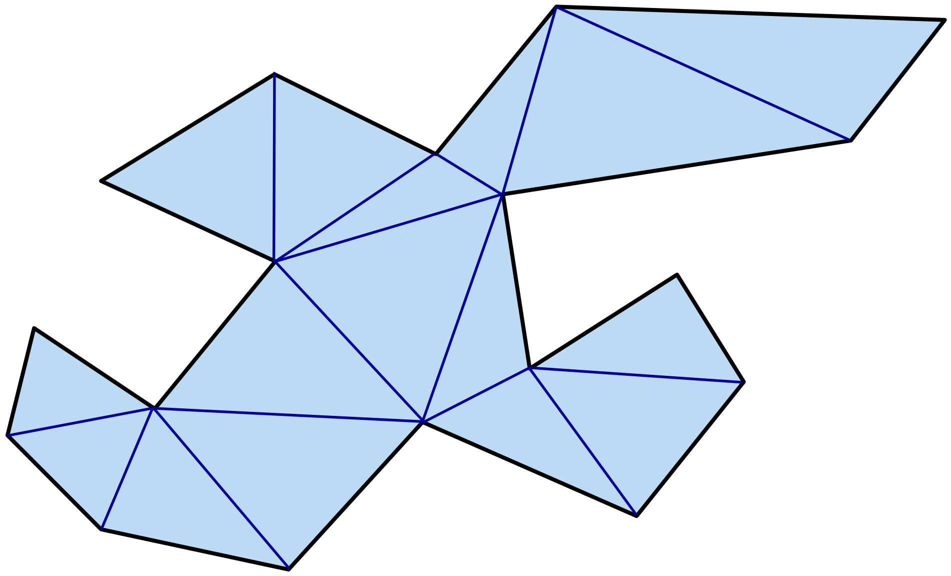
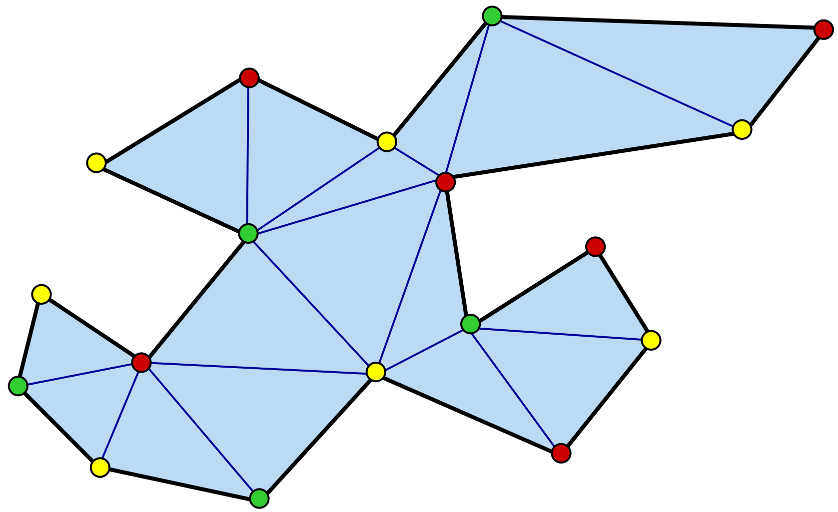
The sweep-line technique

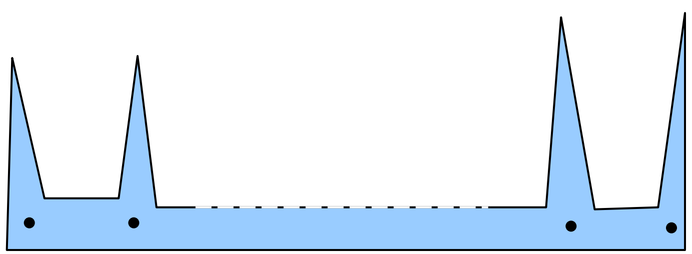
(Segment Intersection and Polygon Triangulation)



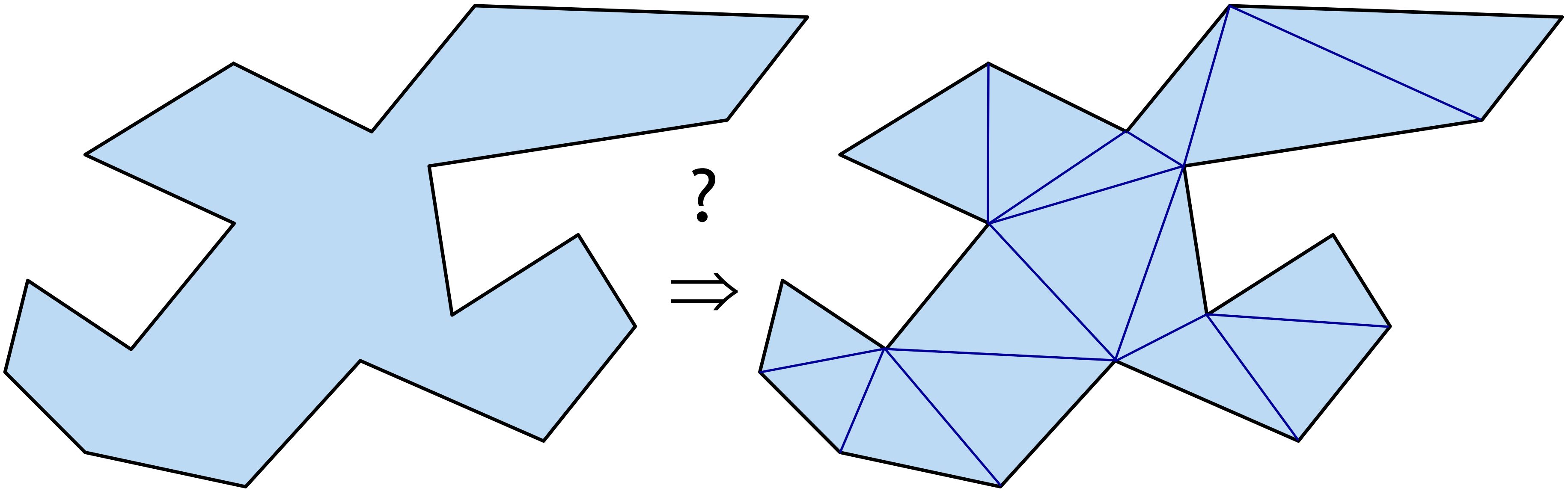
Recall

- Every simple polygon has a diagonal.
- Every simple polygon with n vertices can be decomposed into n-2 triangles.
- Every triangulated simple polygon can be 3-colourable, which can easily be computed greedily.
- Every simple polygon can **always** be "guarded" by $\lfloor n/3 \rfloor$ guards, and $\lfloor n/3 \rfloor$ guards is sometimes necessary.
- To find a guard set our algorithm requires a triangulation.





Ideas for a triangulation algorithm?



Triangulation algorithm 1

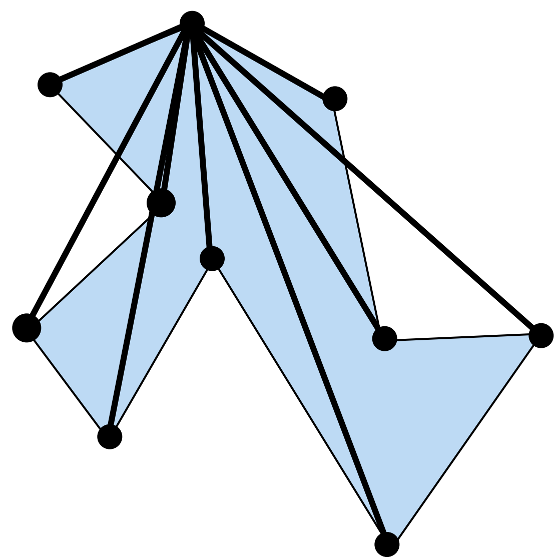
Theorem: Every polygon has a diagonal.

Algorithm 1:

while P not triangulated do

(x,y) := find_valid_diagonal(P)

output (x,y)



Number of potential diagonals? $O(n^2)$

Testing one potential diagonal? $O(n)$

Time complexity:

Test a diagonal $= O(n)$

#diagonals $= O(n^2)$

#iterations $= O(n)$ $\Rightarrow$ $O(n^4)$

Algorithm 1: running time

Assumption: 10^9 instructions per second

Input size: 1 million points = 10^6 points
$\Rightarrow$ running time ~ $n^4/10^9 = 10^{15}$ seconds ~ 32 million years

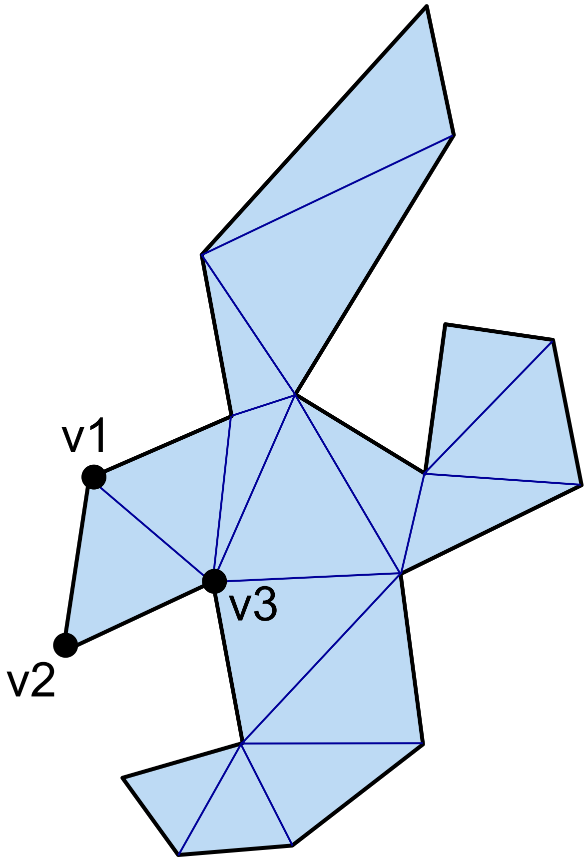
#points in 1 second: 180 points

Triangulation algorithm 2

Theorem: Every polygon has at least two non-overlapping ears.

Algorithm 2:

while n > 3 do	n-3
locate a valid ear tip v2	$O(n^2)$
output diagonal (v1,v3)	O(1)
delete v2 from P	O(1)
	Total: $O(n^3)$

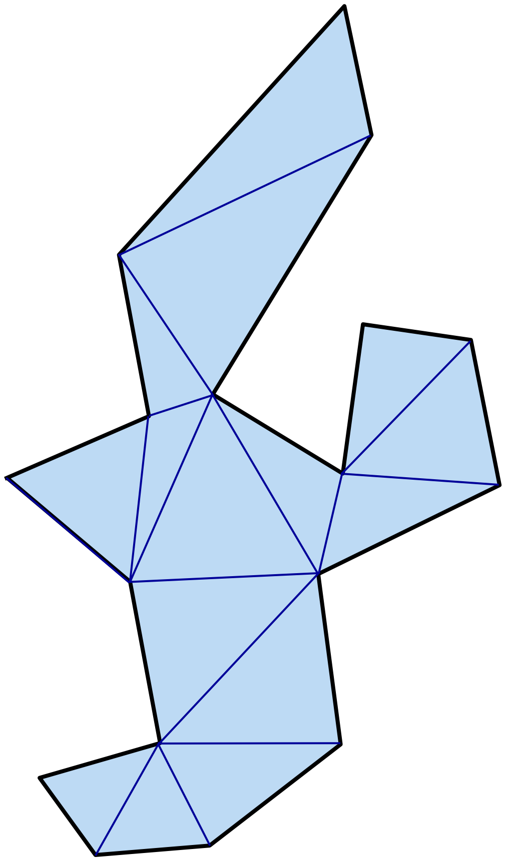


Triangulation algorithm 2

Theorem: Every polygon has at least two non-overlapping ears.

Algorithm 2:

while n > 3 do	n-3
locate a valid ear tip v2	$O(n^2)$
output diagonal (v1,v3)	O(1)
delete v2 from P	O(1)
	Total: $O(n^3)$



Algorithm 2: running time

Assumption: 10^9 instructions per second

Input size: 1 million points = 10^6 points

$\Rightarrow$ running time ~ $n^3/10^9 = 10^9$ seconds ~ 32 years

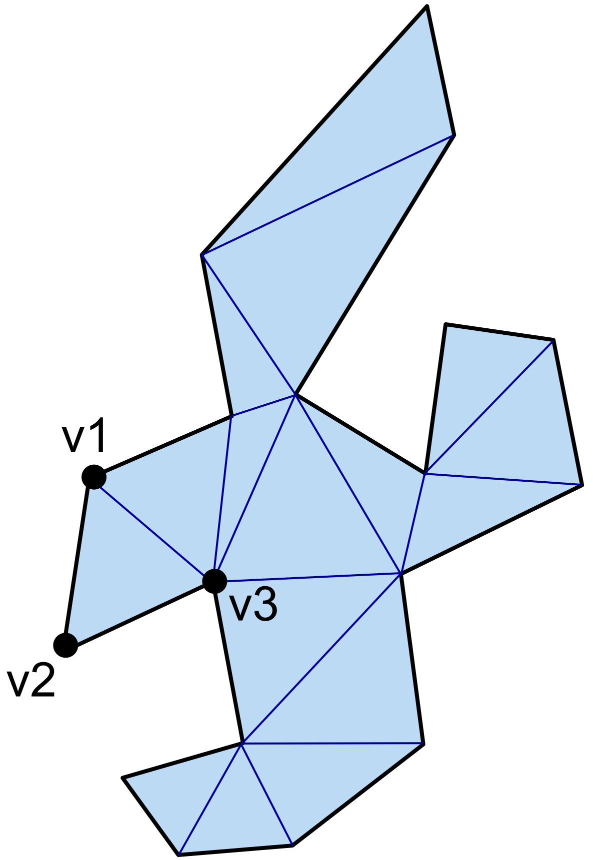
#points in 1 second: 1000 points

Why is triangulation algorithm 2 slow?

Can we speed up Algorithm 2?

Algorithm 2:

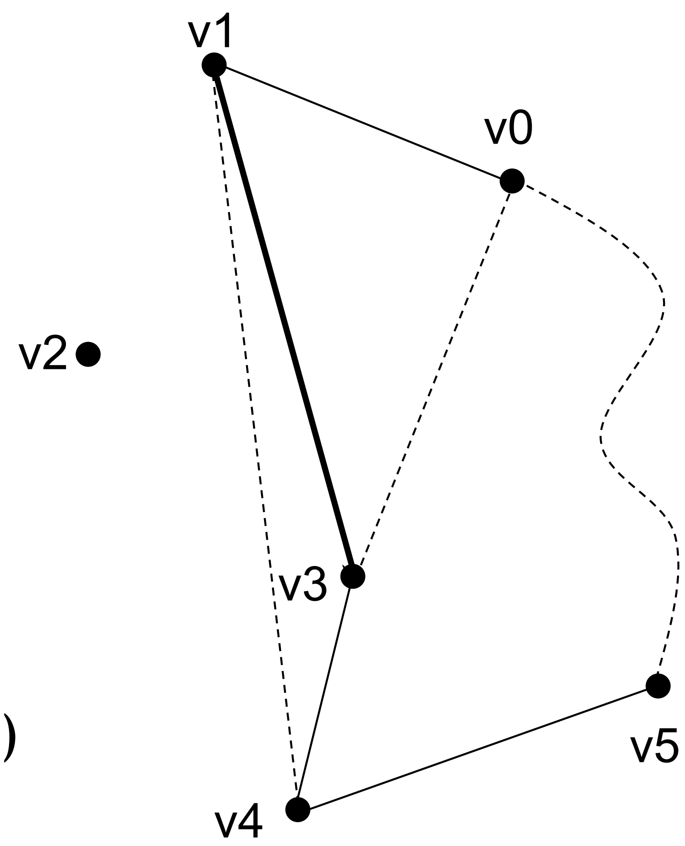
while n > 3 do	n-3
locate a valid ear tip v2	$O(n^2)$
output diagonal (v1,v3)	$O(1)$
delete v2 from P	$O(1)$
	Total: $O(n^3)$



Triangulation algorithm 3

Algorithm 3:

compute all valid ears S	$O(n^2)$
while n > 3 do	n-3
locate a valid ear tip v2	O(1)
output diagonal (v1,v3)	O(1)
delete v2 from P	O(1)
delete (v0,v1,v2) from S	O(n)
delete (v2,v3,v4) from S	O(n)
check ear (v1,v3,v4)	O(n)
check ear (v0,v1,v3)	O(n)
	Total: $O(n^2)$



Algorithm 3: running time

Assumption: 10^9 instructions per second

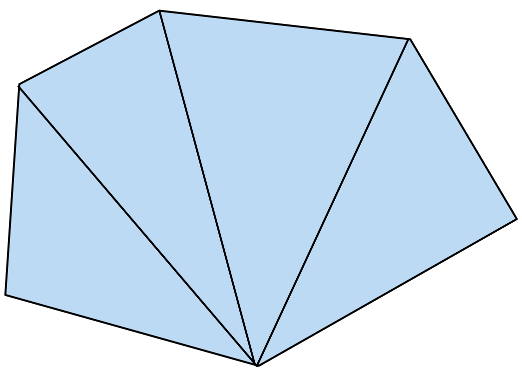
Input size: 1 million points = 10^6 points

$\Rightarrow$ running time ~ $n^2/10^9 = 10^3$ seconds ~ 17 minutes

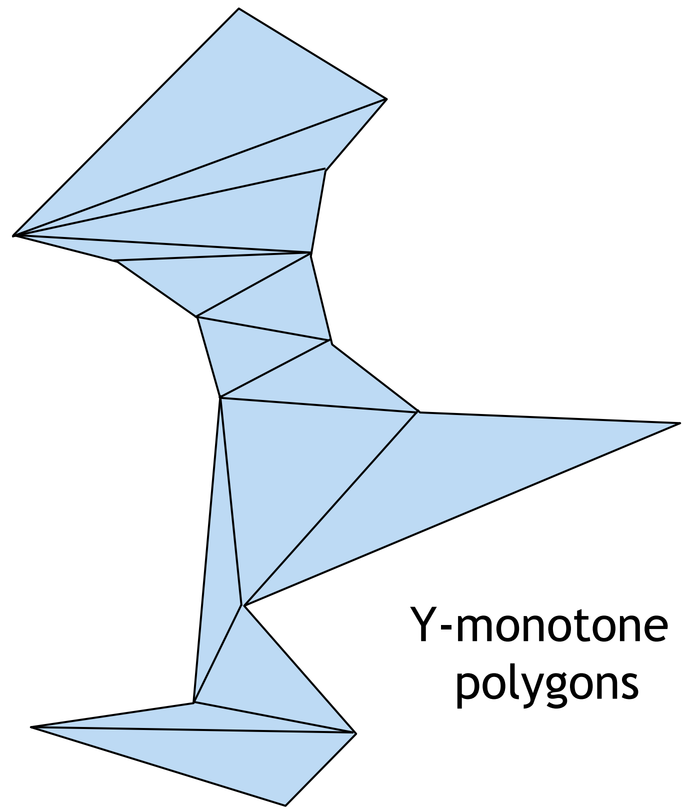
#points in 1 second: 30,000 points

Triangulation algorithm 4

Observation: Some polygons are very easy to triangulate.



Convex polygons



Y-monotone polygons

Polygon triangulation 4

Algorithm 4:

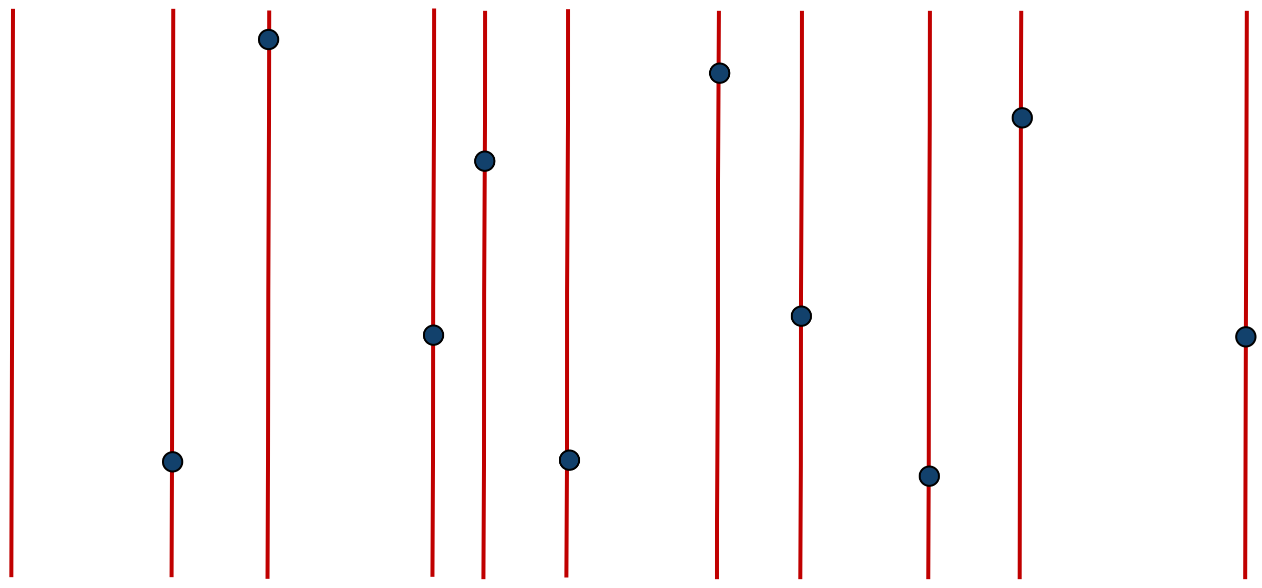
Partition P into y-monotone pieces.	O(n log n)
Triangulate every y-monotone polygon	O(n)

We're going to develop sweepline algorithms for both problems

Theorem: Every simple polygon can be triangulated in O(n log n) time.

Detour into sweepline algorithms

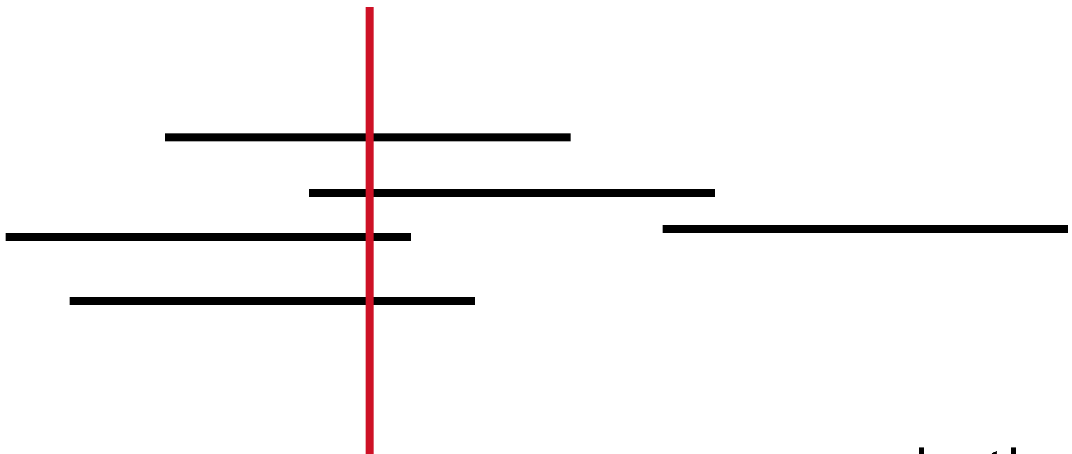
(we'll get back to polygon triangulation later in the lecture)



Depth of interval

Given a set S of n intervals (in 1D) compute the depth of S.

The depth of S is the maximum number of intervals passing over any point.

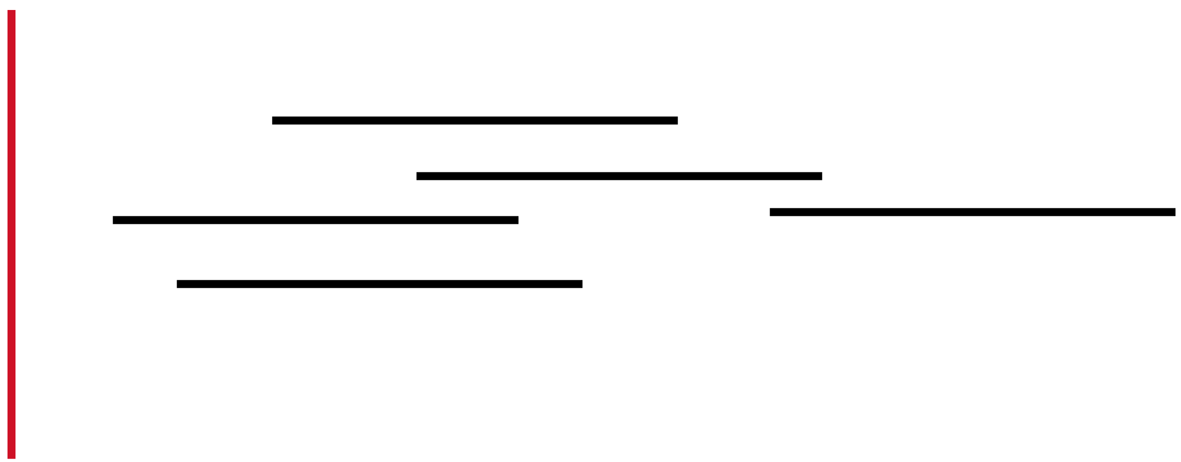


$\Rightarrow$ depth=4

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine "sweeping" a vertical line from left to right while maintaining the current depth.

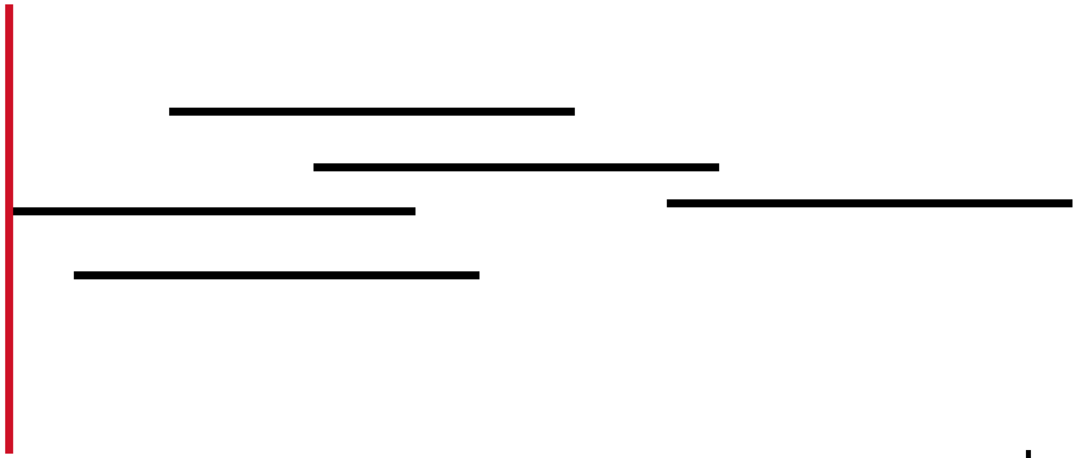


depth = 1

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine "sweeping" a vertical line from left to right while maintaining the current depth.

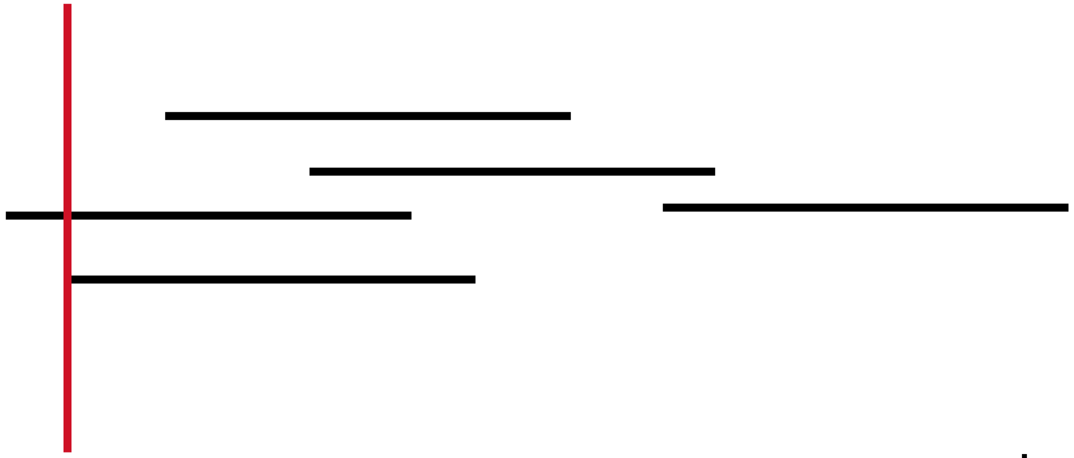


depth = 2

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine “sweeping” a vertical line from left to right while maintaining the current depth.

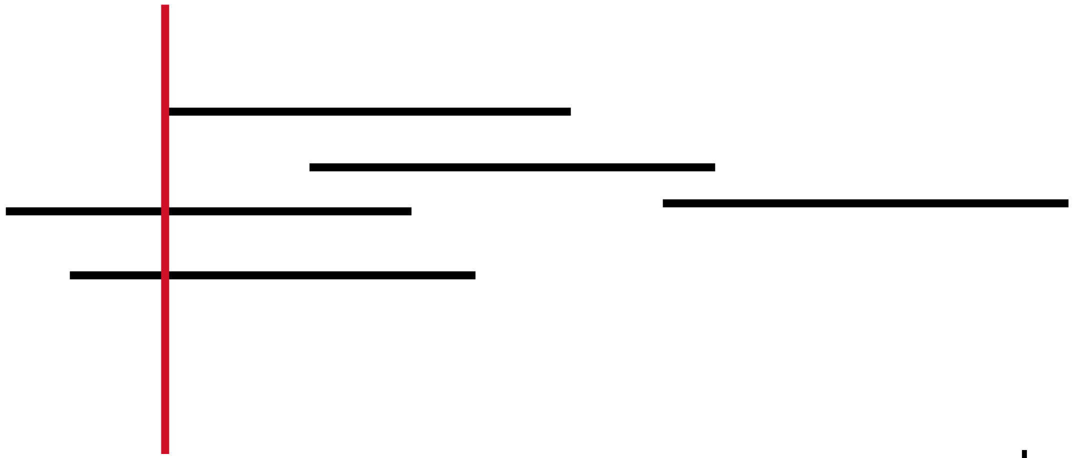


depth = 3

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine "sweeping" a vertical line from left to right while maintaining the current depth.

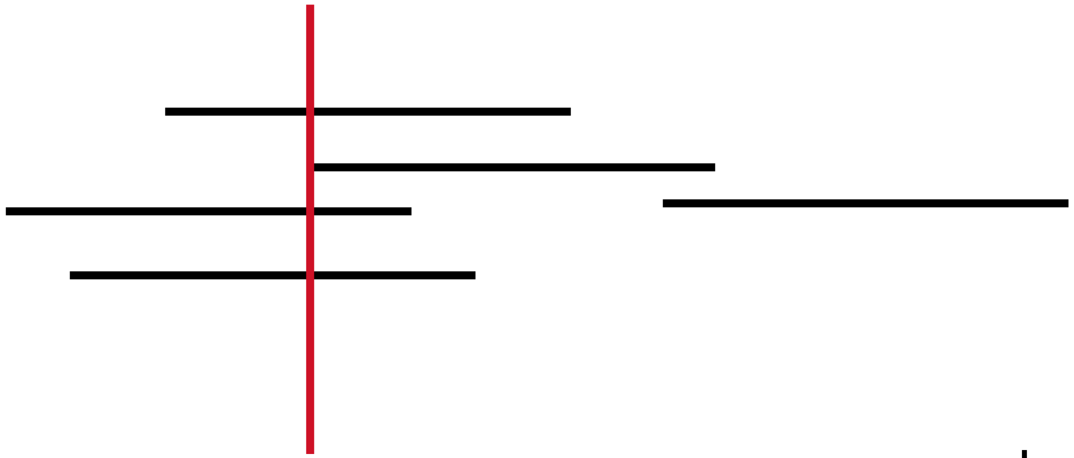


depth = 4

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine "sweeping" a vertical line from left to right while maintaining the current depth.

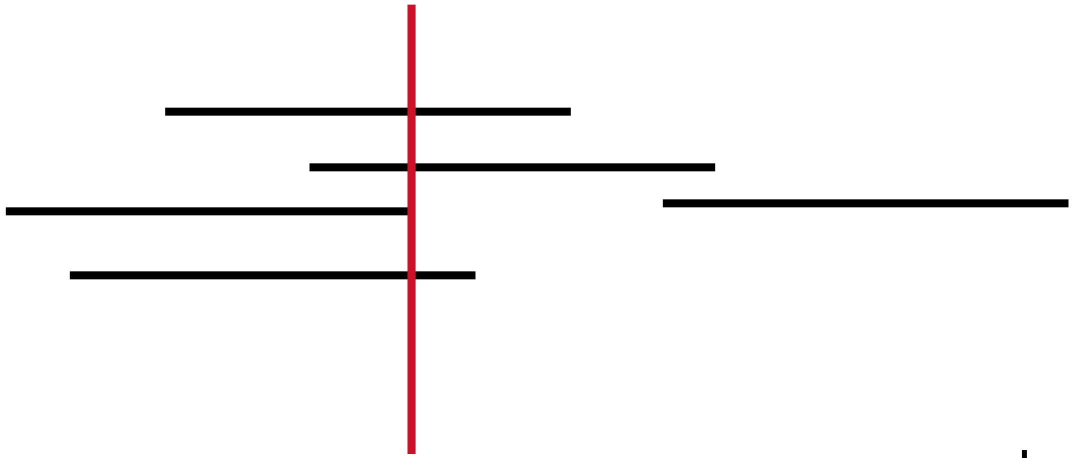


depth = 3

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine “sweeping” a vertical line from left to right while maintaining the current depth.

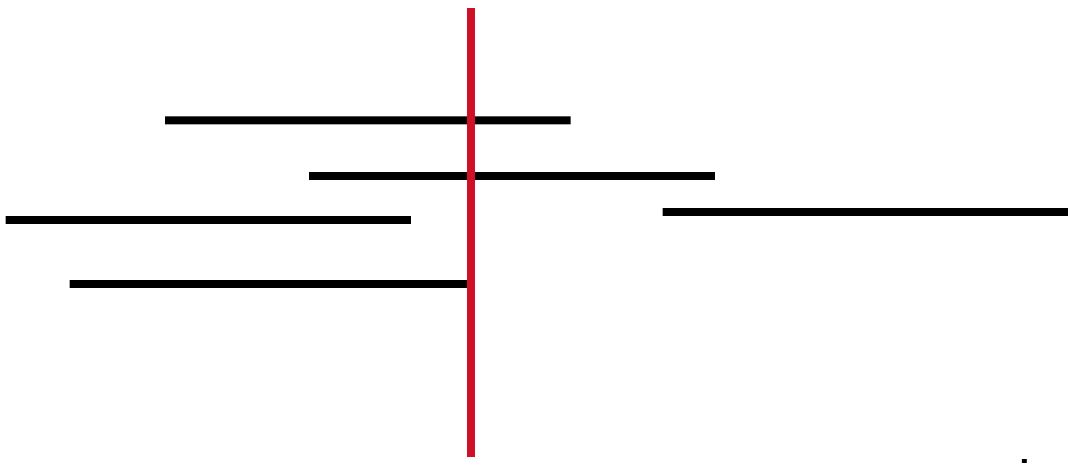


depth = 2

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine "sweeping" a vertical line from left to right while maintaining the current depth.

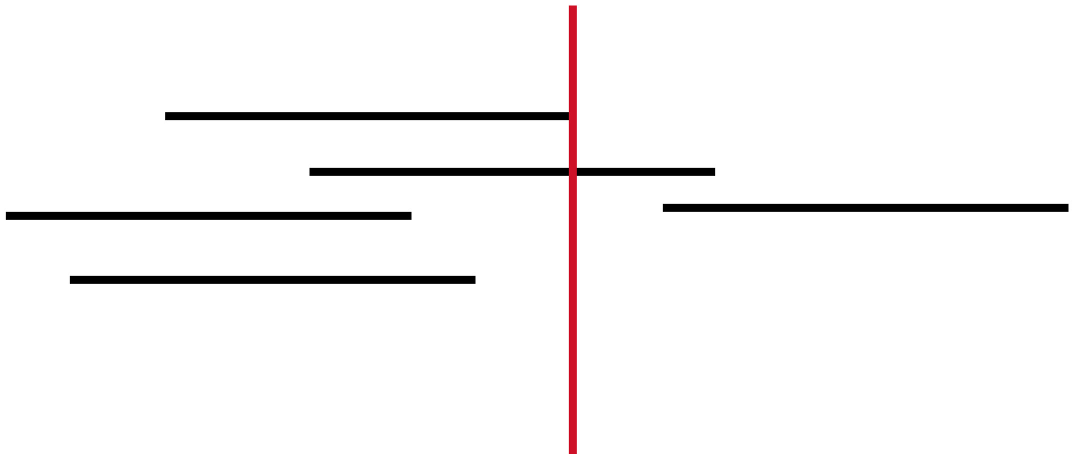


depth = 1

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine "sweeping" a vertical line from left to right while maintaining the current depth.

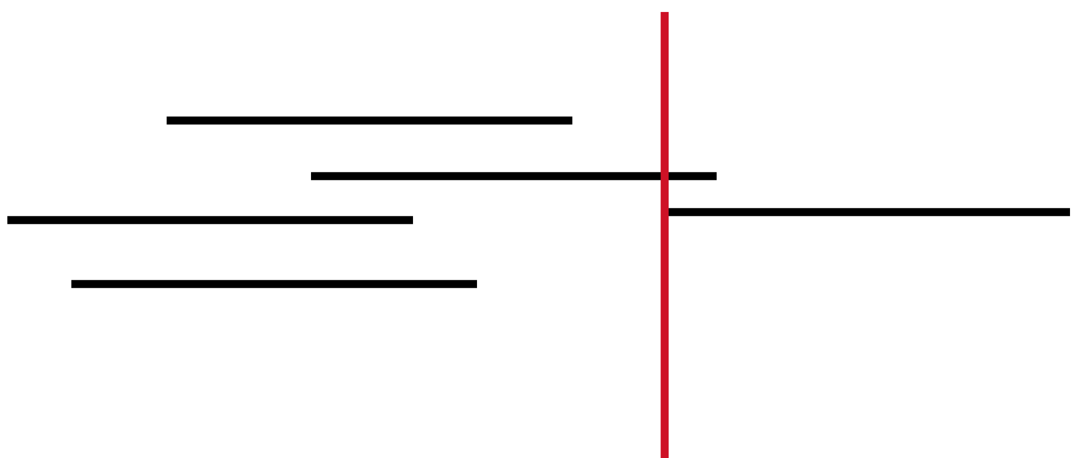


depth = 2

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine “sweeping” a vertical line from left to right while maintaining the current depth.

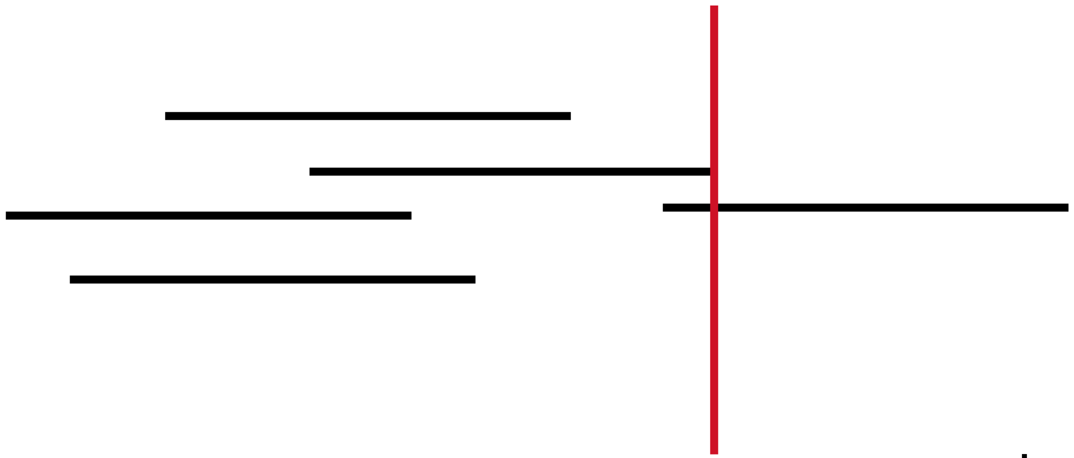


depth = 1

Depth of interval

How can we compute the depth?

The problem can be solved in O(n log n) using a sweepline approach. Imagine "sweeping" a vertical line from left to right while maintaining the current depth.



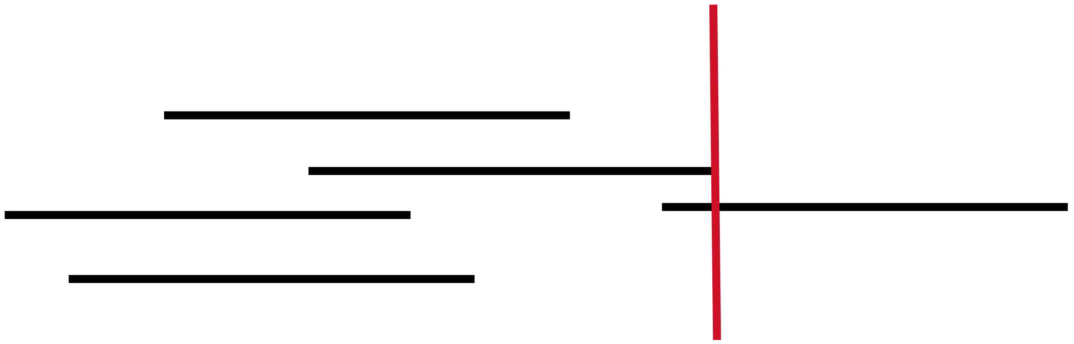
depth = 0

Depth of interval

If we can keep track of the current depth then we can easily also find the maximal depth.

The points where a change of depth may occur are called the event points
$\Rightarrow$ endpoints of the intervals

The sweepline status is the information stored with the sweepline
$\Rightarrow$ current depth and largest depth to the left of the line.



Depth of interval

1. Sort endpoints from left to right $p_1, \ldots, p_{2n}$ O(n log n)
2. currentDepth=0
3. maxDepth=0
4. for i=1 to 2n do

```
if p_i is left endpoint then
    currentDepth = currentDepth + 1
    if maxDepth < currentDepth then
        maxDepth = currentDepth
else  {if p_i is a right endpoint}
    currentDepth = currentDepth – 1
```

5. end {for}
6. Report maxDepth

O(n)

Summary: Depth of intervals

Theorem:

The depth of a set of n intervals in 1D can be computed in O(n log n) time using a sweepline algorithm.

Sweepline

Main idea:

Sweep an “imaginary" line L across the plane while

(1) maintaining the status of L and [current depth]

(2) fulfilling an invariant. [the maximum depth to the left of L has been computed]

› The status of L only changes at certain discrete event points.
[endpoints of segments]

› When the sweep line encounters an event point the status is updated in such a way that the invariant is guaranteed to hold after the event point has been processed.
[updating the depth counter]

Prove correctness

Correctness usually follows immediately from the invariant and the event points.

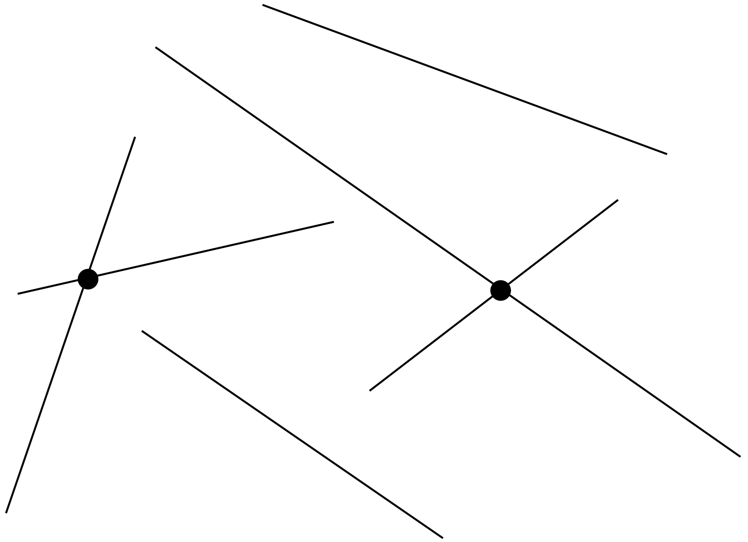
1) Prove that the status can't change between two consecutive event points and

   [if event points are correctly chosen this is usually trivial]

2) prove that the invariant holds before and after an event point is processed.

   [depth counter correct before new event and after an event has been processed]

Segment intersection

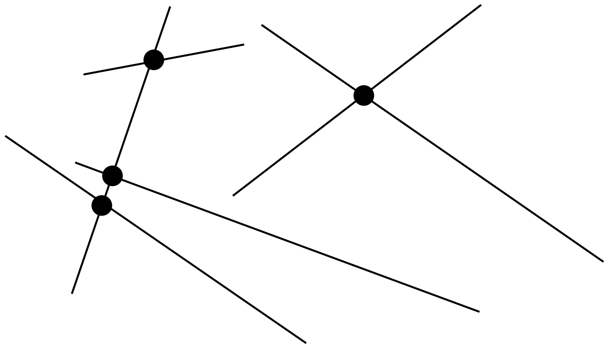


Segment intersection

Input: A set of n line segments $S=\{s_1, s_2, \ldots, s_n\}$ in the plane, represented as pairs of endpoints.

Intersection detection:
Is there a pair of segments in S that intersect?

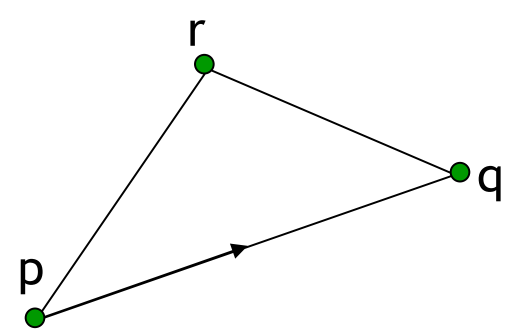
Intersection reporting:
Find all pairs of segments that intersect.



Check left turn a primitive?

How can we check if a point r lies to the left of a line pq?

$\Rightarrow$ Triangle $\Delta(p,q,r)$ is oriented counter-clockwise.



$p=(p_x,p_y)$, $q=(q_x,q_y)$ and $r=(r_x,r_y)$

$$D(p,q,r) = \begin{vmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{vmatrix}$$

$$= (q_x - p_x)(r_y - p_y) - (r_x - p_x)(q_y - p_y)$$

[2 multiplications, 5 subtractions]

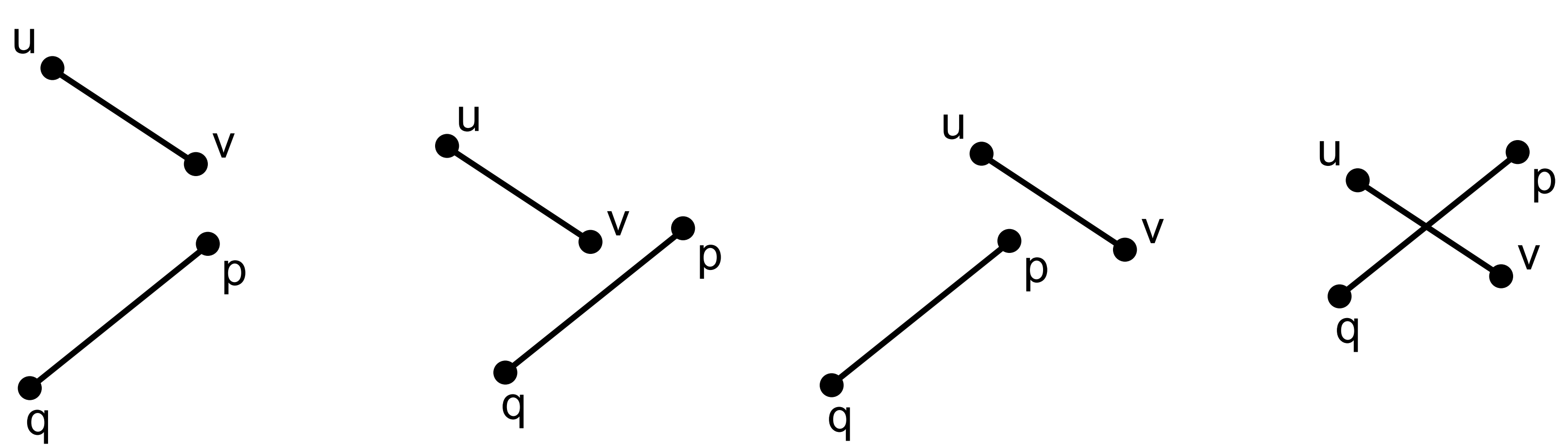
$\Delta(p,q,r)$ is oriented counter-clockwise iff $D(p,q,r) > 0$.

CCW(p,q,r) = true if D(p,q,r)>0 otherwise false

Testing intersection

Test if two segments (p,q) and (u,v) intersect.

```
boolean INTERSECT(Points u, v, p, q)
    return [(CCW(u, v, p) xor CCW (u, v, q)) and
            (CCW(p, q, u) xor CCW (p, q, v))]
```

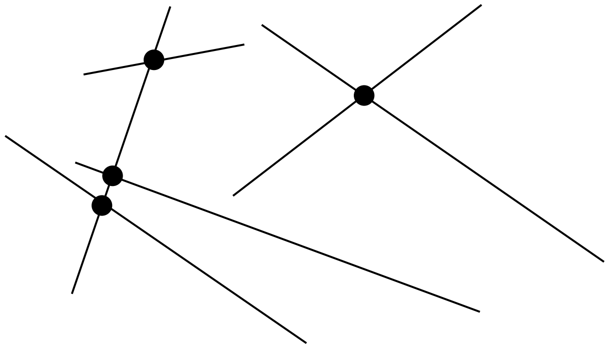


Segment intersection

Input: A set of n line segments $S=\{s_1, s_2, \ldots, s_n\}$ in the plane, represented as pairs of endpoints.

Intersection detection:
Is there a pair of lines in S that intersect?

Intersection reporting:
Find all pairs of segments that intersect.



Brute force algorithm

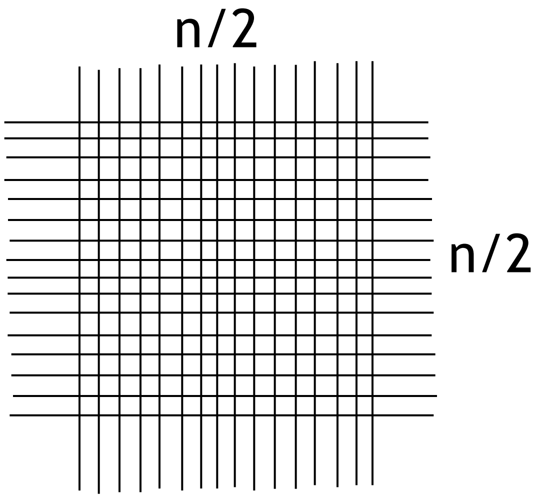
Check every possible pair of segments if they intersect

$\Rightarrow$ $O(n^2)$ time

Can we do better?

Detection? Maybe!

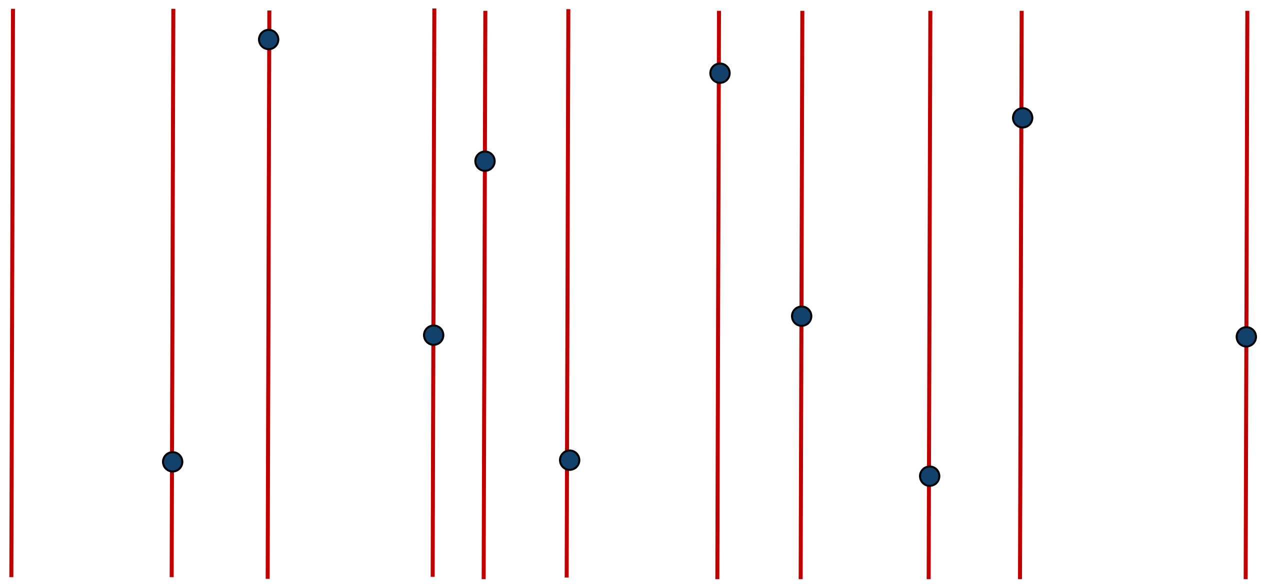
Reporting? Nope!



However, we can try to make the running time sensitive to the size of the output (h).

Design technique

- Simulate sweeping a vertical line from left to right across the plane.
- Events: Discrete points where sweepline status needs to be updated
- Sweepline status: Store information with sweepline
- Maintain invariant: At any point in time, to the left of sweep line everything has been properly processed.



Design technique

- Simulate sweeping a vertical line from left to right across the plane.
- Events: Discrete points where sweepline status needs to be updated
- Sweepline status: Store information with sweepline
- Maintain invariant: At any point in time, to the left of sweep line everything has been properly processed.

Algorithm Generic_Plane_Sweep:

Initialize **sweep line status** S at time $x=-\infty$

Store initial events in **event queue** Q, a priority queue ordered by x-coordinate

while $Q \neq \emptyset$

  // extract next event e:

  e = Q.extractMin();

  // handle event:

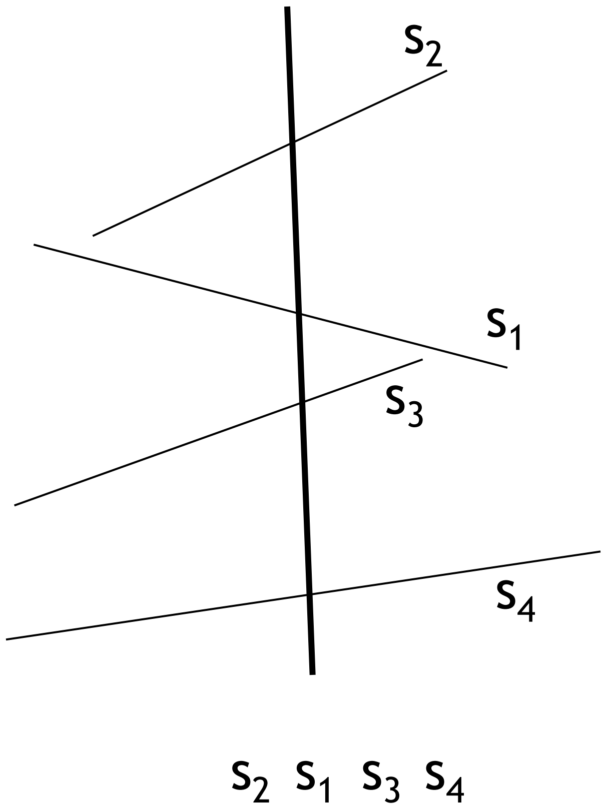
  Update sweep line status

  Discover new upcoming events and insert them into Q

Plane sweep algorithm: intersection detection

Plane sweep (general method):

1. Sweep the input from left to right and stop at event points
2. Maintain invariant (status and structure)
3. At each event point restore invariant



Invariant:

› The order of the segments along the sweep line
› No intersections to the left of the sweepline

Event points? (ignore intersections for now)

end points of the segments

Plane sweep algorithm

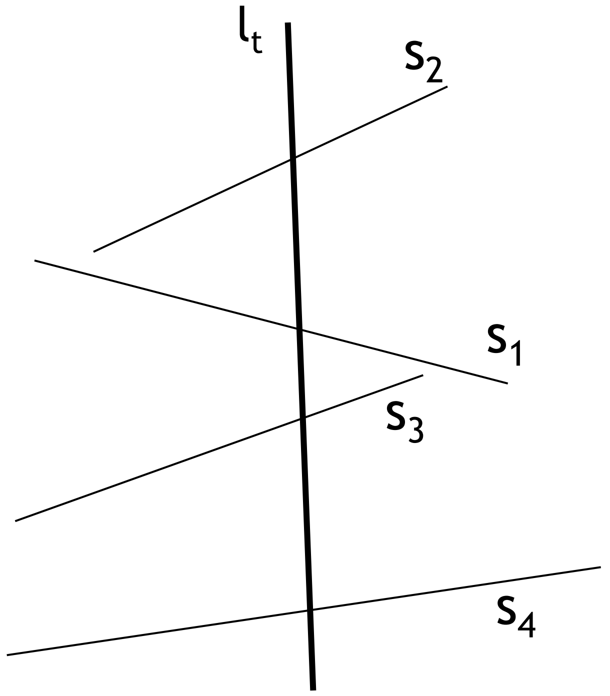
l_t : the vertical line at x=t
S_t : the sequence of the segments that intersects l_t in order from top to bottom.

Idea:

Maintain S_t while l_t moves from left to right

Invariant:

- We know S_t
- No intersections to the left of L

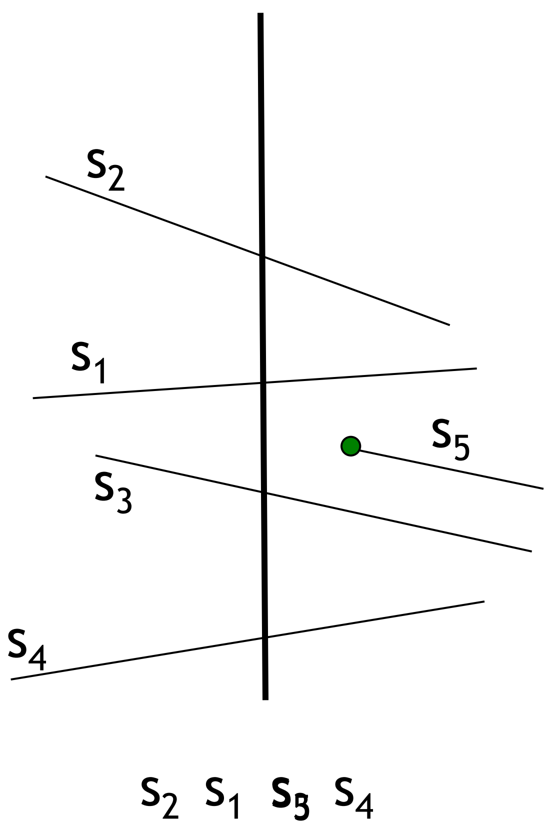


S_t: s_2 s_1 s_3 s_4

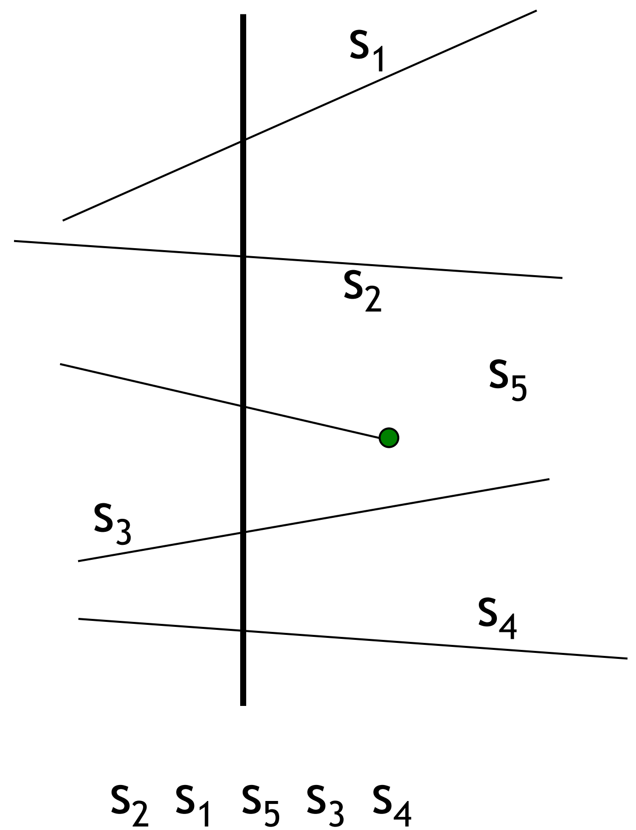
Event handling

Initially: Let t_1, t_2, … , t_{2n} be the x-coordinates of the endpoints

Case 1: t_{i+1} is a left end point



Case 2: t_{i+1} is a right end point



Data structure

We need to store S_t in a data structure that supports fast insertions and deletions.

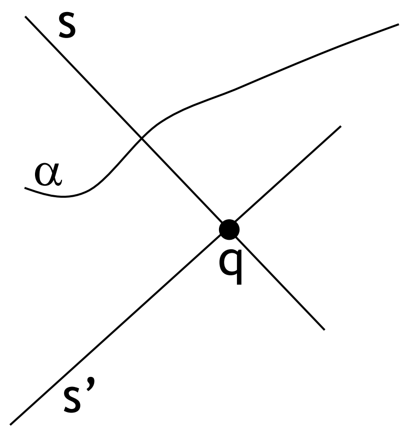
Structure: Balanced binary search trees
Each update can be done in O(log n) time

Problem: We did not check intersections! How can we look for intersections?

Intersection points

Observation: Let q be the leftmost intersection point, where q is an intersection point between the segments s and s' with x-coordinate t then s and s' are adjacent in S_t.

Proof:
Assume the opposite, i.e., S_t = (….s…α…s'…)

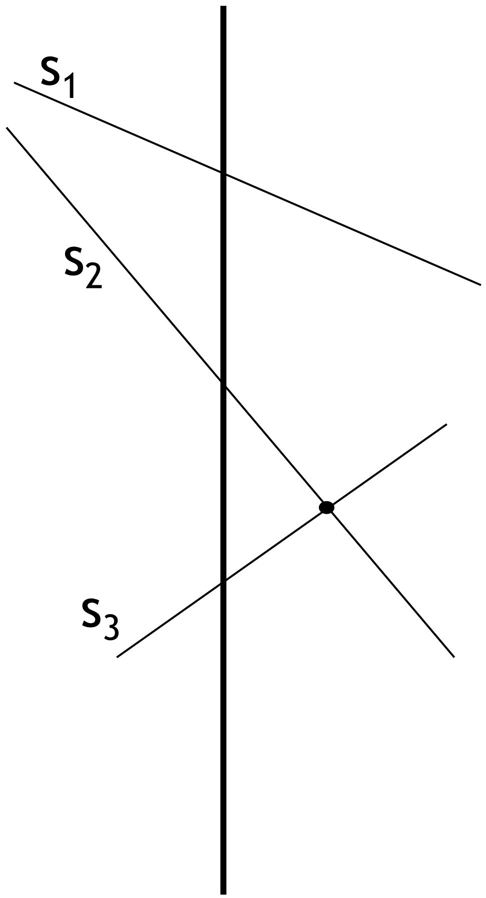


1. Right endpoint of α must be to the right of q.
2. If q below α then α intersects s to the left of q.
   $\Rightarrow$ contradicts that q is leftmost intersection
3. Similarly, q cannot lie above α

$\Rightarrow$ contradiction!

Intersection point

Conclusion: To detect an intersection we only need to check adjacent segments in S_t.



Algorithm

Algorithm DetectIntersection(S)

1. Store the segments S_t in a balanced binary search tree T w.r.t. the order along l_t.

2. When deleting a segment in T two segments become adjacent. We can find them in O(log n) time and check if they intersect.

3. When inserting a segment s_i in T it becomes adjacent to two segments. We can find them in O(log n) time and check if they intersect s_i.

4. If we find an intersection we're done!

Time complexity?

Algorithm - detection

Every endpoint is an event point $\Rightarrow$ 2n event points

Insert segment s

Add s to T:	O(log n)
Check neighbours:	$2 \times$ O(log n)

Delete segment s

Remove s from T:	O(log n)
Check new neighbours:	$2 \times$ O(log n)

Total: O(n log n)

Intersection reporting

How can we change the algorithm to report all intersections?

Event points = endpoints plus intersection points

Where does the order along l_t change?

With the new event points we can run the algorithms as before (with minor modifications).

Running time: $O(n \log n + h \log n)$

Lower bound

Can we do better that $O(n \log n)$?

Element Uniqueness problem $\Omega(n \log n)$
Given a set of real numbers, are they distinct?

Element Uniqueness is a simpler version than our problem

Segment intersection

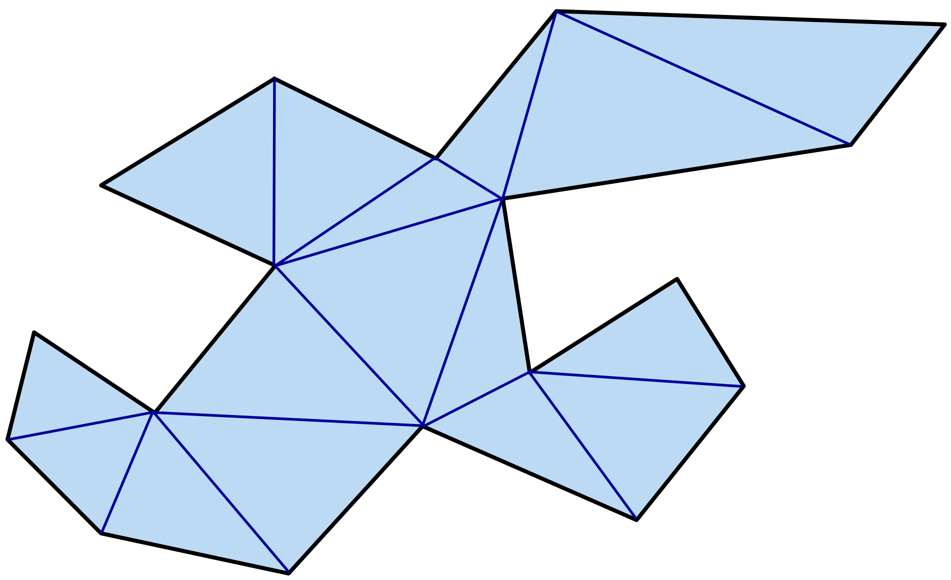
Sweep-line technique

[Shamos & Hoey’75], [Lee & Preparata’77], [Bentley & Ottman’79]

Intersection reporting

- $O(n \log n + h \log n)$ time [Bentley & Ottmann’79]
- $O(n \log^2 n / \log\log n + h)$ [Chazelle’86]
- $O(n \log n + h)$ [Chazelle & Edelsbrunner’88]
- $O(n \log n + h)$ [Balaban’95] (also works for curves)

Back to polygon triangulation



Polygon triangulation 4

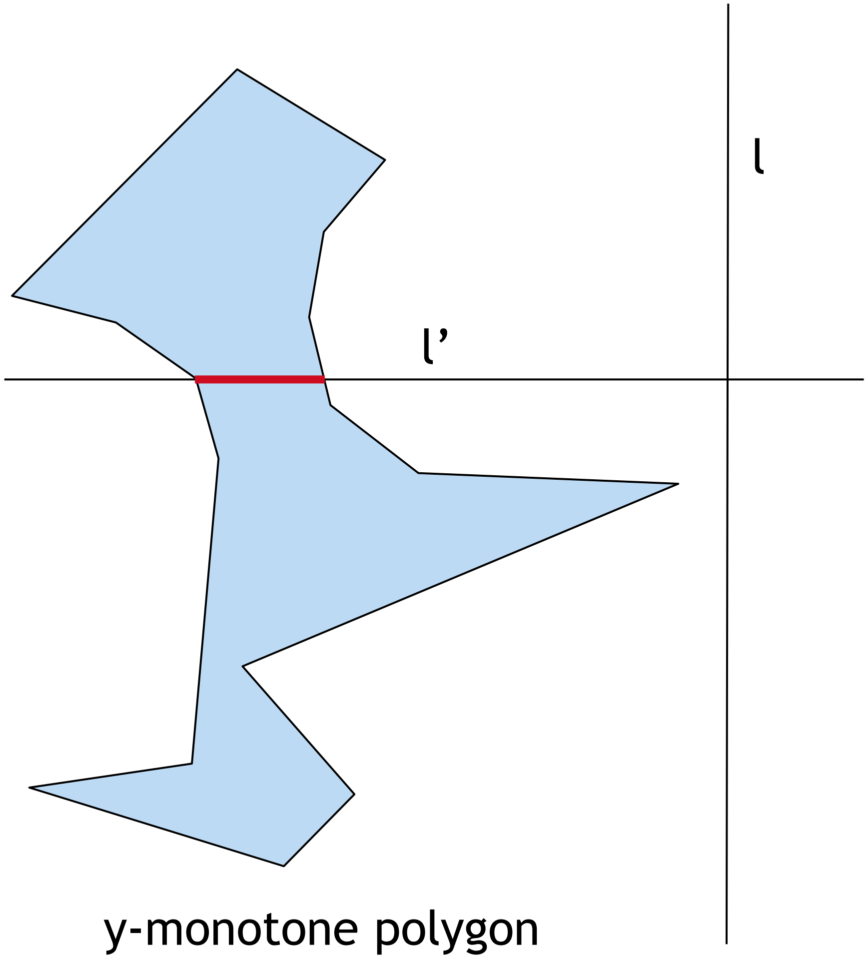
Algorithm 4:

Partition P into y-monotone pieces.	O(n log n)
Triangulate every y-monotone polygon	O(n)

Theorem: Every simple polygon can be triangulated in O(n log n) time.

l-monotone polygon

A simple polygon is called monotone w.r.t. a line l if for any line l' perpendicular to l, the intersection of the polygon with l' is connected (y-monotone, if l = y-axis).



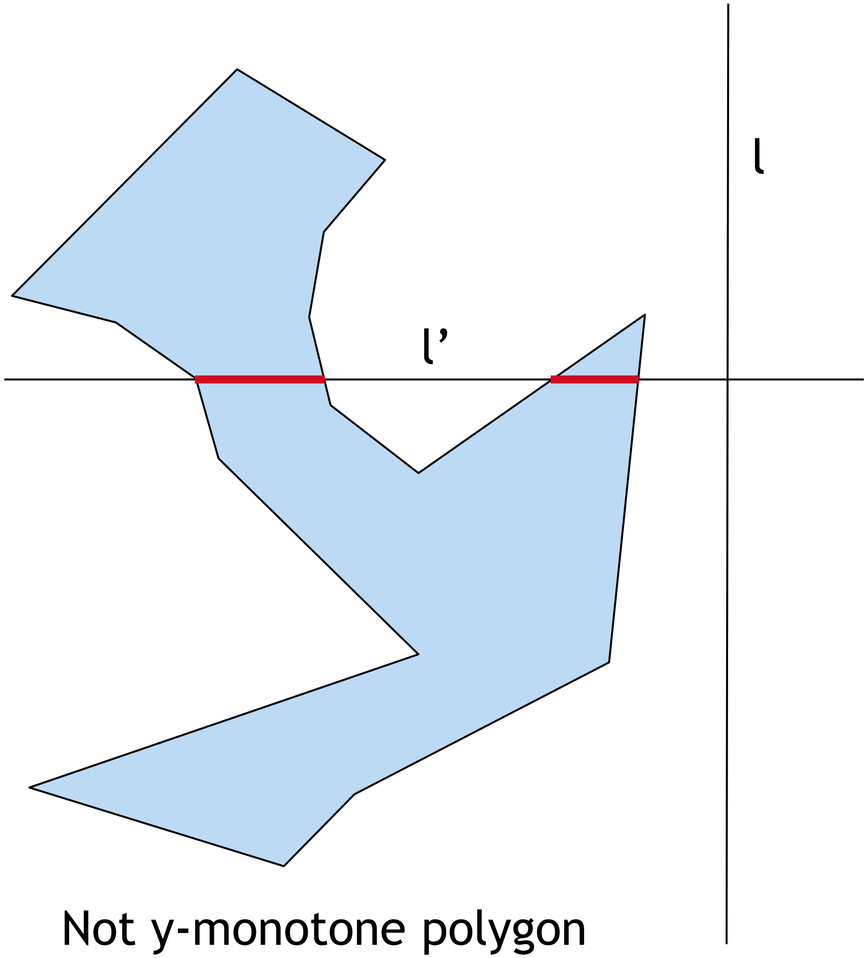
y-monotone polygon

Observation: if P is l-monotone then P consists of two l-monotone chains.

l-monotone polygon

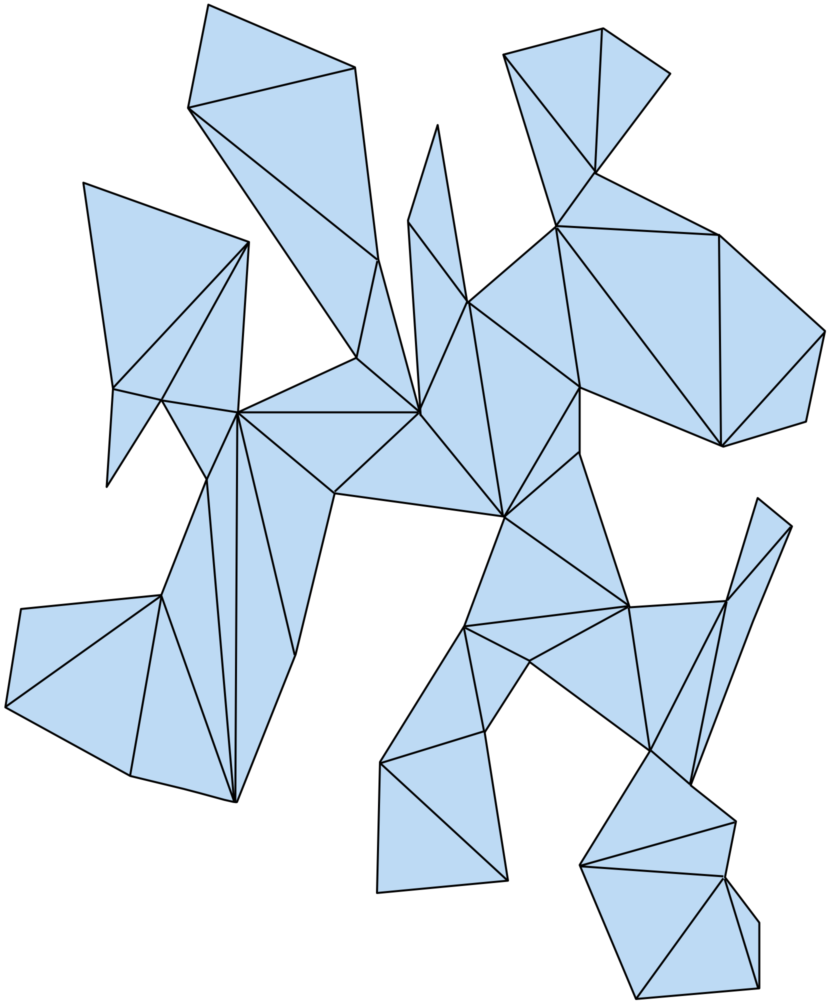
A simple polygon is called monotone w.r.t. a line l if for any line l' perpendicular to l, the intersection of the polygon with l' is connected (y-monotone, if l = y-axis).

Observation: if P is l-monotone then P consists of two l-monotone chains.



Not y-monotone polygon

Triangulate simple polygon



Main idea:

1. Partition P into y-monotone polygons

   Time O(n log n)

2. Triangulate each y-monotone polygon

   Time $O(n_i)$

Prove:

A simple polygon can be triangulated in O(n log n) time.

Triangulate monotone polygon

Idea:

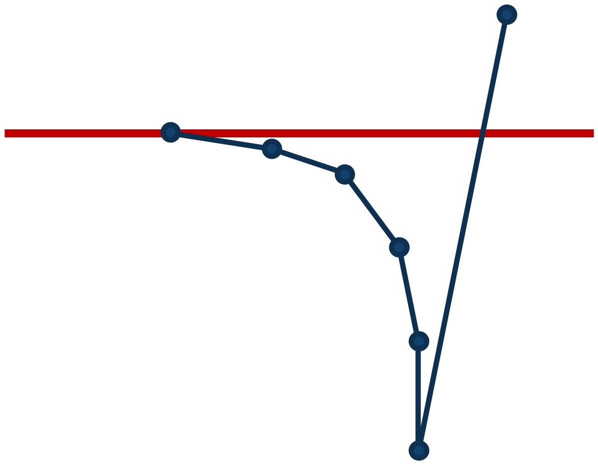
Use a plane sweep algorithm.

Informal invariant: Try to triangulate everything you can below the sweep line by adding diagonals and then remove the triangulated region from further consideration.

Plane sweep algorithm

Plane sweep (general method):

1. Sweep the input from bottom to top and stop at event points
2. Maintain invariant
3. At each event point restore invariant



Event points?

Input points

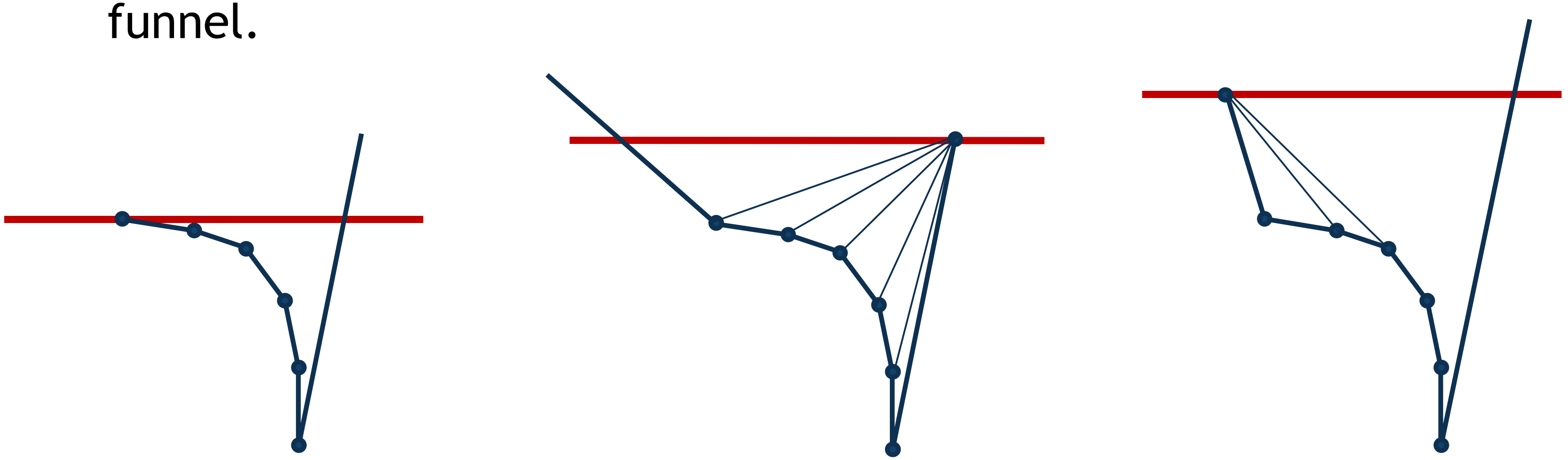
Invariant:

- The part below the sweep-line that has not been triangulated forms a funnel (one side being a segment and the other a concave chain).
- A stack containing all the vertices below the sweep-line that may need more diagonals.

Plane sweep algorithm

Three cases:

1. The new point extends the “funnel chain”
2. The new point is on the opposite side of the “funnel chain”
3. The new point lie on the chain side but does not extend the funnel.



Add edges from the new vertex v to all vertices below that are visible from v!

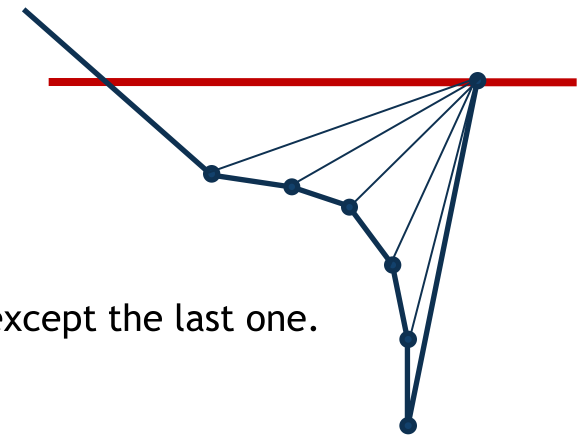
Triangulate monotone polygon

- merge the vertices of the left and right chains of P into y-sorted order, say, u_1, u_2, … , u_n.
- push u_1 and u_2 into an initially empty stack S.
- **for** $j \leftarrow 3$ .. n-1 **do**

  if u_j and $v_{top} \leftarrow$ top(S) are on different chains

  then pop all vertices from S
  add a diagonal between u_j and each popped vertex except the last one.
  push u_{j-1} and u_j onto S.

  else pop one vertex from S
  pop all vertices from S that are visible from u_j
  and add a diagonal between u_j and each popped vertex.
  push last popped vertex back onto S.
  push u_j onto S.



- add diagonals from the last vertex u_n to all stack vertices except first and last.

end

Triangulate monotone polygon

- merge the vertices of the left and right chains of P into y-sorted order, say, u_1, u_2, … , u_n.
- push u_1 and u_2 into an initially empty stack S.
- **for** $j \leftarrow 3$ .. n-1 **do**

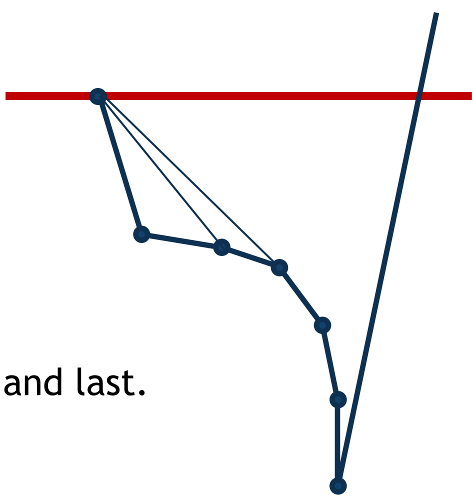
  if u_j and $v_{top} \leftarrow$ top(S) are on different chains

  then pop all vertices from S
  add a diagonal between u_j and each popped vertex except the last one.
  push u_{j-1} and u_j onto S.

  else pop one vertex from S
  pop all vertices from S that are visible from u_j
  and add a diagonal between u_j and each popped vertex.
  push last popped vertex back onto S.
  push u_j onto S.

- add diagonals from the last vertex u_n to all stack vertices except first and last.

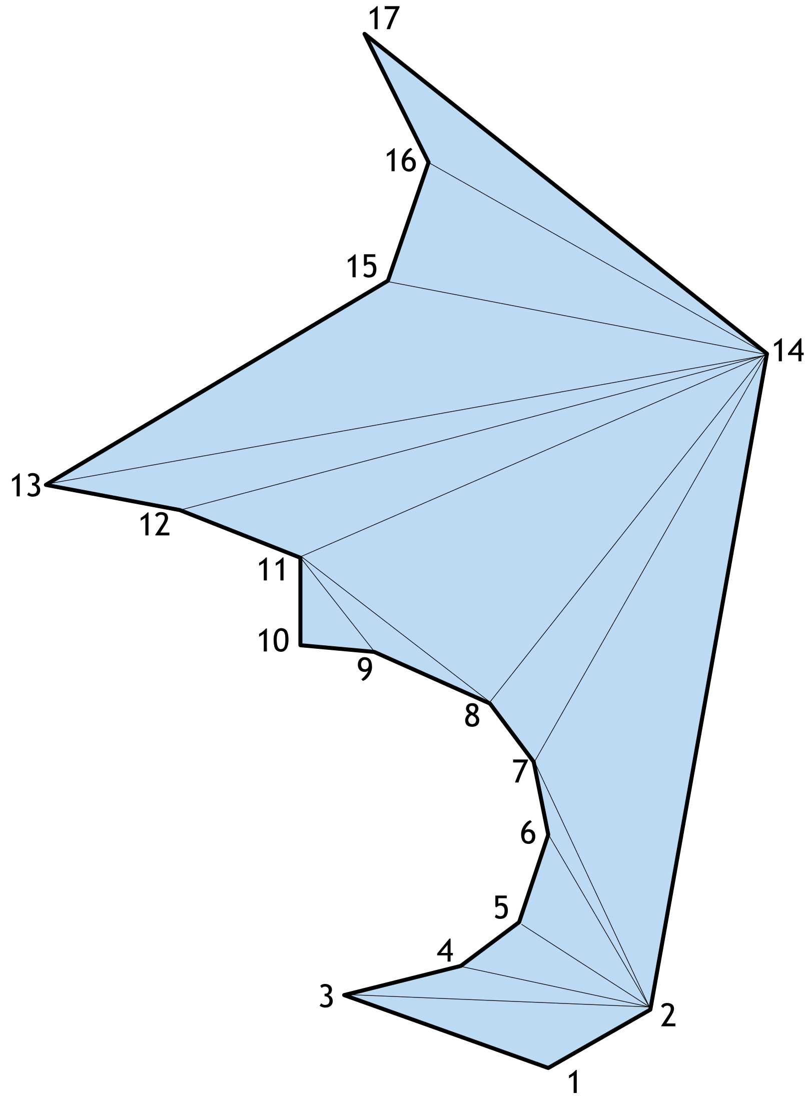
end



Triangulate monotone polygon

- Advance along y-sorted vertex-list from bottom to top.

- For each vertex v in y-sorted order, add downward visible diagonals from v to all visible vertices, starting from most recent & backwards.



Triangulate monotone polygon

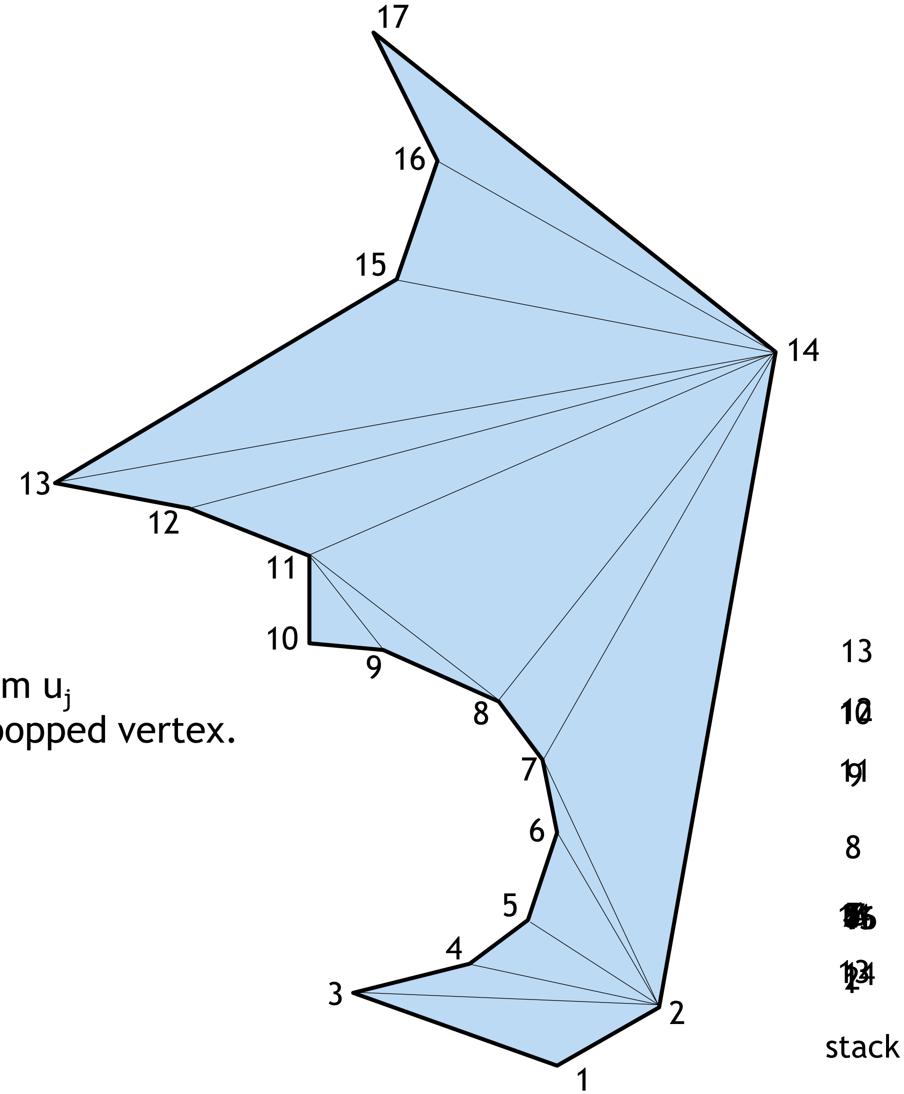
push u_1 and u_2 into an initially empty stack S.

for $j \leftarrow 3$ .. n-1 **do**

if u_j and $v_{top} \leftarrow$ top(S) are on different chains

then pop all vertices from S
add a diagonal between u_j and each popped vertex except the last one.
push u_{j-1} and u_j onto S.

else pop one vertex from S
pop all vertices from S that are visible from u_j and add a diagonal between u_j and each popped vertex.
push last popped vertex back onto S.
push u_j onto S.



Running time?

1. merge the vertices of the left and right chains of P into y-sorted order, say, $u_1, u_2, \ldots, u_n$.
2. push u_1 and u_2 into an initially empty stack S.
3. **for** $j \leftarrow 3 \,..\, n-1$ **do**
   - **a.** **if** u_j and $v_{top} \leftarrow top(S)$ are on different chains
   - **b.** **then** pop all vertices from S
     add a diagonal between u_j and each popped vertex except the last one.
     push u_{j-1} and u_j onto S.
   - c. **else** pop one vertex from S
     pop all vertices from S that are visible from u_j
     and add a diagonal between u_j and each popped vertex.
     push last popped vertex back onto S.
     push u_j onto S.
4. add diagonals from the last vertex u_n to all stack vertices except first and last.

Step 1: O(n) time

Step 3: n times - each iteration may take O(n) time $\Rightarrow$ O(n^2) time

Can it be improved?

How many vertices are pushed onto the stack at each iteration?

At most 2! $\Rightarrow$ O(n) time

Result

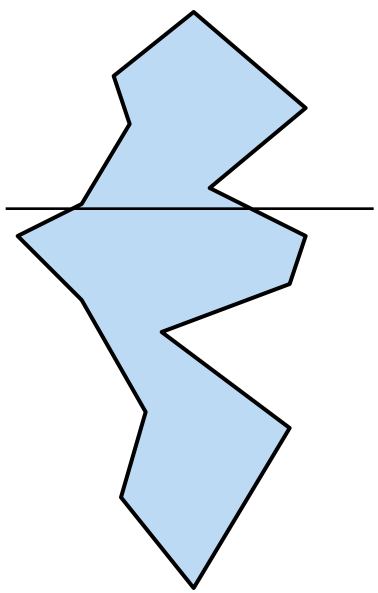
Theorem:

A y-monotone polygon can be triangulated in O(n) time!

Partition into monotone polygons

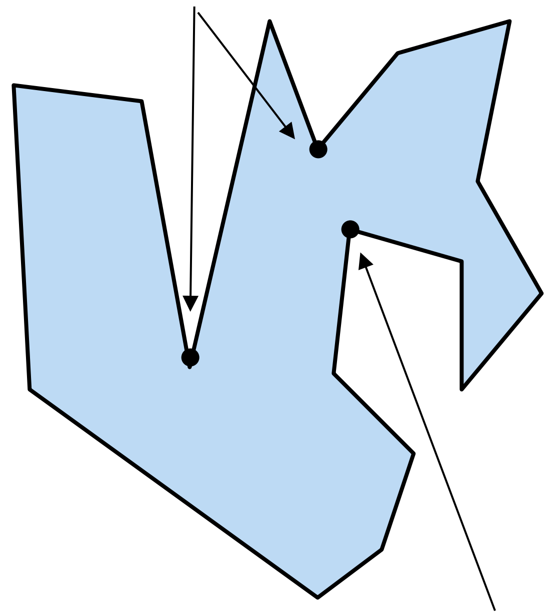
Fact: P is y-monotone if and only if it does not have any cusps (no split or merge vertices).

Subdivide the simple polygon into monotone sub-polygons by adding diagonals to split and merge vertices.



y-monotone

Merge vertex: concave local y-min vertex.

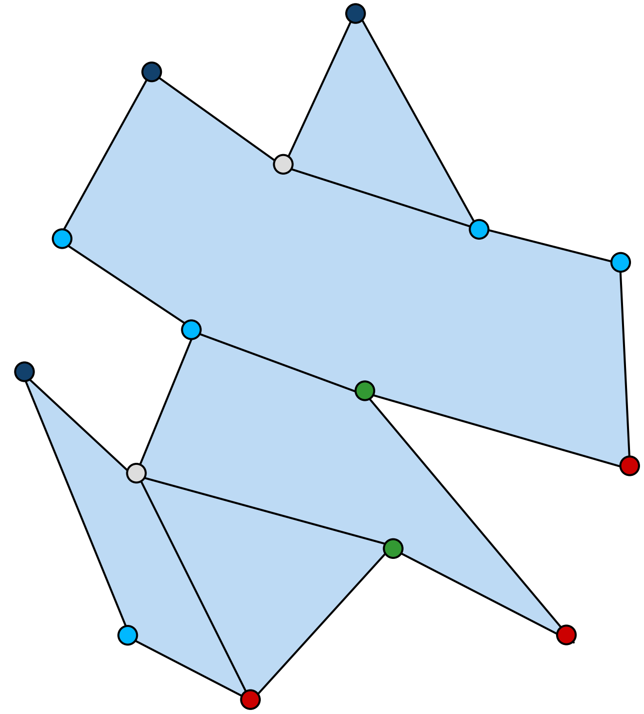


Split vertex: concave local y-max vertex.

Monotonic partitioning

Idea:

- sweep the polygon from top to bottom
- keep track of all regions that the sweep line intersects.
- When two regions merge, or one is split, add edges to separate the regions into monotonic parts.
- Events? vertices
- Sweep line data structure? binary search tree (BST) that keeps tracks of the order of the edges intersecting the sweep line



Plane sweep algorithm

Plane sweep (general method):

1. Sweep the input from top to bottom and stop at event points
2. Maintain invariant
3. At each event point restore invariant

Event points?

Vertices of polygon, sorted in decreasing order by *y*-coordinate (no new events will be added).

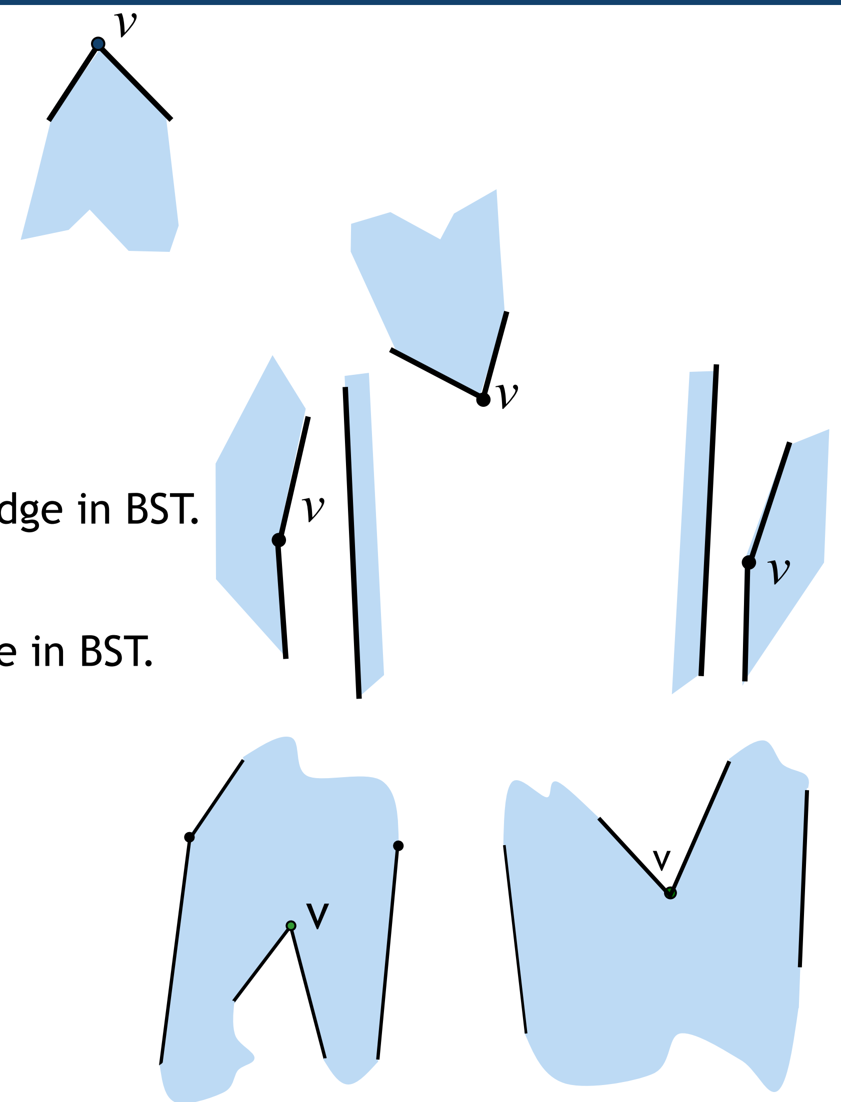
Invariant:

1. The part of the polygon above the last processed event point is partitioned into y-monotone polygons.
2. We know the order of the segments along the sweep line (stored in a balanced binary tree).
3. We know the "helper" of each edge intersecting the sweep-line

Sweep Line Algorithm

Event processing of vertex *v*:

1. **Start vertex:**
   - Insert the two edges into BST.
2. **End vertex:**
   - Delete incident edges from BST.
3. **Left chain vertex:**
   - Replace upper edge with lower edge in BST.
4. **Right chain vertex:**
   - Replace upper edge with lower edge in BST.
5. **Split vertex**
   - Insert the two edges into BST.
6. **Merge vertex**
   - Delete incident edges from BST.



THE UNIVERSITY OF SYDNEY

Plane sweep algorithm

Plane sweep (general method):

1. Sweep the input from top to bottom and stop at event points
2. Maintain invariant
3. At each event point restore invariant

Event points?

Vertices of polygon, sorted in decreasing order by *y*-coordinate (no new events will be added).

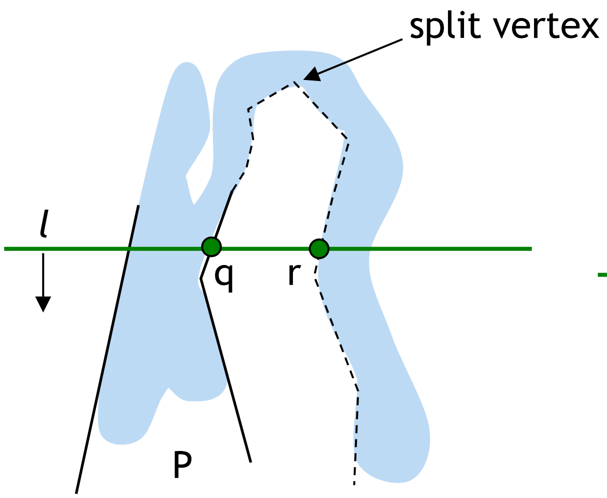
Invariant:

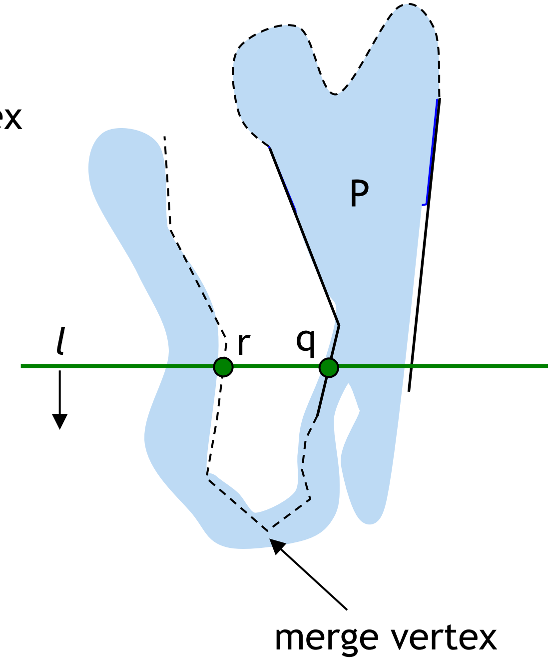
1. The part of the polygon above the last processed event point is partitioned into y-monotone polygons.
2. We know the order of the segments along the sweep line (stored in a balanced binary tree).
3. We know the “helper” of each edge intersecting the sweep-line

Monotonic partitioning

Lemma: A polygon is y-monotone if it has no split vertices or merge vertices.

Proof: Assume P is not y-monotone. Prove that P has a split or merge vertex.





On the shortest walk from q to r there must be some highest (or lowest) point. This point must be a split (or merge) vertex.

Plane sweep algorithm

Plane sweep (general method):

1. Sweep the input from top to bottom and stop at event points
2. Maintain invariant
3. At each event point restore invariant

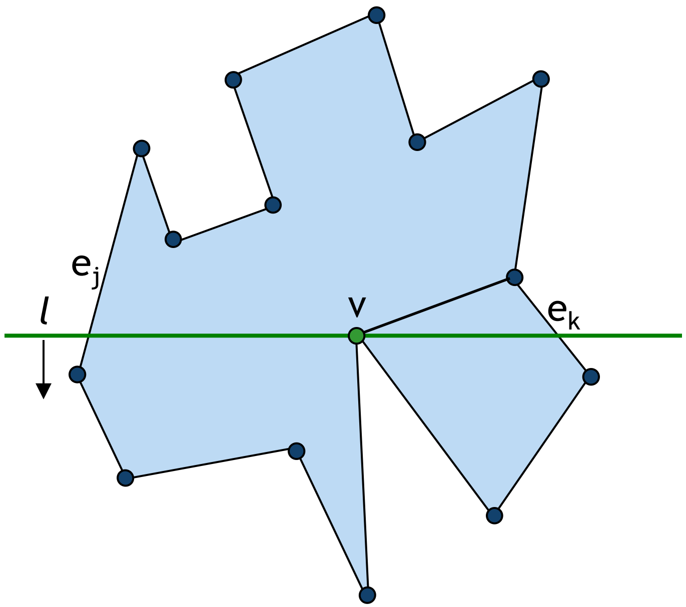
Event points?

Vertices of polygon, sorted in decreasing order by *y*-coordinate (no new events will be added).

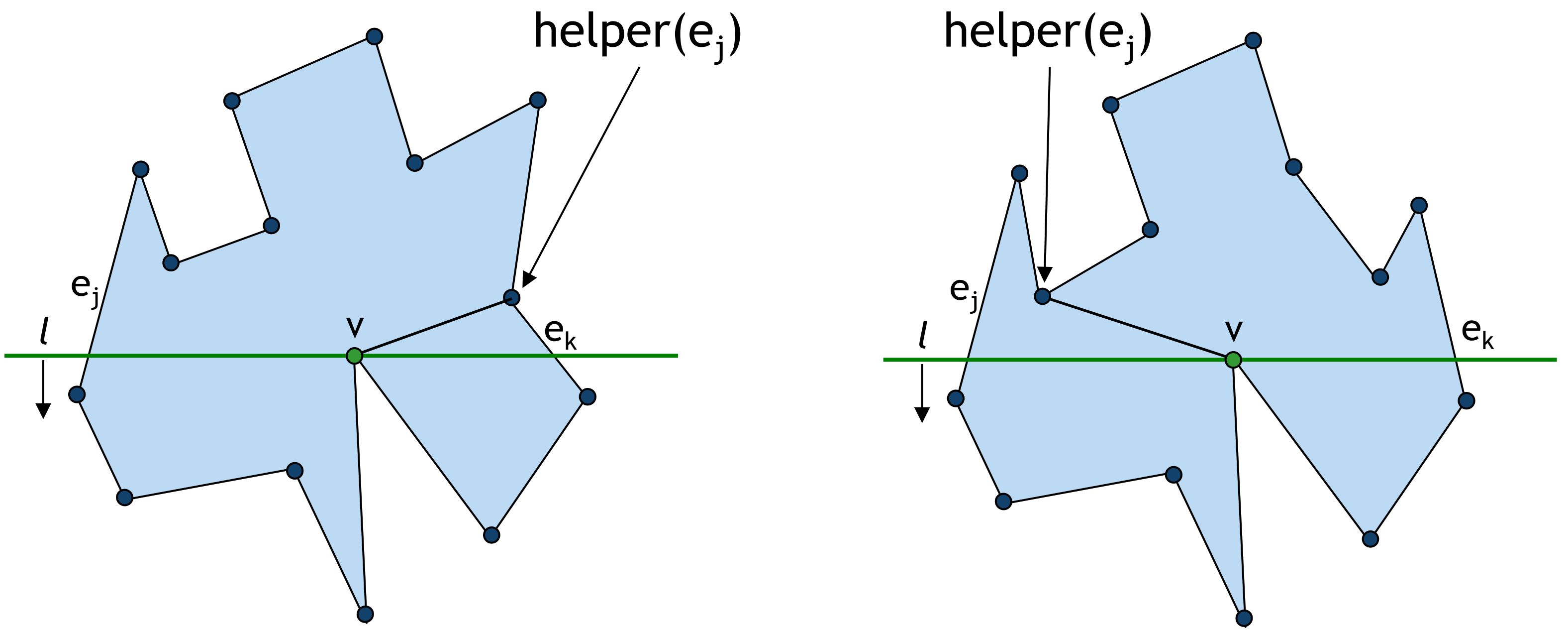
Invariant:

1. The part of the polygon above the last processed event point is partitioned into y-monotone polygons.
2. We know the order of the segments along the sweep line (stored in a balanced binary tree).
3. We know the "helper" of each edge intersecting the sweep-line

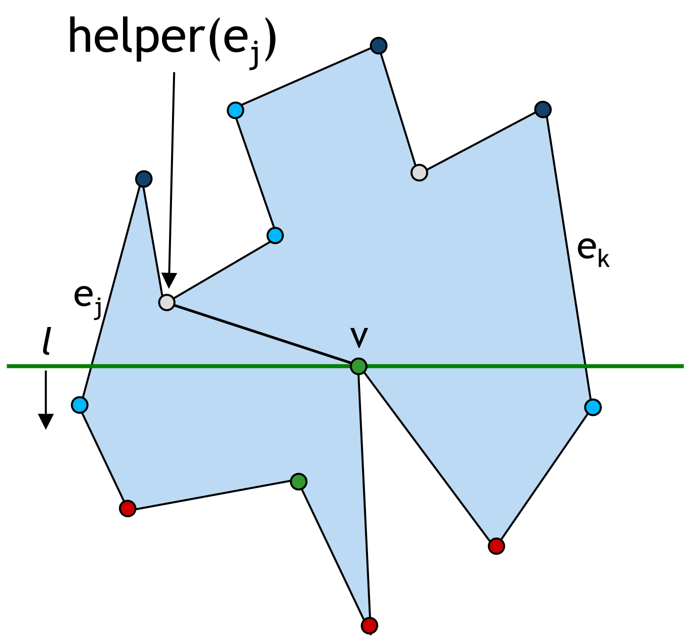
Helper



Helper



Monotonic partitioning: Algorithm



helper(e_j) = the lowest vertex to the right of e_j above the sweep line such that the horizontal segment connecting the vertex to e_j lies inside P.

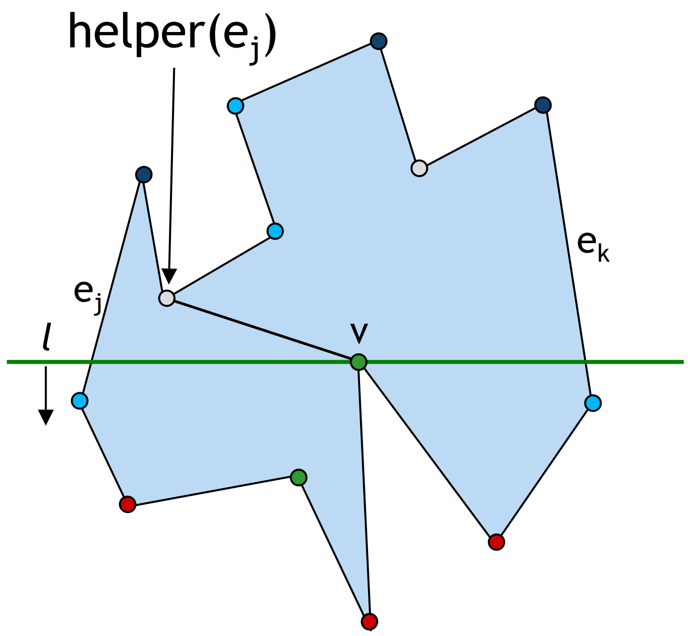
The upper endpoint of e_j can be the helper.

We can always connect v to the helper of e_j.

Monotonic partitioning: Algorithm

Goal: add diagonals from each split vertex to a vertex above it. Which one?

A vertex close to it?



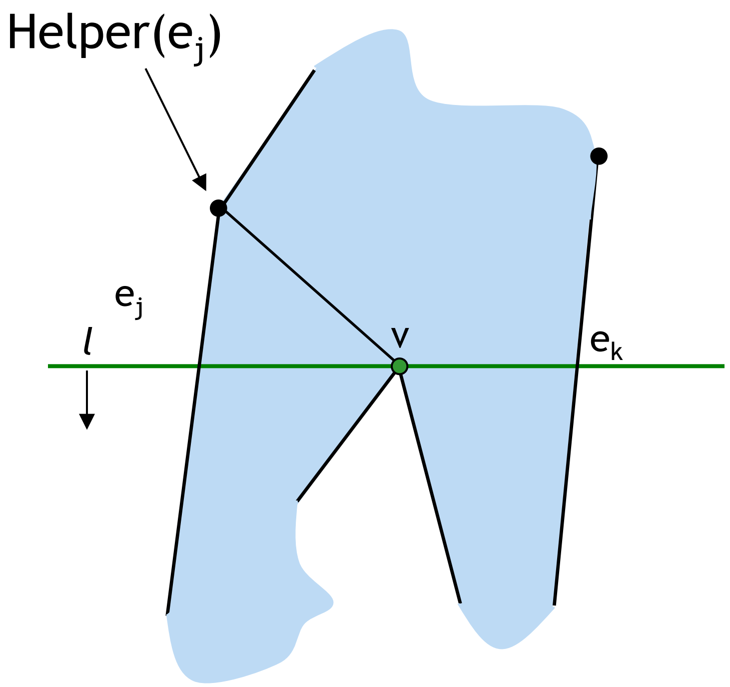
Consider a split vertex v. Let e_j (e_k) be the edge immediately to the left (right) of v along sweep line.

We can always connect v to the helper of e_j.

Monotonic partitioning: Algorithm

Goal: add diagonals from each split vertex to a vertex above it. Which one?

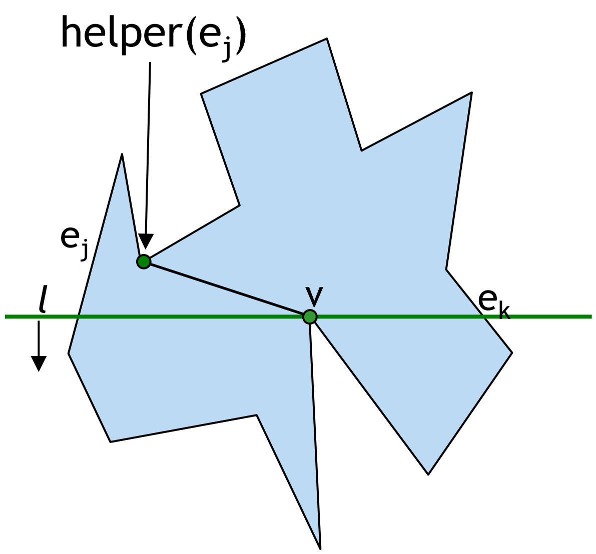
A vertex close to it?



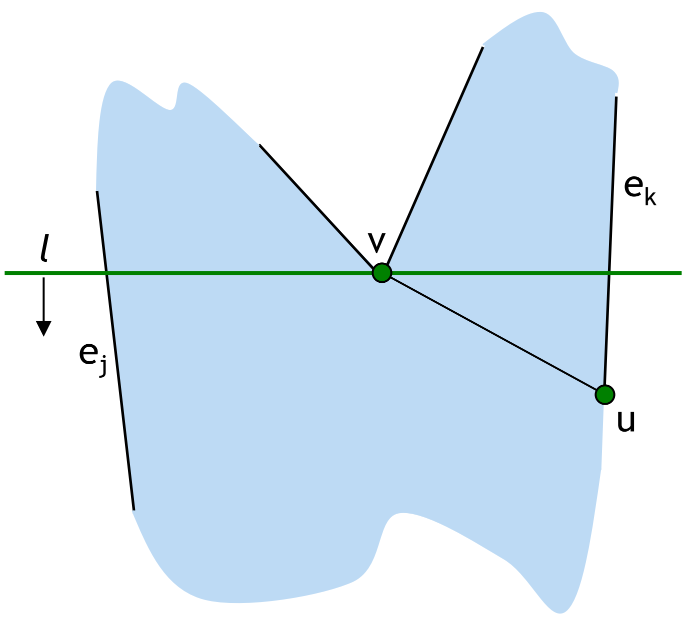
Consider a split vertex v. Let e_j (e_k) be the edge immediately to the left (right) of v along sweep line.

We can always connect v to the helper of e_j.

Removal of split and merge vertices



Split vertex: Add an edge to helper(e_j) or to top vertex of e_j.
Set helper(e_j) := v
Insert v's incident edges to BST.



Merge vertex: (split nodes in reverse)
Set helper(e_j) := v
Aim is to connect v to the highest vertex below the sweep line in between e_j and e_k.
Remove v's incident edges from BST.

Plane sweep algorithm

Plane sweep (general method):

1. Sweep the input from top to bottom and stop at event points
2. Maintain invariant
3. At each event point restore invariant

Event points?

Vertices of polygon, sorted in decreasing order by *y*-coordinate.
(No new events will be added)

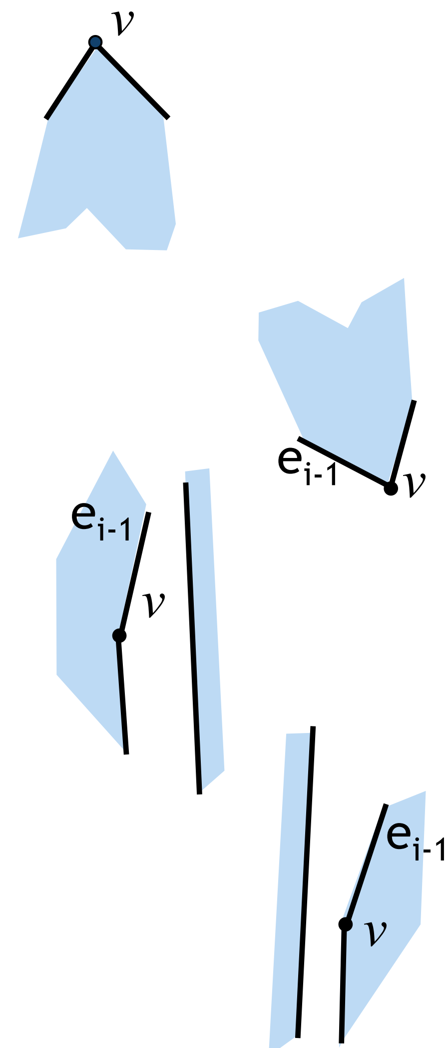
Invariant:

1. The part of the polygon above the last processed event point is partitioned into y-monotone polygons.
2. We know the order of the segments along the sweep line (stored in a balanced binary tree).
3. We know the “helper” of each edge intersecting the sweep-line

Sweep Line Algorithm

Event processing of vertex *v*:

1. **Split vertex**
2. **Merge vertex**
3. **Start vertex:**
   – Insert the two edges into BST.
   – Set helper of left edge to *v*.
4. **End vertex:**
   – Delete incident edges from BST.
   – If u=helper(e_{i-1}) is a merge vertex then add (u,v)
5. **Left chain vertex:**
   – If u=helper(e_{i-1}) is a merge vertex then add (u,v)
   – Replace upper edge with lower edge in BST.
   – Make *v* helper of new edge.
6. **Right chain vertex:** (similar to 5)



Monotone partition: Analysis

Correctness? There are no split or merge vertices remaining.

Many cases to consider for correctness of event handling. See the book for detailed proof.

- Helpers are correctly updated
- Merge vertices are correctly resolved
- Split vertices are correctly resolved
- Added diagonals do not intersect each other

Monotone partition: Analysis

Time complexity?

Sort the vertices into an event queue Q.

We have n events and in each event we perform:

- One query on Q (which event to process),
- at most one query in T (the binary search tree),
- at most two deletions on T, and
- at most two insertions on T.

Each update operation can be performed in $O(\log n)$ time

Total time: $O(n \log n)$

Triangulate simple polygon

Main idea:

1. Partition P into y-monotone polygons

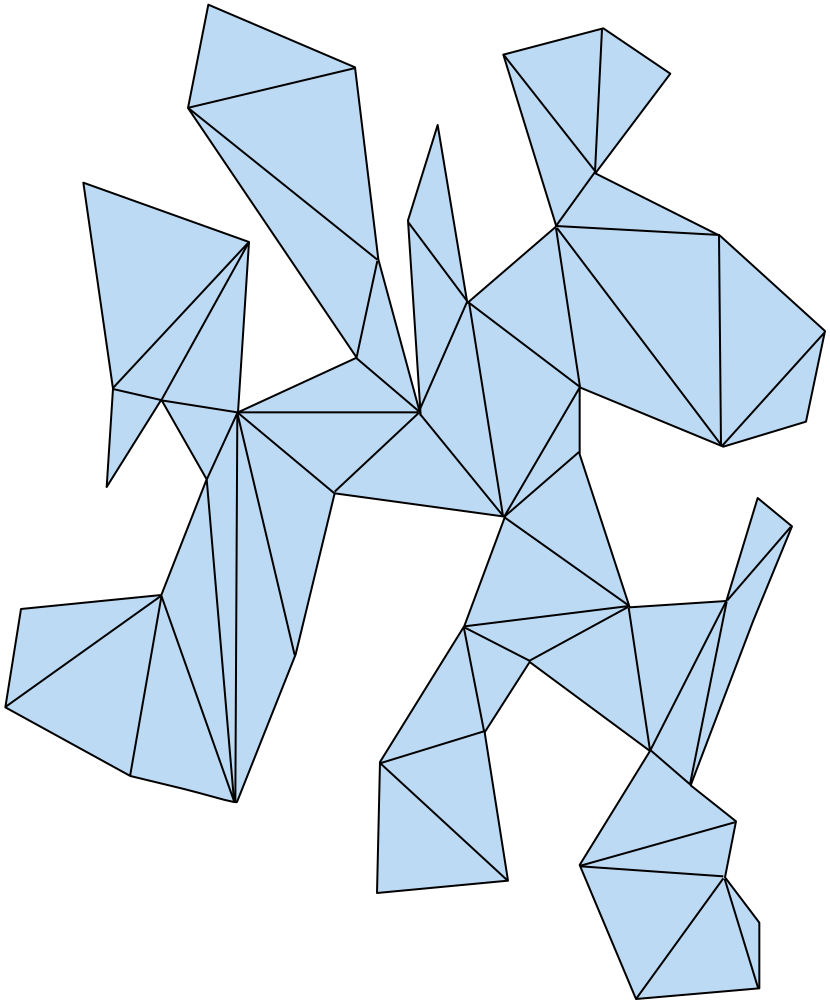
   Time O(n log n)

2. Triangulate each y-monotone polygon

   Time $O(n_i)$

Prove:

A simple polygon can be triangulated in O(n log n) time.



Algorithm 4: running time

Assumption: 10^9 instructions per second

Input size: 1 million points = 10^6 points

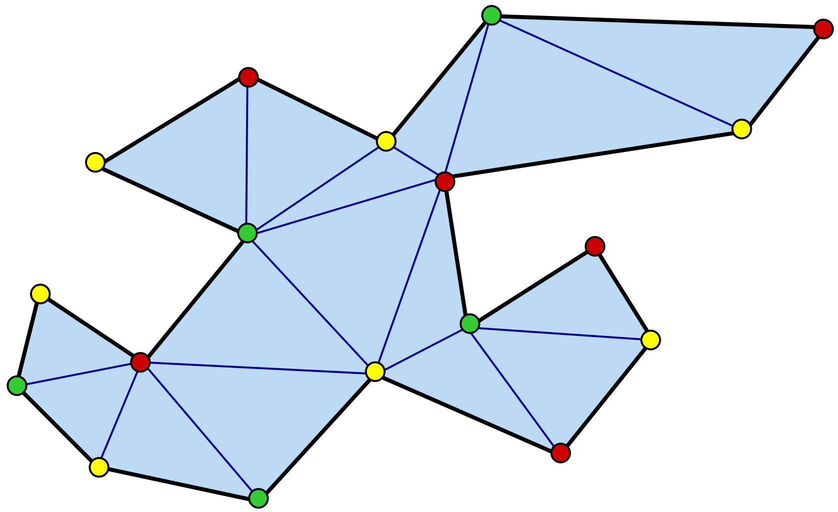
$\Rightarrow$ running time ~ $n \log n / 10^9$ = 0.006 seconds

#points in 1 second: 100,000,000 points

Algorithm	Complexity	Time (sec)	Points/sec
1	$O(n^4)$	32M yrs	180
2	$O(n^3)$	32 yrs	1000
3	$O(n^2)$	1020 sec	30k
4	$O(n \log n)$	0.06 sec	100M

Previously

- Every simple polygon with n vertices can be decomposed into n-2 triangles.
- Every triangulated simple polygon can be 3-colourable.
- Every simple polygon can be “guarded” by n/3 guards, and n/3 guards is sometimes necessary.
- To find a guard set our algorithm requires a triangulation.
  Today: O(n log n)



Polygon triangulation

› O(n log n) time	[Garey, Johnson, Preparata & Tarjan'78]
› O(n loglog n)	[Tarjan & van Wijk'88]
› O(n log* n)	[Clarkson et al.'89]
› O(n)	[Chazelle'90]
› O(n) randomised	[Amato, Goodrich & Ramos'00]

Open problem: Is there a simple O(n)-time algorithm?