Computational Geometry
[csci 3250]

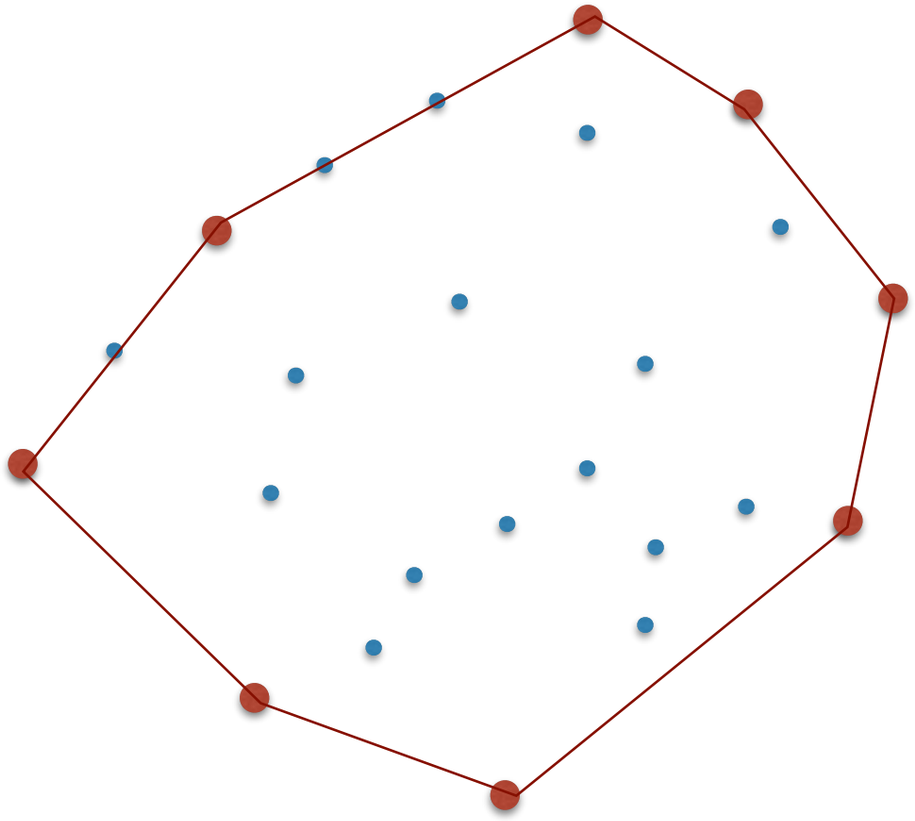
Laura Toma

Bowdoin College

Convex hull

- Given a set P of points in 2D, CH(P) is the smallest convex polygon that contains all points of P

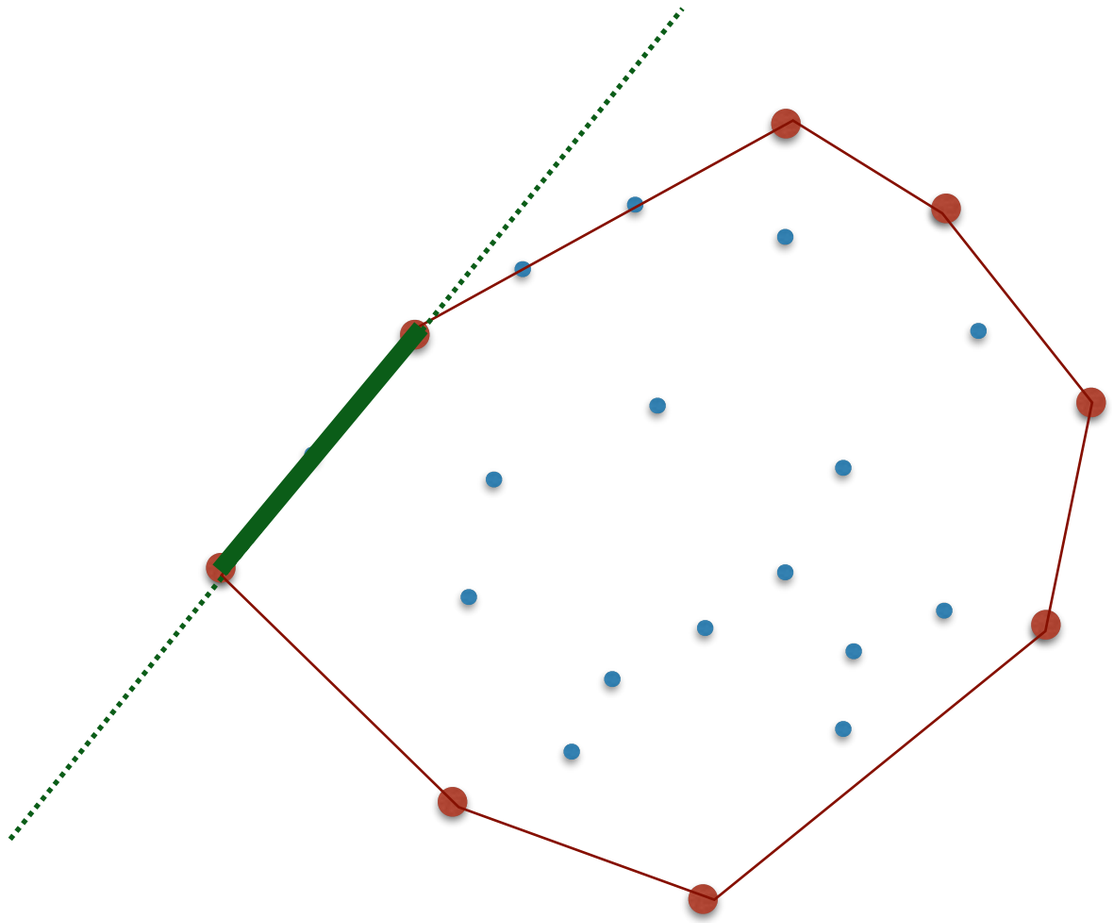
- Properties of CH
  - all internal angles are < 180
  - all edges of CH are extreme and all extreme edges of P are on CH
  - all vertices of CH are extreme and all extreme vertices of P are on CH



Convex hull

- Given a set P of points in 2D, CH(P) is the smallest convex polygon that contains all points of P

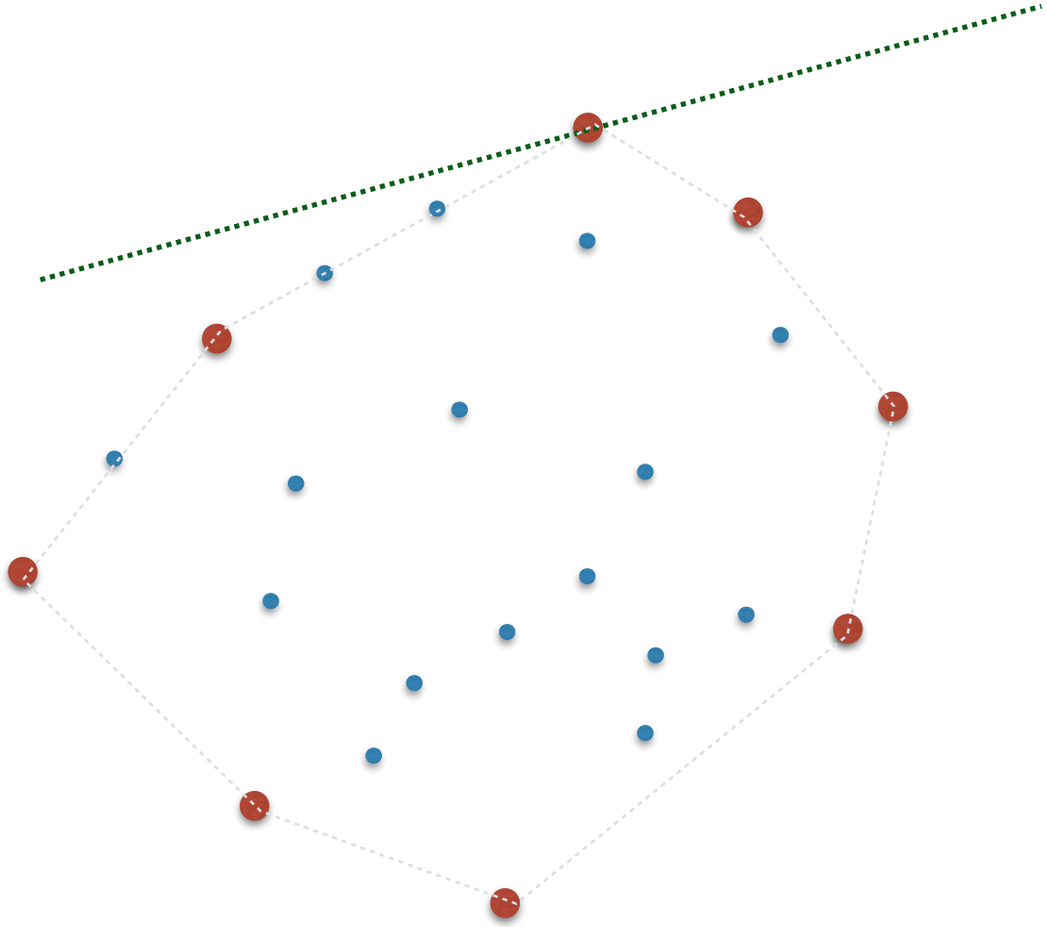
- Properties of CH
  - all internal angles are < 180
  - all edges of CH are extreme and all extreme edges of P are on CH
  - all vertices of CH are extreme and all extreme vertices of P are on CH



Convex hull

- Given a set P of points in 2D, CH(P) is the smallest convex polygon that contains all points of P

- Properties of CH
  - all internal angles are < 180
  - all edges of CH are extreme and all extreme edges of P are on CH
  - all vertices of CH are extreme and all extreme vertices of P are on CH



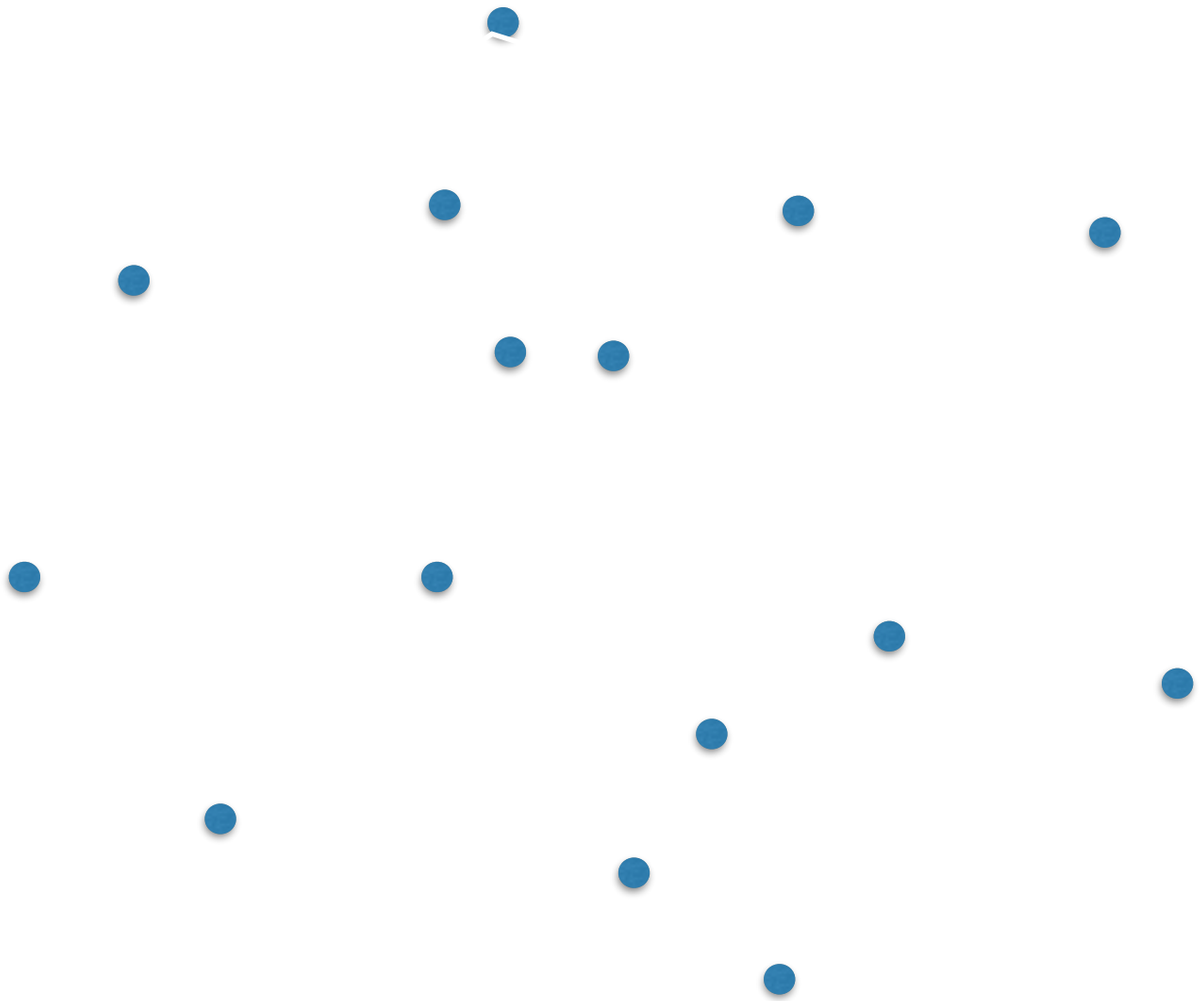
Outline

- Last time:
  - CH properties and 3 algorithms:
    - Gift wrapping
    - Quickhull
    - Graham scan

- Today
  - More algorithms
    - Andrew's monotone chain algorithm
    - Incremental
    - Divide-and-conquer
  - Lower bound

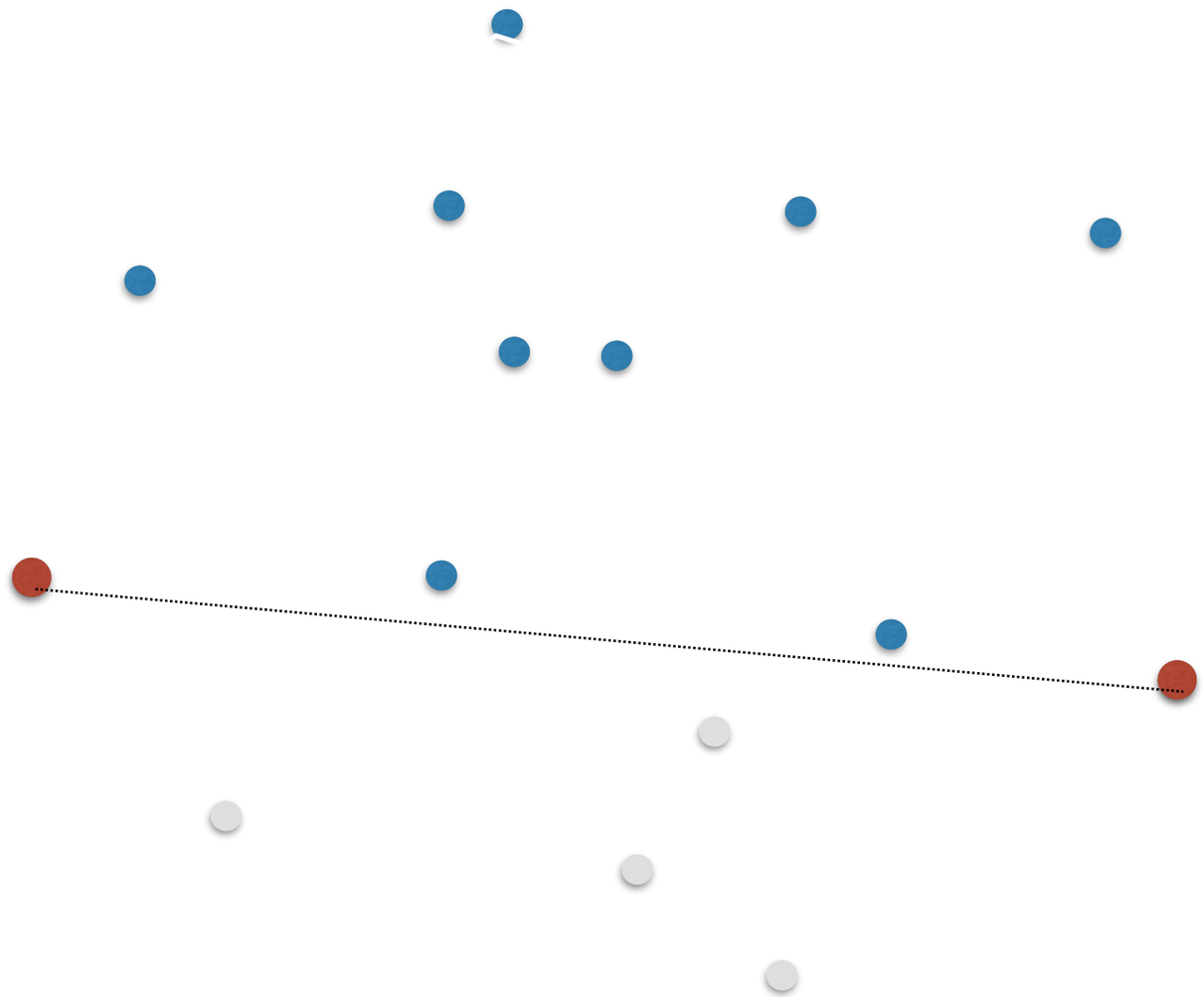
Andrew's Monotone Chain Algorithm (1979)

- Alternative to Graham's scan



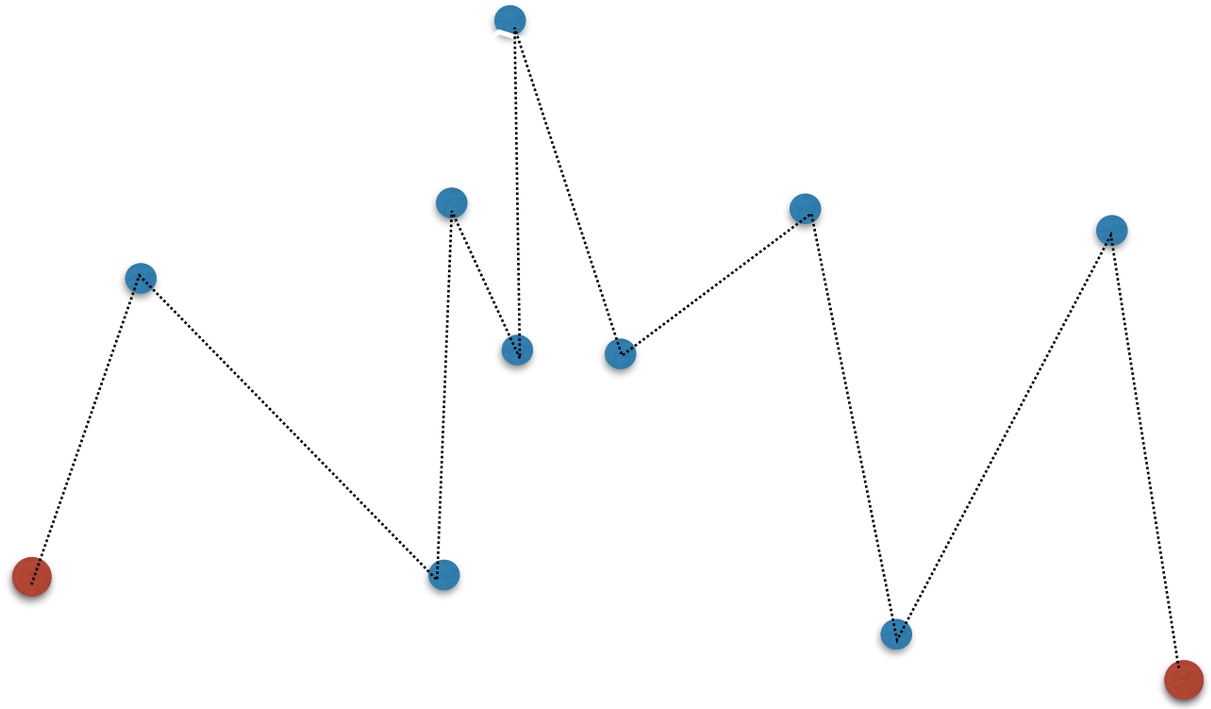
Andrew's Monotone Chain Algorithm (1979)

- Find upper hull and lower hulls separately



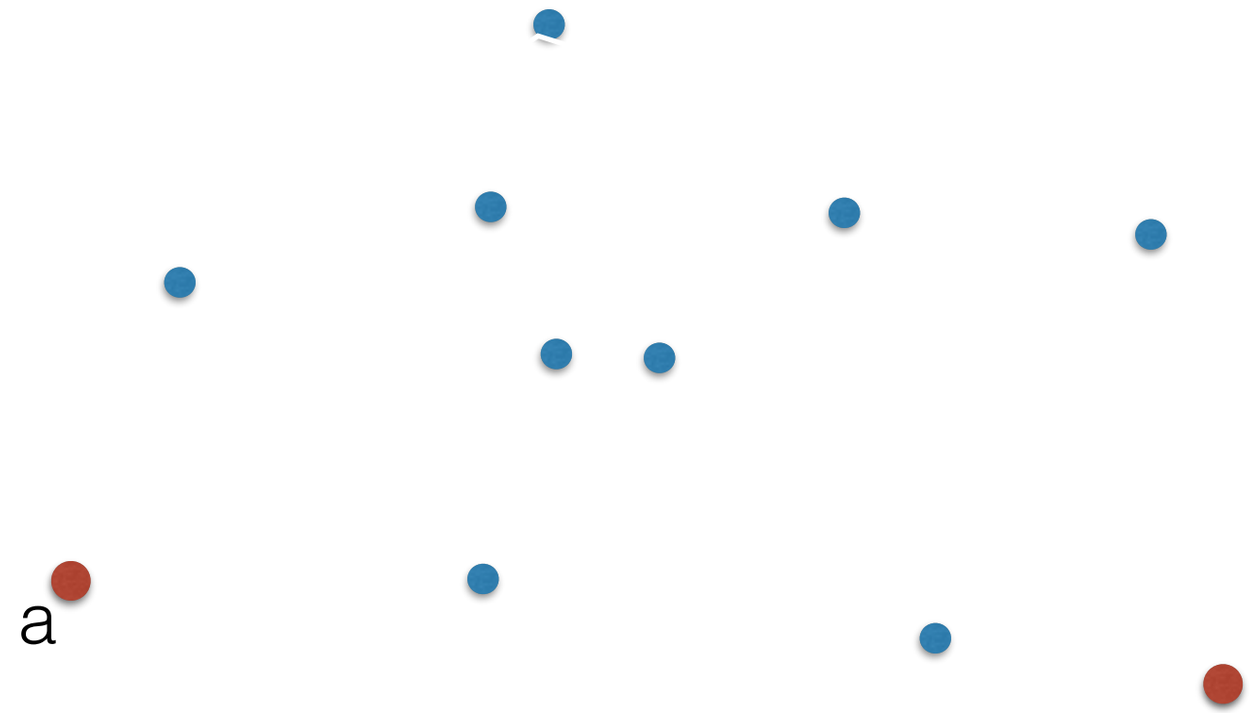
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



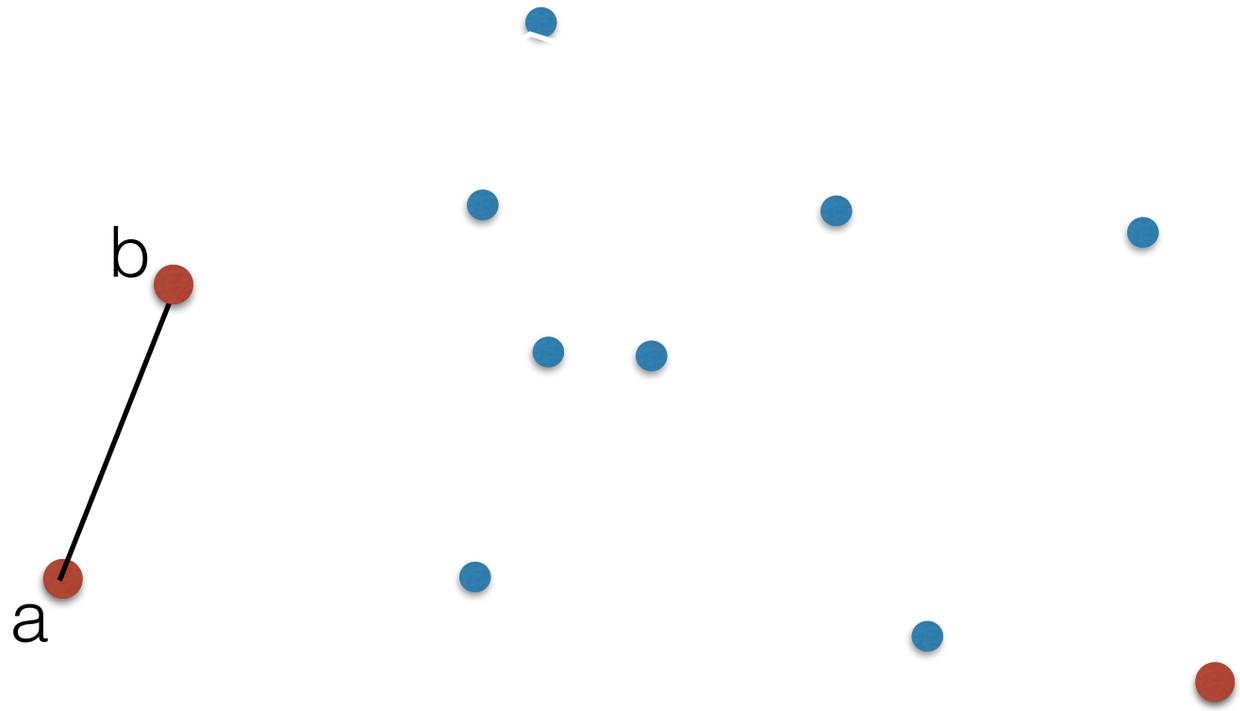
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



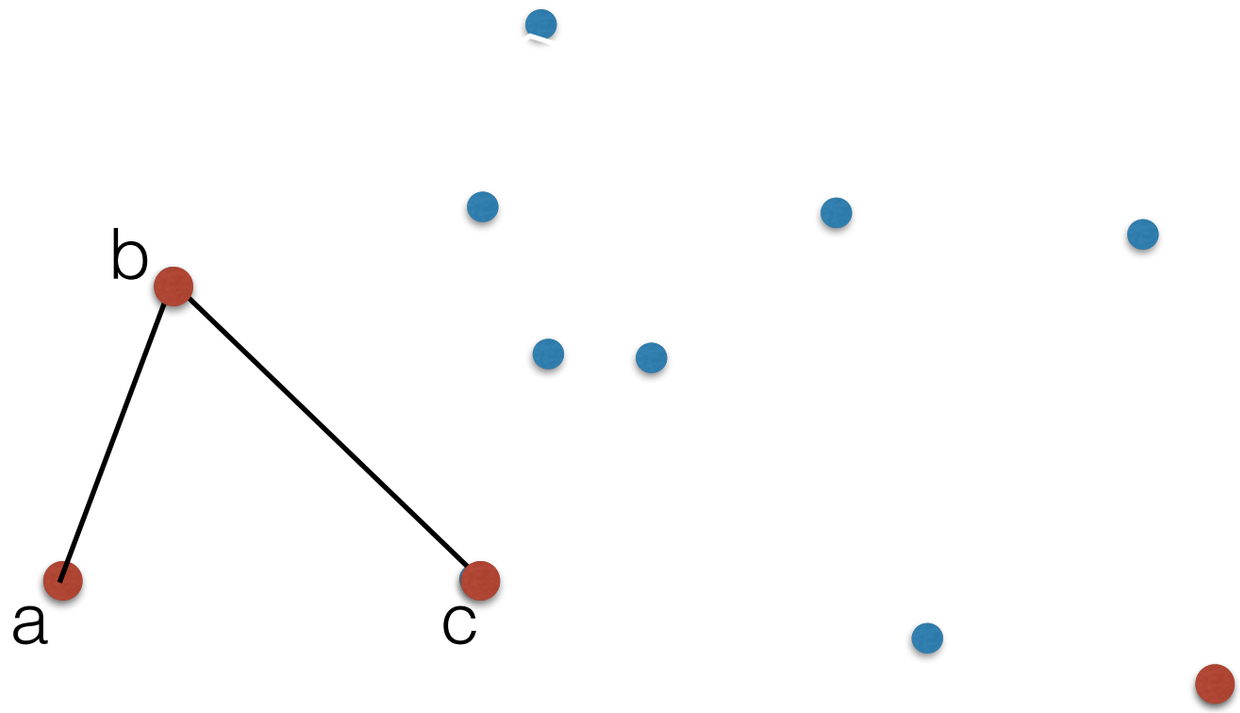
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



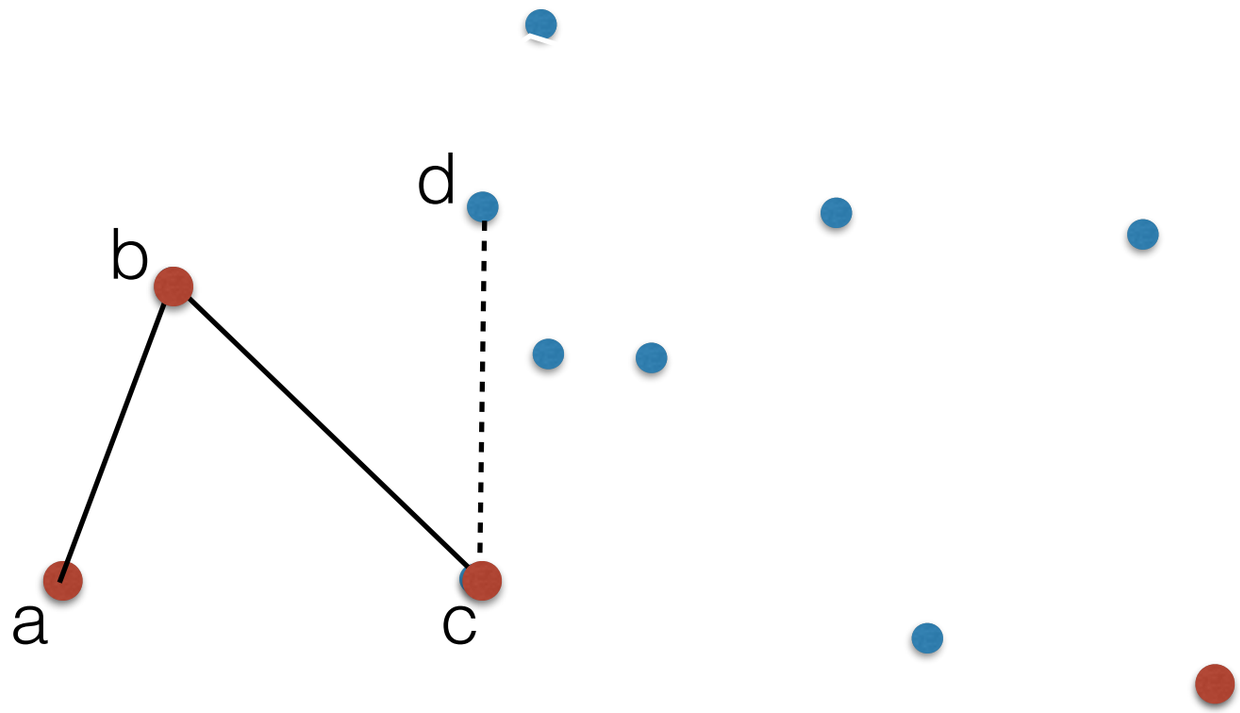
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



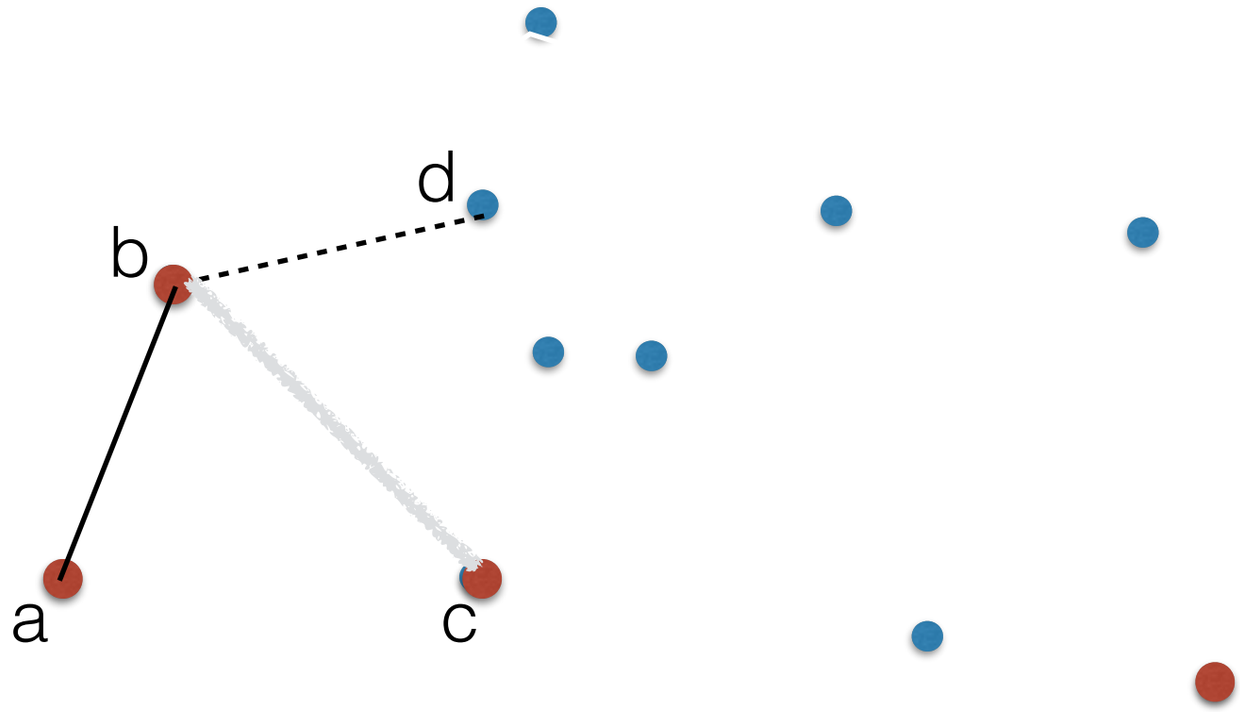
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



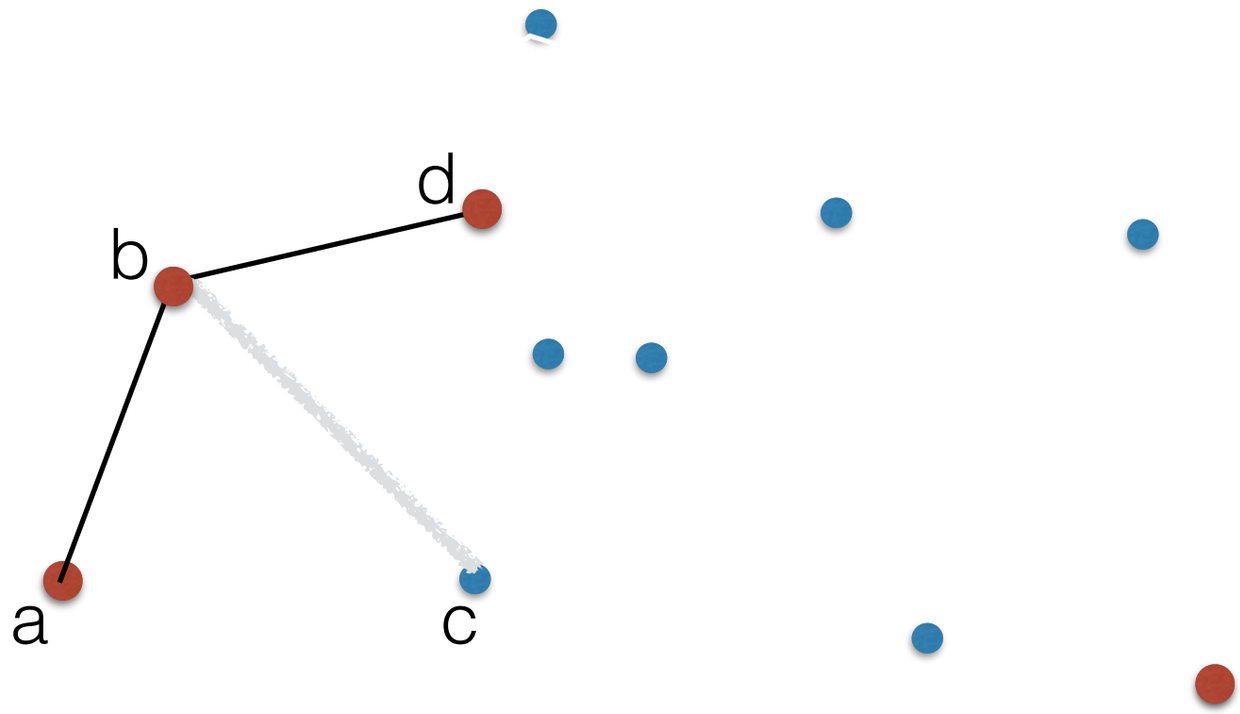
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



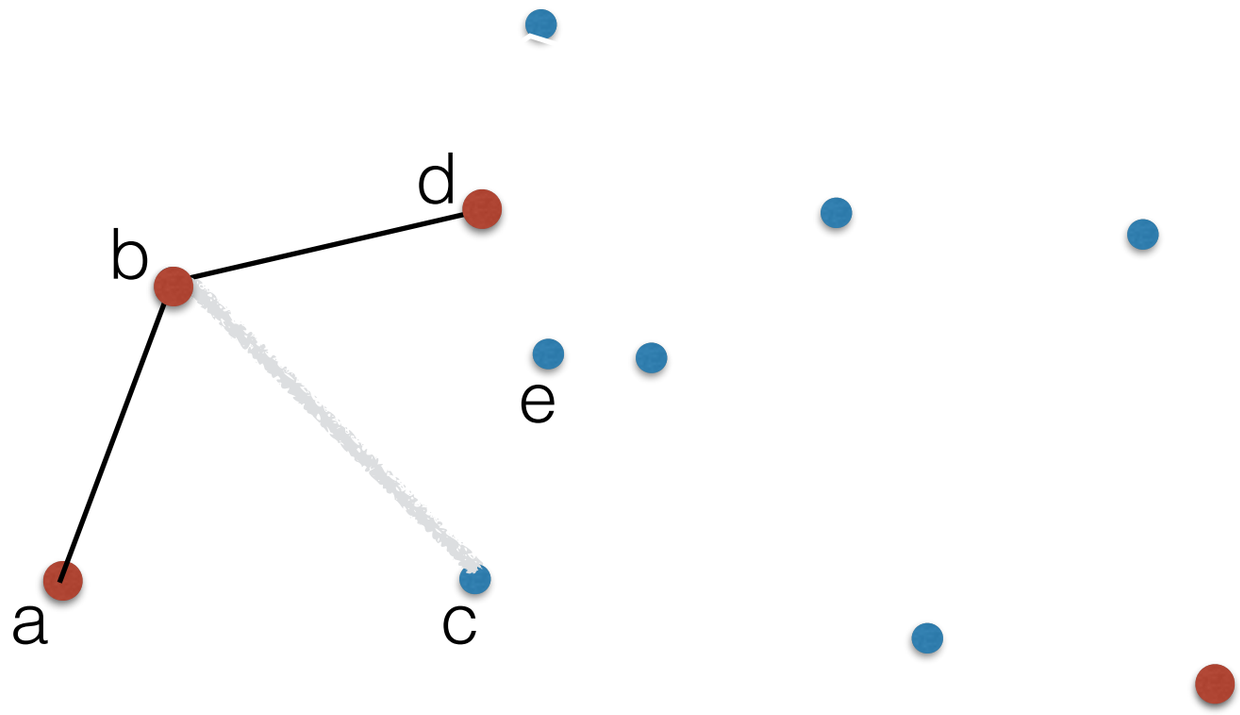
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



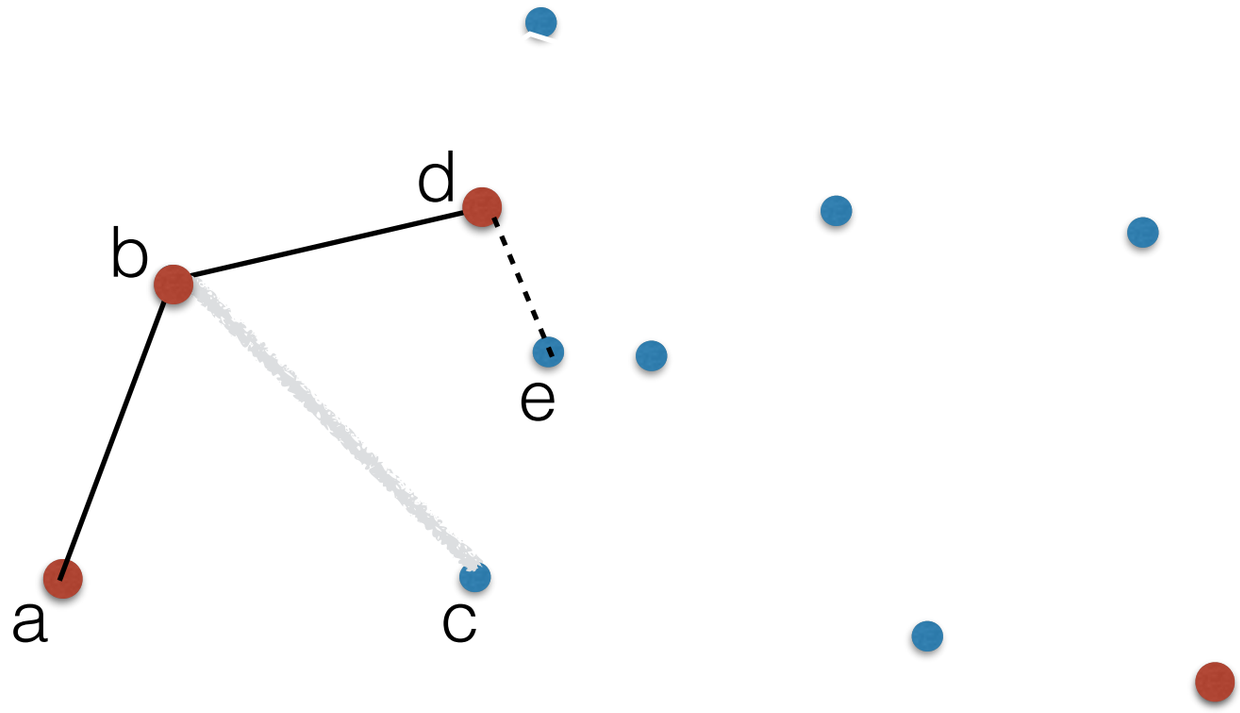
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



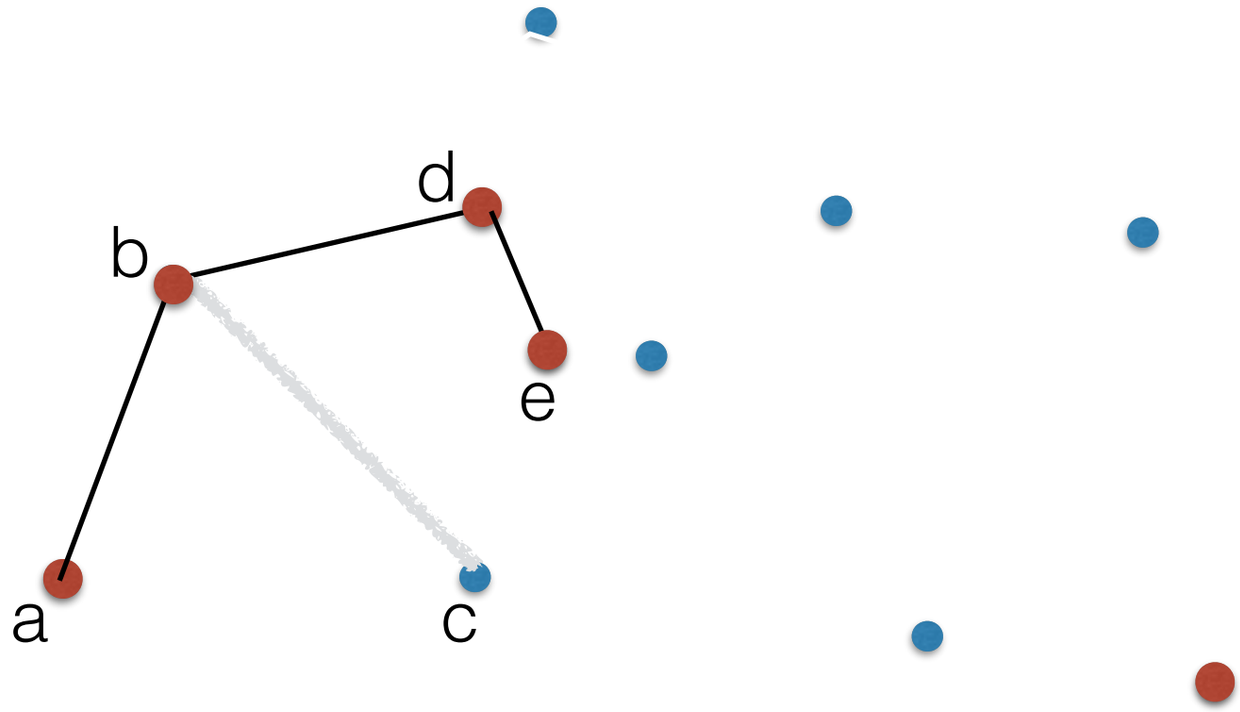
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



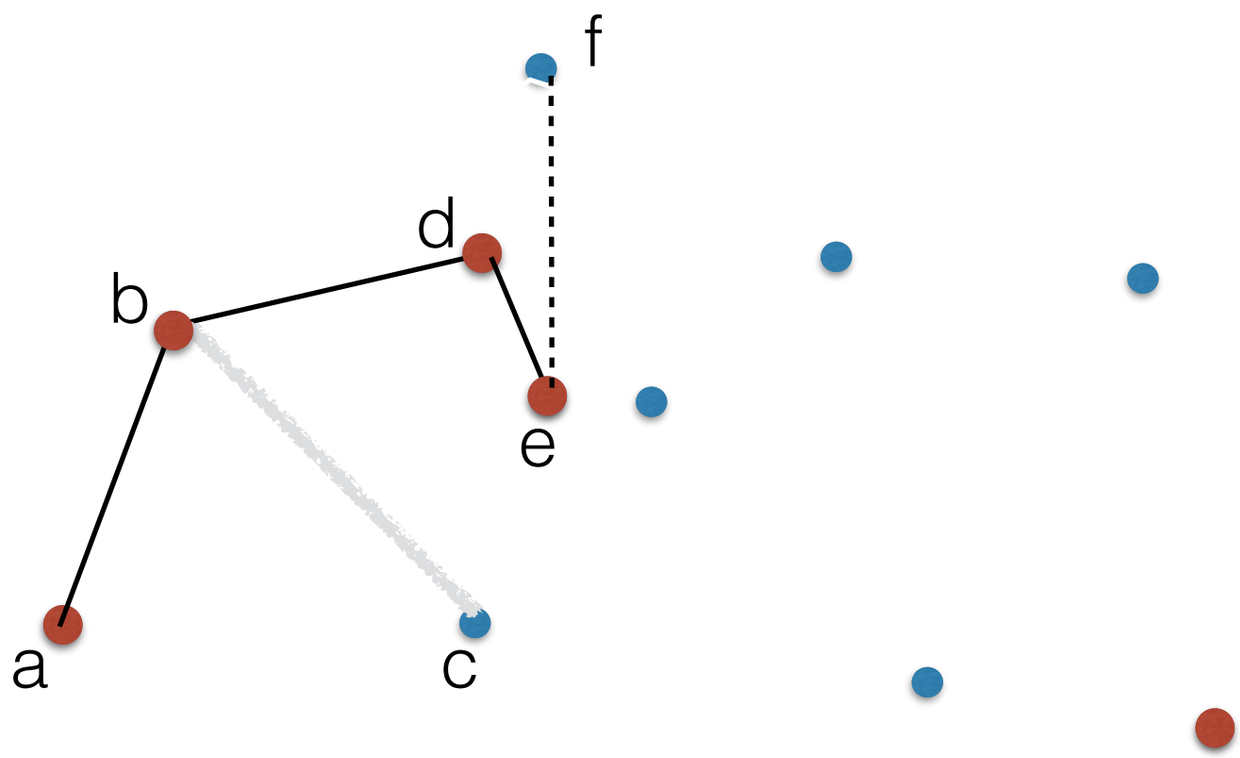
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



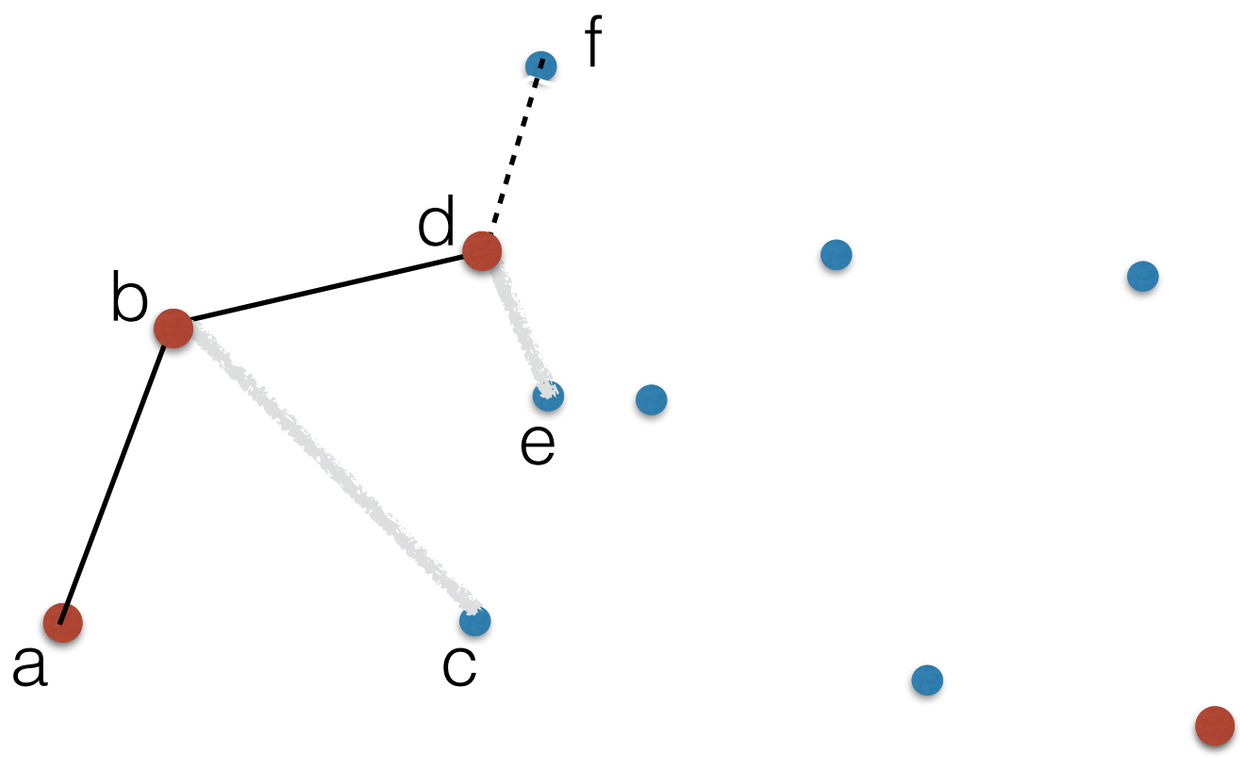
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



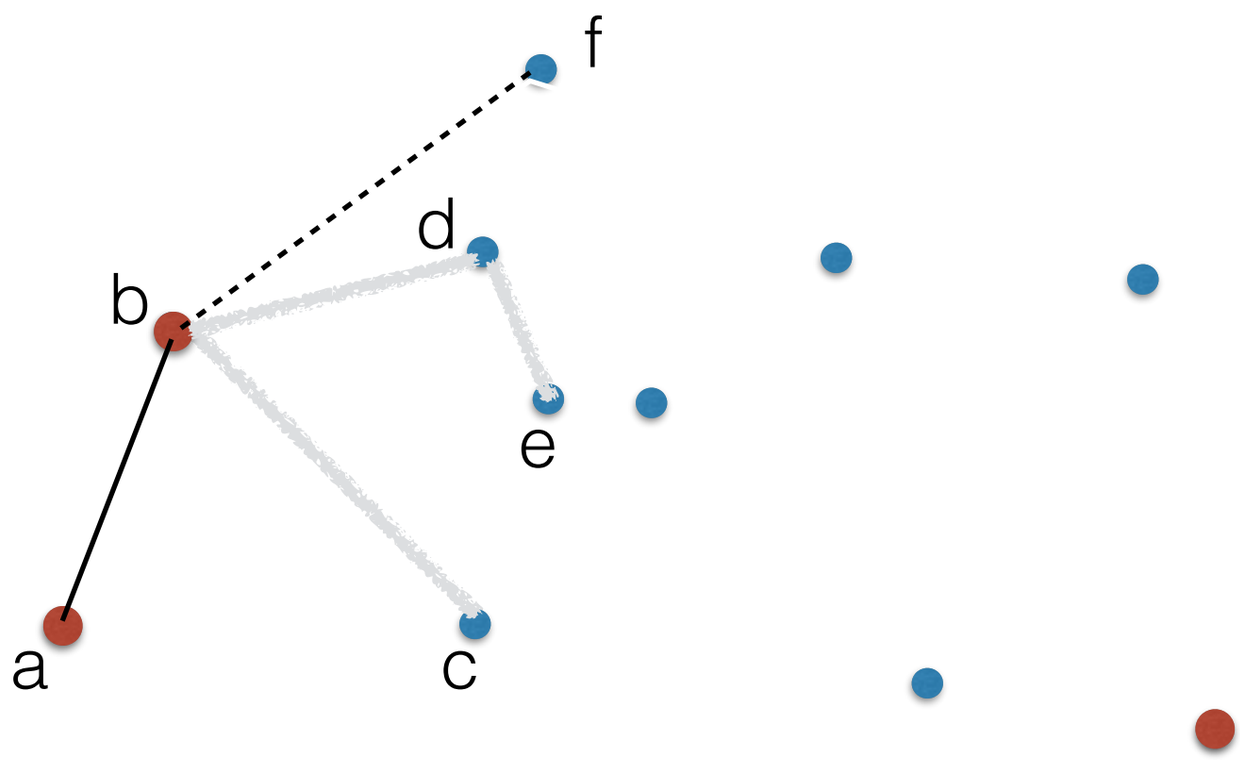
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



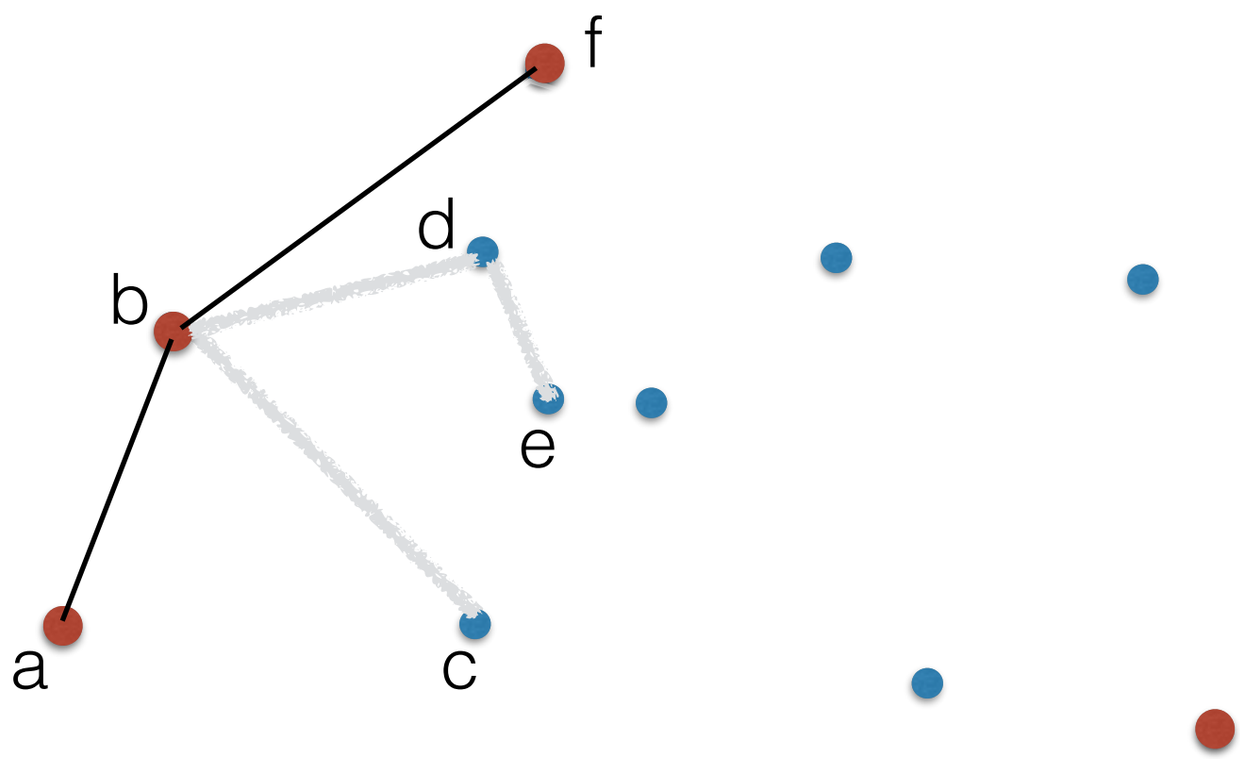
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



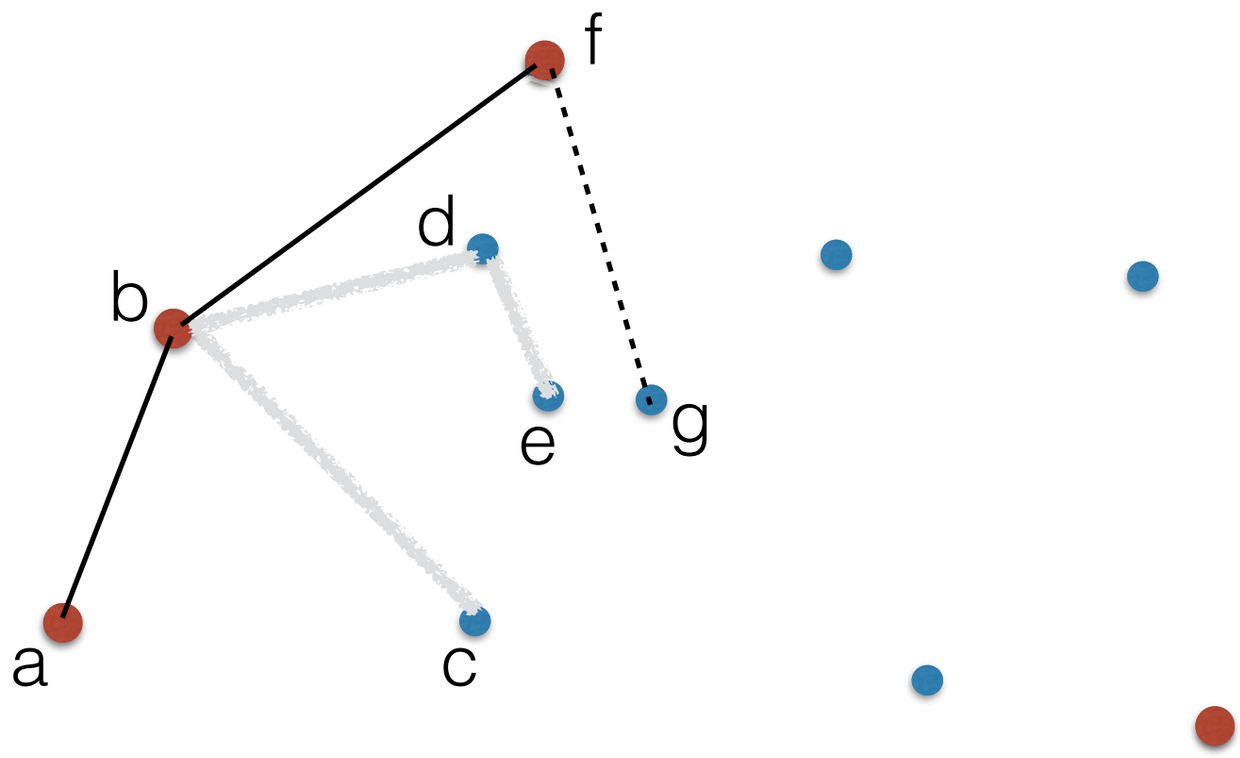
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



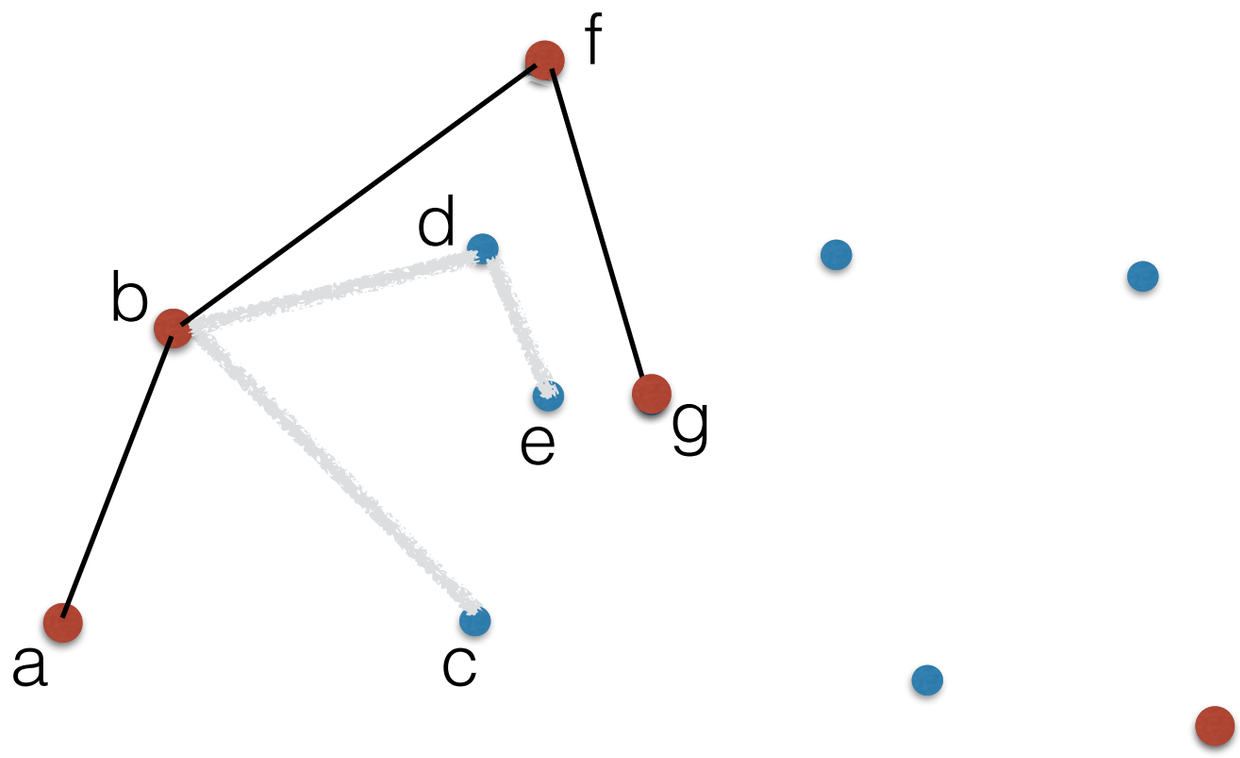
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



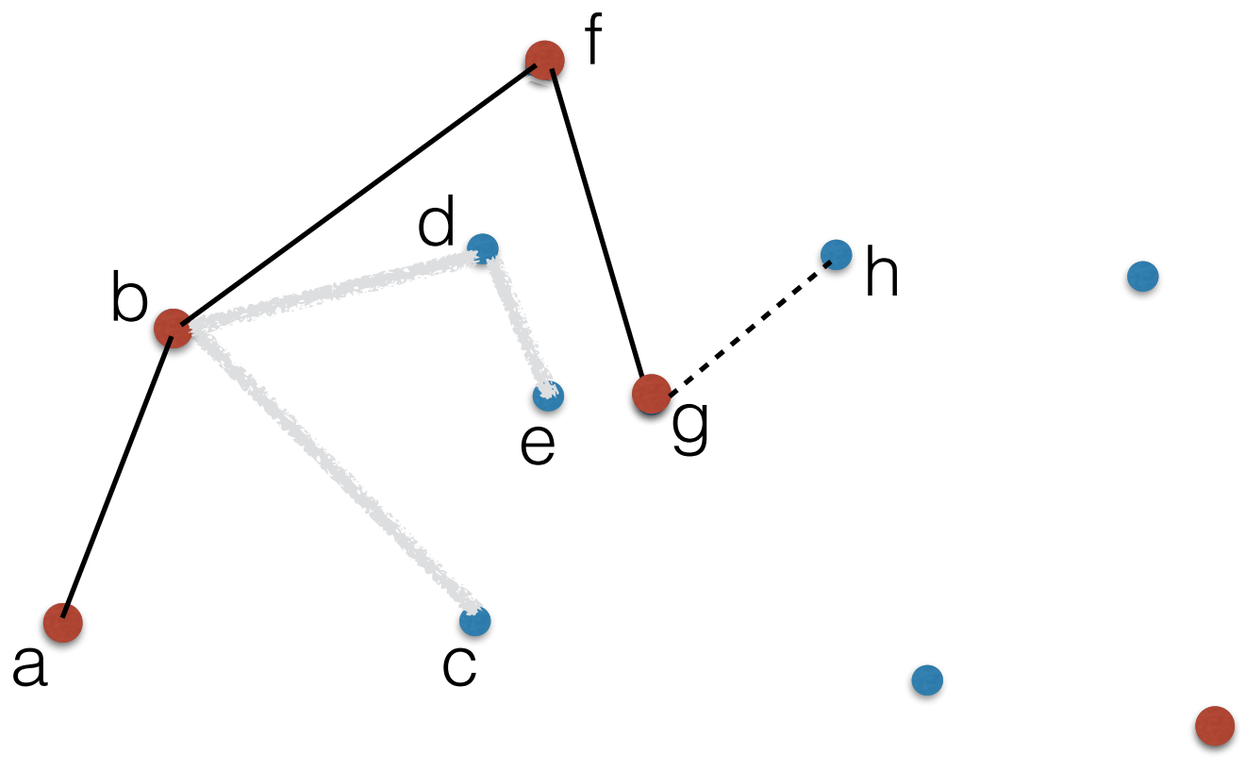
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



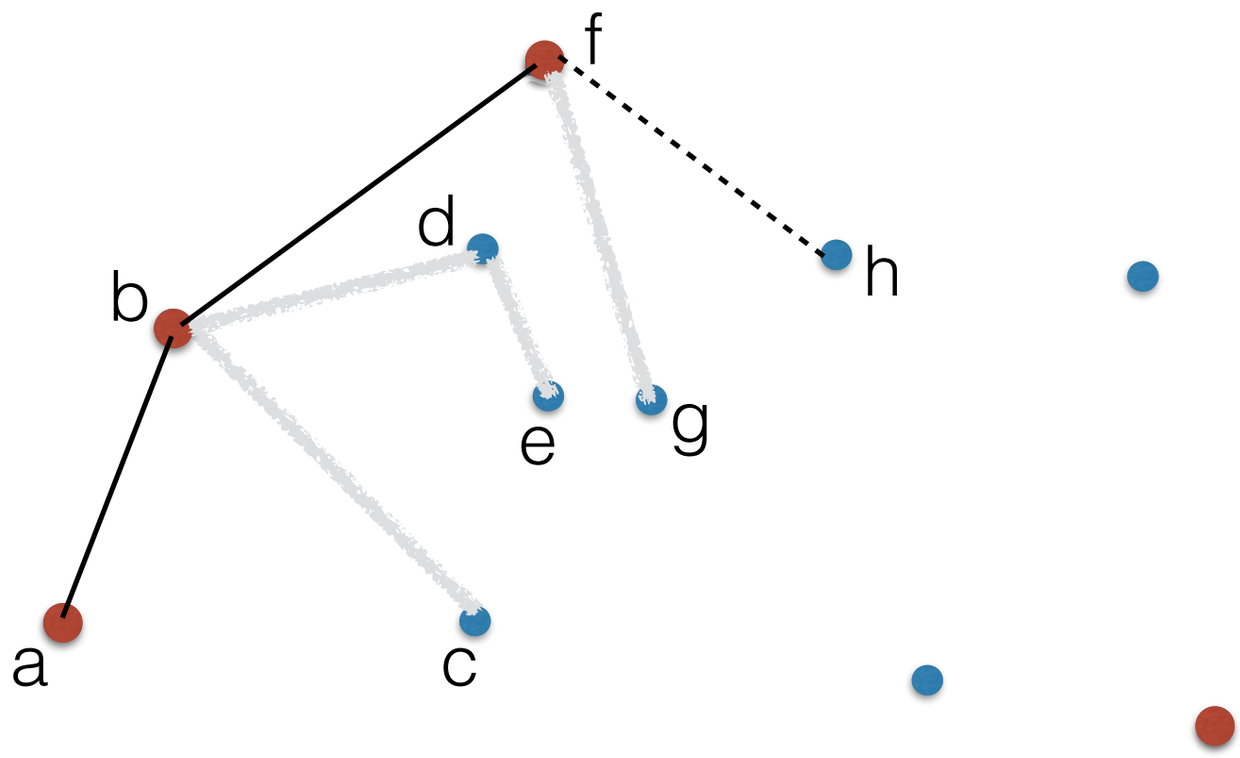
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



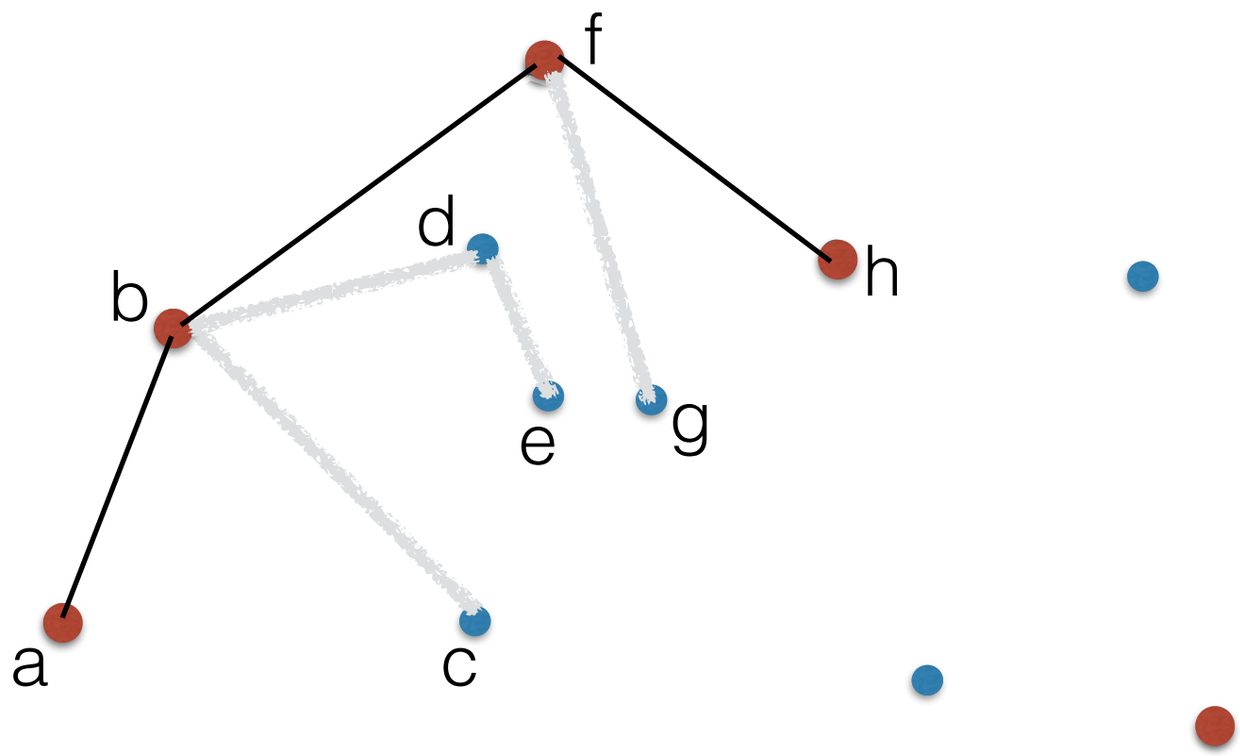
Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



Andrew's Monotone Chain Algorithm (1979)

- Idea: Sort and traverse points lexicographically



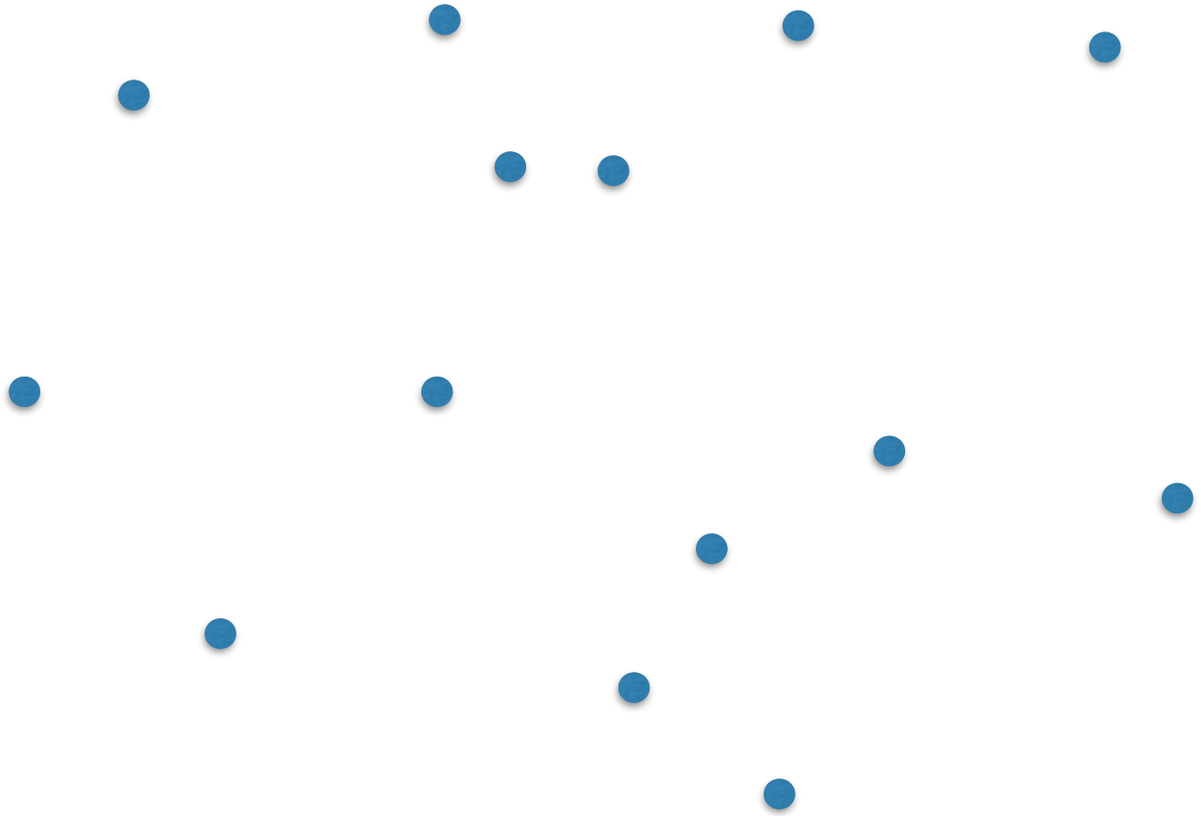
Andrew's Monotone Chain Algorithm (1979)

- Alternative to Graham's scan
- Idea: Sort and traverse points lexicographically (instead of radially)
- Runs in sort + scan
- Sorting lexicographically is faster than sorting radially

Incremental algo for CH

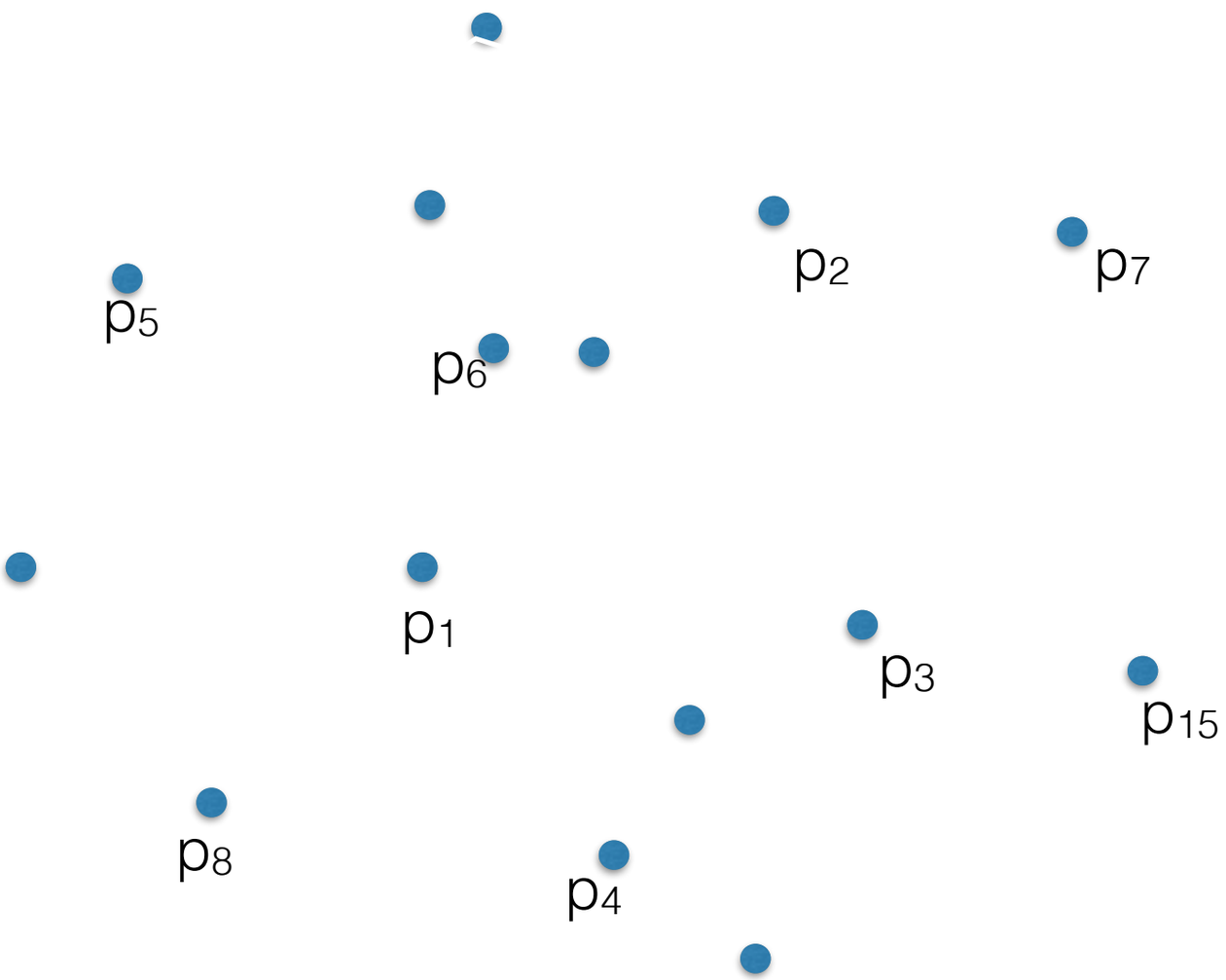
- Idea: traverse the points one at a time and maintain the CH of points read so far
  - CH = {}
  - for i=1 to n
    - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)





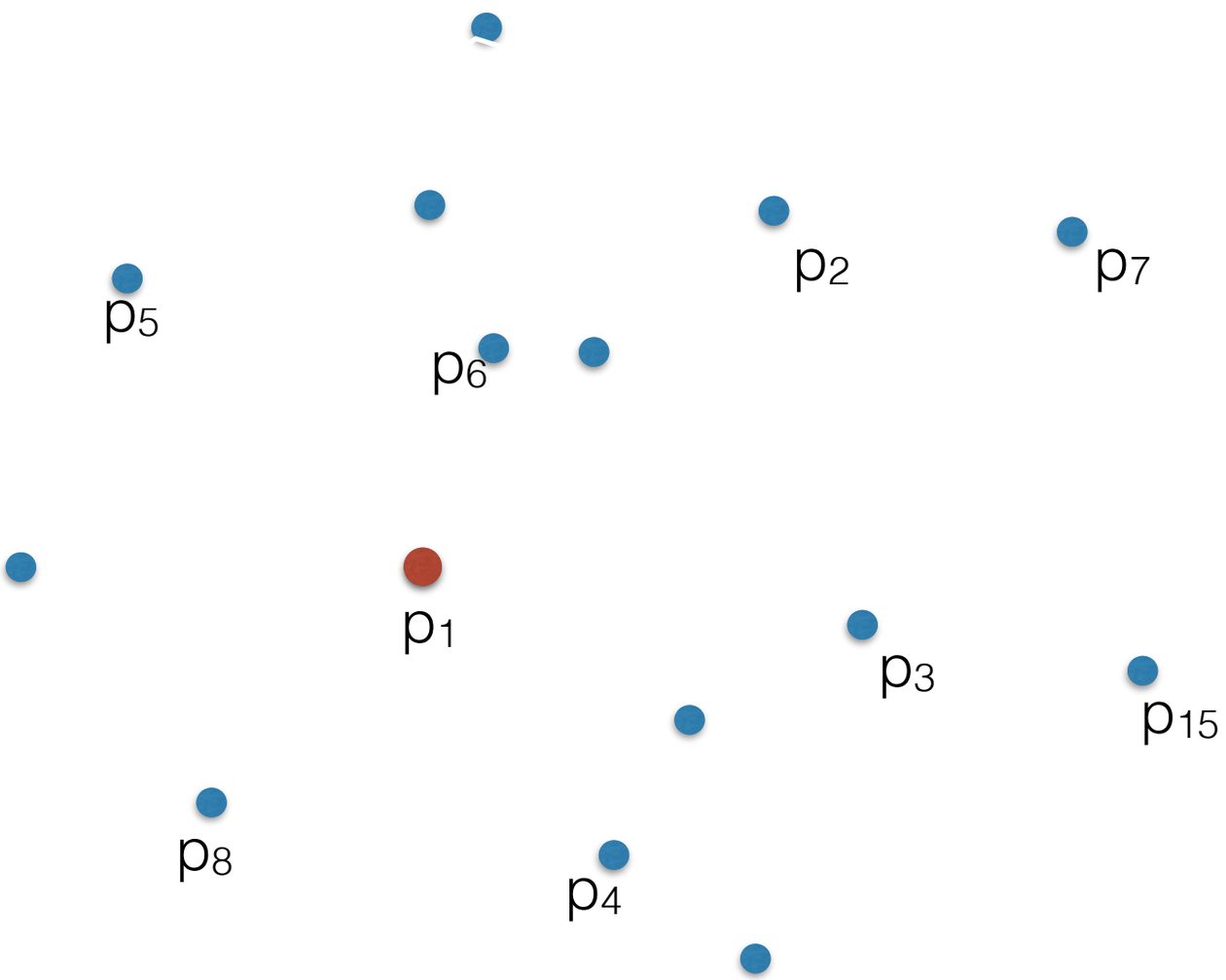
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



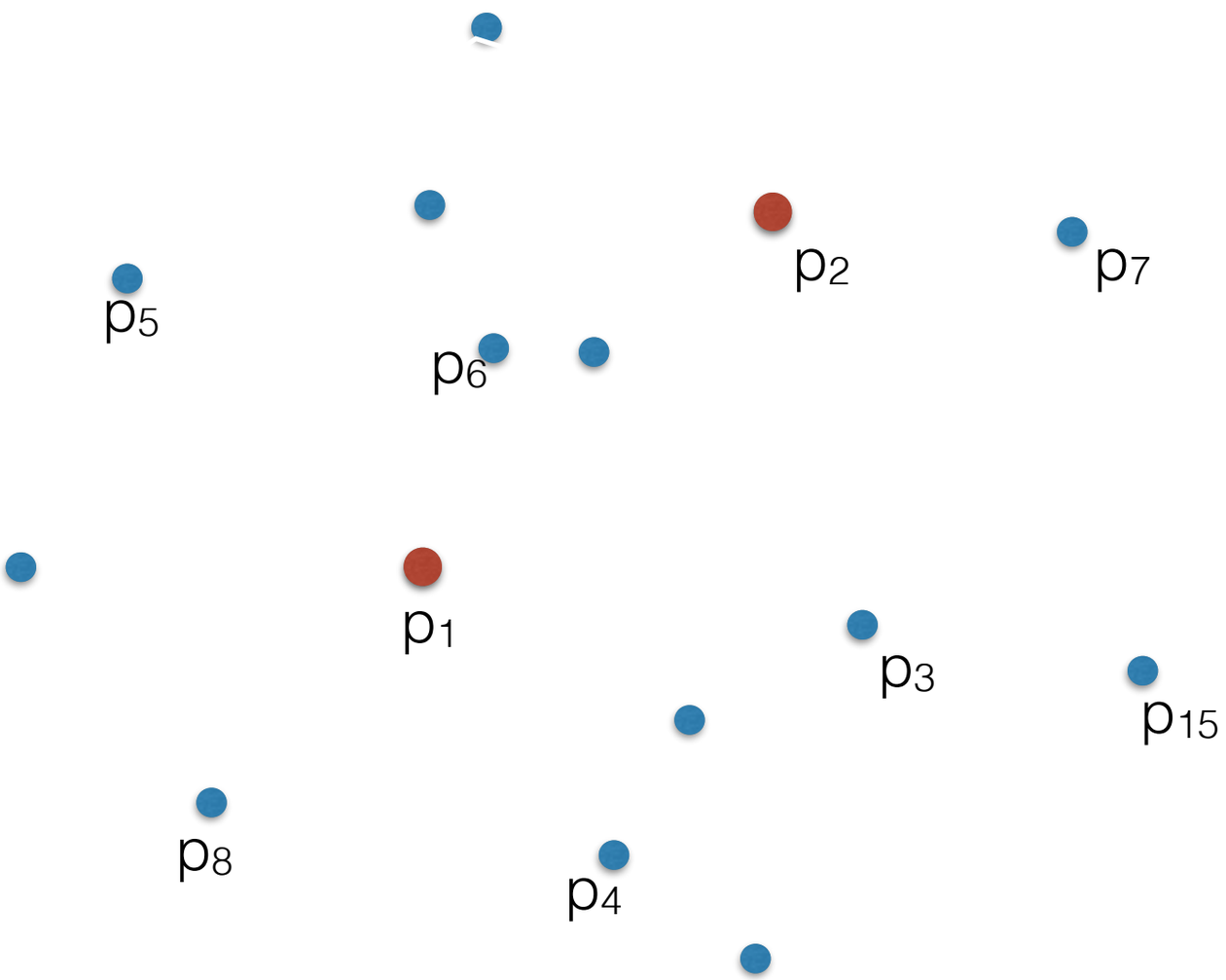
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



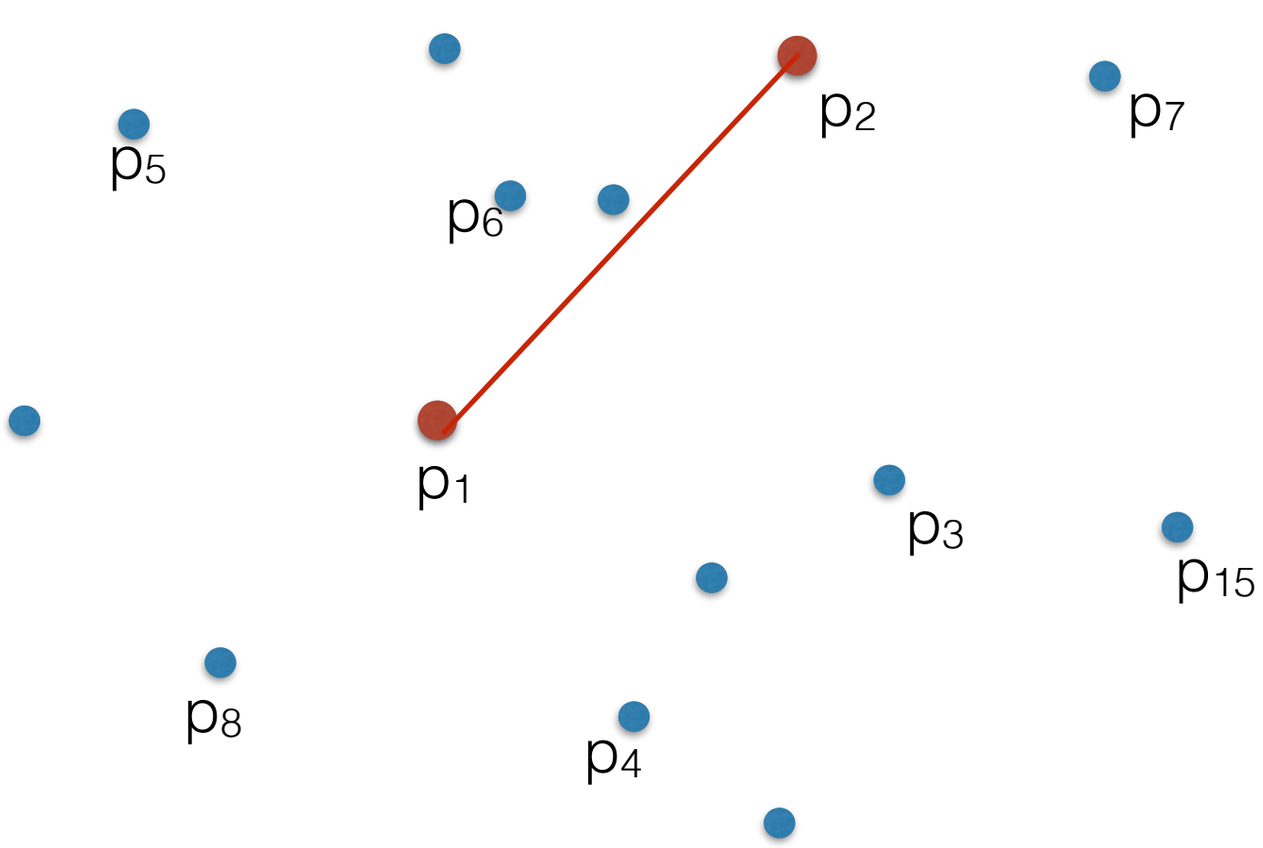
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



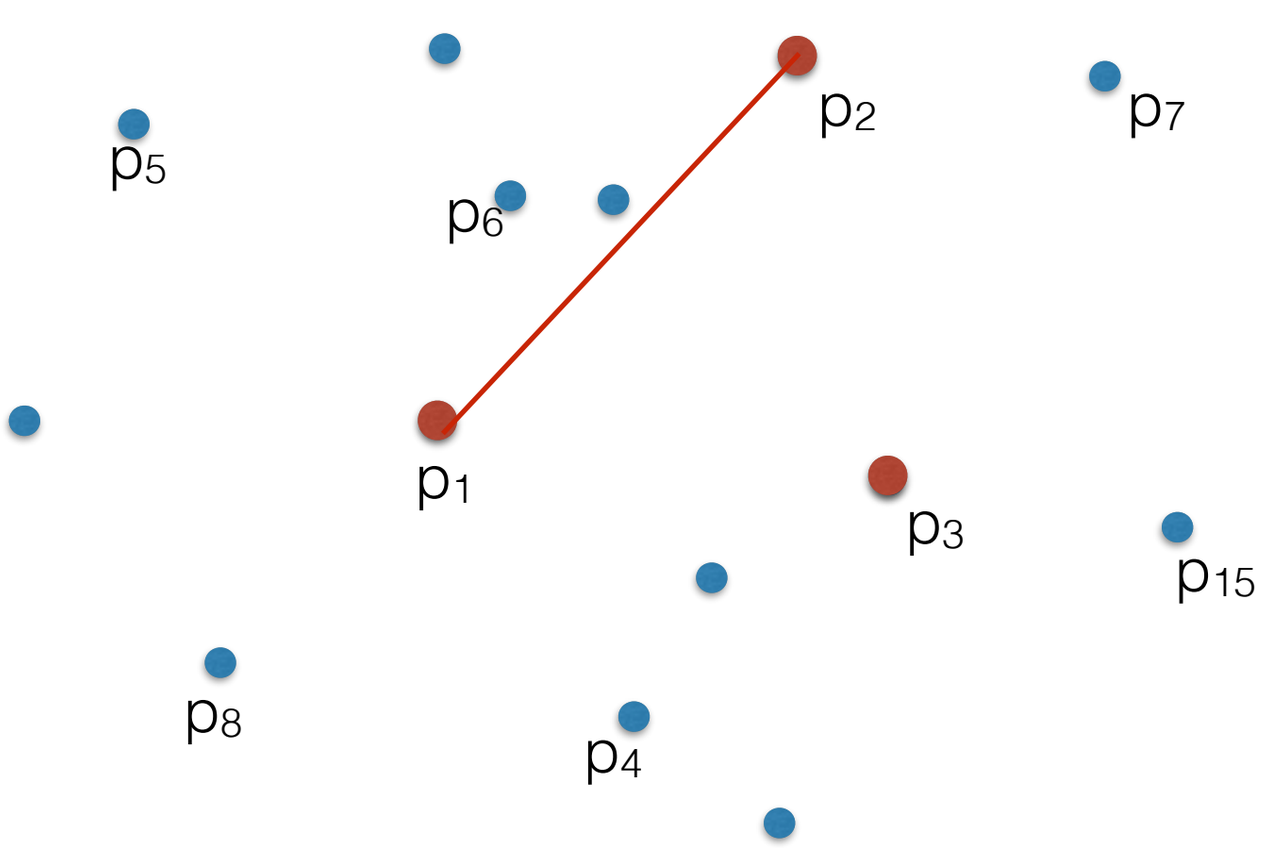
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



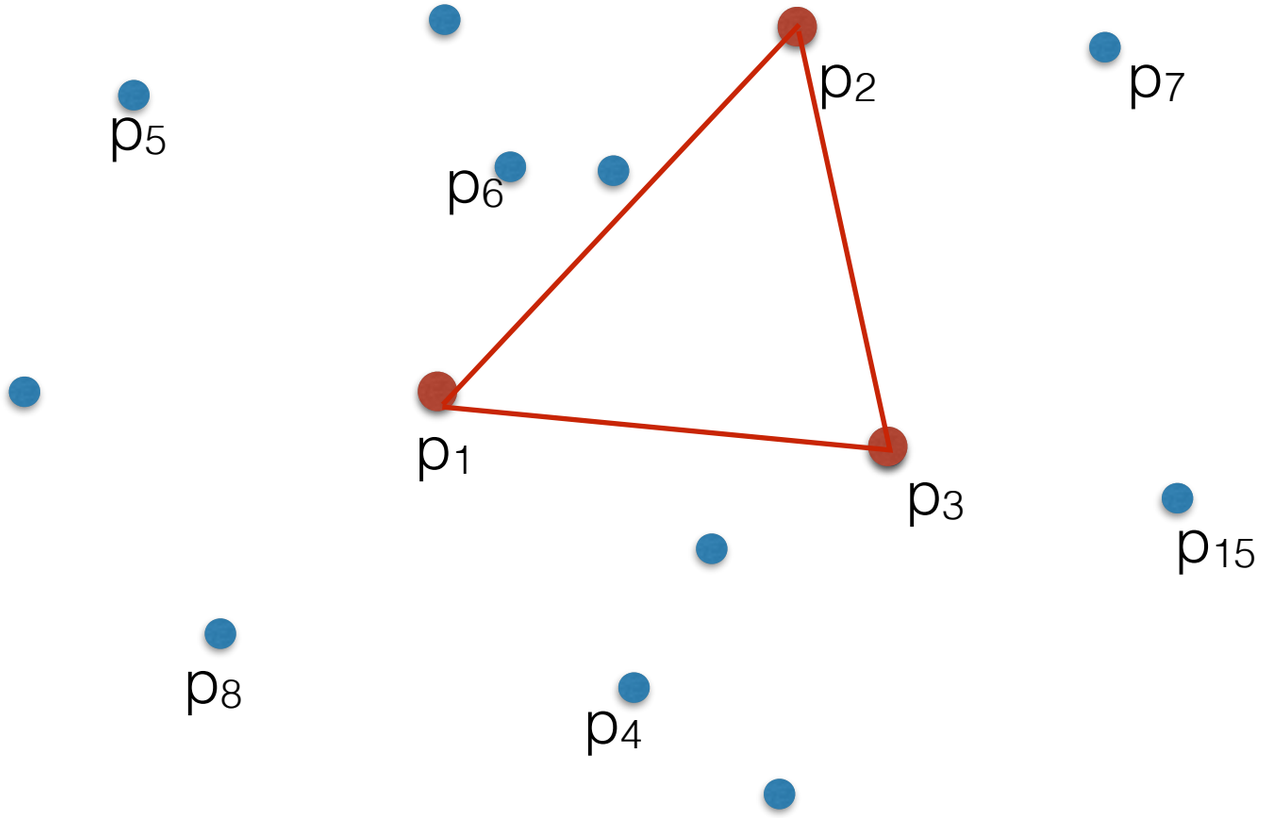
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



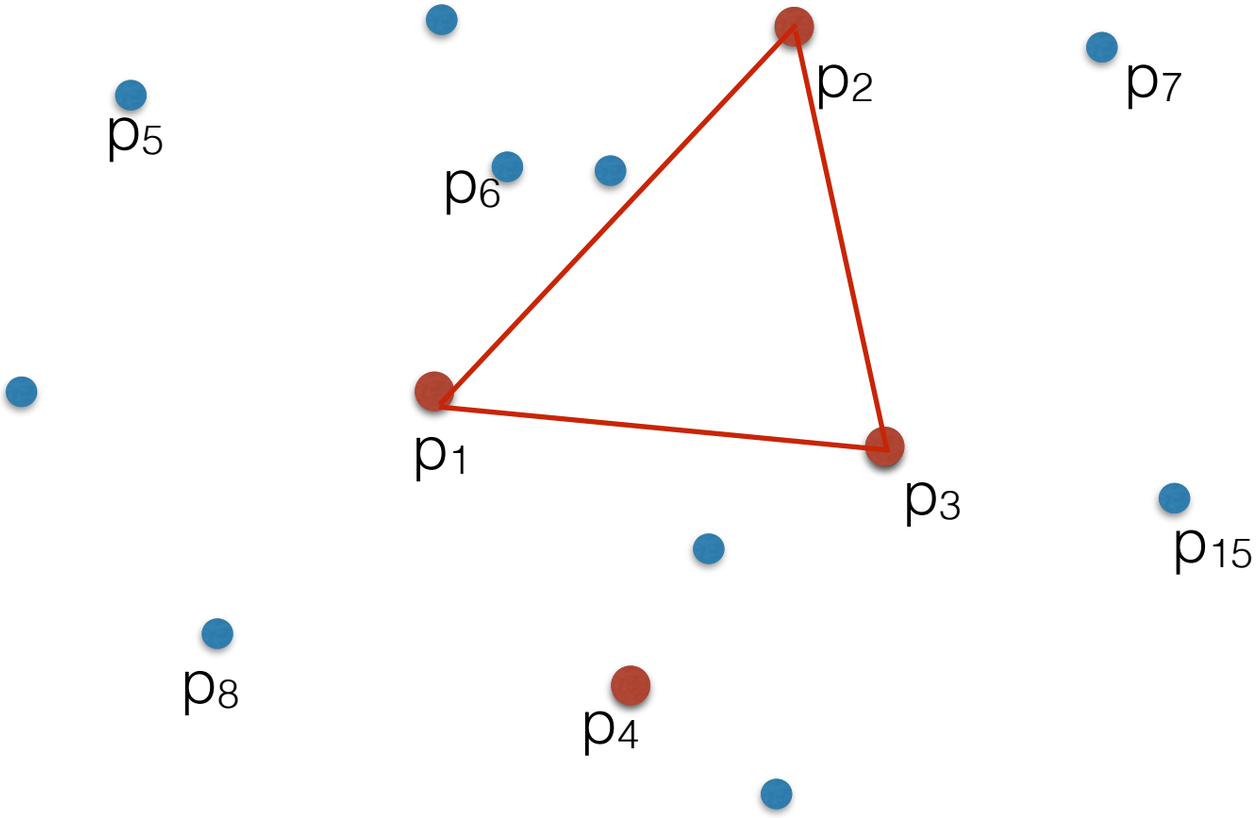
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



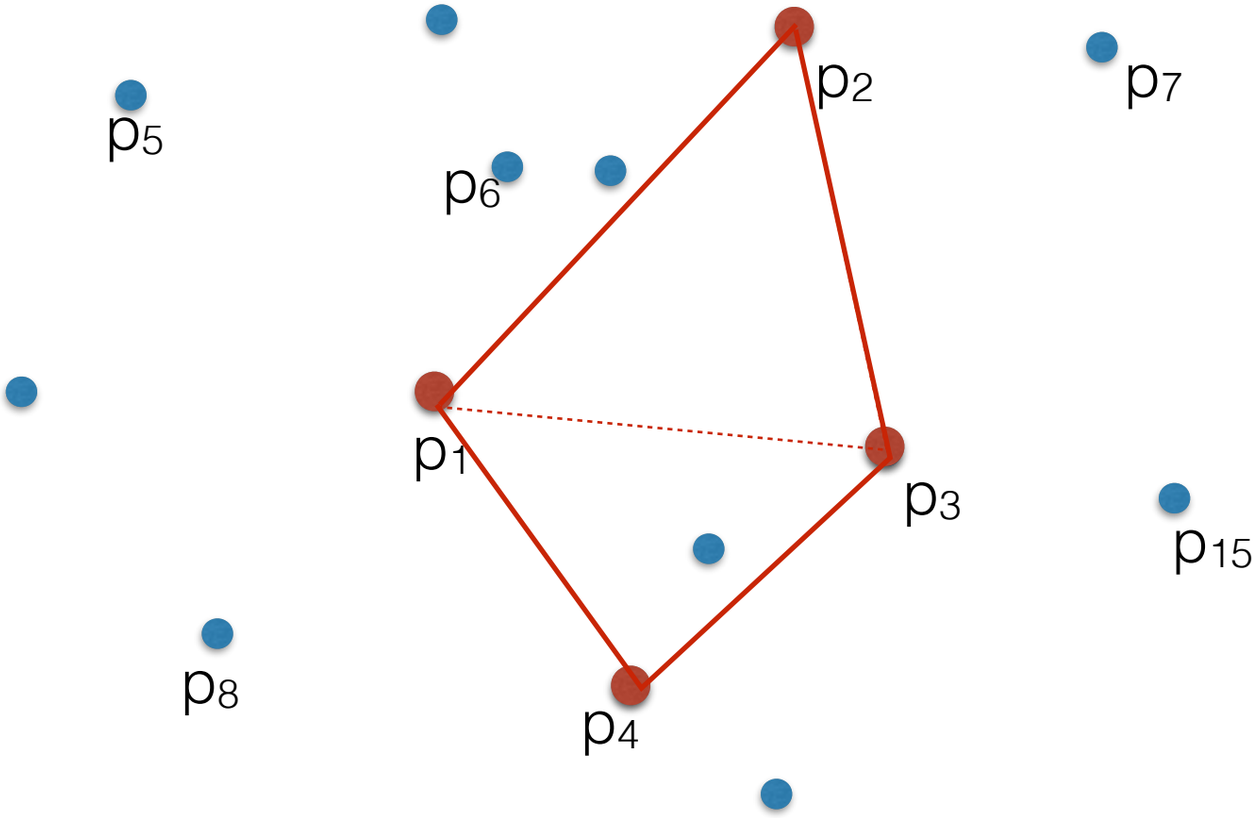
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



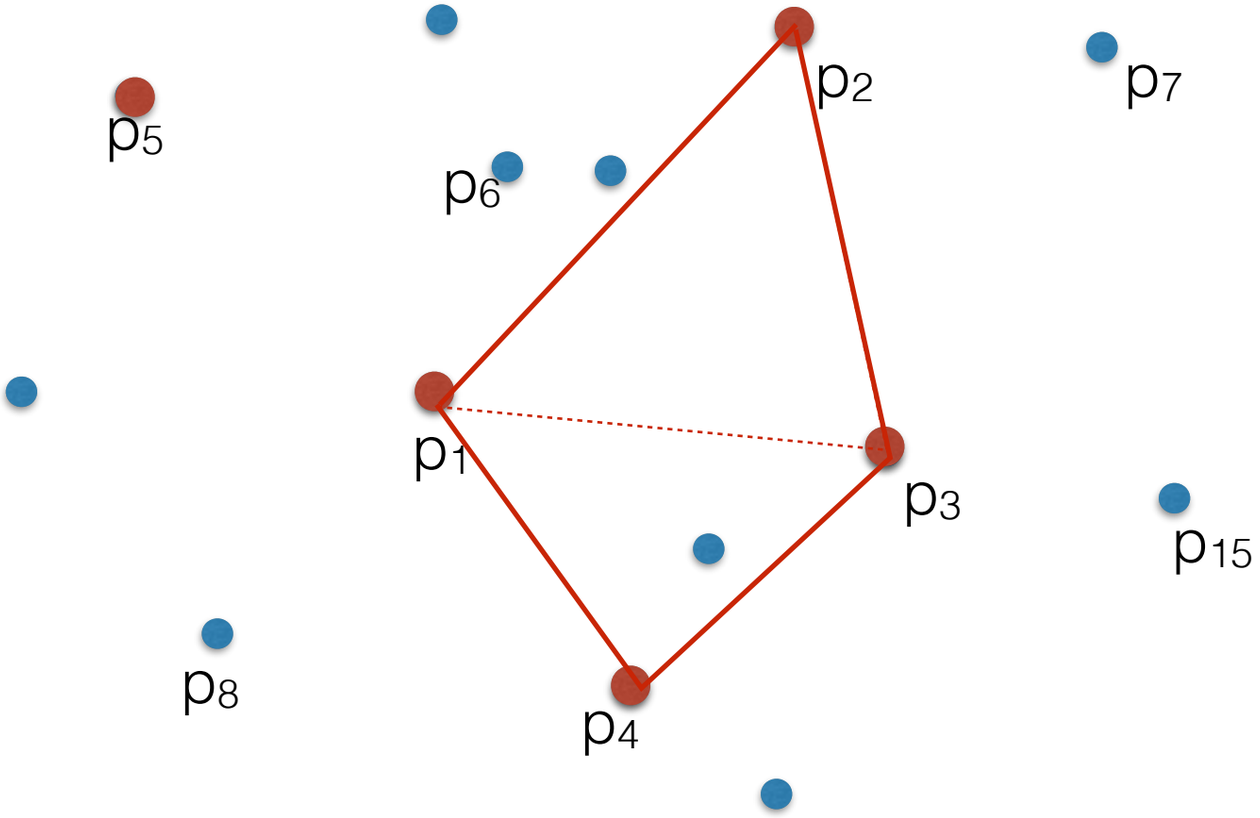
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



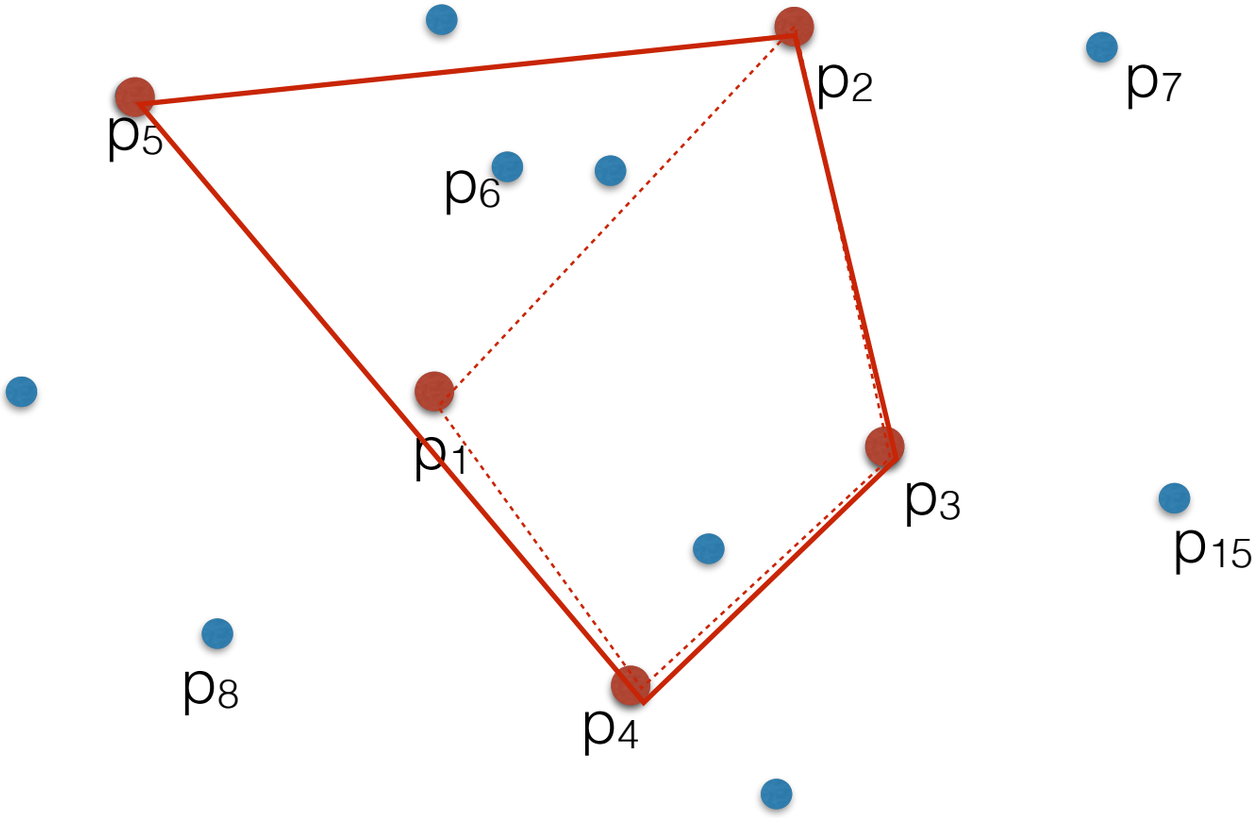
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
  - CH = {}
  - for i=1 to n
    - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



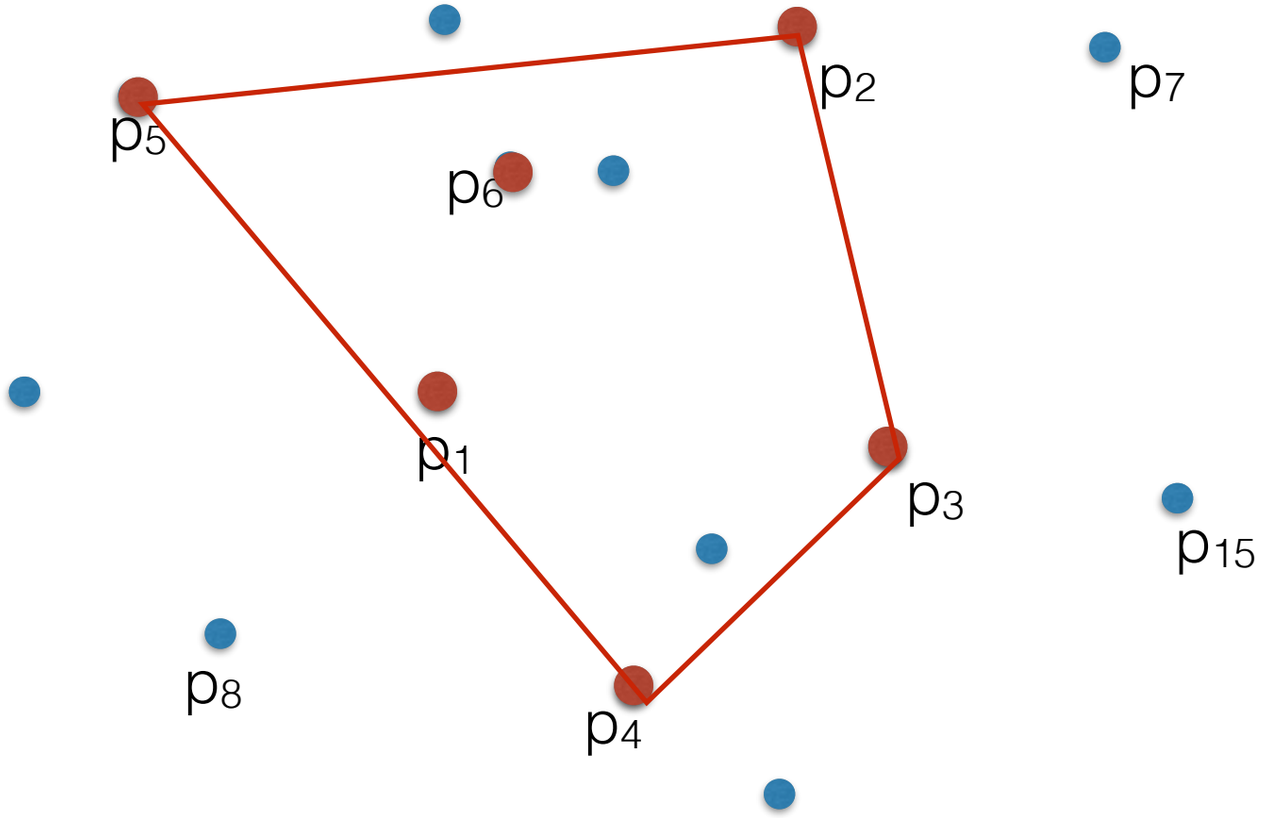
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



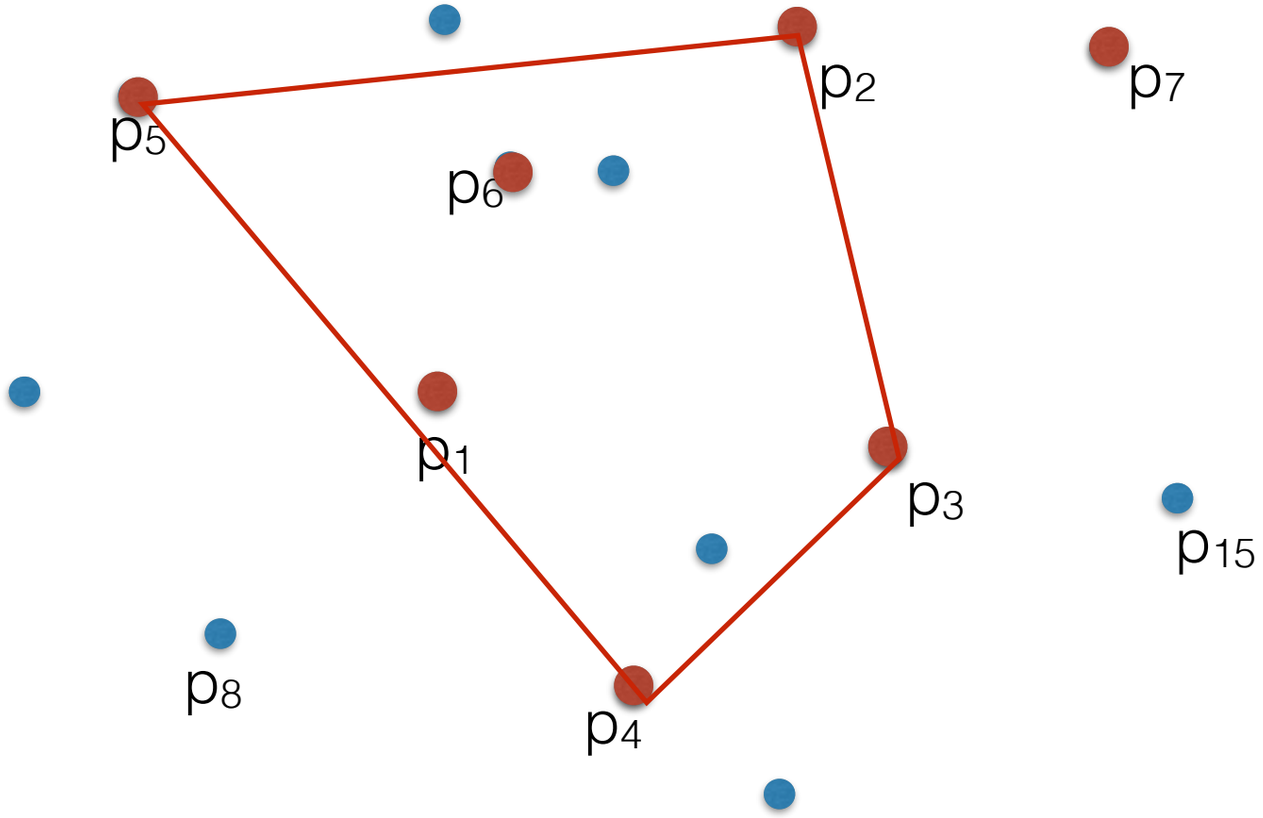
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



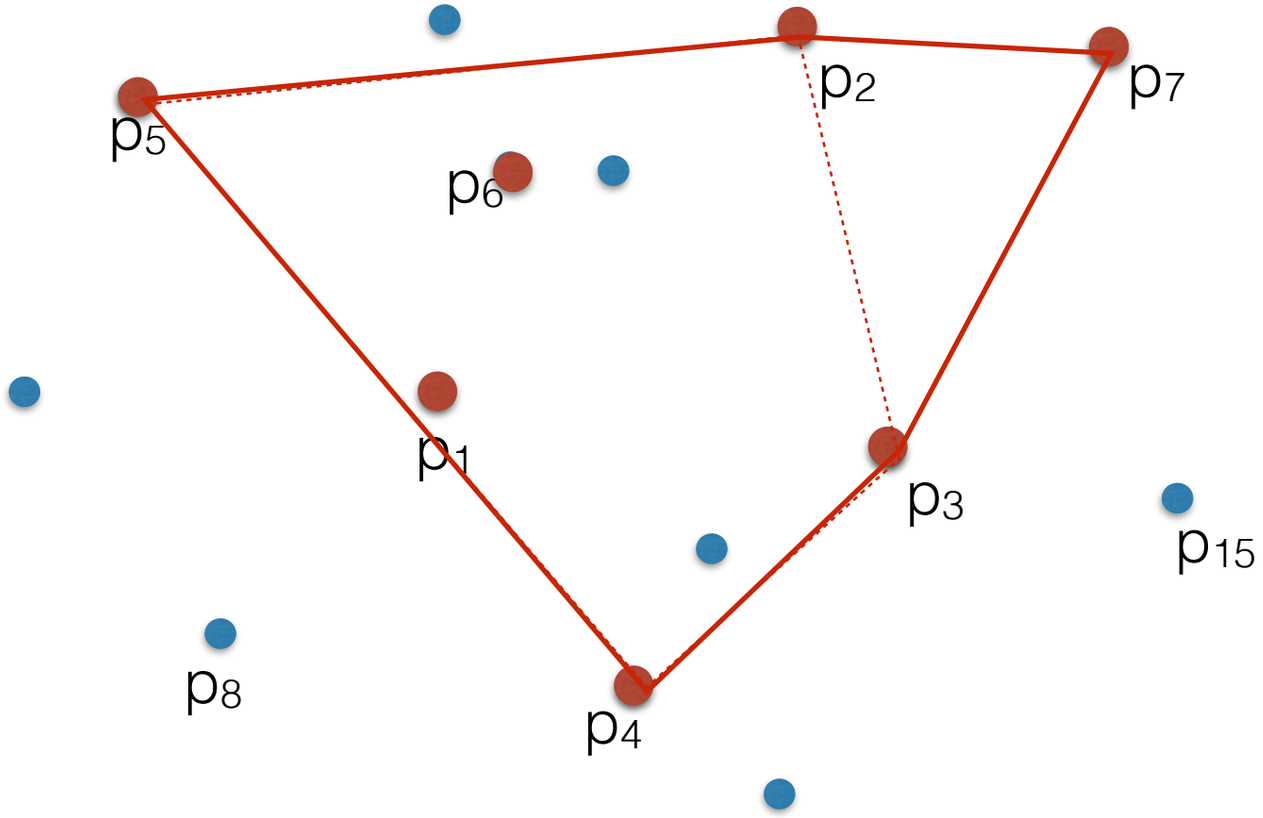
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



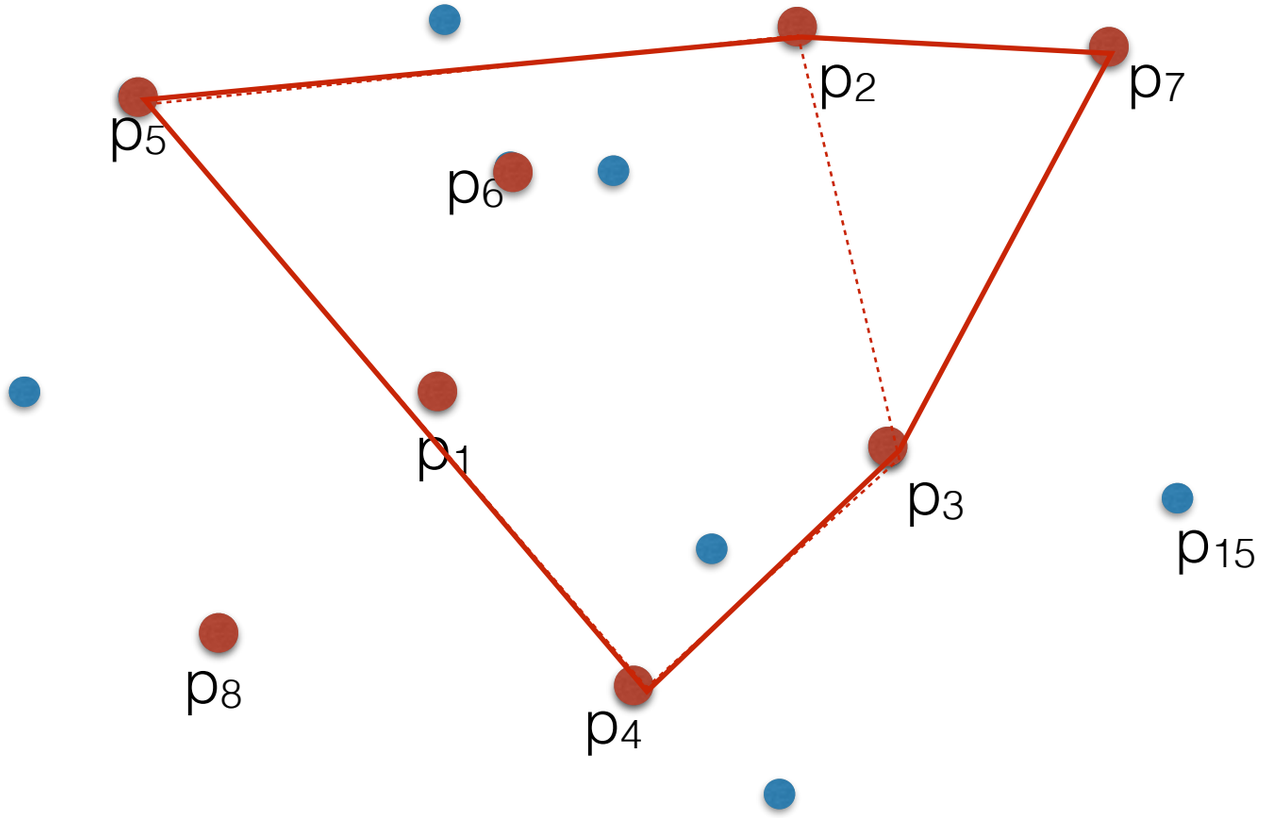
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



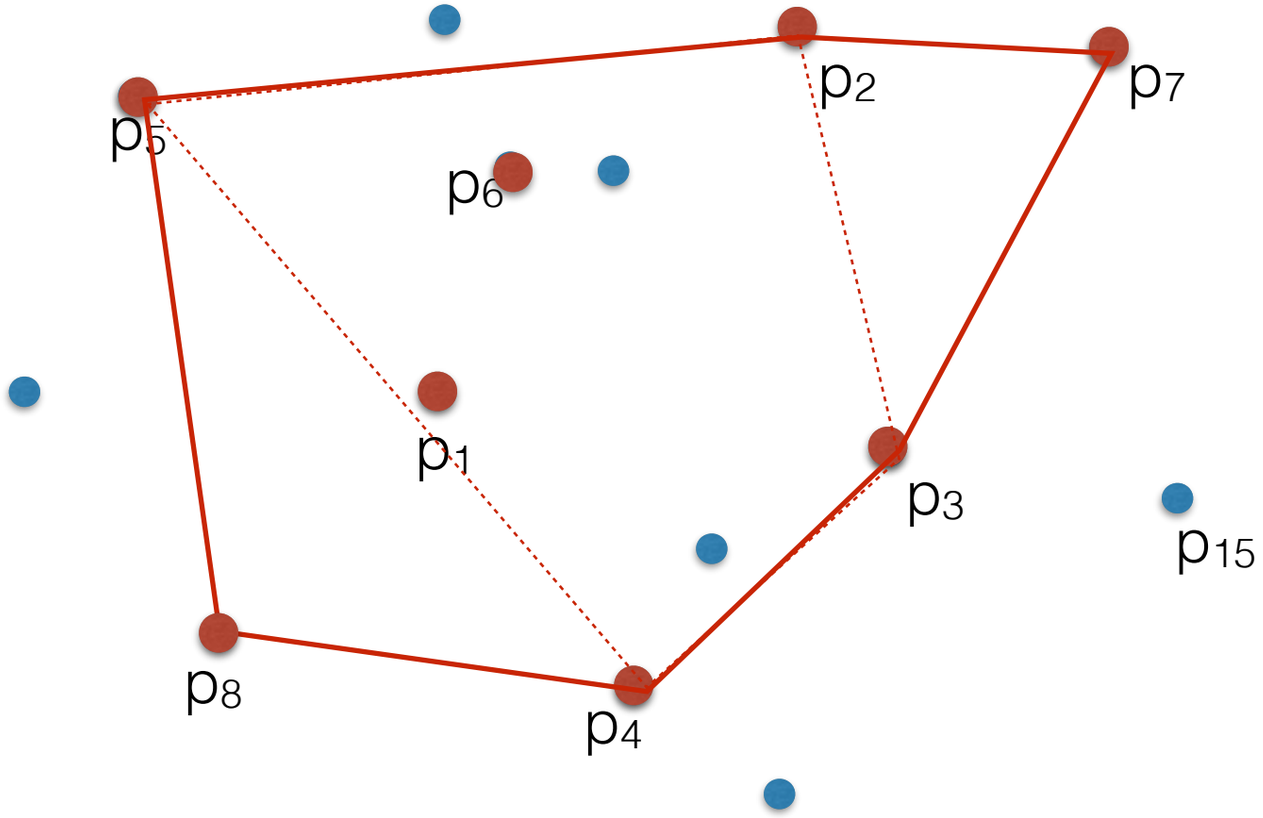
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



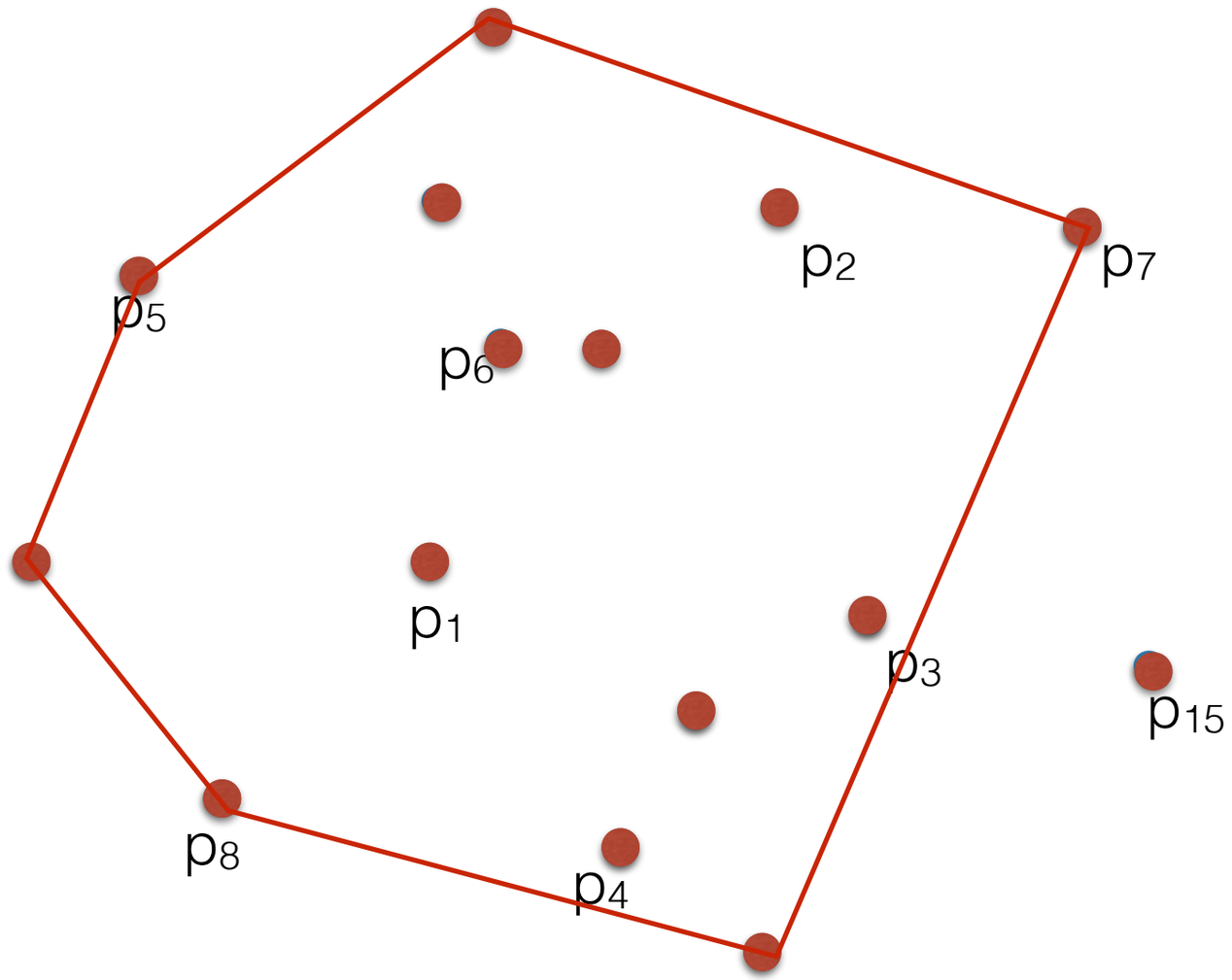
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



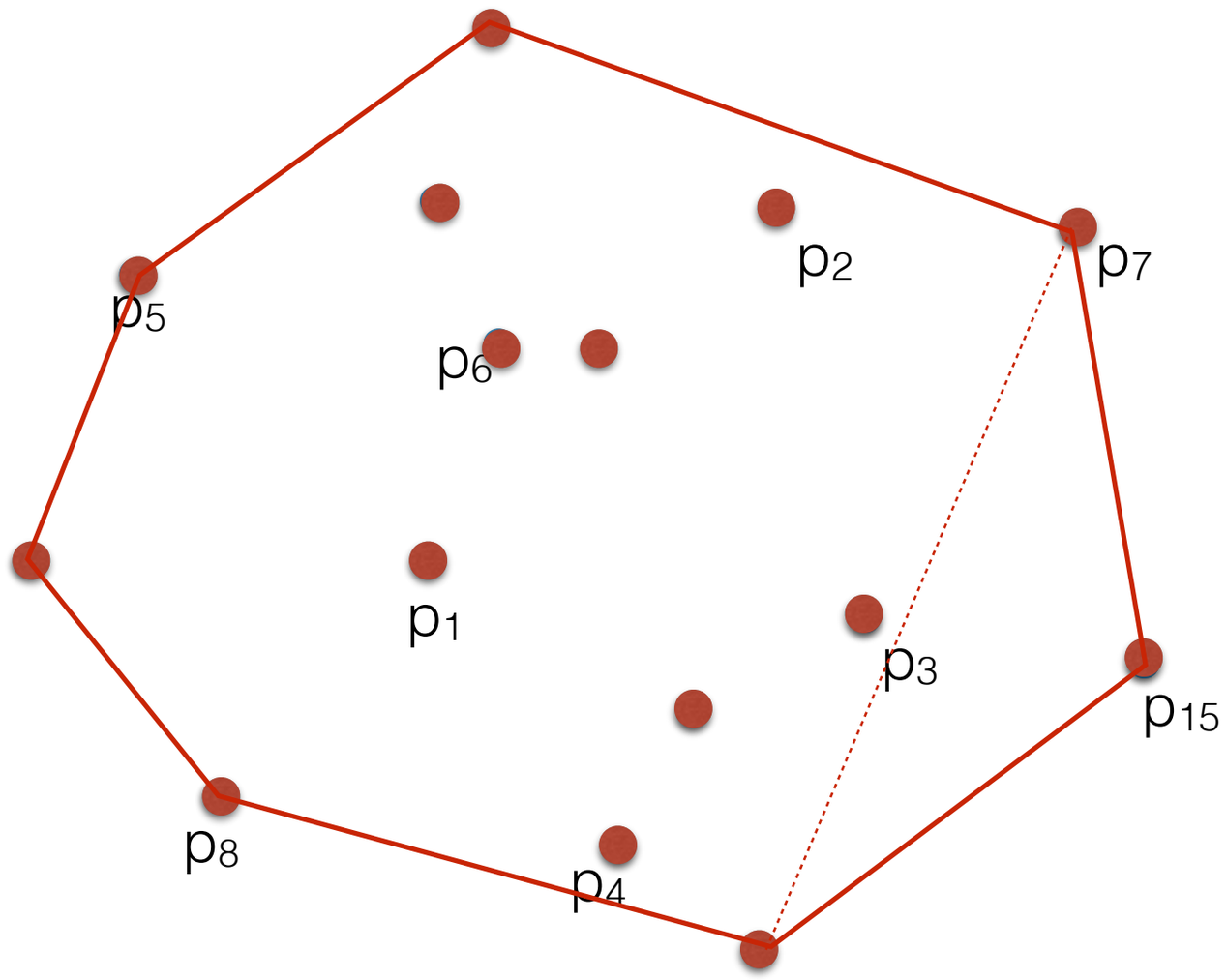
Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
    - CH = {}
    - for i=1 to n
        - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)



Incremental algo for CH

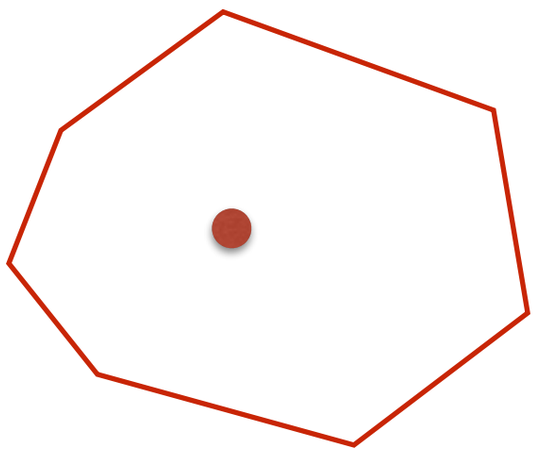
- Idea: traverse the points one at a time and maintain the CH of points read so far
  - CH = {}
  - for i=1 to n
    - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)

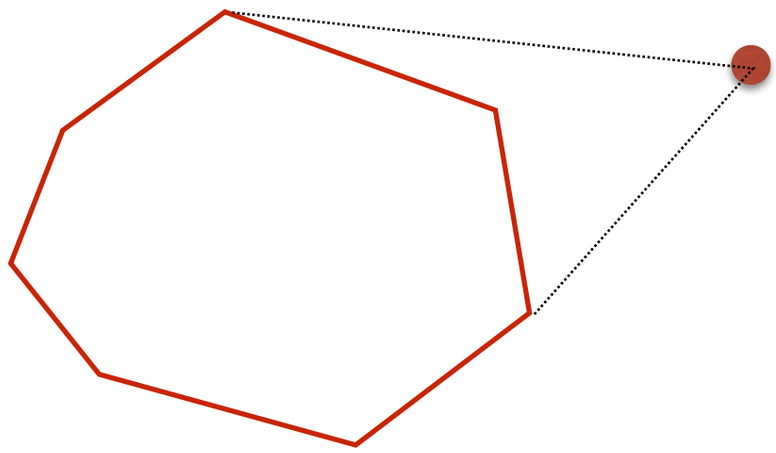
- The basic operation is adding a point to a convex polygon
  - How many cases can you think of?
  - How would you handle each?

Incremental algo for CH

- Idea: traverse the points one at a time and maintain the CH of points read so far
  - CH = {}
  - for i=1 to n
    - add p_i into CH(p_1..p_{i-1}) to compute CH(p_1..p_i)

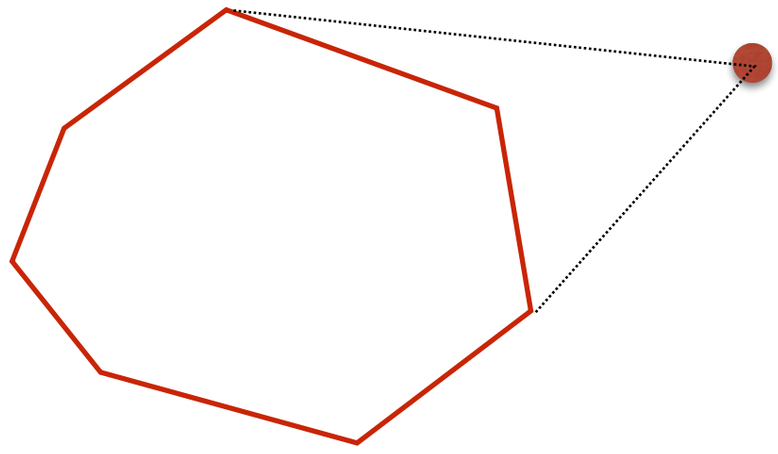
- The basic operation is adding a point to a convex polygon
  - CASE 1: p is in polygon
  - CASE 2: p outside polygon





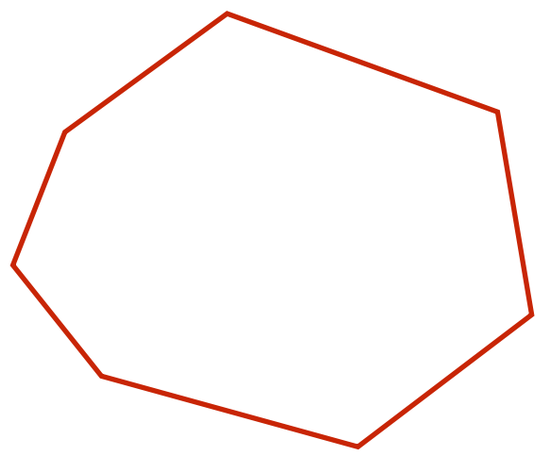
Incremental algo for CH

- Questions
  - What's a good representation for a polygon?
  - A point-in-polygon test ?
  - How to handle CASE 2 ?



Representing a polygon

A polygon is essentially a list of vertices in boundary order; the convention is ccw order.

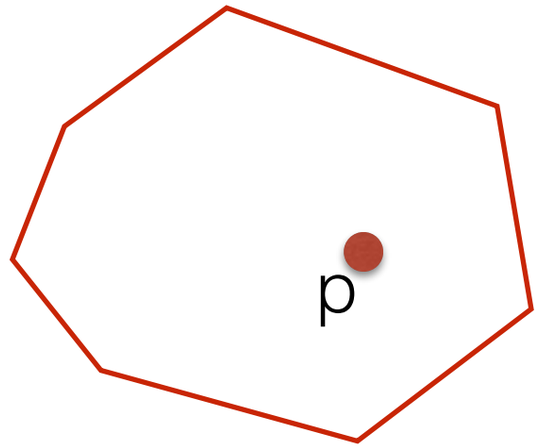


```
typedef struct _polygon{
        int k; //number of vertices
        Point* vertices; //the vertices, ccw in boundary order
  } Polygon;

  or

Vector<Point>          //note: the vertices, ccw in boundary order
```

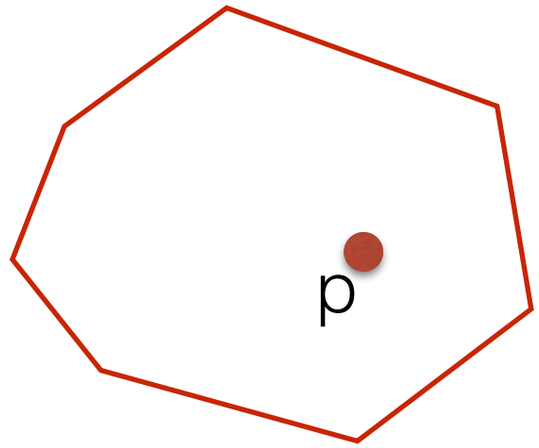
Point in convex polygon



//return TRUE iff p on the boundary or inside H; H is convex a polygon

bool point_in_polygon(point p, polygon H)

Point in convex polygon

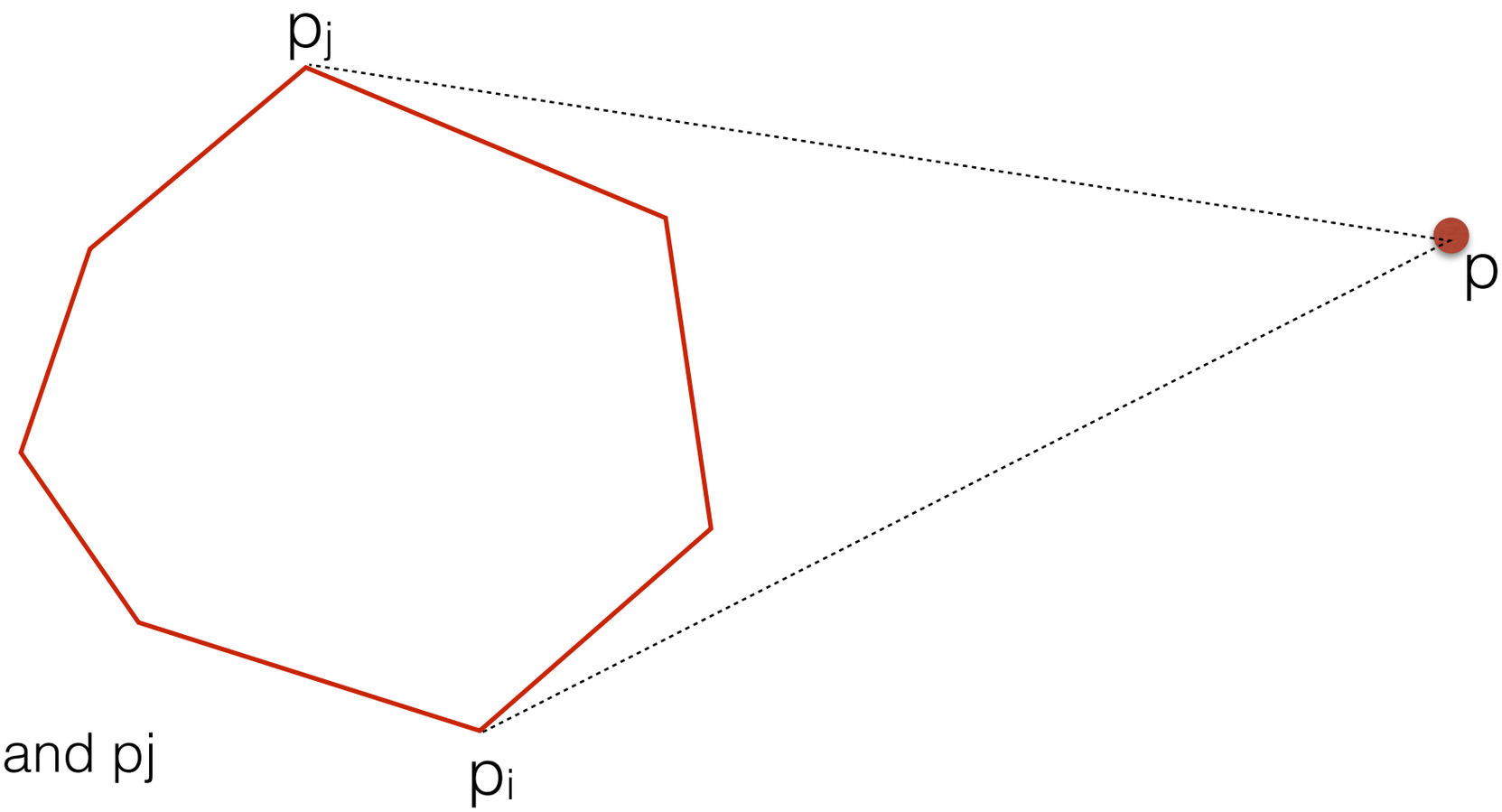



```
//return TRUE iff p on the boundary or inside H; H is convex a polygon
bool point_in_polygon(point p, polygon H)
   //p is inside if and only if it is on or to the left of all edges, oriented ccw
//note:  this is NOT true for a non-convex polygon — can you show a
//counter-example?
```

Analysis:

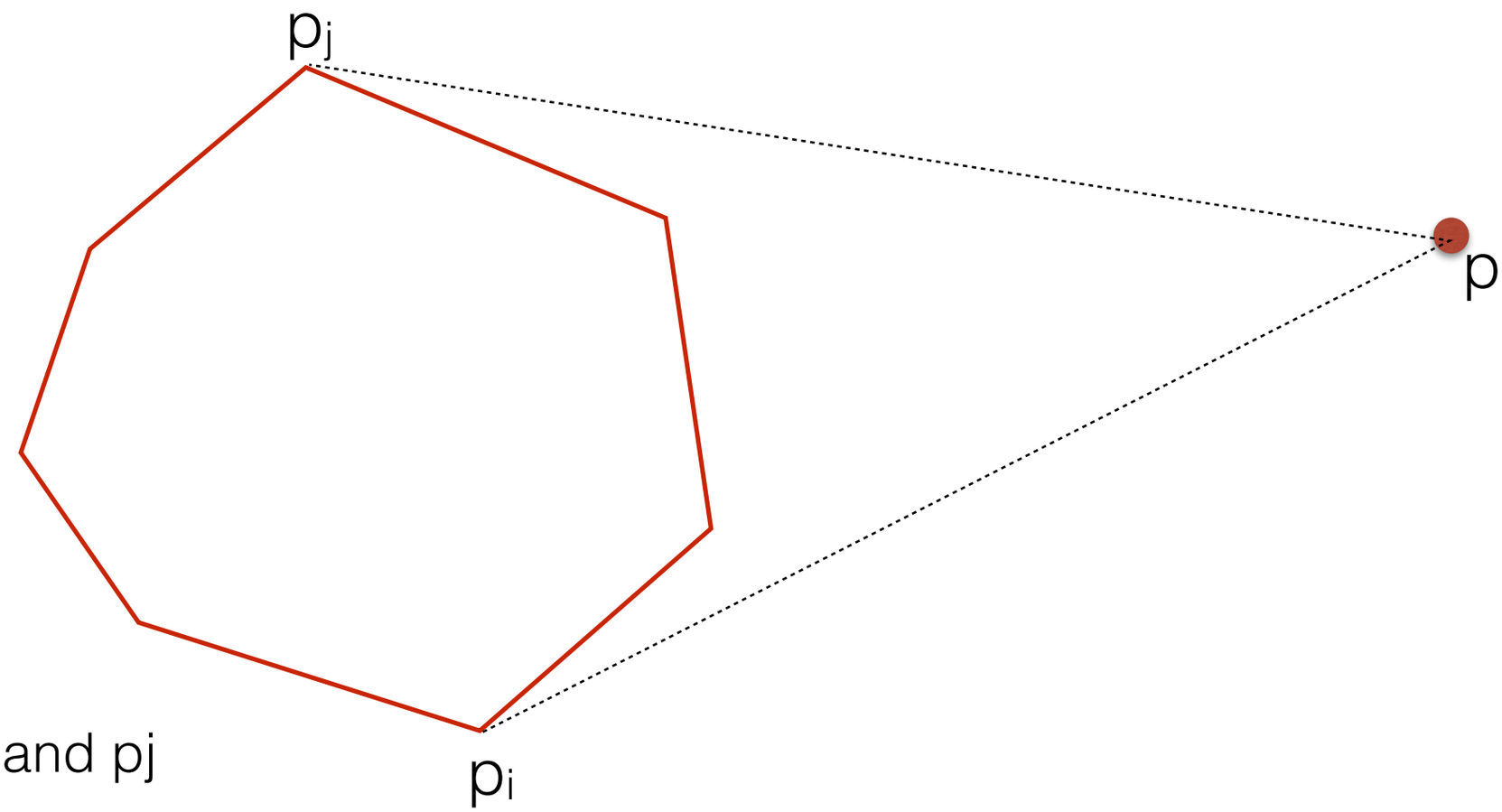
- O(k) where k is the size of the polygon

Case 2:



We want to find pi and pj

Case 2:



We want to find pi and pj

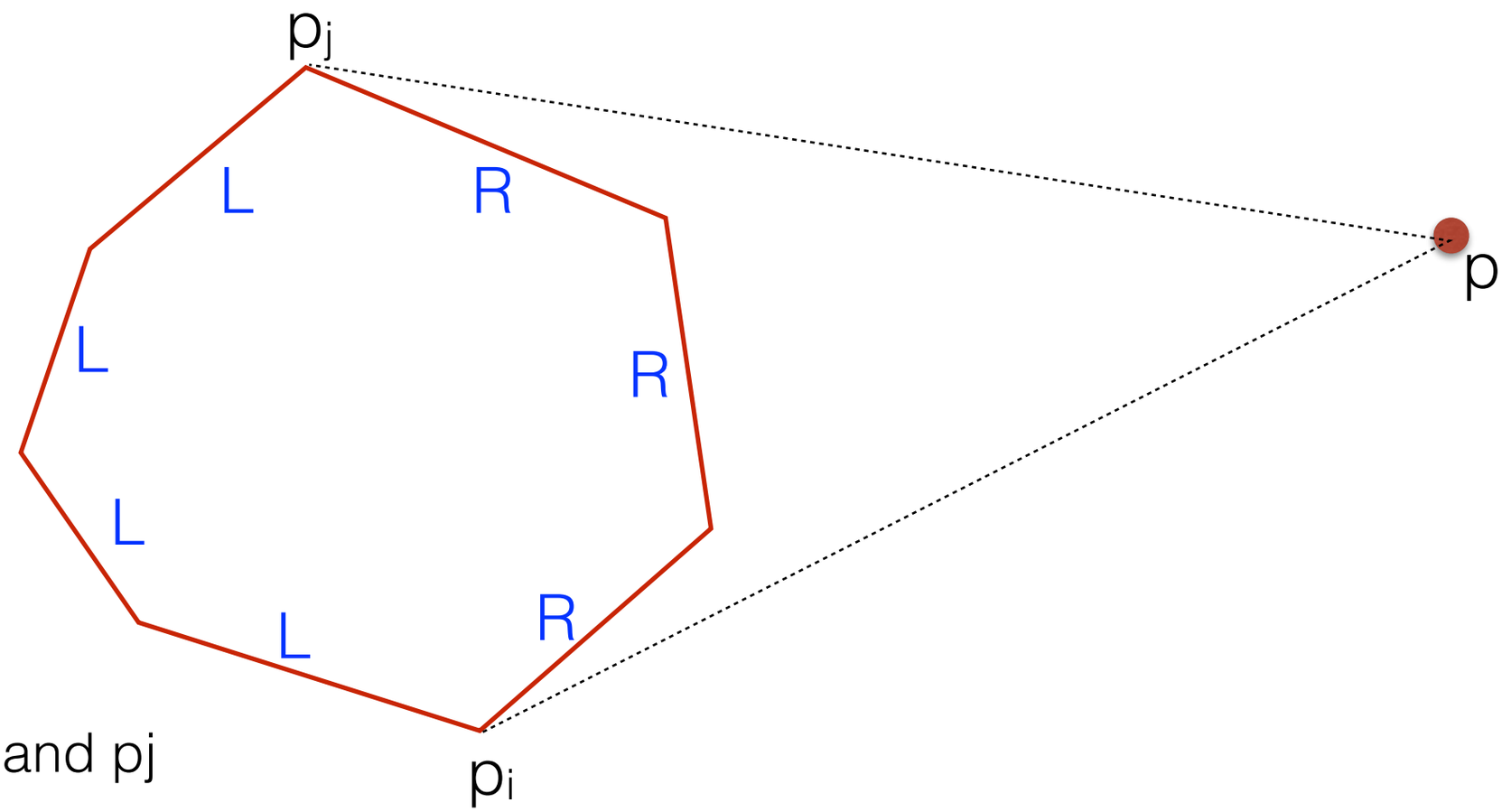
Hint: Check the orientation of p wrt the edges of the polygon.

What do you notice?

How can you use this to find the tangent points?

How long does it take?

Case 2:



We want to find pi and pj

Hint: Check the orientation of p wrt the edges of the polygon.

What do you notice?

How can you use this to find the tangent points?

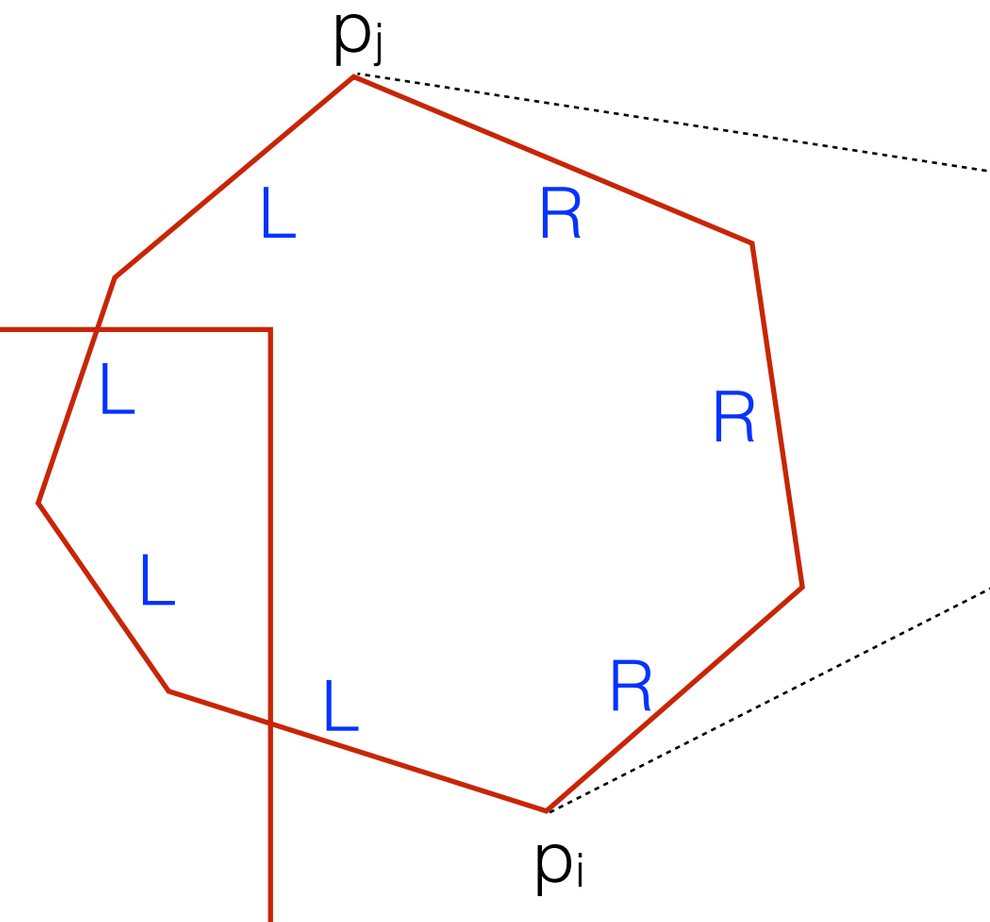
How long does it take?

Finding tangent points

point p outside H

let H = [p_0, p_1,…, p_{k-1}], H convex

- *for i=0 to k-1 do*
  - *prev = ((i == 0)? k-1: i-1);*
  - *next = (i==k-1)? 0; k+1);*
  - *if XOR (p is left-or-on (p_{prev}, p_i), p is left-or-on(p_i, p_{next}))*
  - *then p_i is a tangent point*



After finding p_i and p_j, how would you update H?

Incremental algo for CH

- *H = [p1, p2, p3]*
- *for i=4 to n do*
  - *//add p_i to H*
  - *if point_in_polygon(p, H)*
    - *//do nothing*
  - *else*
    - *let p_i be the tangent point where orientation changes from L to R*
    - *let p_j be the tangent point where orientation changes from R to L*

      //note: p_i not necessarily before p_j in the vertex array of H
    - *cut out the part from p_i to p_j in H (note: view H as wrapping around) and replace it with vertex p*

Incremental algo for CH

- *H = [p1, p2, p3]*
- *for i=4 to n do*
  - *//add p_i to H*
  - *if point_in_polygon(p, H)*
    - *//do nothing*
  - *else*
    - *let p_i be the tangent point where orientation changes from L to R*
    - *let p_j be the tangent point where orientation changes from R to L*

      //note: p_i not necessarily before p_j in the vertex array of H
    - *cut out the part from p_i to p_j in H (note: view H as wrapping around) and replace it with vertex p*

Work out a couple of examples and show how p_i could come before p_j in H or the other way around.

Incremental algo for CH

- *H = [p1, p2, p3]*
- *for i=4 to n do*
  - *//add p_i to H*
  - *if point_in_polygon(p, H)*
    - *//do nothing*
  - *else*
    - *let p_i be the tangent point where orientation changes from L to R*
    - *let p_j be the tangent point where orientation changes from R to L*

      //note: p_i not necessarily before p_j in the vertex array of H
    - *cut out the part from p_i to p_j in H (note: view H as wrapping around) and replace it with vertex p*

Analysis:

Incremental algo for CH

- *H = [p1, p2, p3]*
- *for i=4 to n do*
  - *//add p_i to H*
  - *if point_in_polygon(p, H)* ← O(i)
    - *//do nothing*
  - *else*
    - *let p_i be the tangent point where orientation changes from L to R* ← O(i)
    - *let p_j be the tangent point where orientation changes from R to L*

      //note: p_i not necessarily before p_j in the vertex array of H
    - *cut out the part from p_i to p_j in H (note: view H as wrapping around) and replace it with vertex p*

Analysis: $\text{SUM}_i\ O(i) = O(n^2)$

Incremental algo for CH

- An optimal incremental algorithm is possible!

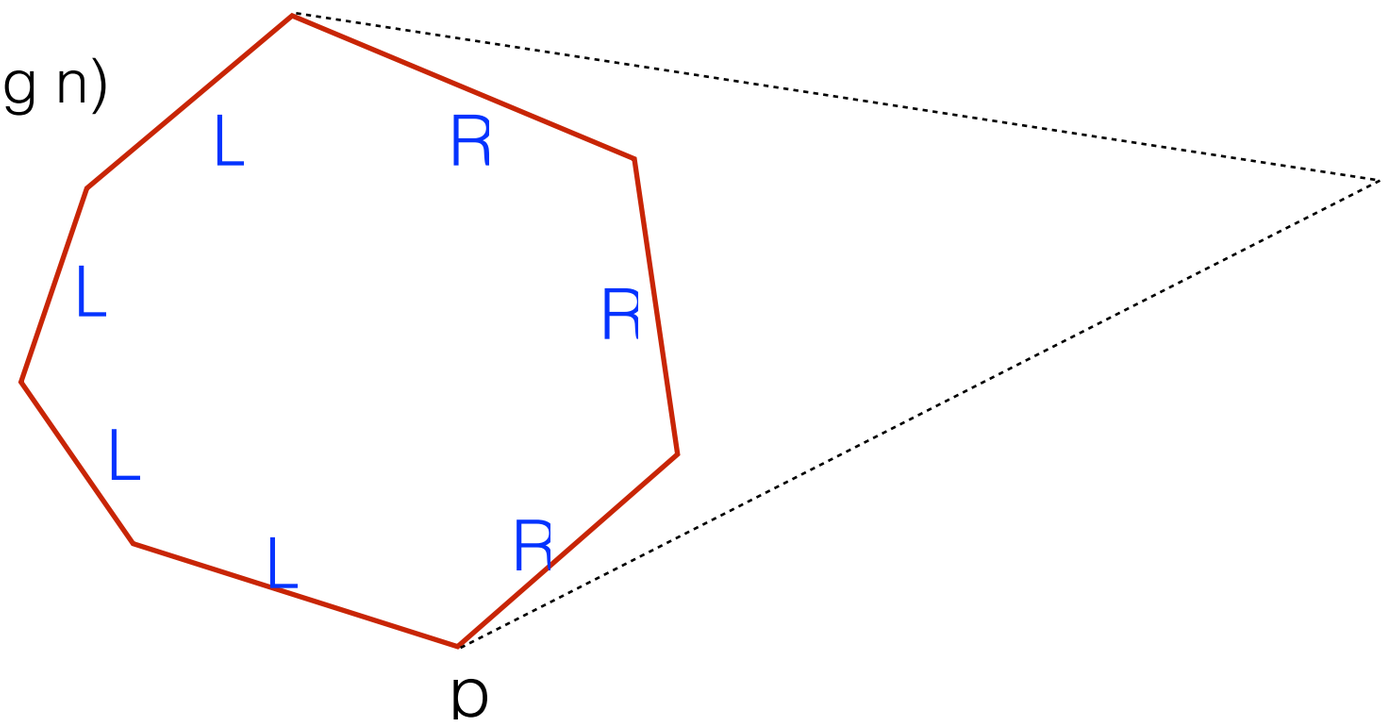
  Ideas:

  1. pre-sort the points by their x-coordinates

  What does this give us?

  2. replace the search for tangent points with some sort of binary search

- Overall such that it runs in O(n lg n)



CH via divide-and-conquer

- Yet another illustration of divide-and-conquer paradigm
- Runs in O(n lg n)
- Somewhat complicated in 2D
- Extends nicely to 3D

Divide-and-conquer

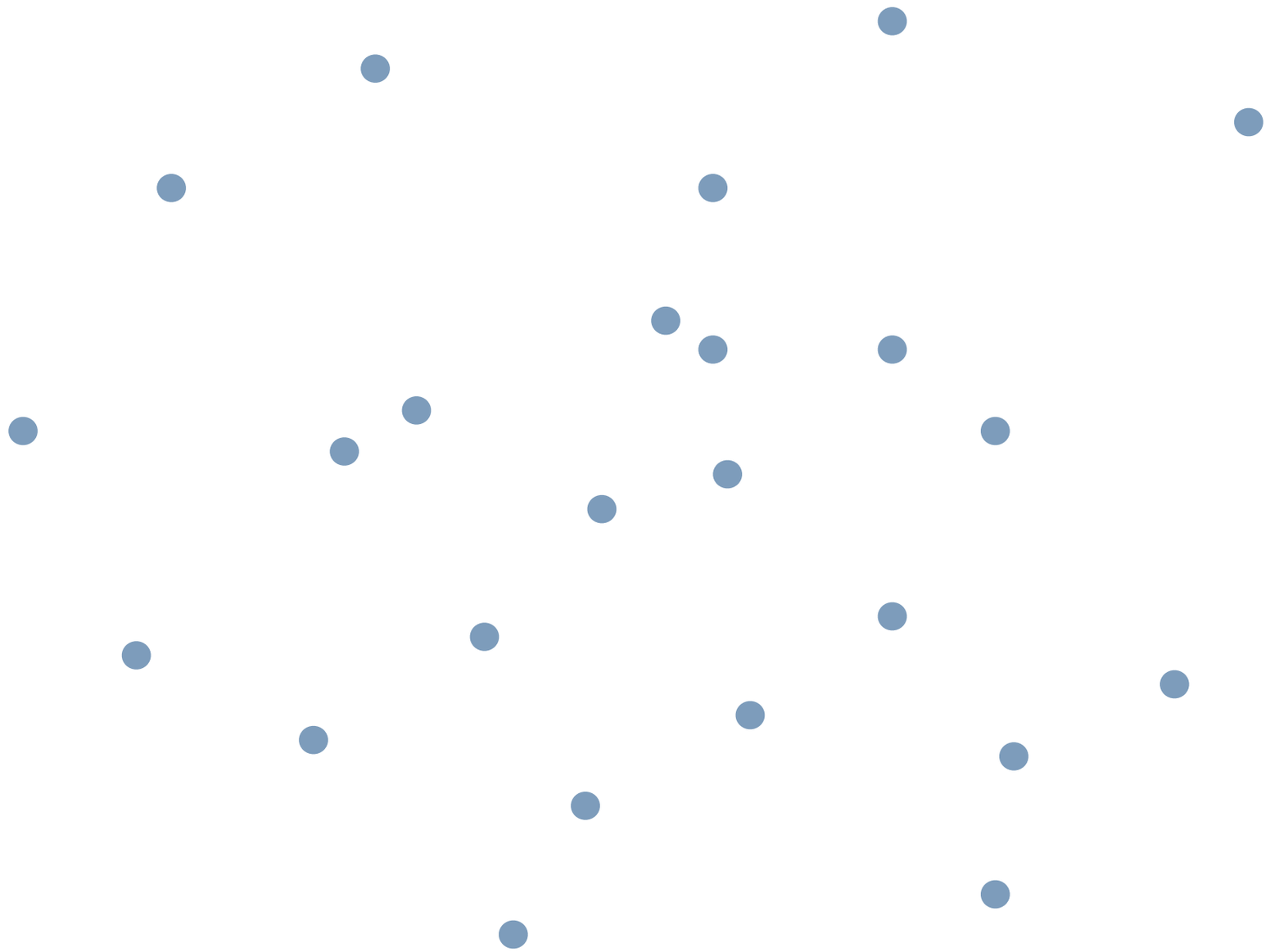
```
DC(input P)
    if P is small, solve and return
    else
        //divide
        divide input P into two halves, P1 and P2
        //recurse
        result1 = DC(P1)
        result2 = DC(P2)
        //merge
        do_something_to_figure_out_result_for_P

        return result
```

Analysis: T(n) = 2T(n/2) + O(merge phase)

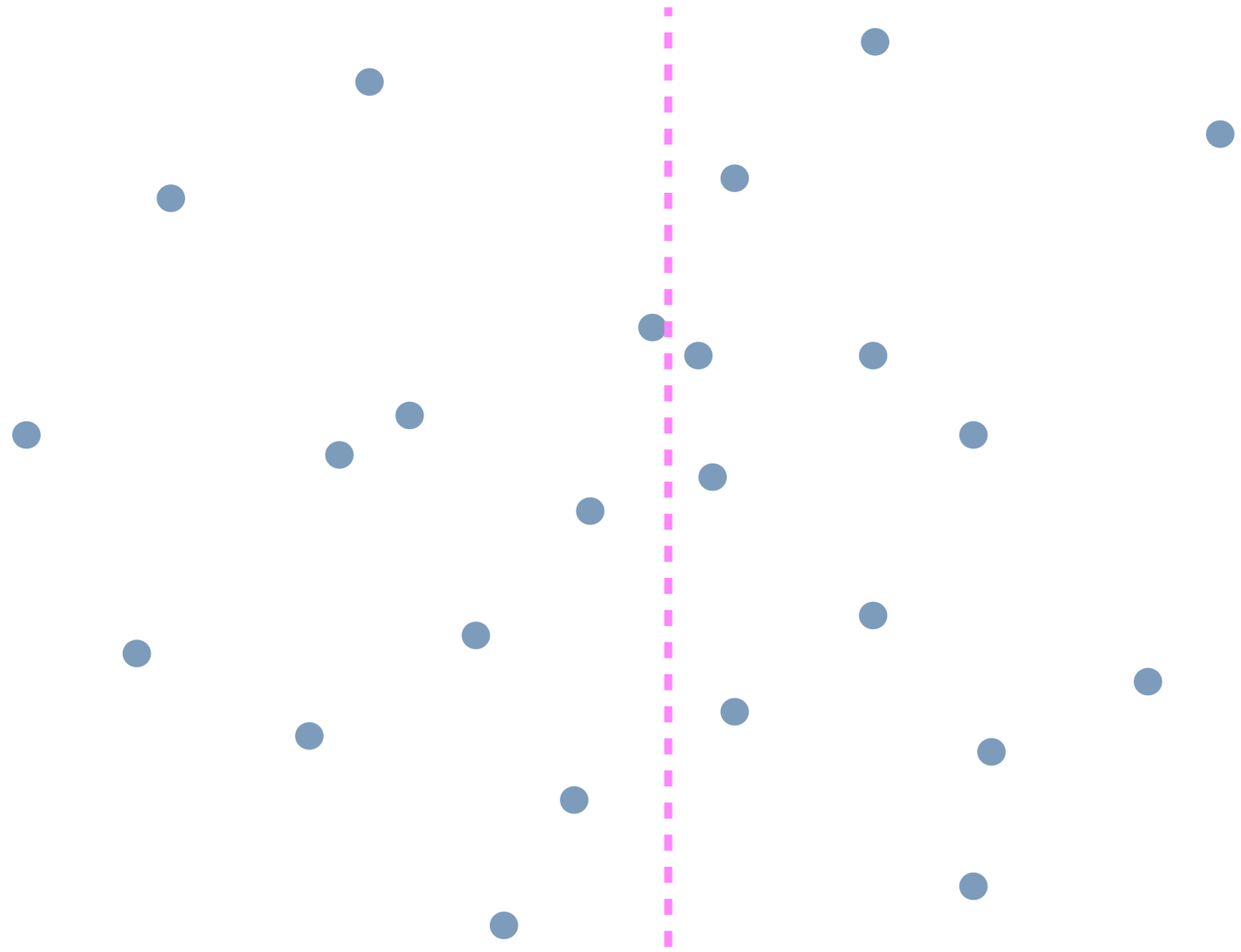
- *if merge phase is **O(n)**: T(n) = 2T(n/2) + **O(n)** => O(n lg n)*

CH via divide-and-conquer



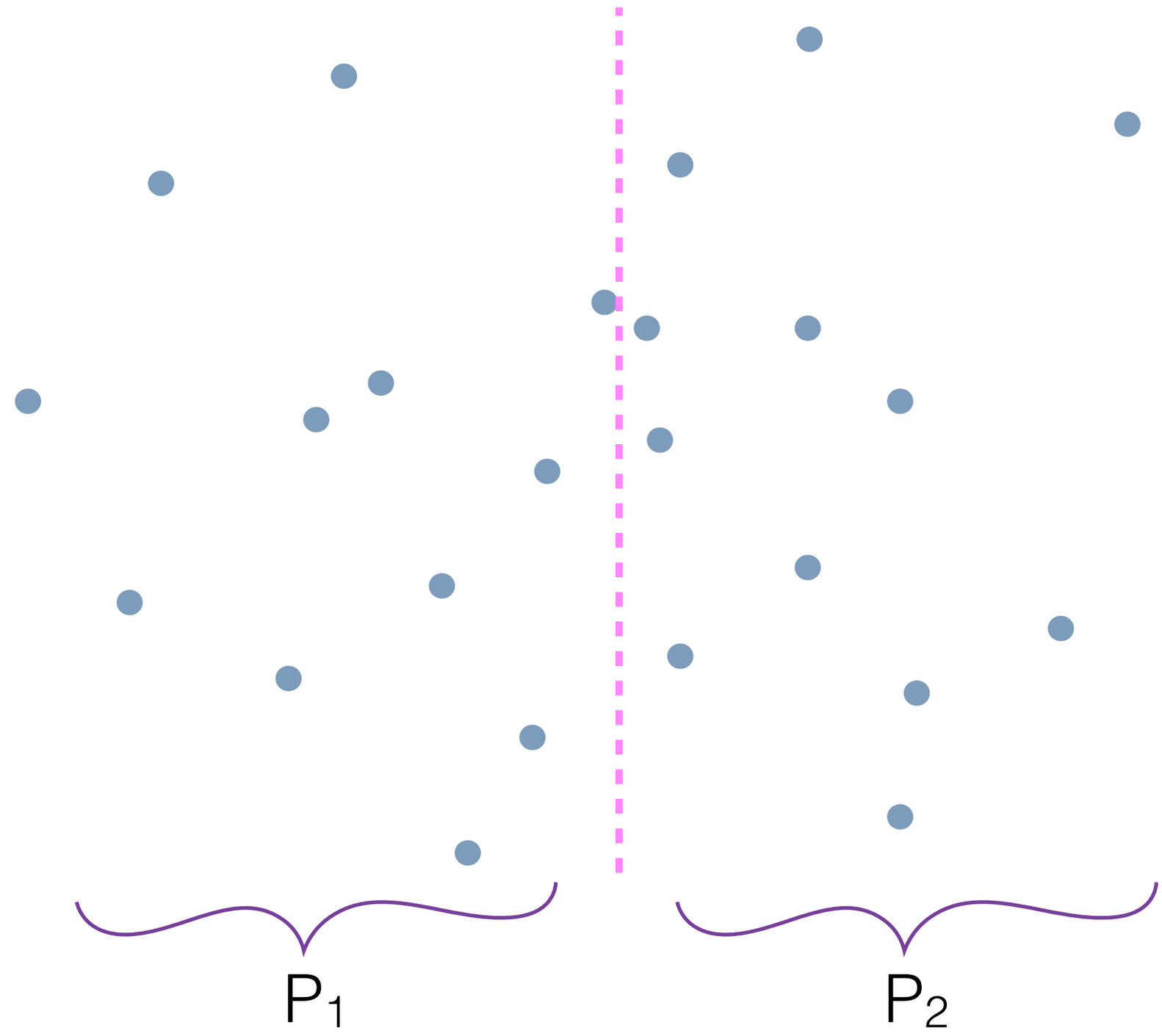
CH via divide-and-conquer

- find vertical line that splits P in half



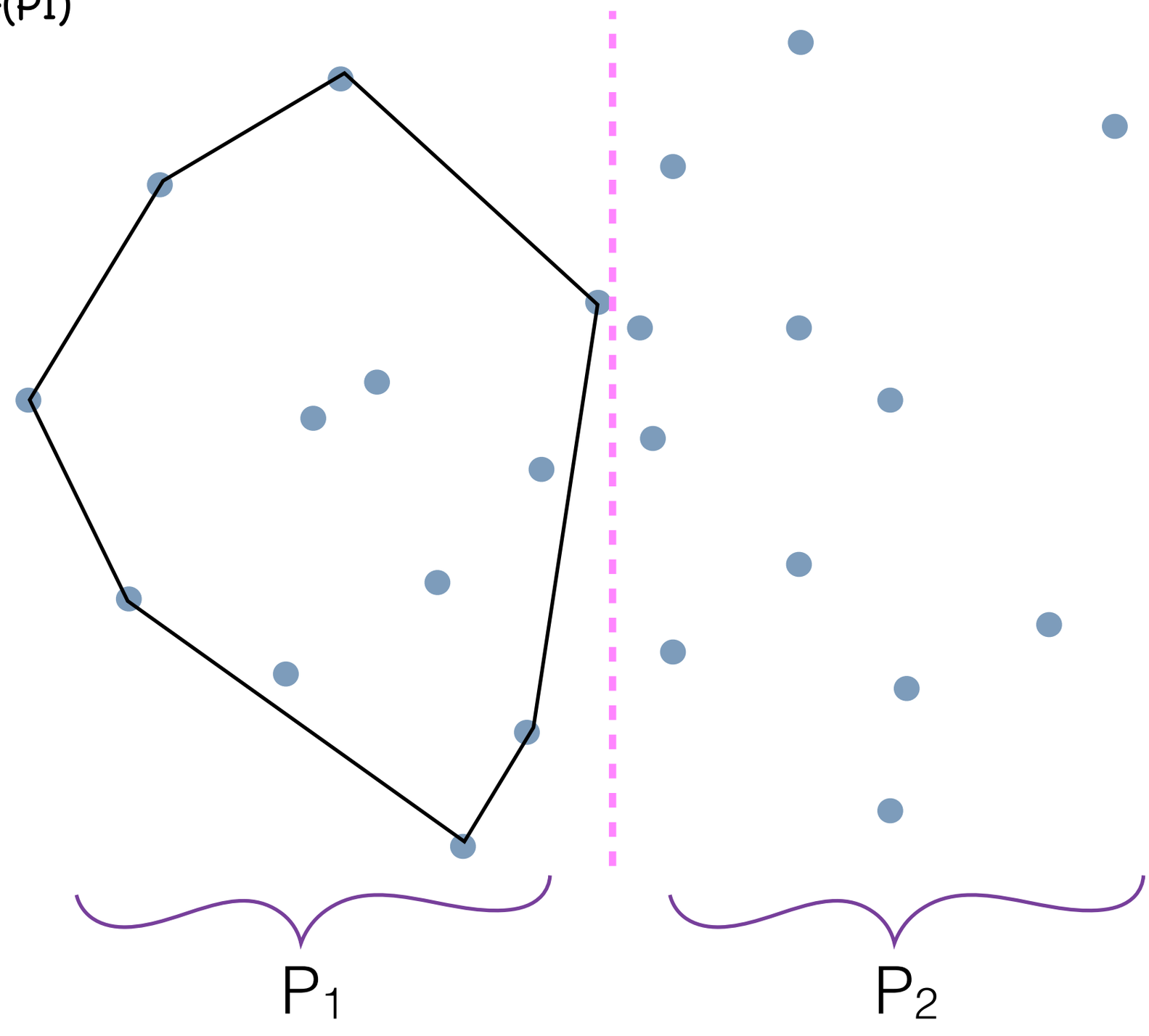
CH via divide-and-conquer

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line



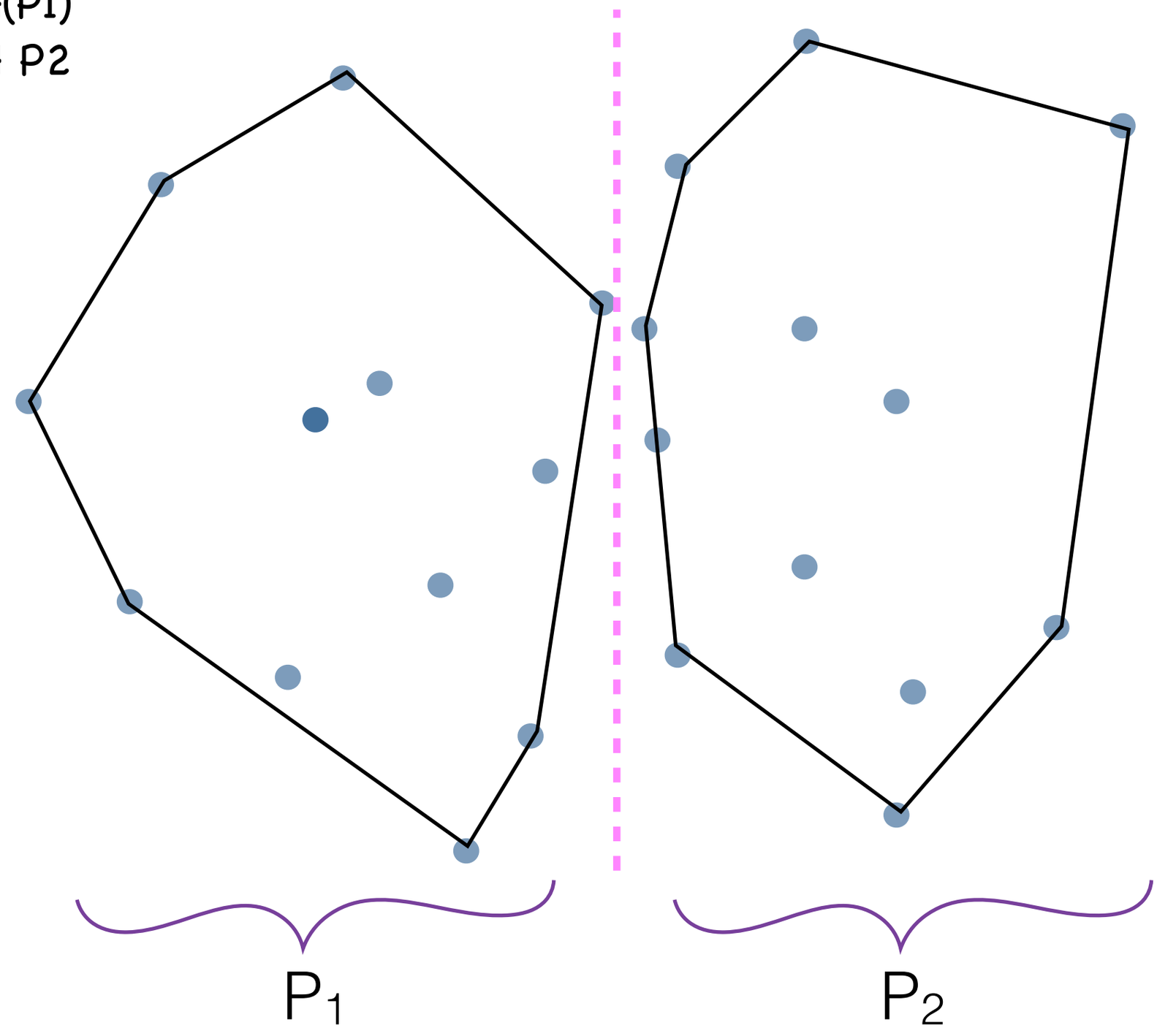
CH via divide-and-conquer

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find CH(P1)



CH via divide-and-conquer

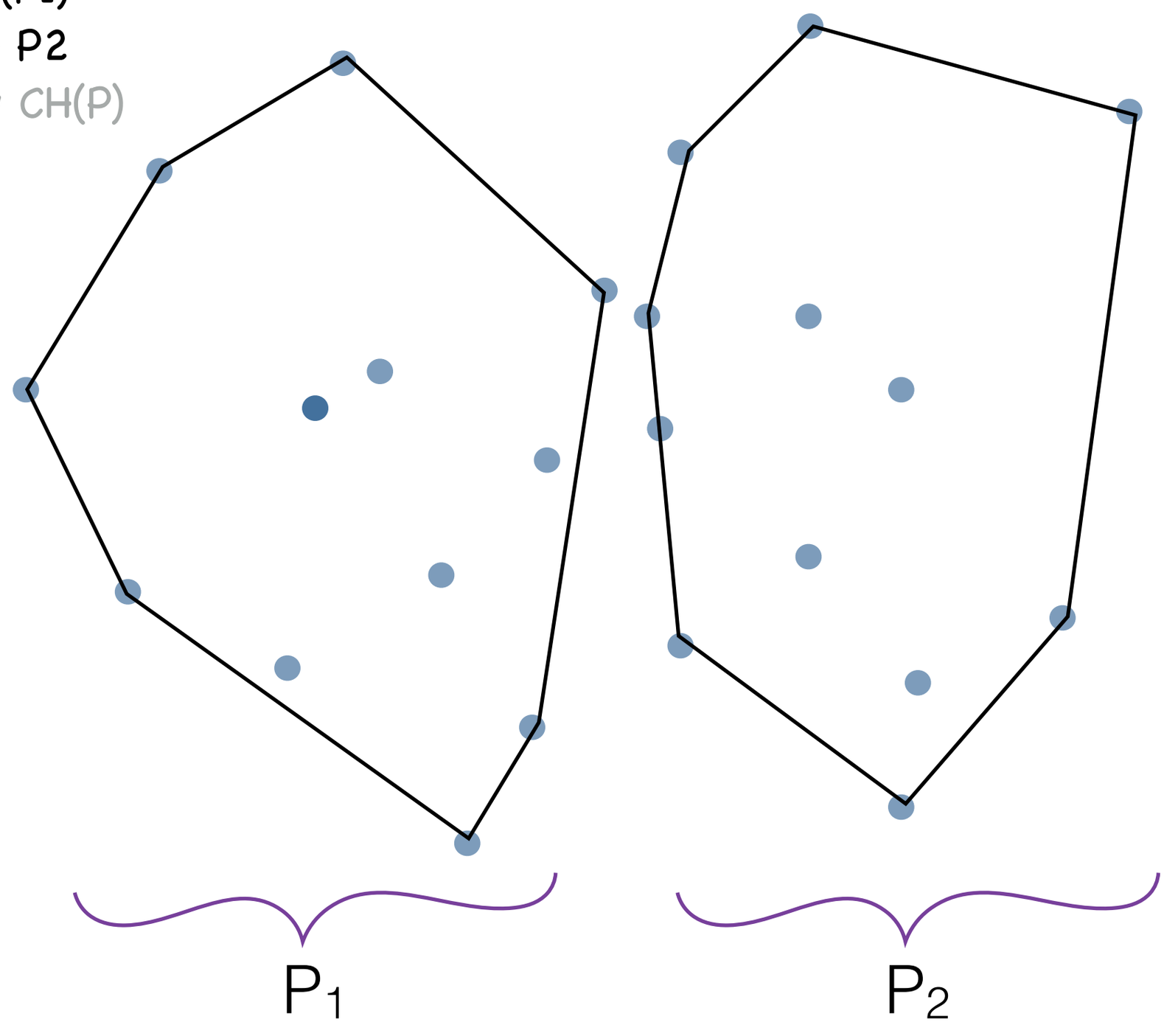
- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find CH(P1)
- recursively find CH P2



CH via divide-and-conquer

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find CH(P1)
- recursively find CH P2

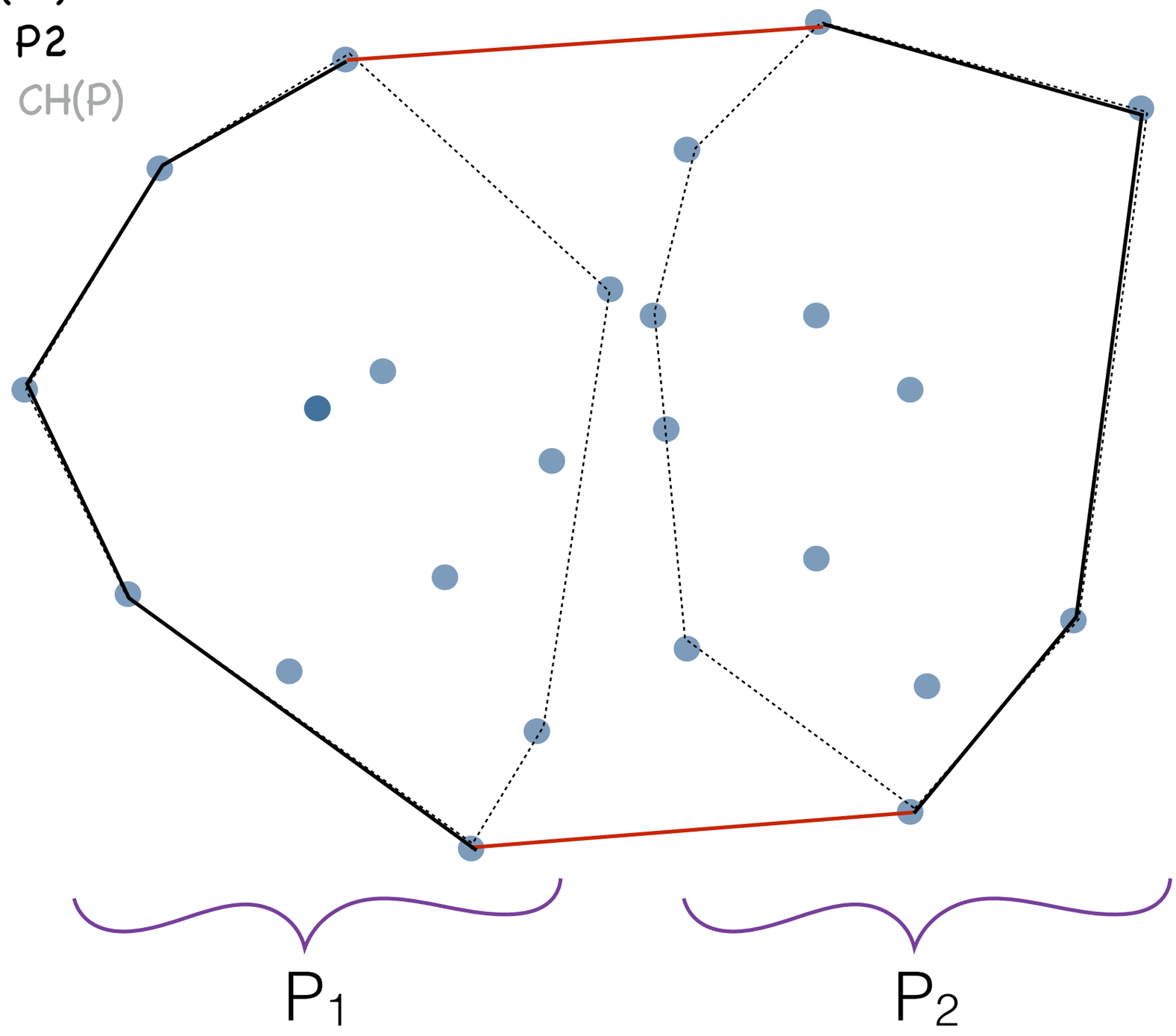
  //now get somehow CH(P)



CH via divide-and-conquer

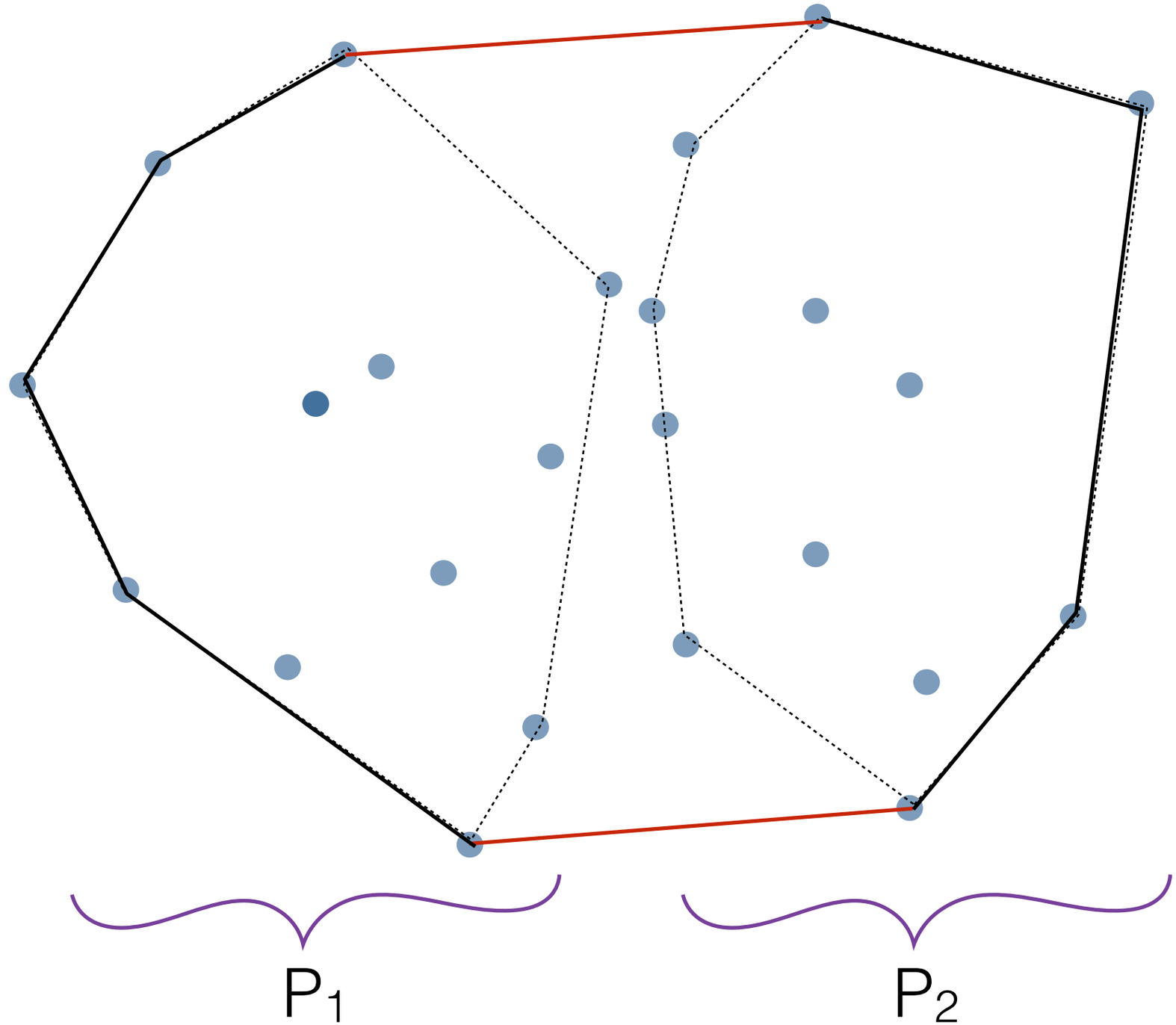
- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find CH(P1)
- recursively find CH P2

  //now get somehow CH(P)



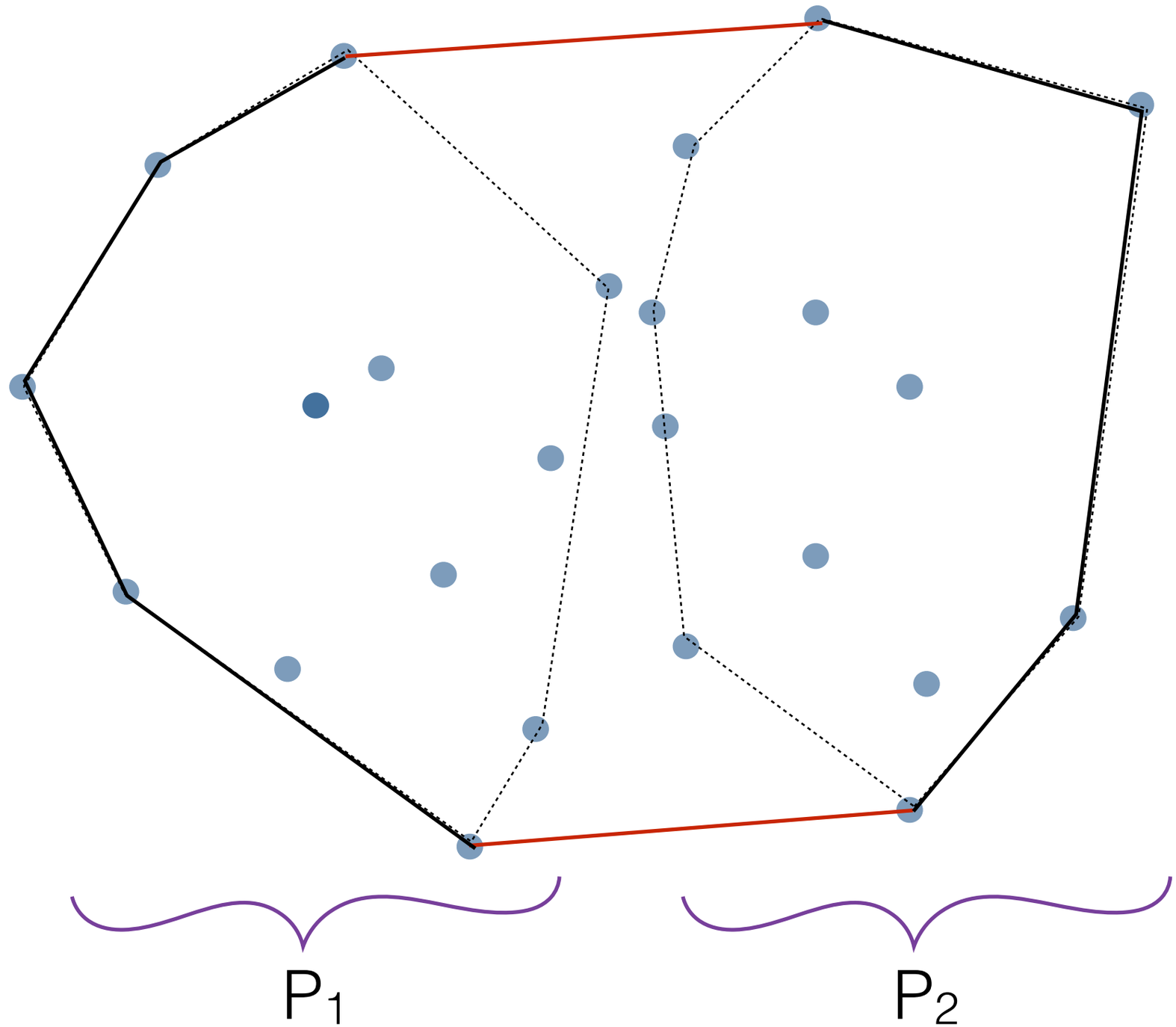
Merging two hulls..in linear time

- Need to find the two tangents



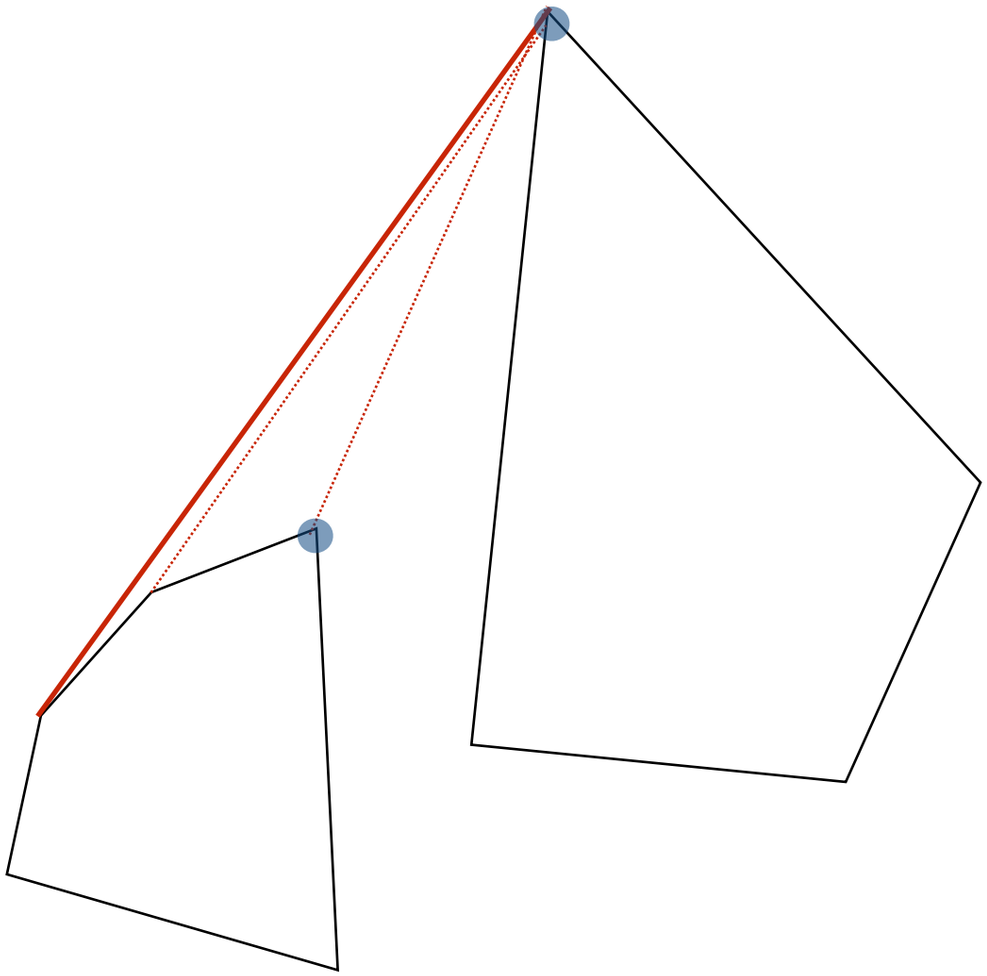
Merging two hulls..in linear time

- Is the upper tangent always between top-most points in P_1 and P_2?



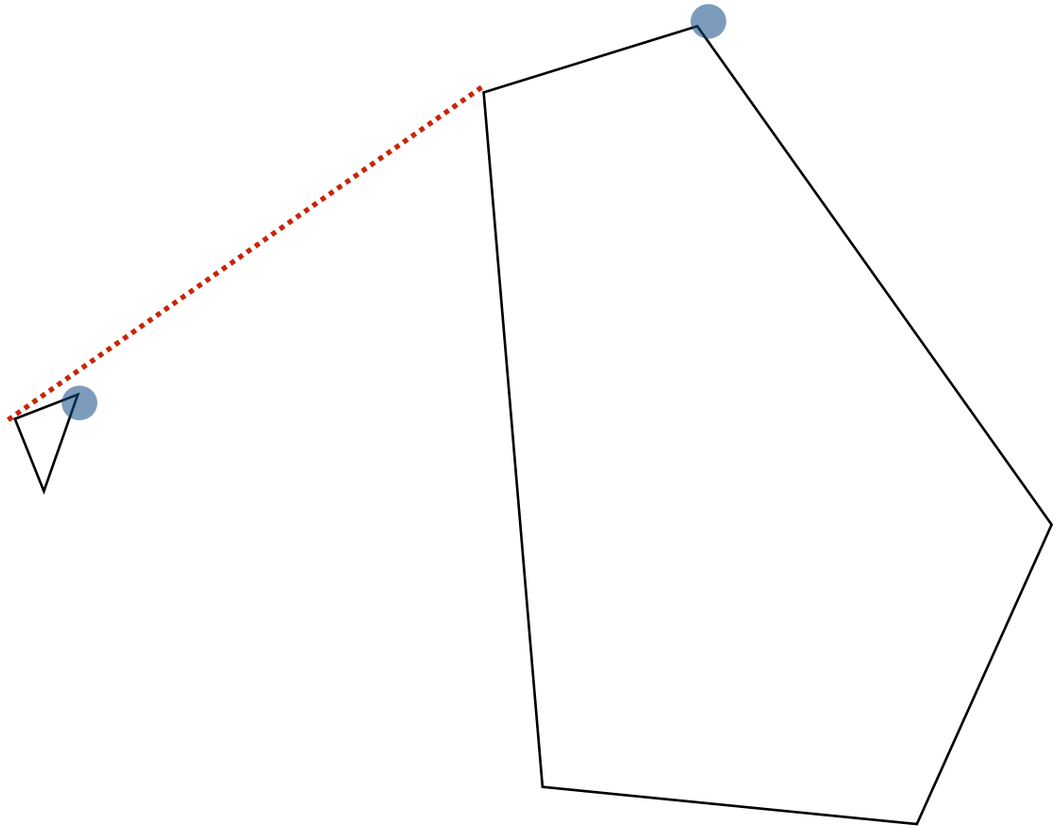
Merging two hulls..in linear time

- Is the upper tangent always between top-most points in P_1 and P_2?
  - Not necessarily…



Merging two hulls..in linear time

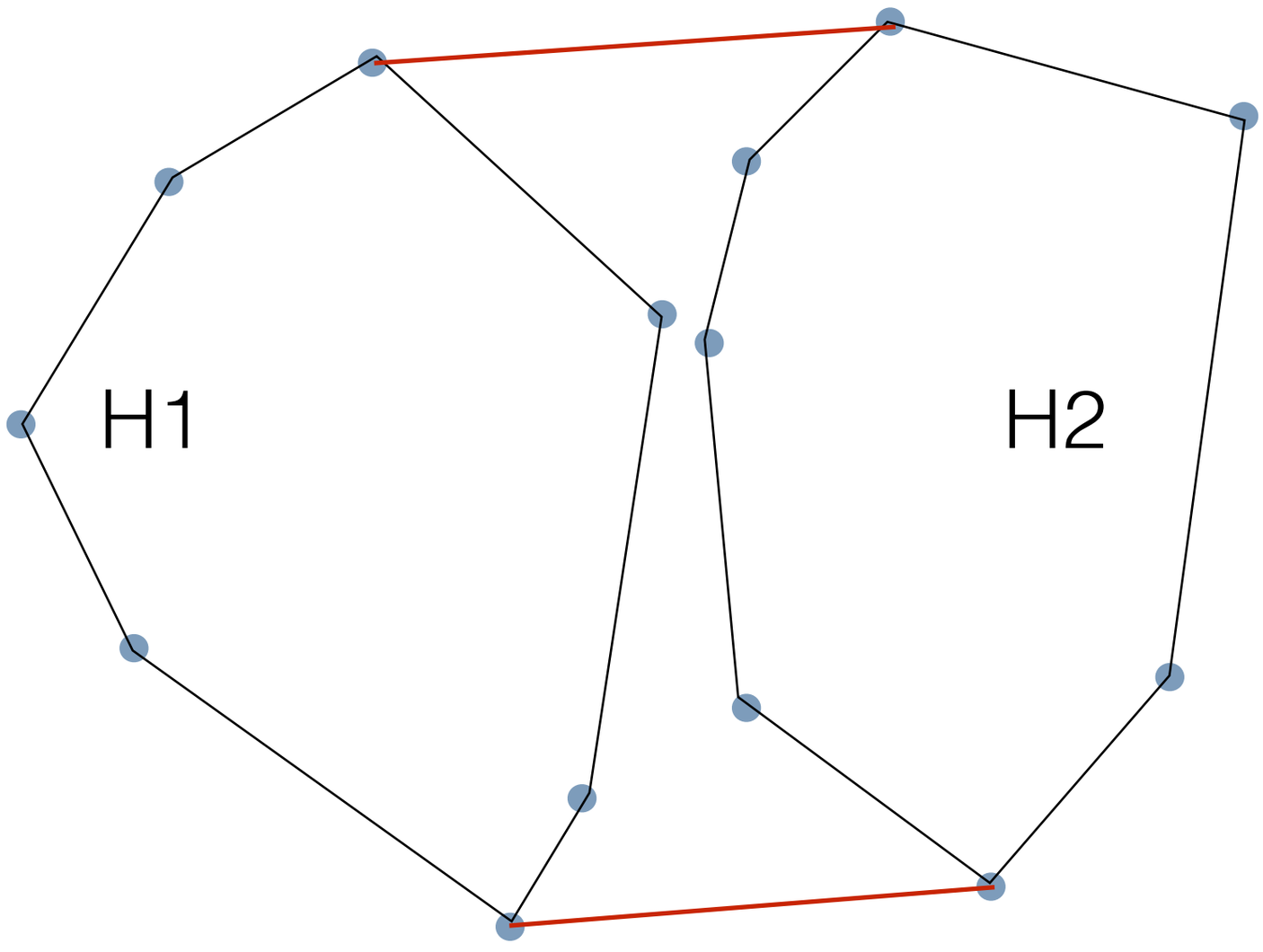
- Note: the top-most point is on the CH, but not necessarily on the upper tangent



Merging two hulls..in linear time

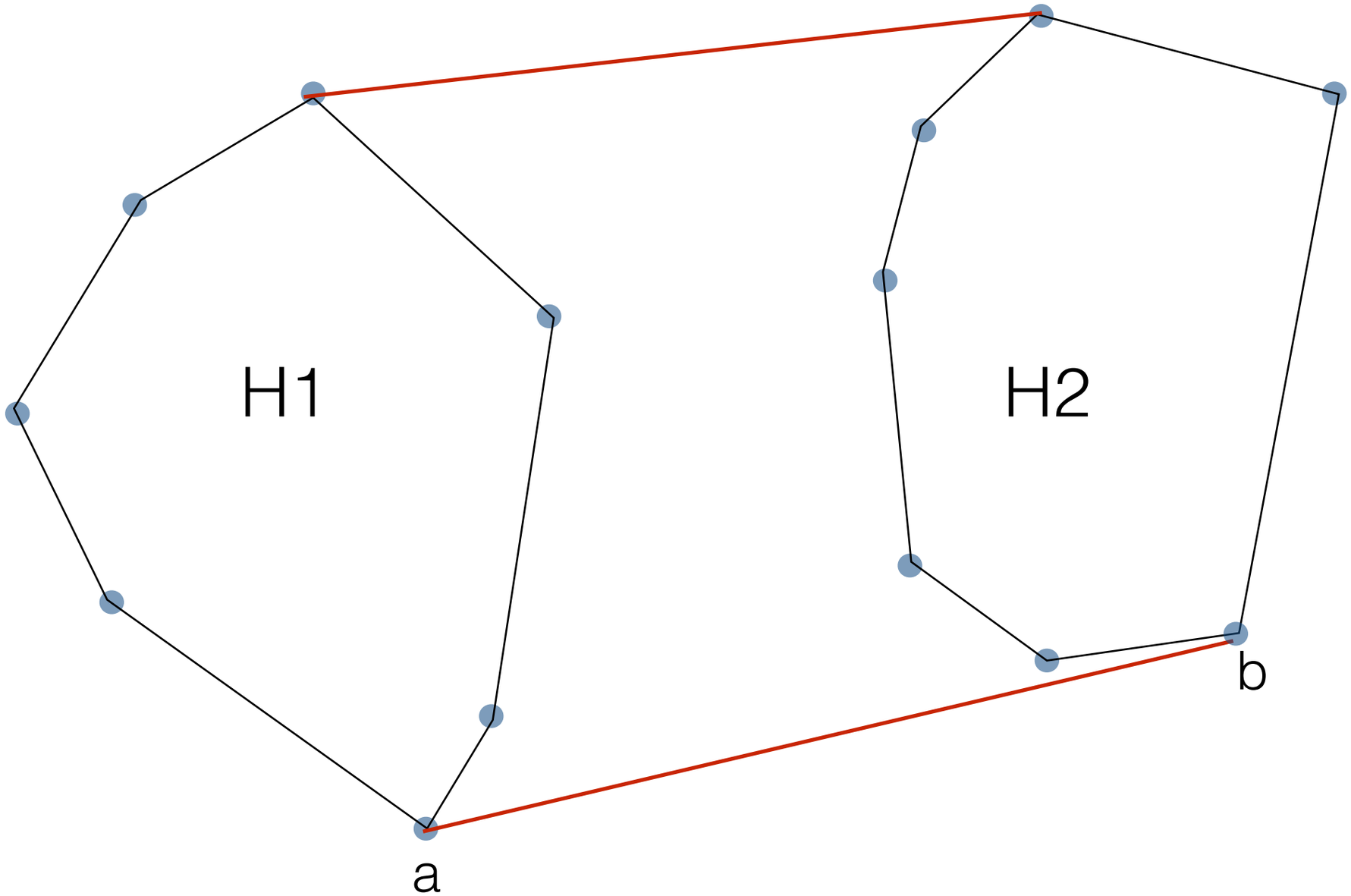
- Naive algorithm: try all segments (a,b) with a in H_1 and b in H_2

=> $O(n^2)$ merge, $O(n^2 \lg n)$ CH algorithm



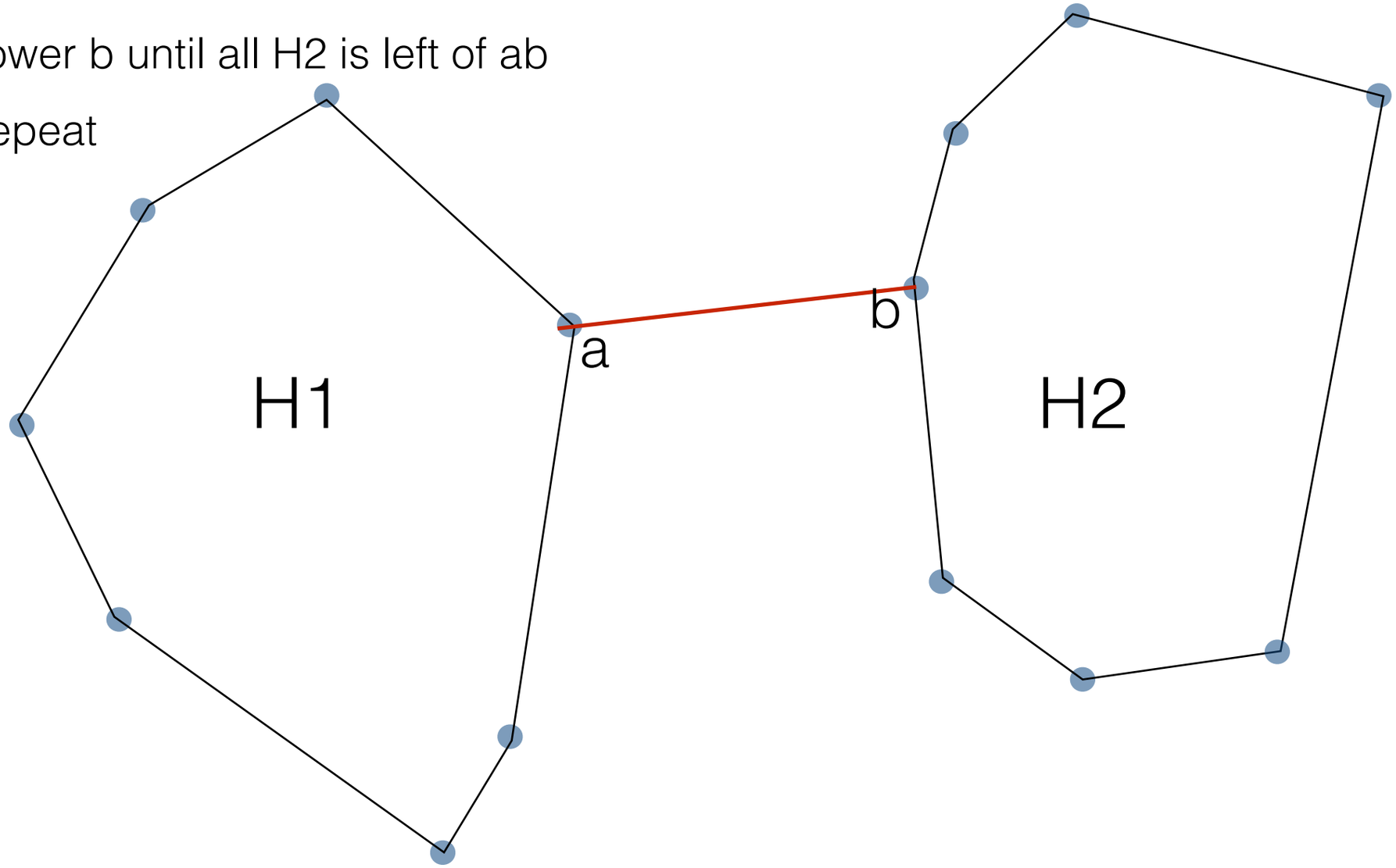
Merging two hulls..in linear time

- Claim: all points in H1 and H2 are to the left of ab



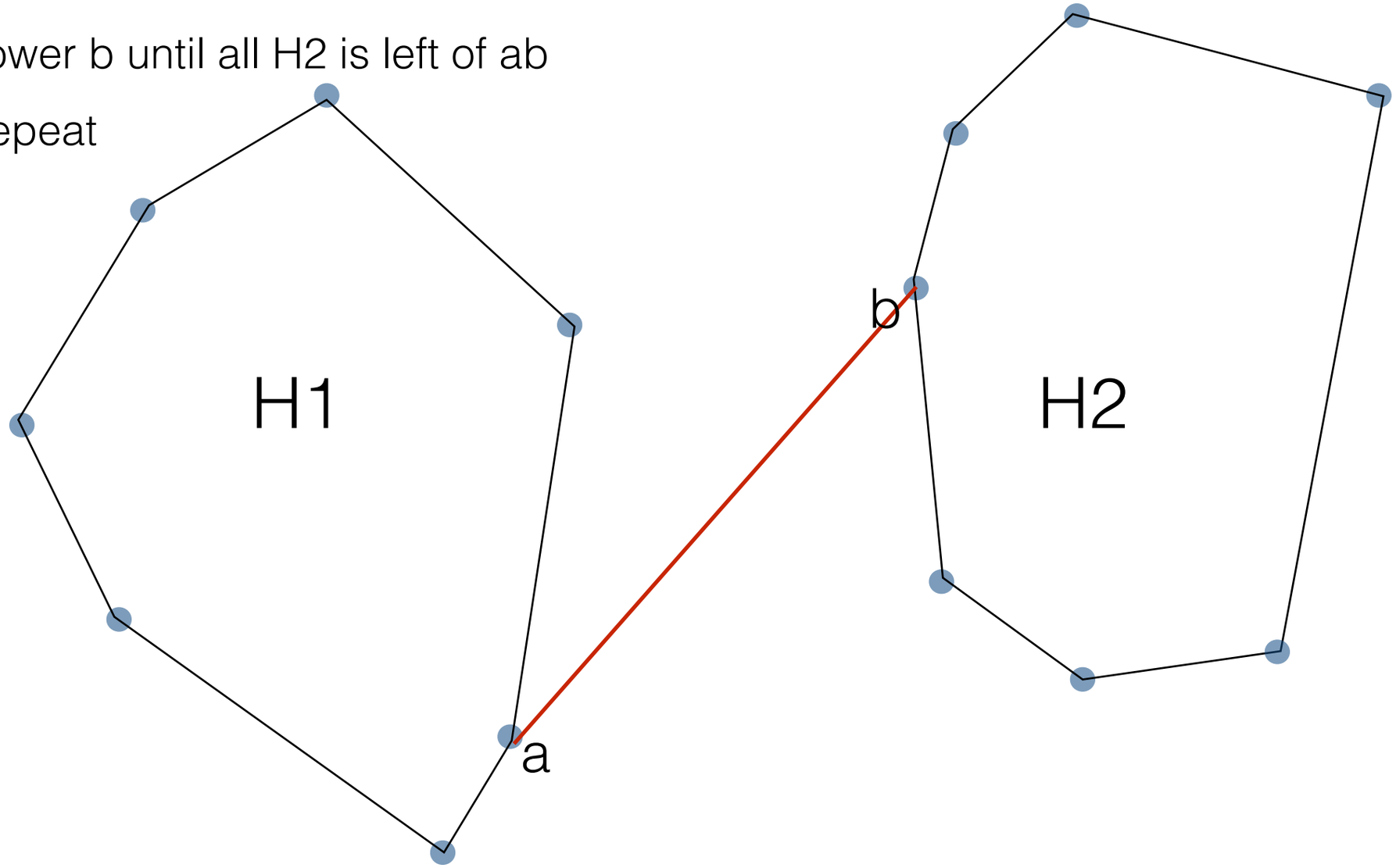
Merging two hulls..in linear time

- Idea:
  - start with a = rightmost point in H1, b = leftmost point in H2
  - lower a until all H1 is left of ab
  - lower b until all H2 is left of ab
  - repeat



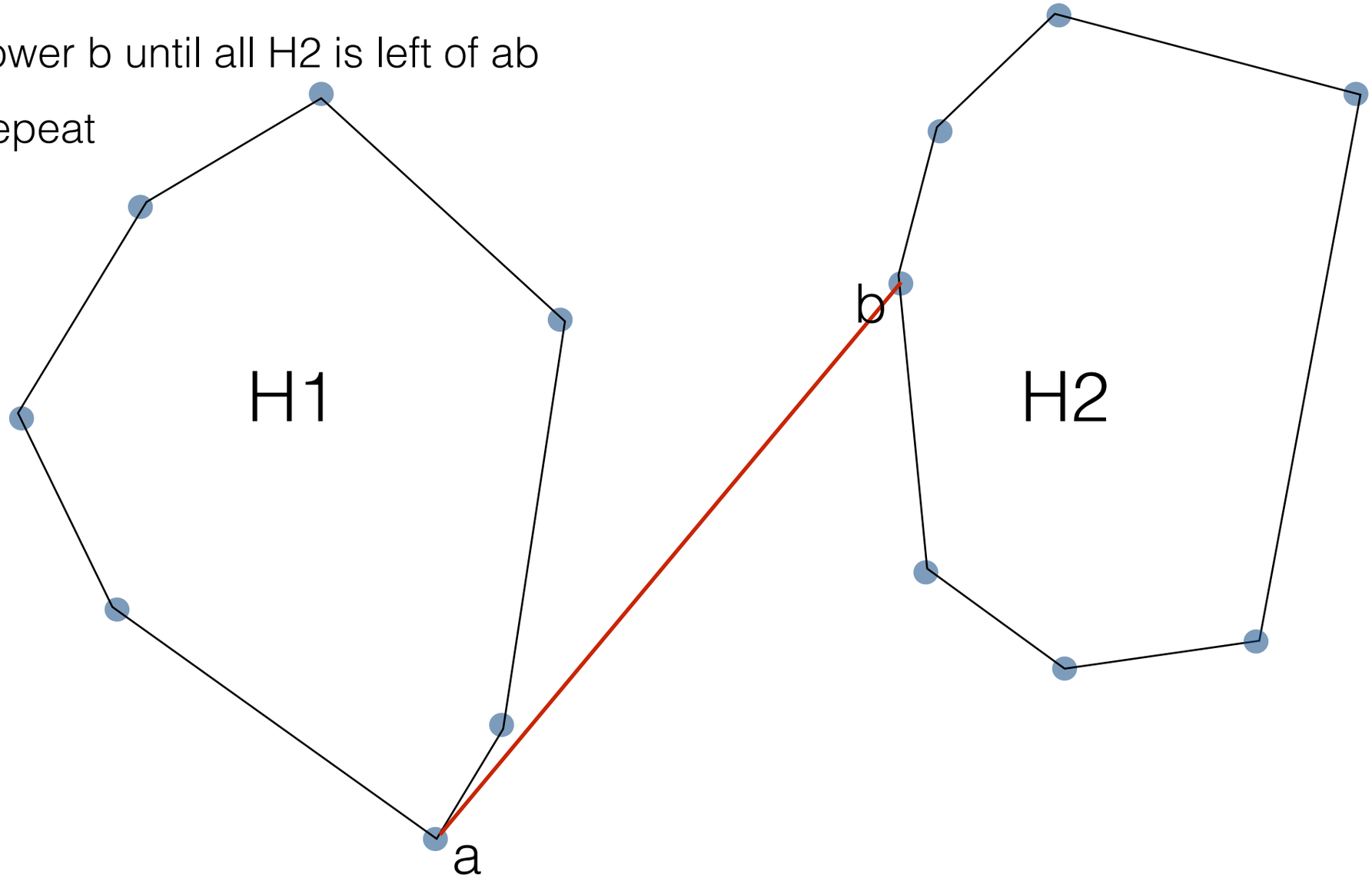
Merging two hulls..in linear time

- Idea:
  - start with a = rightmost point in H1, b = leftmost point in H2
  - lower a until all H1 is left of ab
  - lower b until all H2 is left of ab
  - repeat



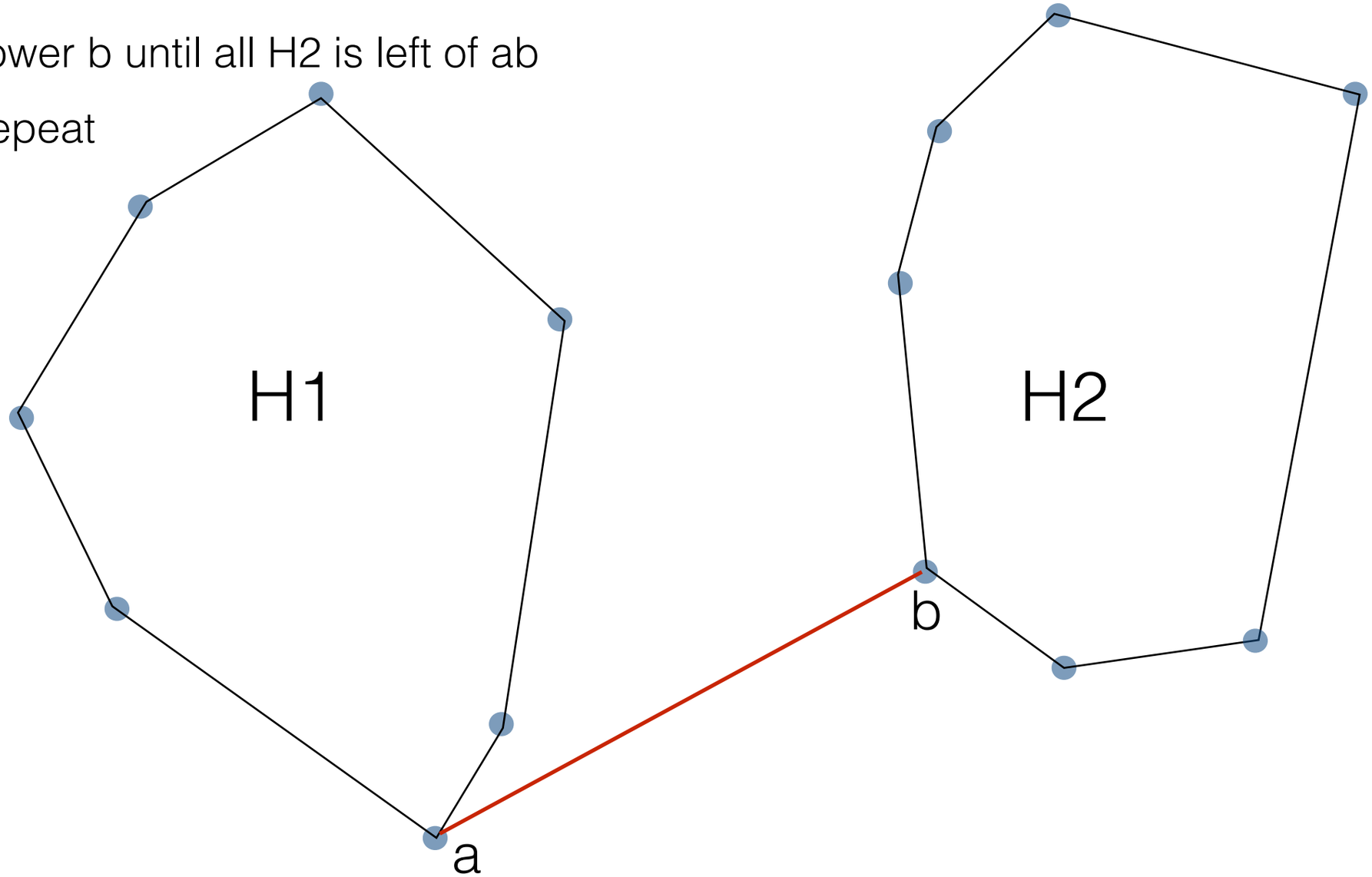
Merging two hulls..in linear time

- Idea:
  - start with a = rightmost point in H1, b = leftmost point in H2
  - lower a until all H1 is left of ab
  - lower b until all H2 is left of ab
  - repeat



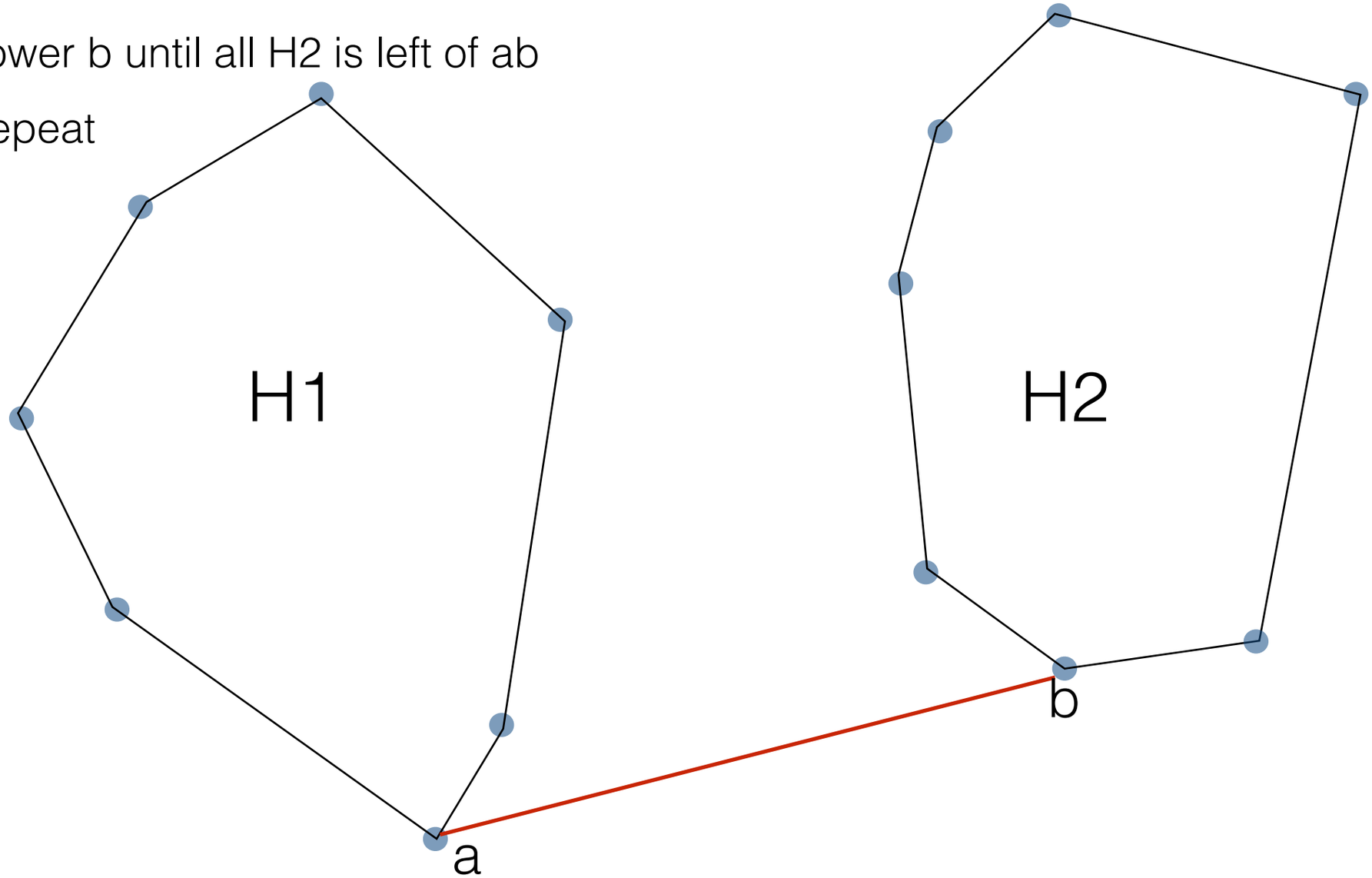
Merging two hulls..in linear time

- Idea:
  - start with a = rightmost point in H1, b = leftmost point in H2
  - lower a until all H1 is left of ab
  - lower b until all H2 is left of ab
  - repeat



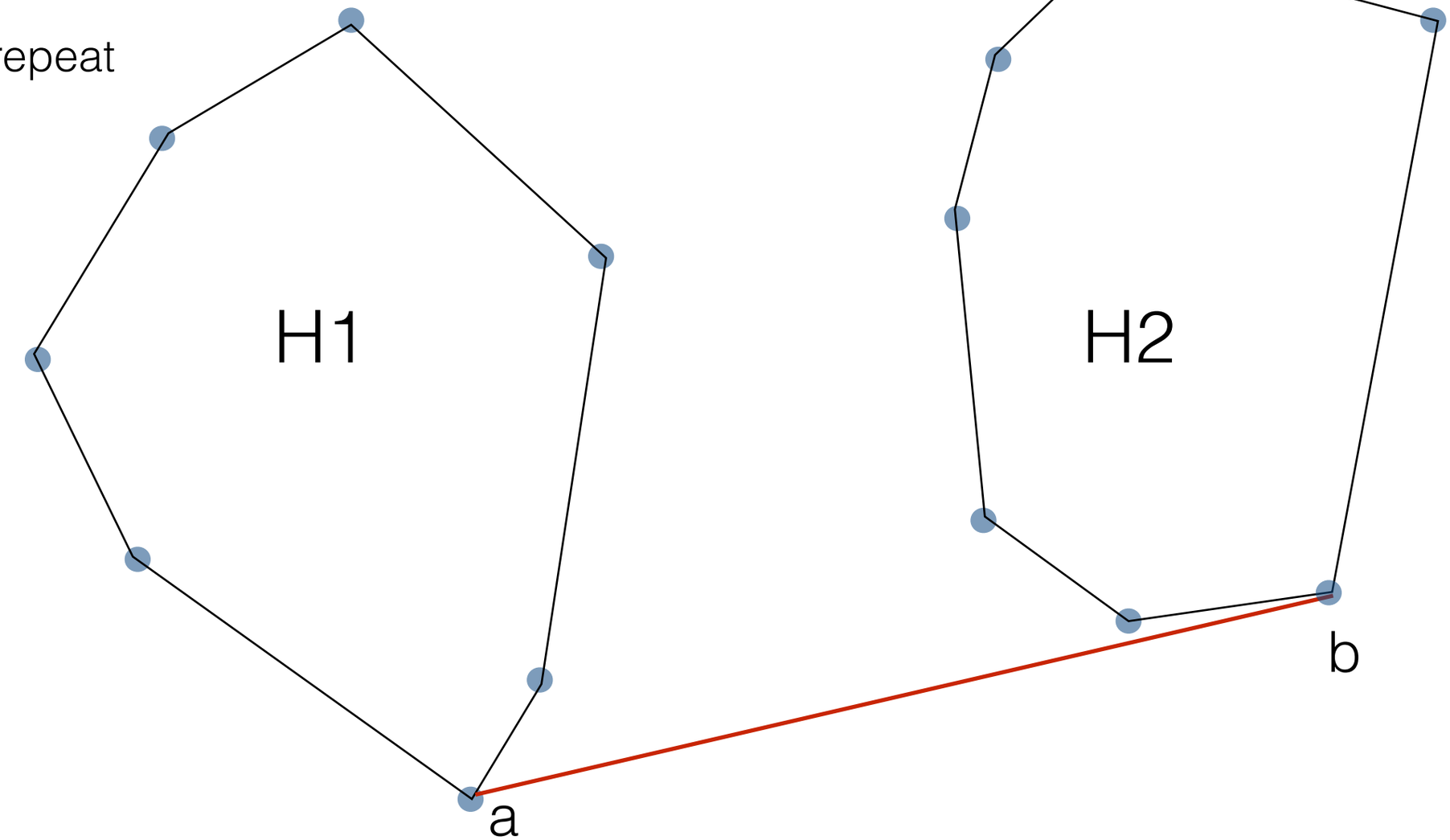
Merging two hulls..in linear time

- Idea:
  - start with a = rightmost point in H1, b = leftmost point in H2
  - lower a until all H1 is left of ab
  - lower b until all H2 is left of ab
  - repeat



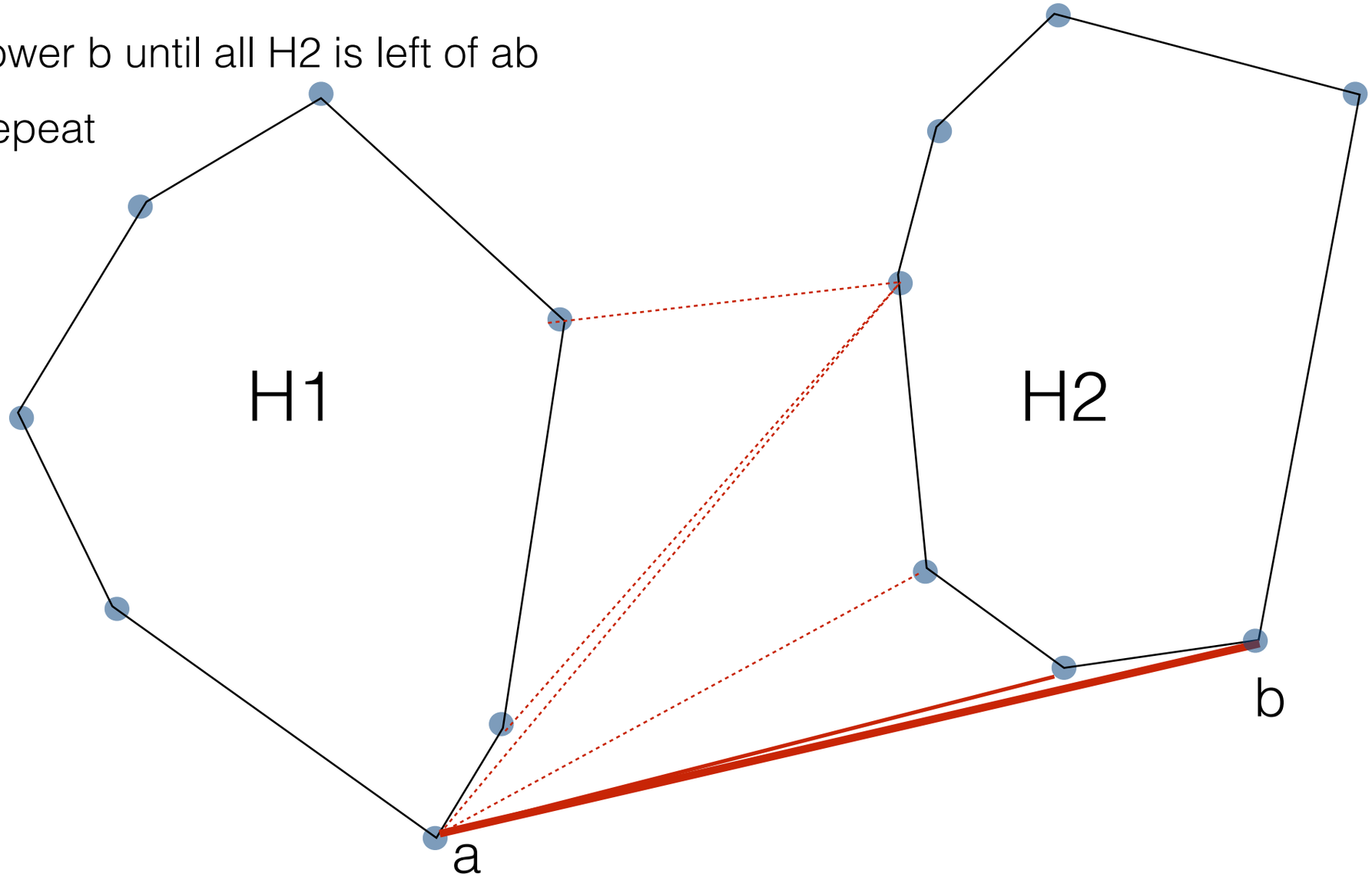
Merging two hulls..in linear time

- Idea:
  - start with a = rightmost point in H1, b = leftmost point in H2
  - lower a until all H1 is left of ab
  - lower b until all H2 is left of ab
  - repeat



Merging two hulls..in linear time

- Idea:
  - start with a = rightmost point in H1, b = leftmost point in H2
  - lower a until all H1 is left of ab
  - lower b until all H2 is left of ab
  - repeat



- Not completely trivial to show that the algorithm terminates; $O(n_1) + O(n_2)$

Convex hull: summary

Naive	$O(n^3)$	
Gift wrapping	O(nh)	1970
Graham scan	O(n lg n)	1972
Quickhull	$O(n^2)$	1977
Divide-and-conquer	O(n lg n)	1977
Incremental	O(n lg n)	1984

Can we do better?

Convex hull lower bound

- For a given CH algorithm, its worst case running time is the largest running time on an input of size n

$$\max_{P,\,|P|=n} \{T(n) \mid T(n) \text{ is the running time on input } P\}$$

- CH lower bound
  - What is the best possible worst-case running time?;
  - What is the worst-case running time of the best possible CH algorithm?
  - Consider all possible CH algorithms, and take the overall smallest worst-case running time

$$\min_{A} \max_{P,\,|P|=n} \{T(n) \mid T(n) \text{ is the running time of algorithm } A \text{ on input } P\}$$

Sorting lower bound

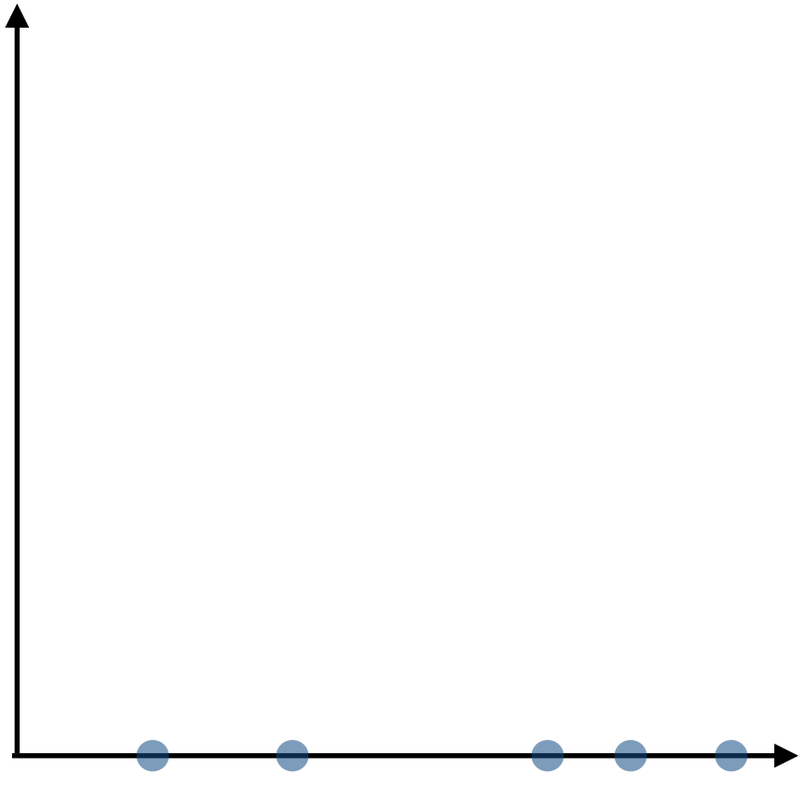
Theorem:

- Any sorting algorithm that uses only comparisons must take at least Omega(n lg n) in the worst case.

- We'll prove an Omega(n lg n) lower bound for CH by showing that sorting reduces to computing the CH
  - i.e. sorting can be solved by an instance of convex hull

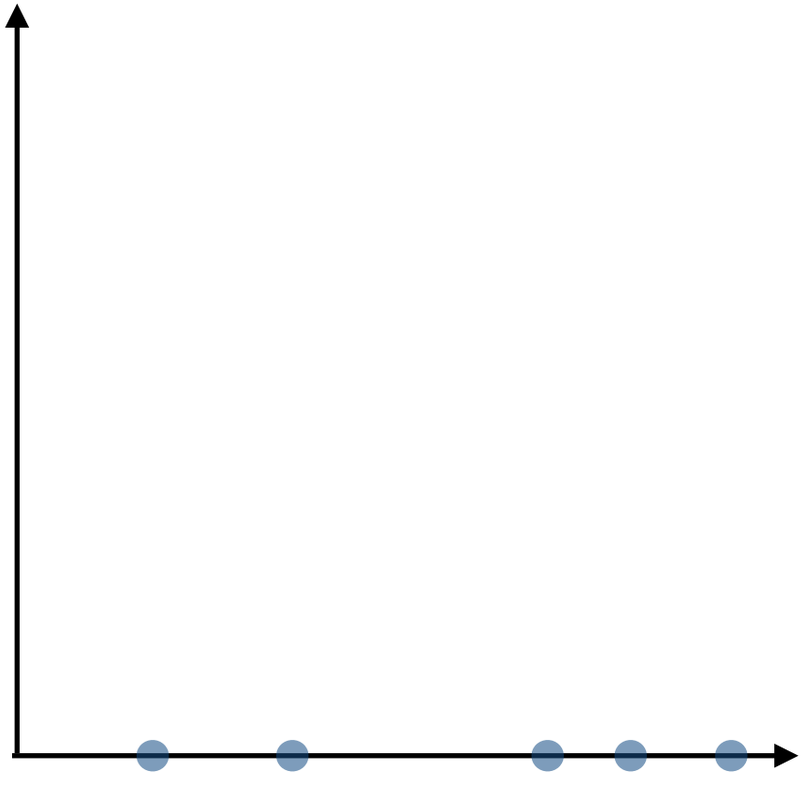
Convex Hull lower bound

- Assume we are given a set of of numbers x_1, x_2, …x_n to sort.



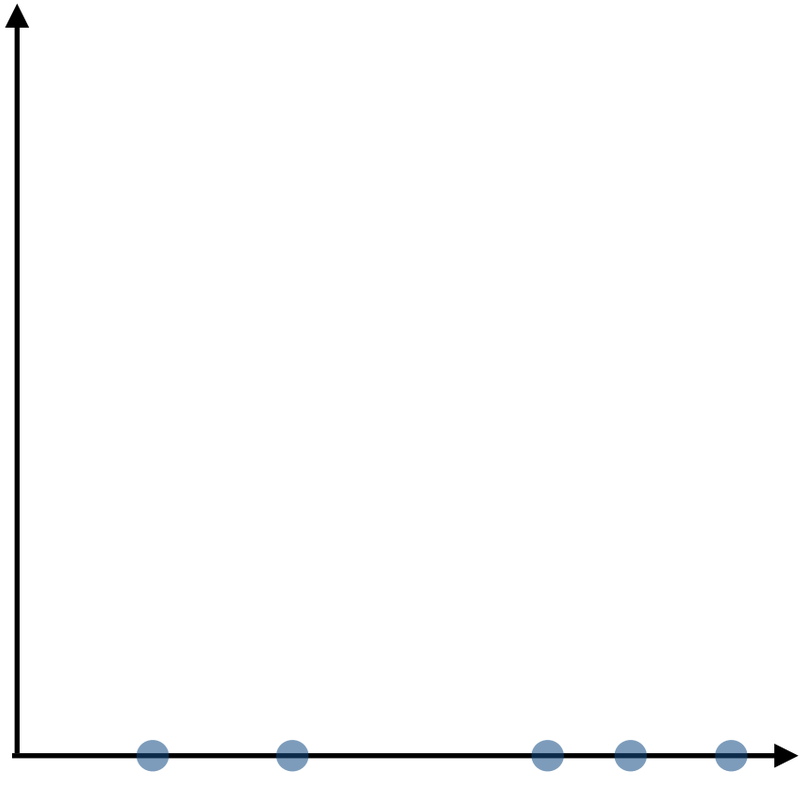
Convex Hull lower bound

- Suppose we have an algorithm that constructs the CH of a set of n points in worst-case T(n) time.



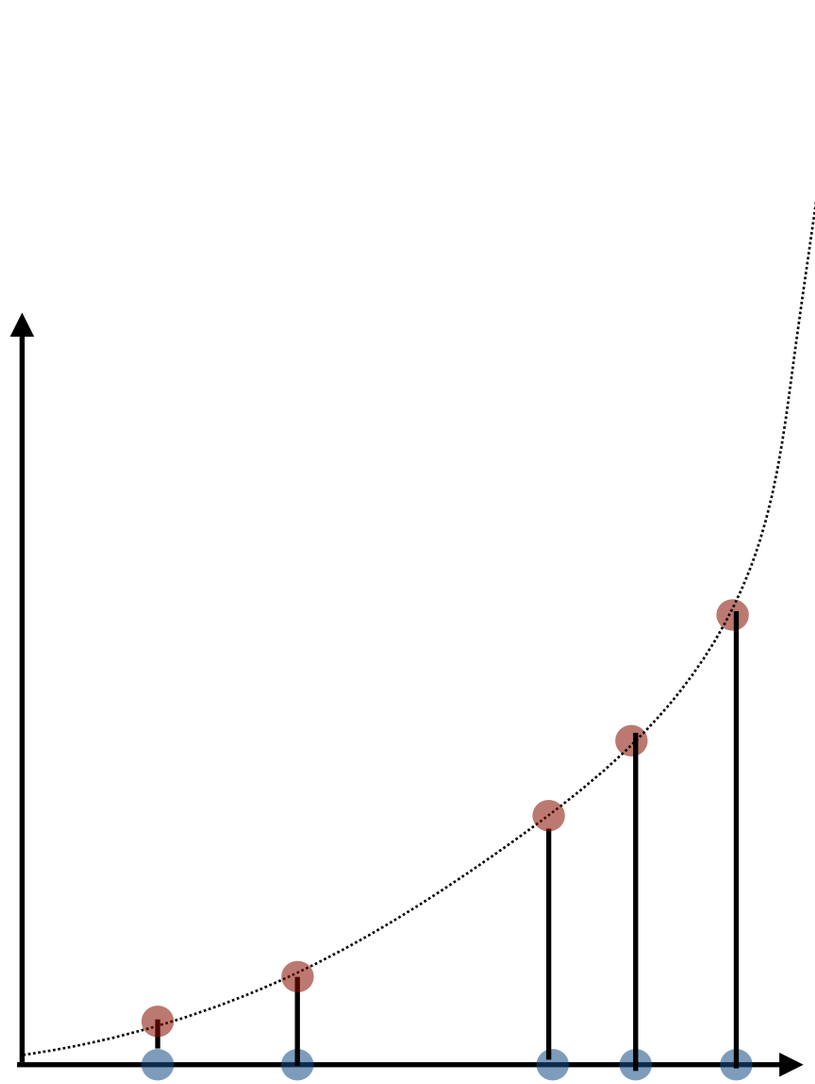
Convex Hull lower bound

- Our goal is to argue that there exists some instance of a convex hull problem that sorts our numbers.



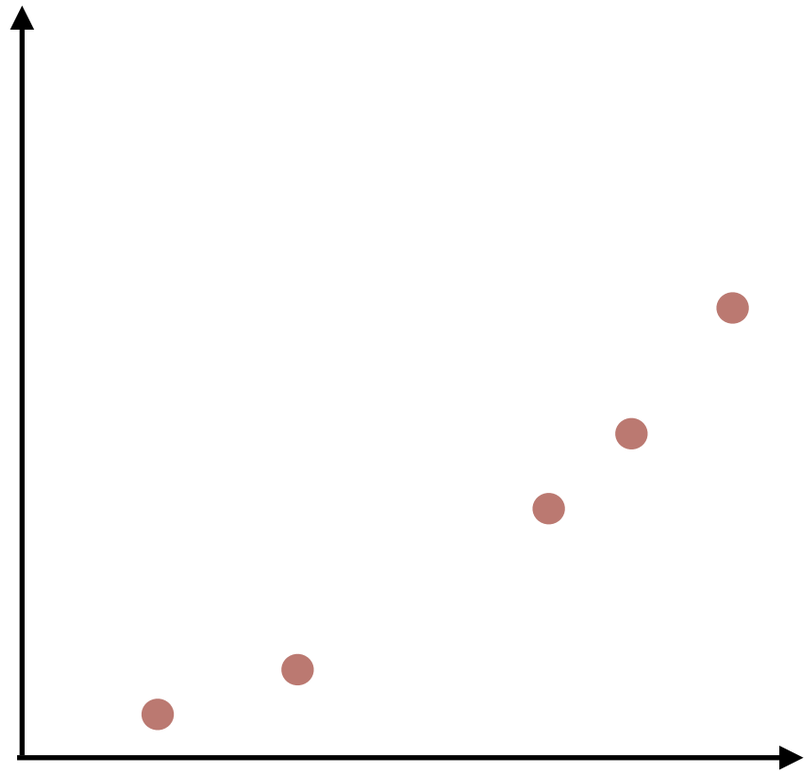
Convex Hull lower bound

- Form a set of 2D points (x_i, x_i^2).



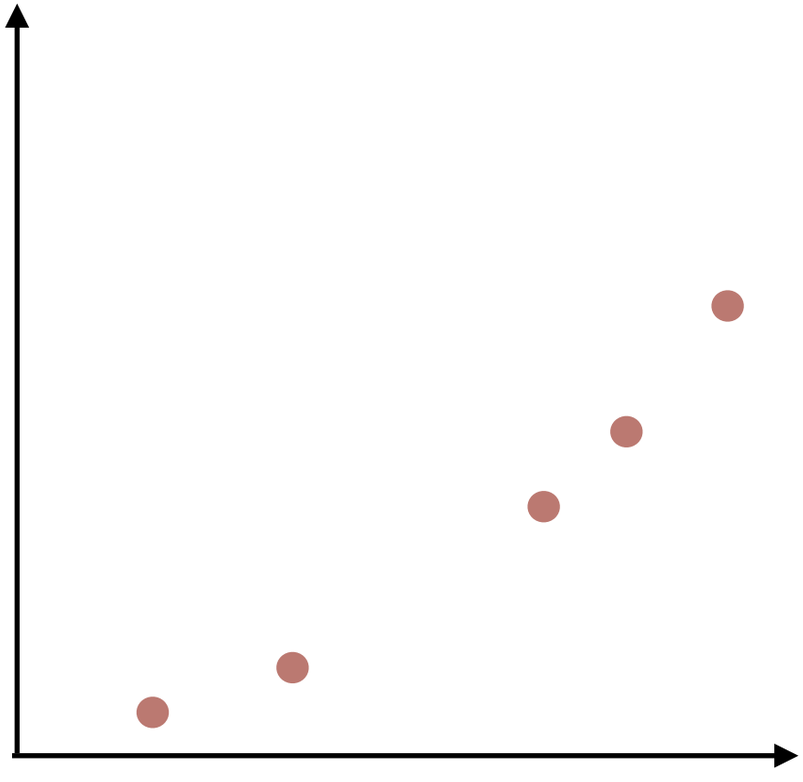
Convex Hull lower bound

- Form a set of 2D points (x_i, x_i^2).



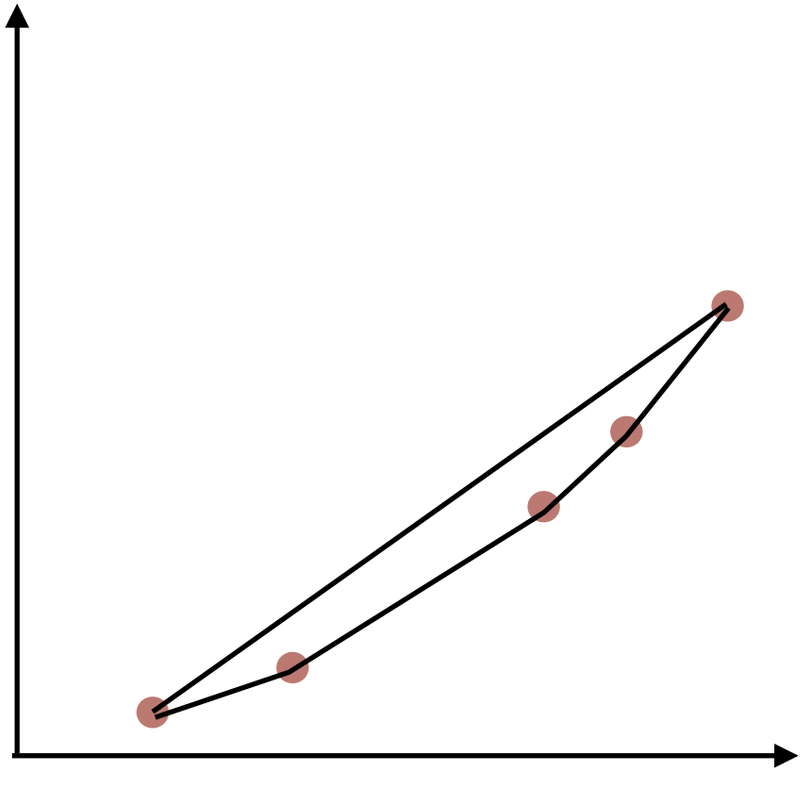
Convex Hull lower bound

- Run the CH algorithm to construct their convex hull.



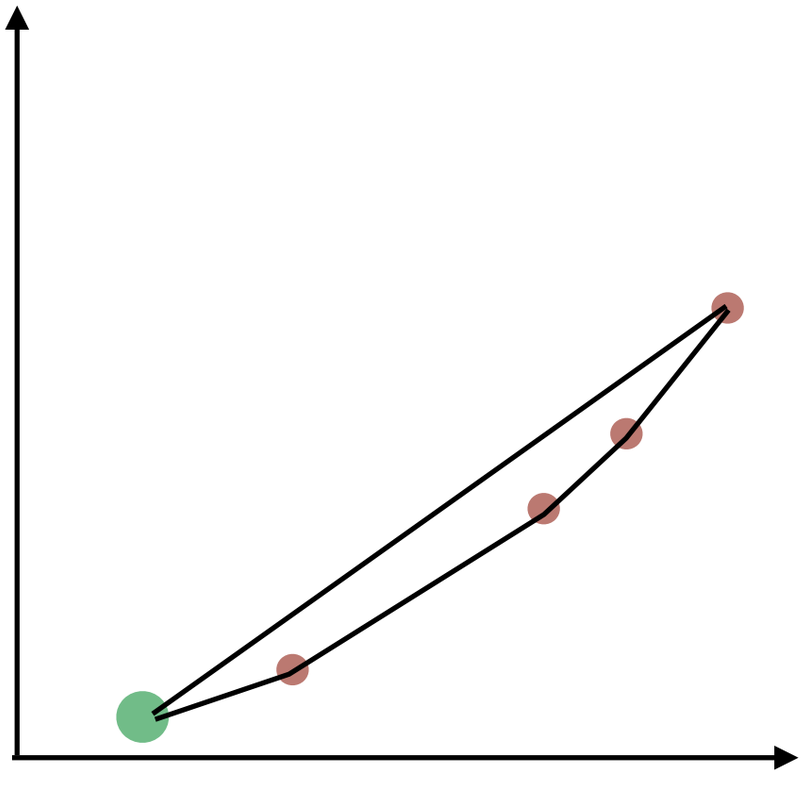
Convex Hull lower bound

- They fall on a parabola, so clearly every point is on the hull



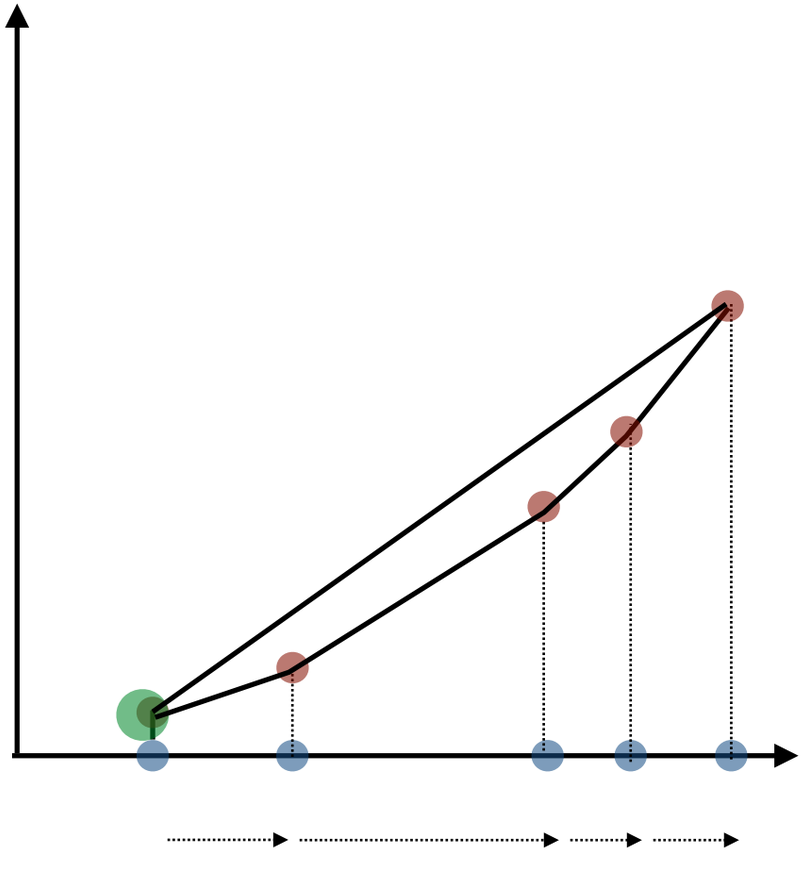
Convex Hull lower bound

- Find the lowest point on the hull, and walk from in ccw order.



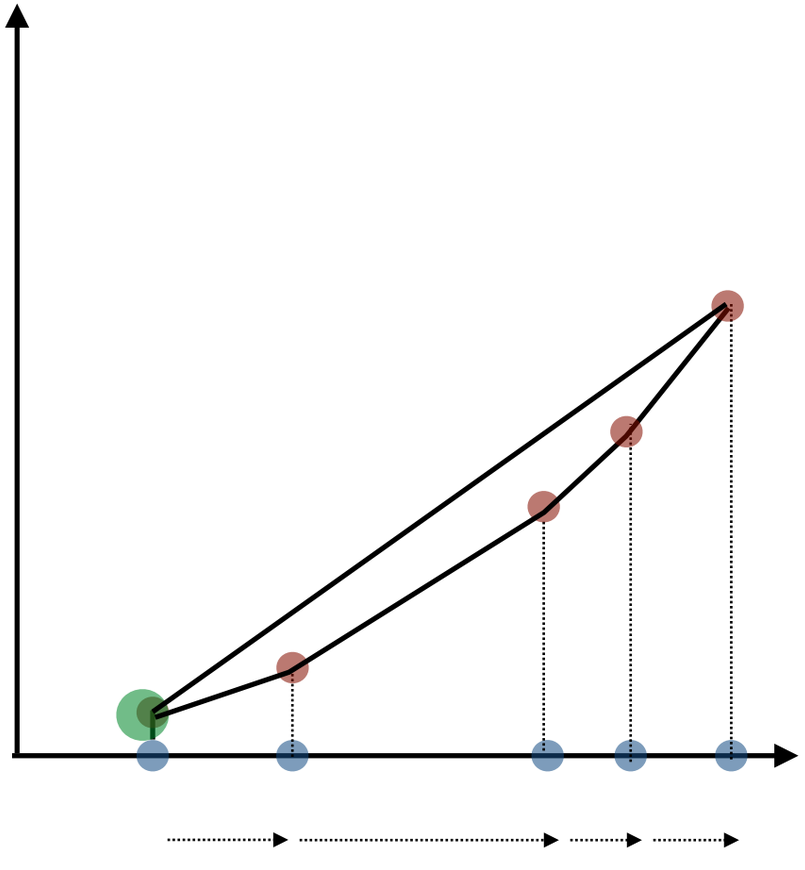
Convex Hull lower bound

- Find the lowest point on the hull, and walk from in ccw order.
- This is sorted order!



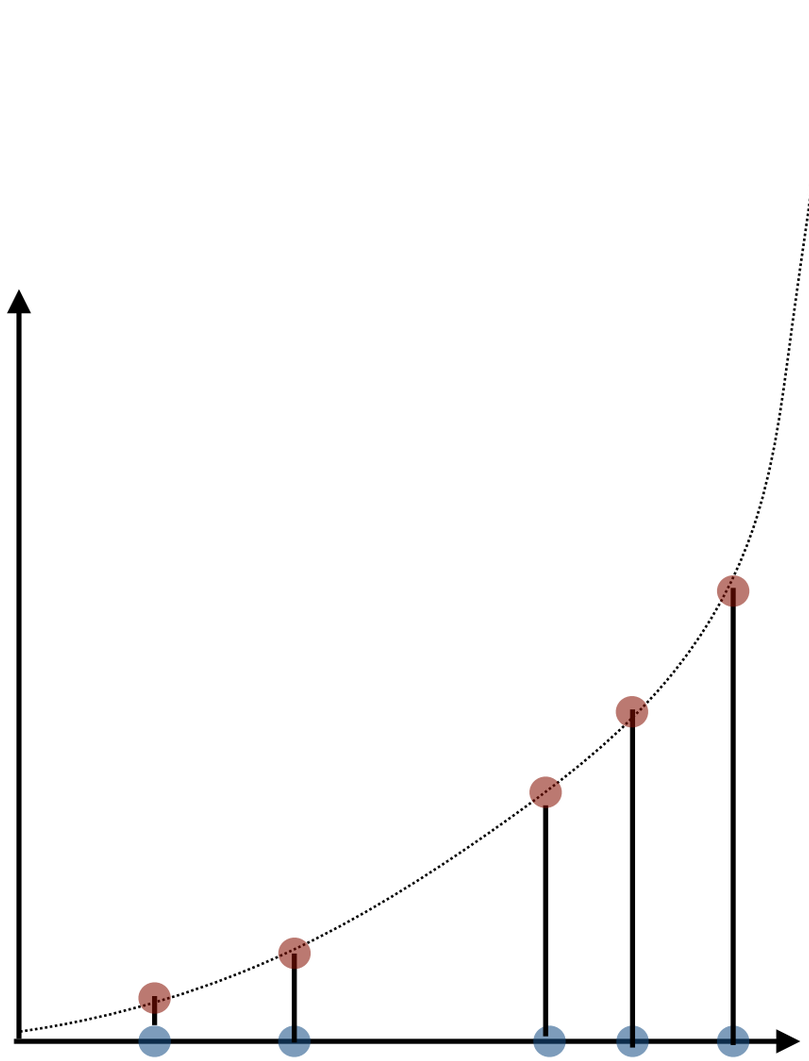
Convex Hull lower bound

- If we could find the CH faster than n lg n, then we could sort faster than n lg n! Impossible!



Convex Hull lower bound

- Assume we are given a set of of numbers x_1, x_2, …x_n to sort.
- Suppose we have an algorithm that constructs the CH of a set of n points in worst-case T(n) time.
- Our goal is to argue that there exists some instance of a convex hull problem that sorts our numbers.
- Form a set of 2D points (x_i, x_i^2).
- They fall on a parabola.
- Run the CH algorithm to construct their CH. Clearly every point is on the CH!
- Find the lowest point a on the hull. Start from a and walk along the CH in ccw order. This is sorted order. Thus we used a CH algo to sort in T(n) time plus O(n).
- If T(n) is smaller than n lg n, then we could sort faster than n lg n! Impossible!

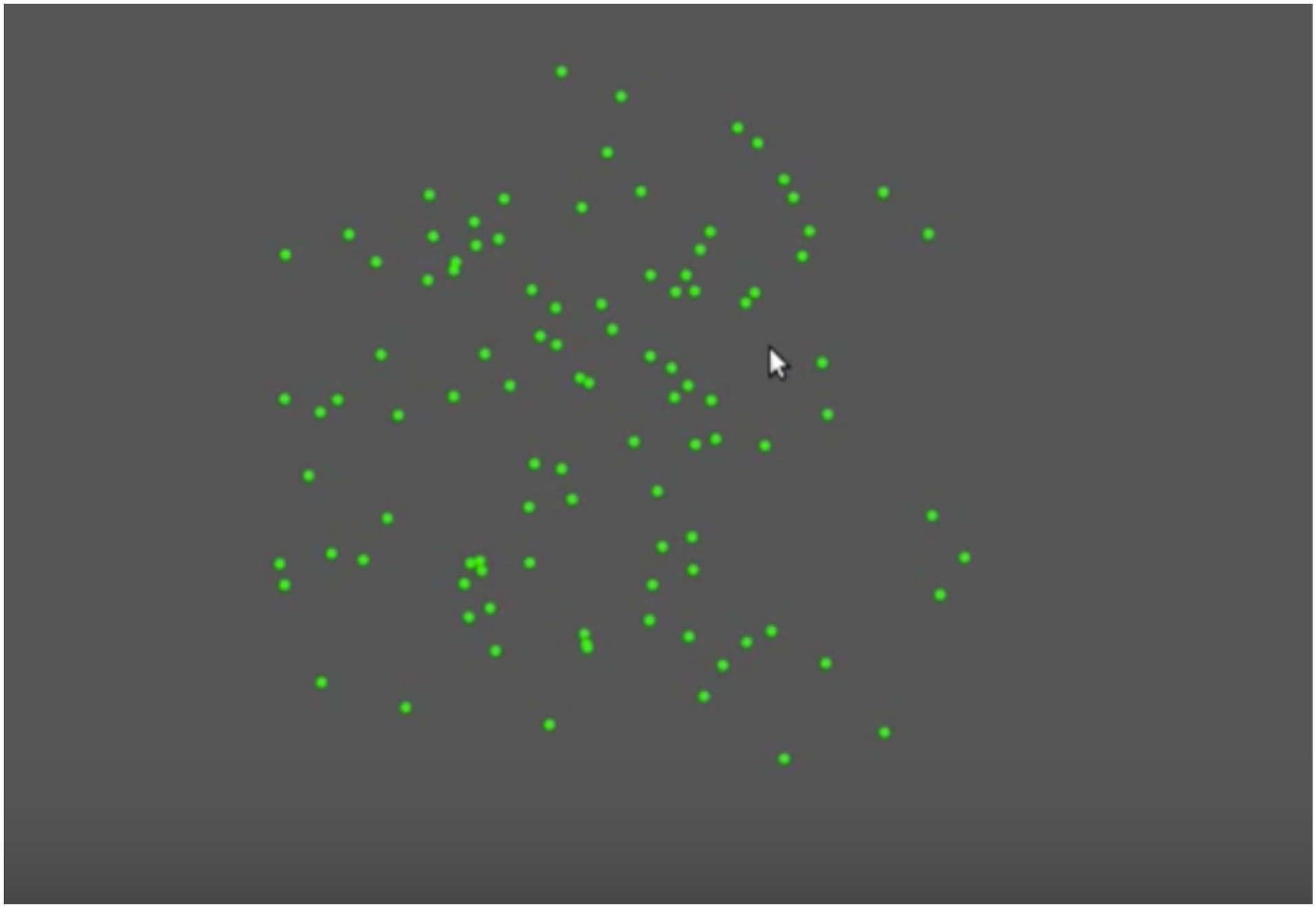


Convex Hull lower bound

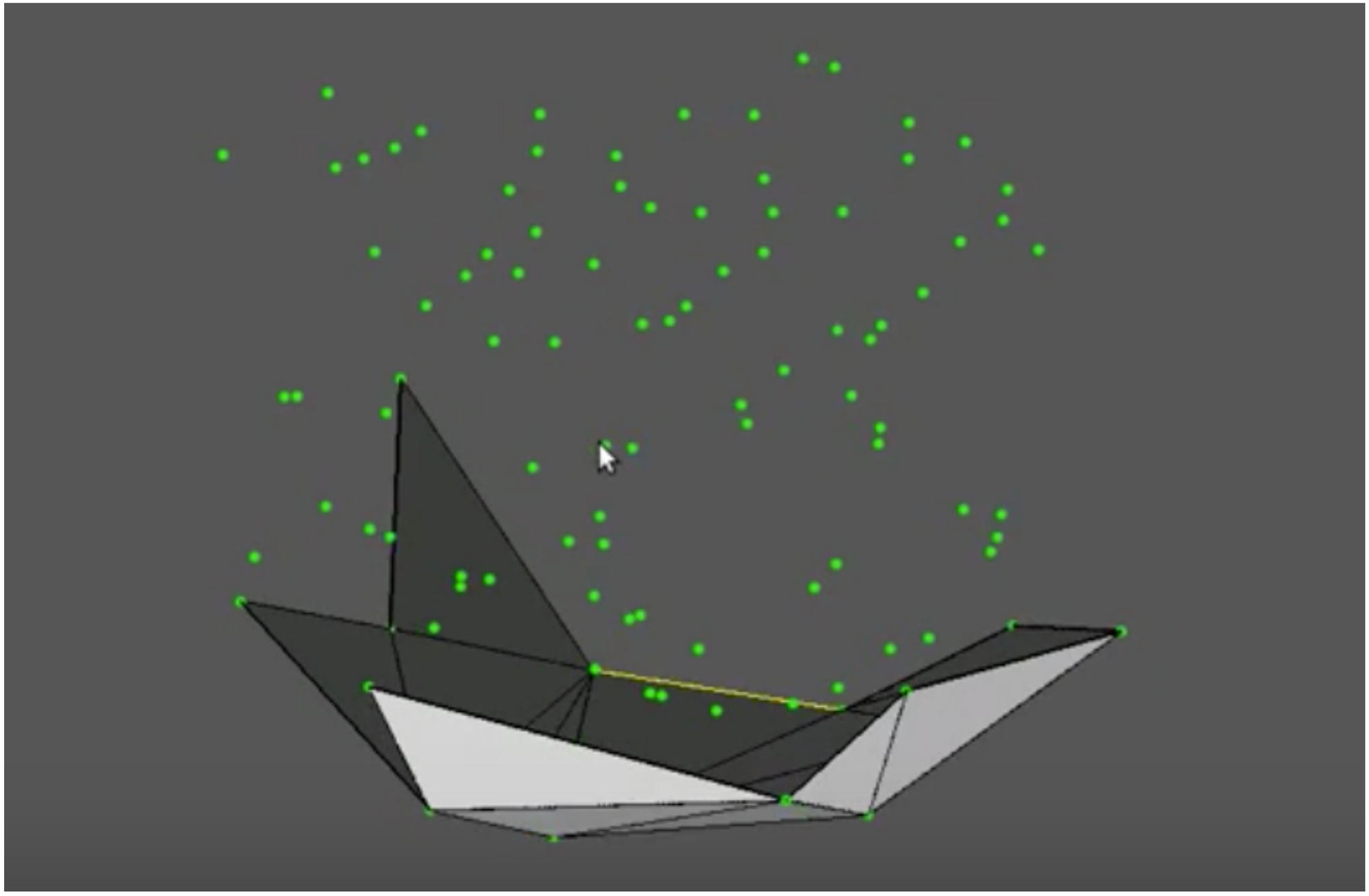
- What we actually proved is that
  - Any CH algorithm that produces the boundary in order must take Omega (n lg n) in the worst case.

- What if we did not want the boundary in order? Can the CH be constructed faster?
  - It was an open problem for a while
  - Finally, it was established that a convex hull algorithm, even if it does not produce the boundary in order, still needs Omega (n lg n)

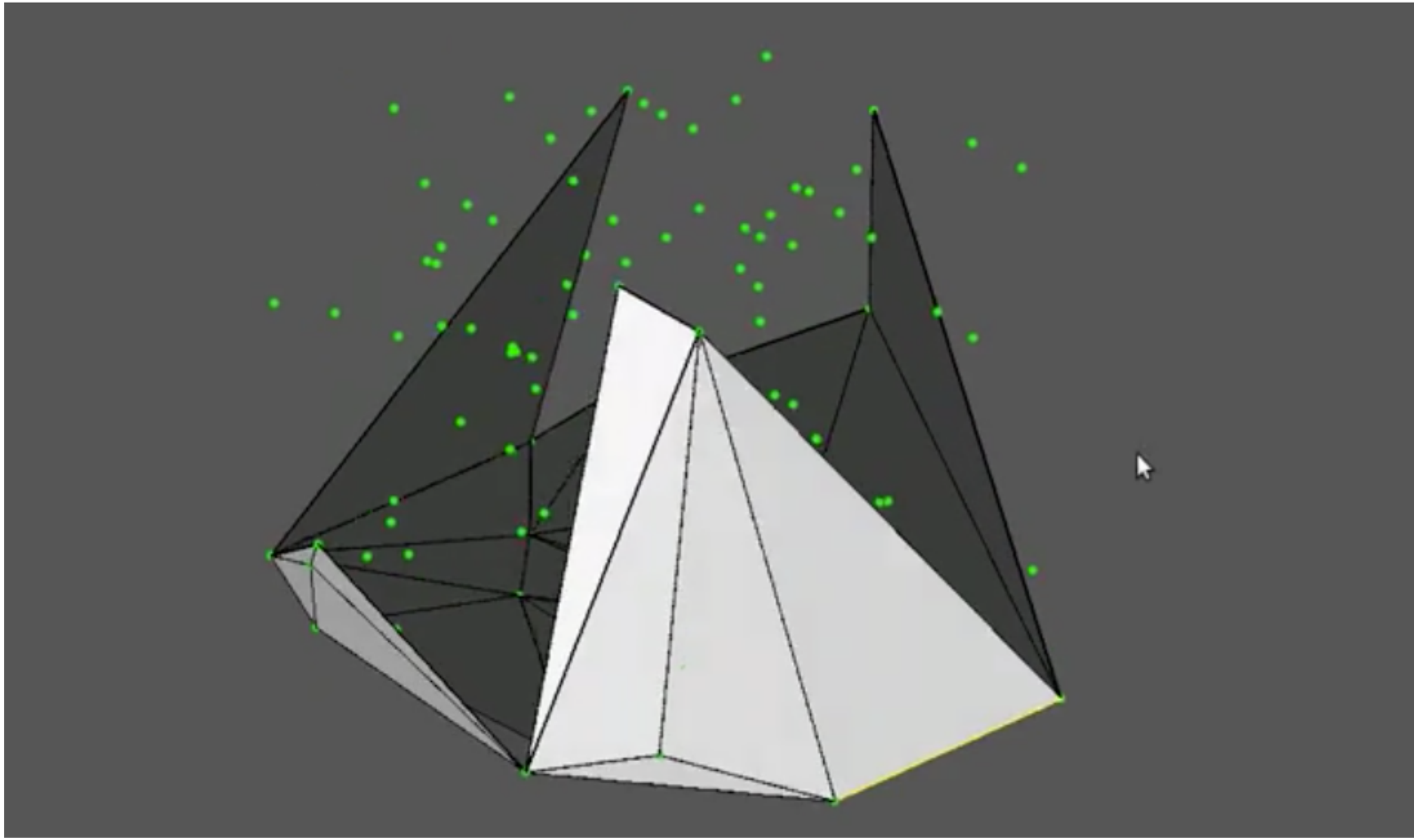
Convex Hull in 3D



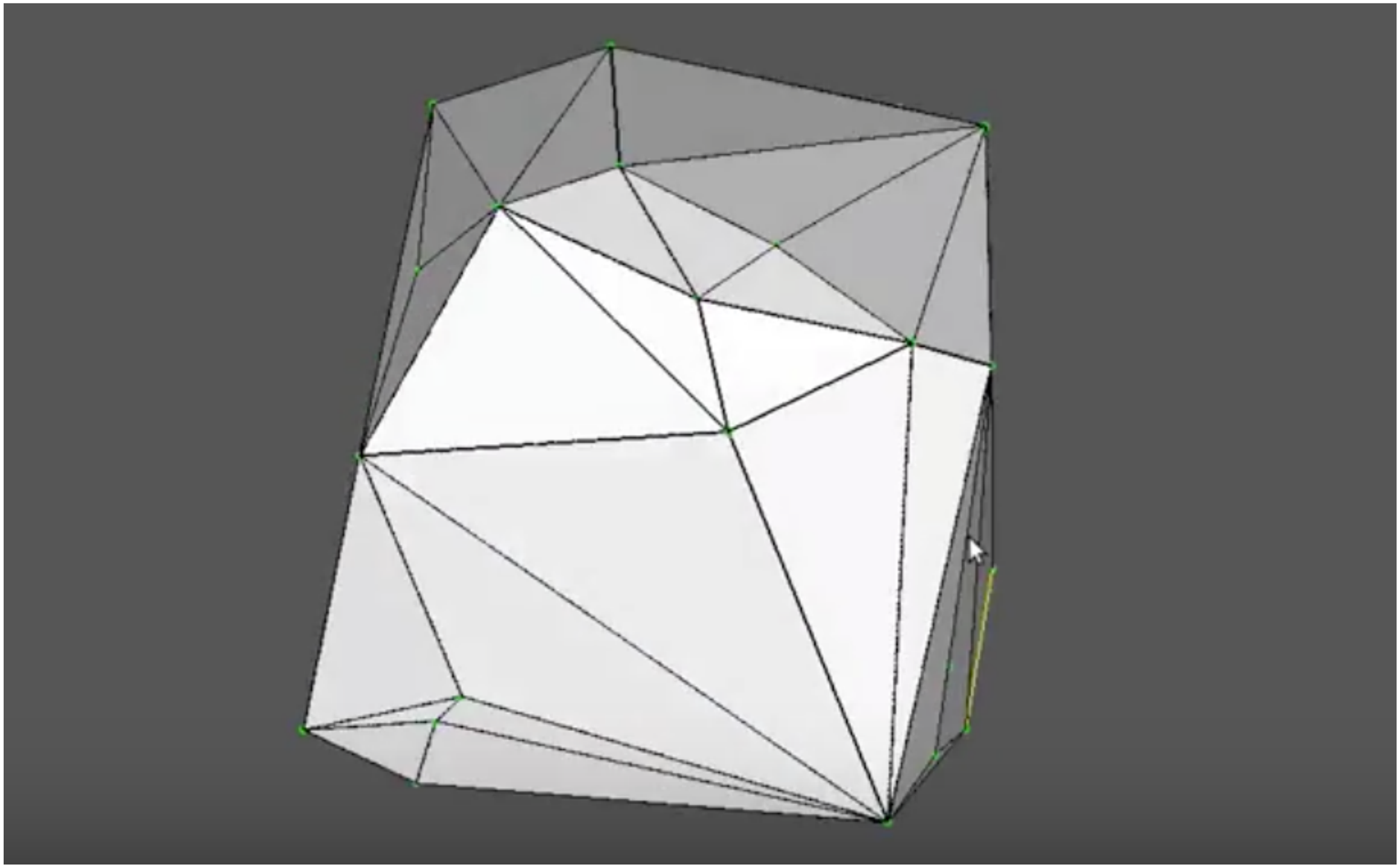
Convex Hull in 3D



Convex Hull in 3D



Convex Hull in 3D



Convex Hull in 3D

- YouTube
  - Video of CH in 3D (by Lucas Benevides)
  - Fast 3D convex hull algorithms with CGAL

Convex Hull: from 2D to 3D?

2D convex hull algorithms

- Naive (check all edges if extreme; $O(n^3)$)
  - Does this extend to 3D?
  - YES! for each triplet, check if all points on the same side; $O(n^4)$
- **Gift wrapping** (find extreme edges in order)
  - Does this extend to 3D?
  - YES! For many years it the primary algorithm for 3D; O(nh)
- Graham scan
  - Does this extend to 3D?
  - NO!
- Quickhull
  - YES!
- **Divide-and-conquer**
  - Does this extend to 3D?
  - YES! O(n lg n) in 3D