Planar Convex Hulls (III)

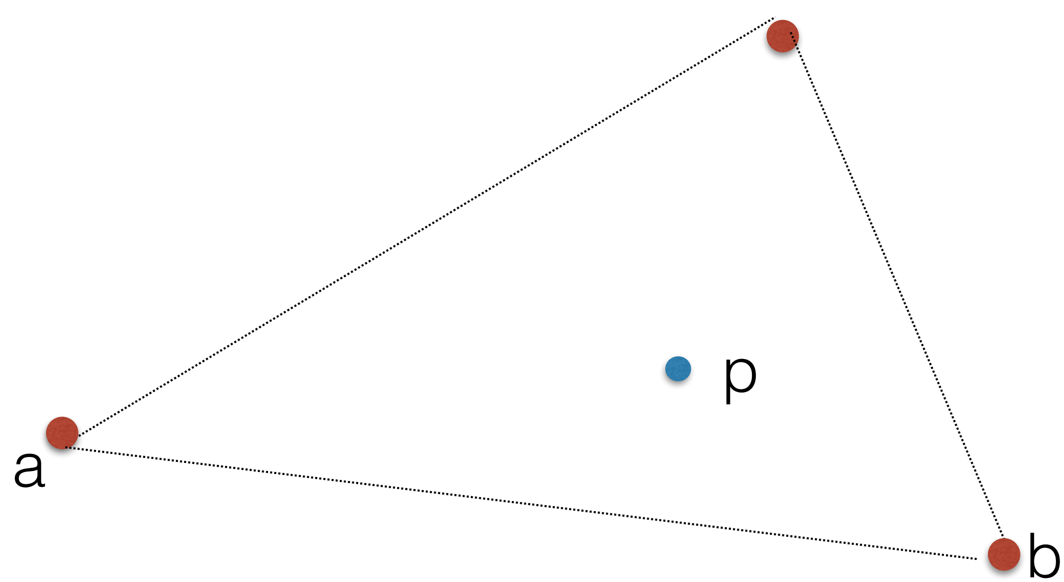
Computational Geometry [csci 3250]
Laura Toma
Bowdoin College

Algorithms for computing the convex hull

- Last time
  - Brute force
  - Gift wrapping
  - Graham scan
  - Quickhull
- Today
  - Andrew's monotone chain
  - Lower bound
  - Other algorithms:
    - Incremental hull
    - Divide-and -conquer hull

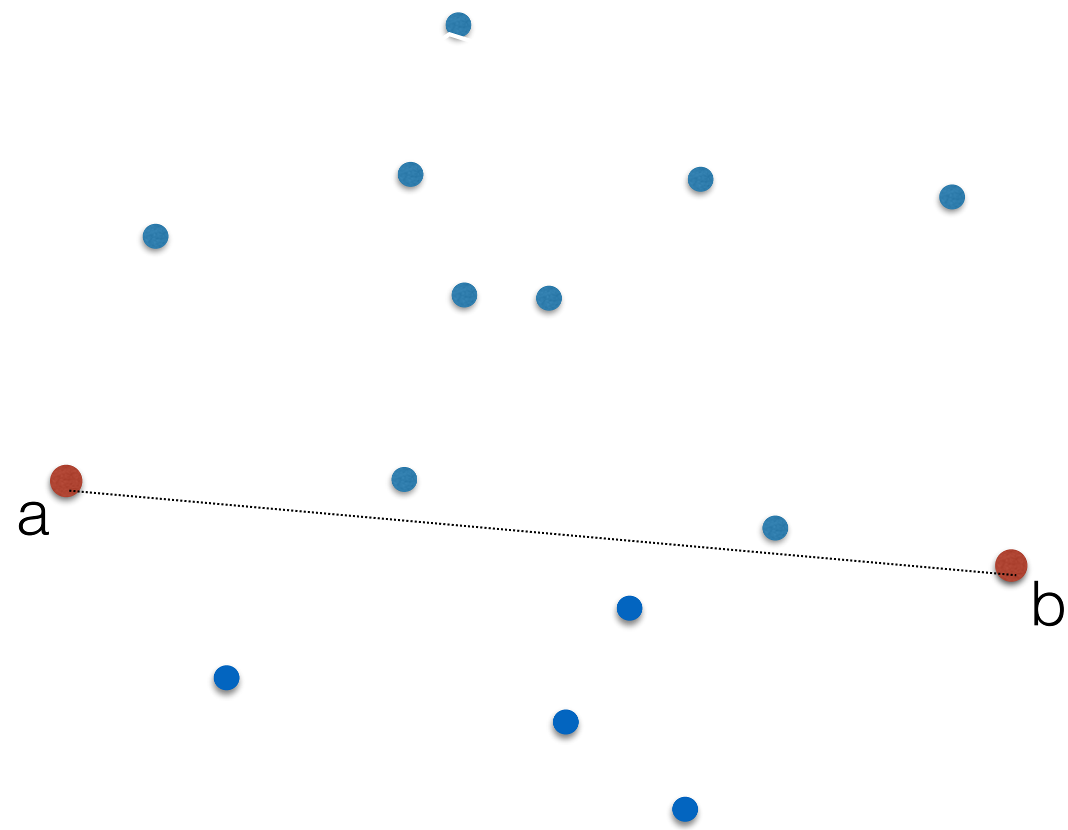
Classwork: Given a point p and a triangle a, b, c

```
//return true if p is inside (or on) abc, and false otherwise
bool isInside (p, a, b, c)
```



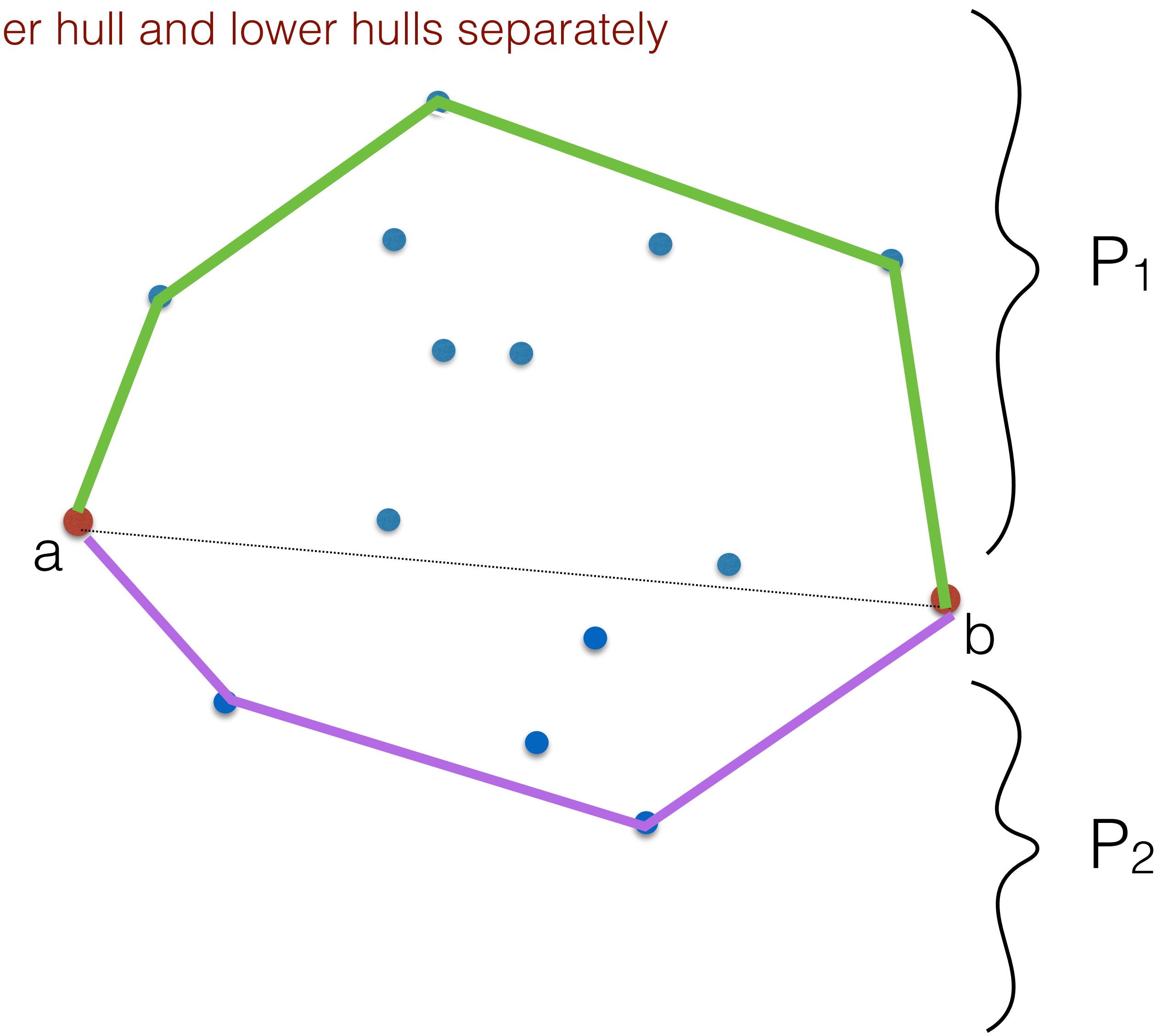
Andrew's Monotone Chain Algorithm

- Alternative to Graham's scan, **faster in practice**
- Idea: Find upper hull and lower hulls separately

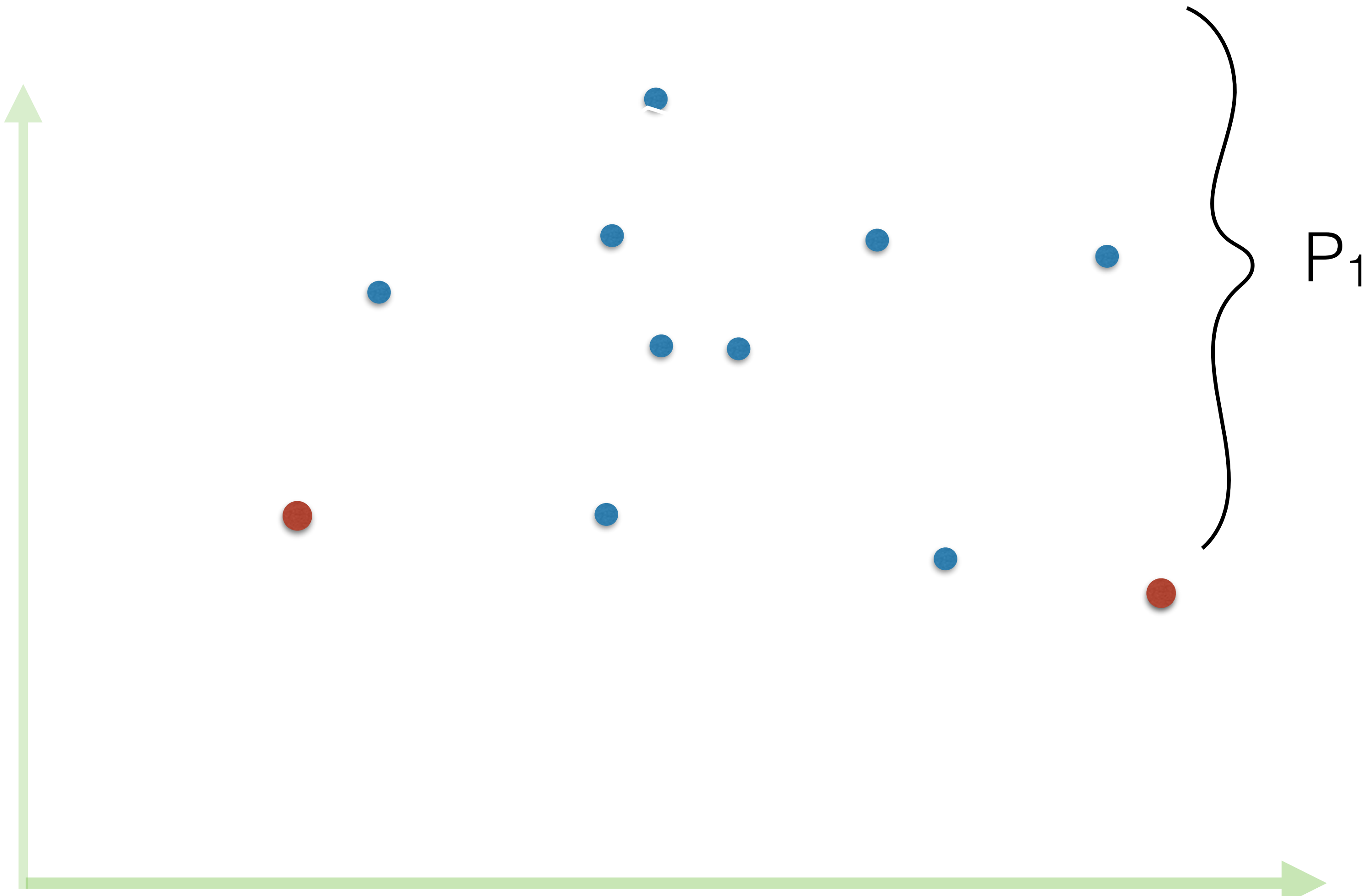


Andrew's Monotone Chain Algorithm

- Alternative to Graham's scan, **faster in practice**
- Idea: Find upper hull and lower hulls separately

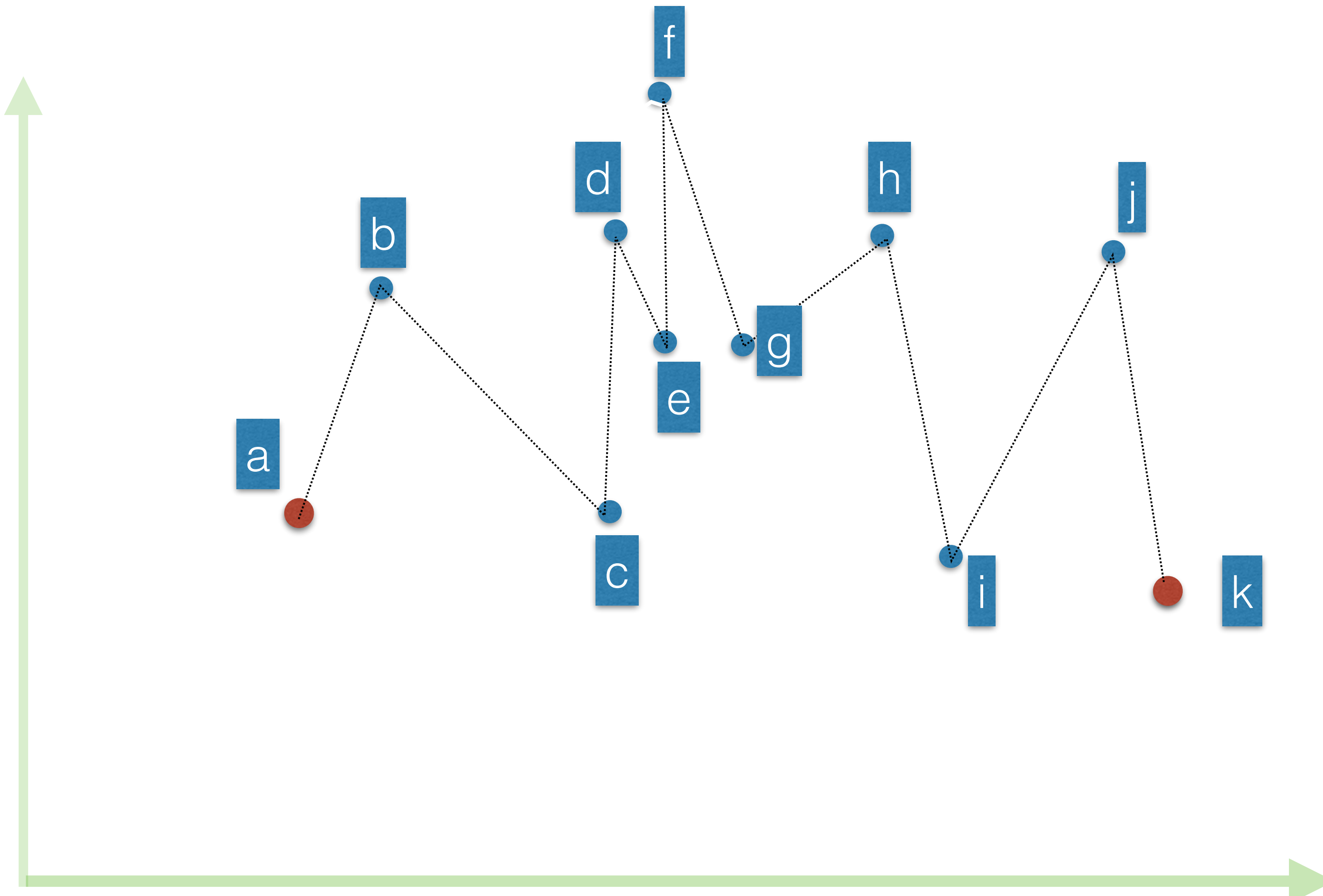


- Order these points in (x,y) lexicographic order



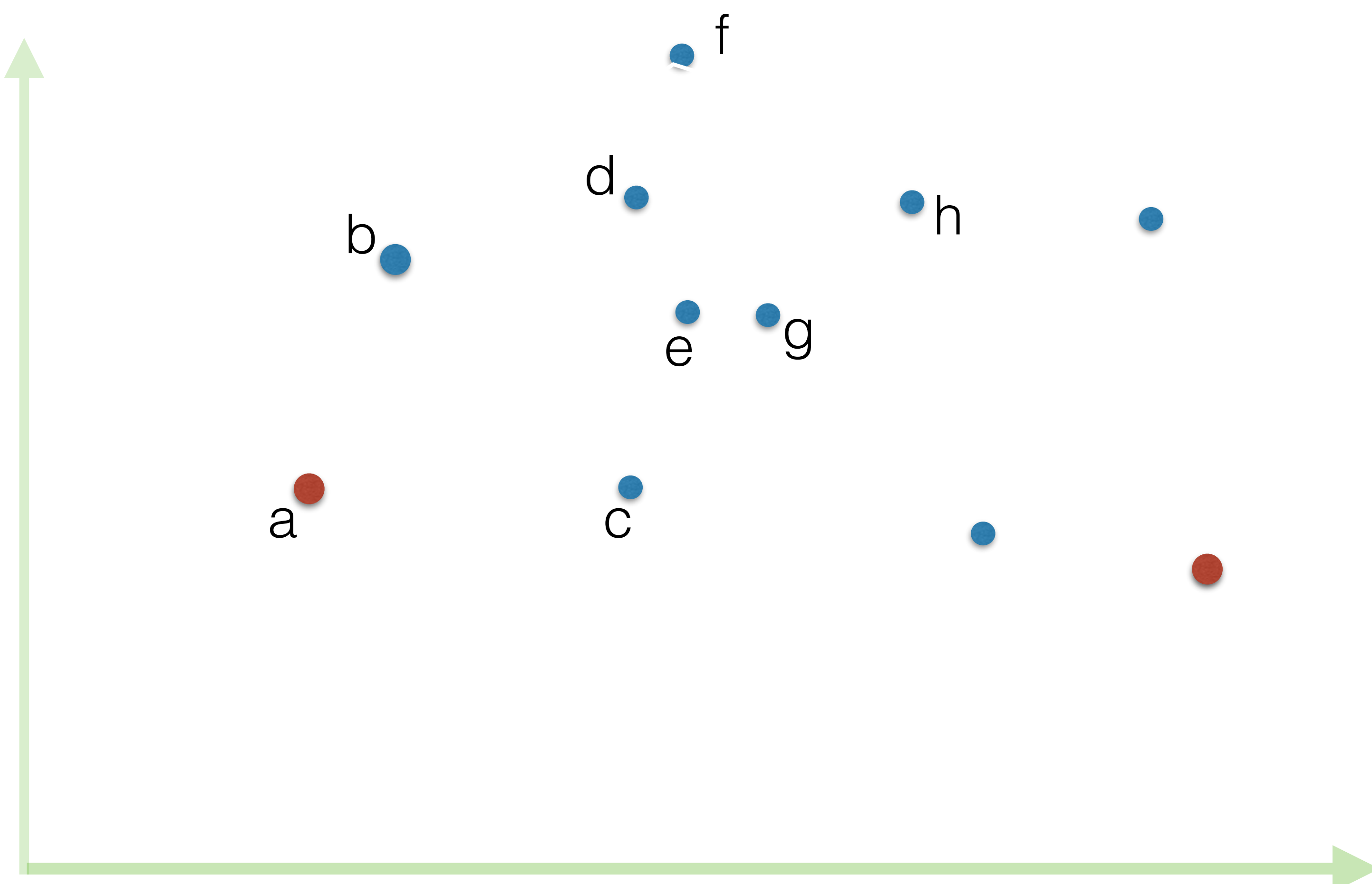
called: lexicographic order

- Order these points in (x,y) order (first by x, second by y)



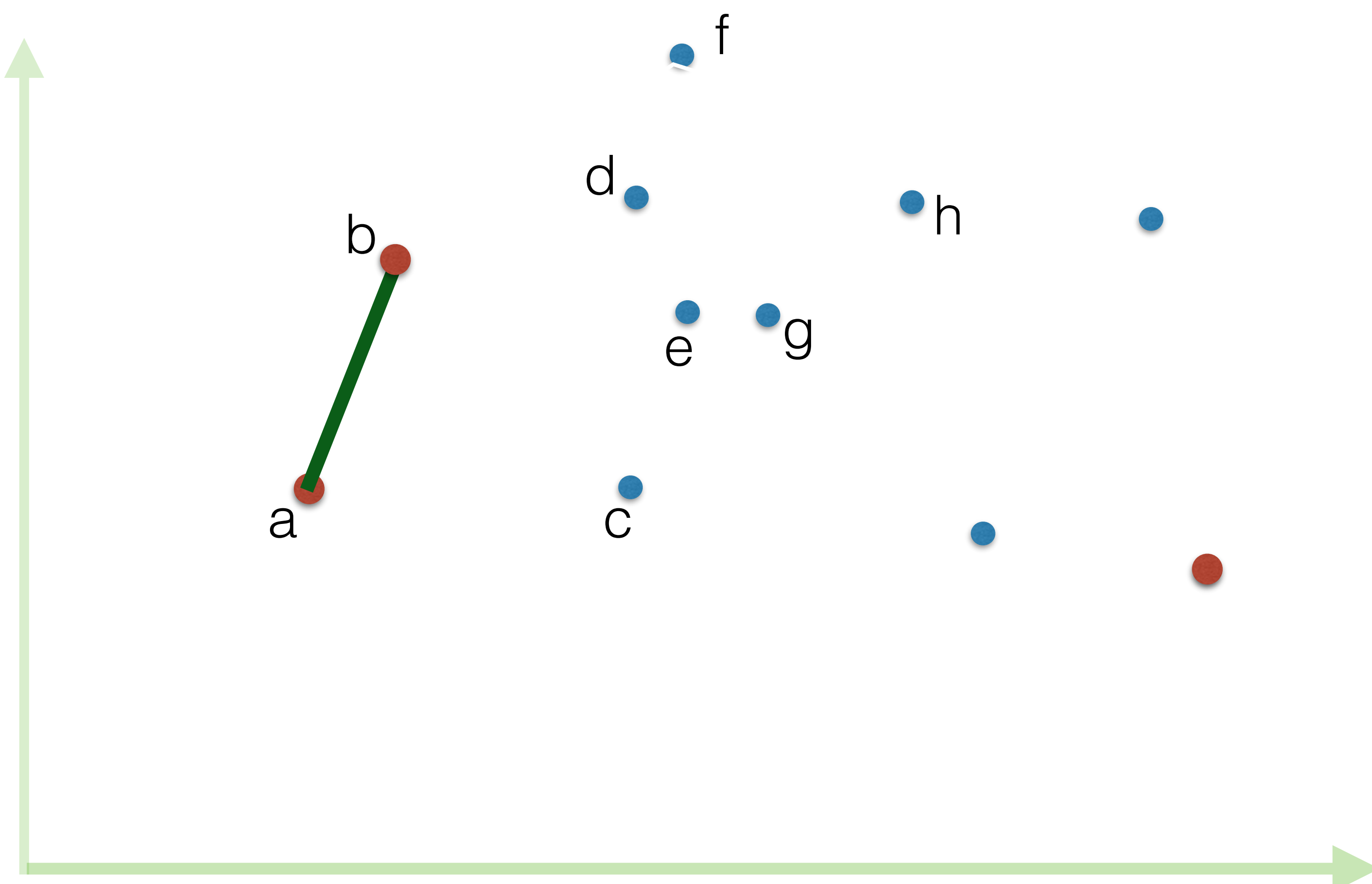
Finding the upper hull of P1

- Traverse points in (x,y) order and build the upper hull, like in Graham scan

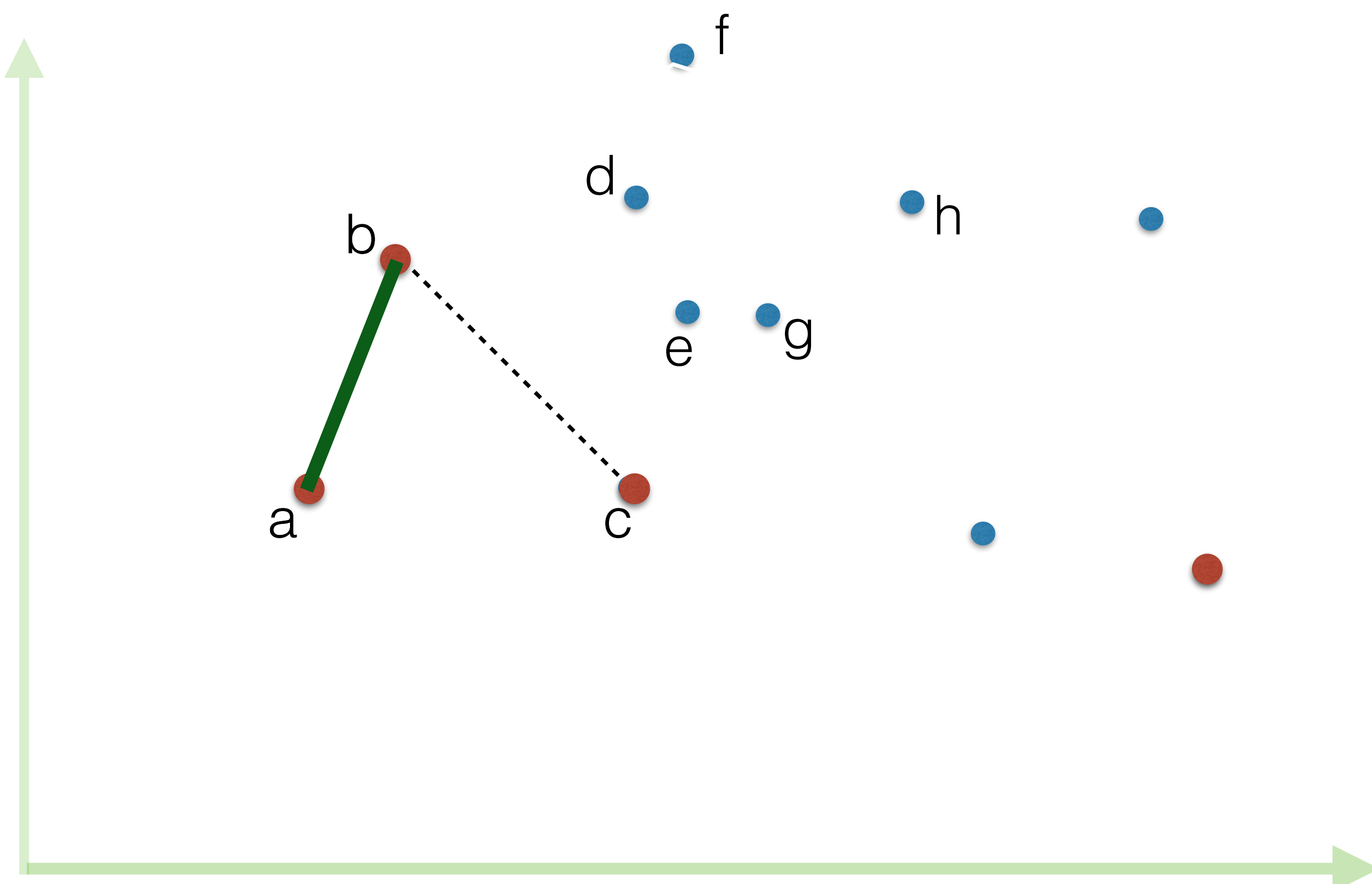


Finding the upper hull of P1

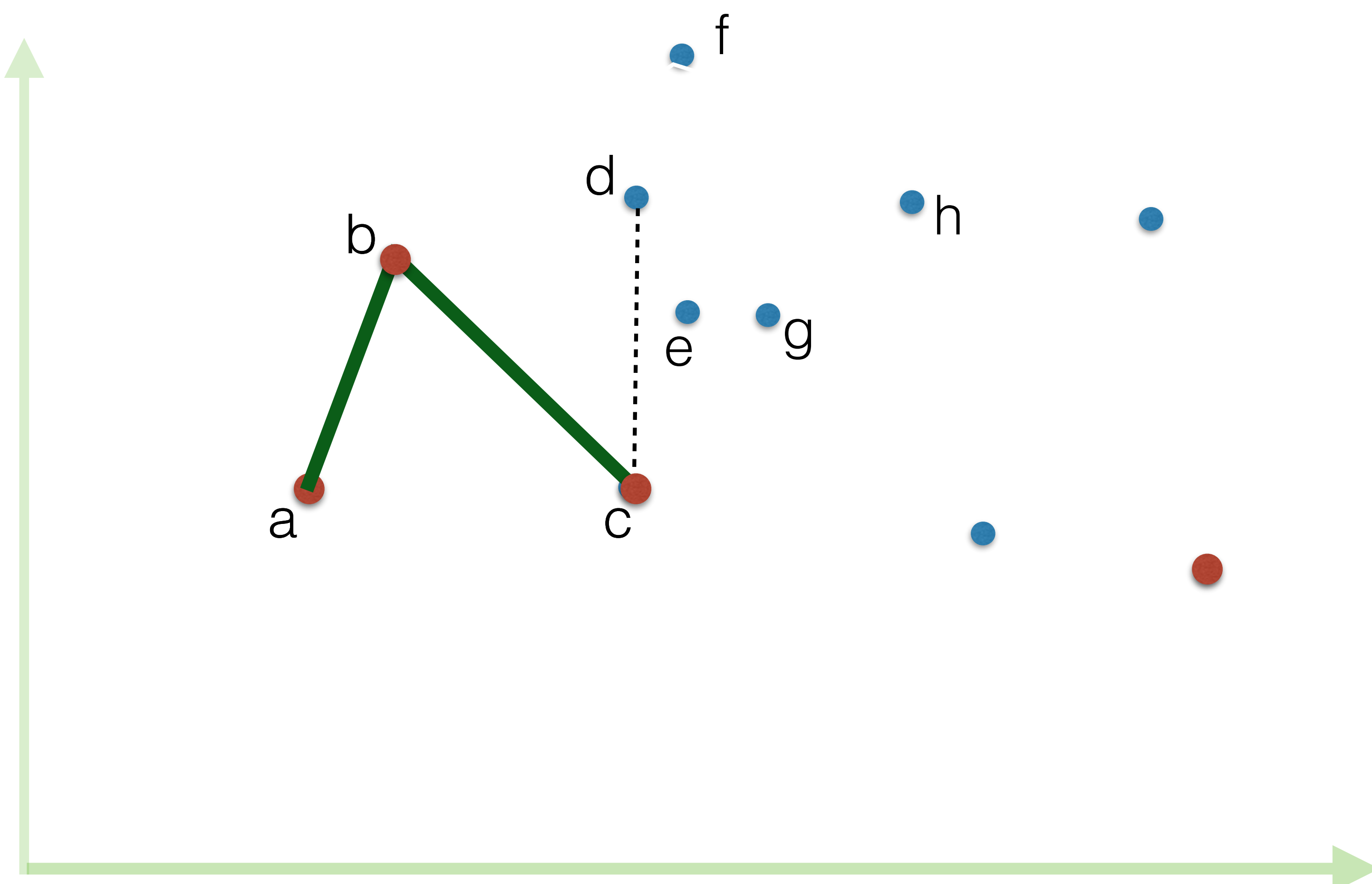
- Traverse points in (x,y) order and build the upper hull, like in Graham scan



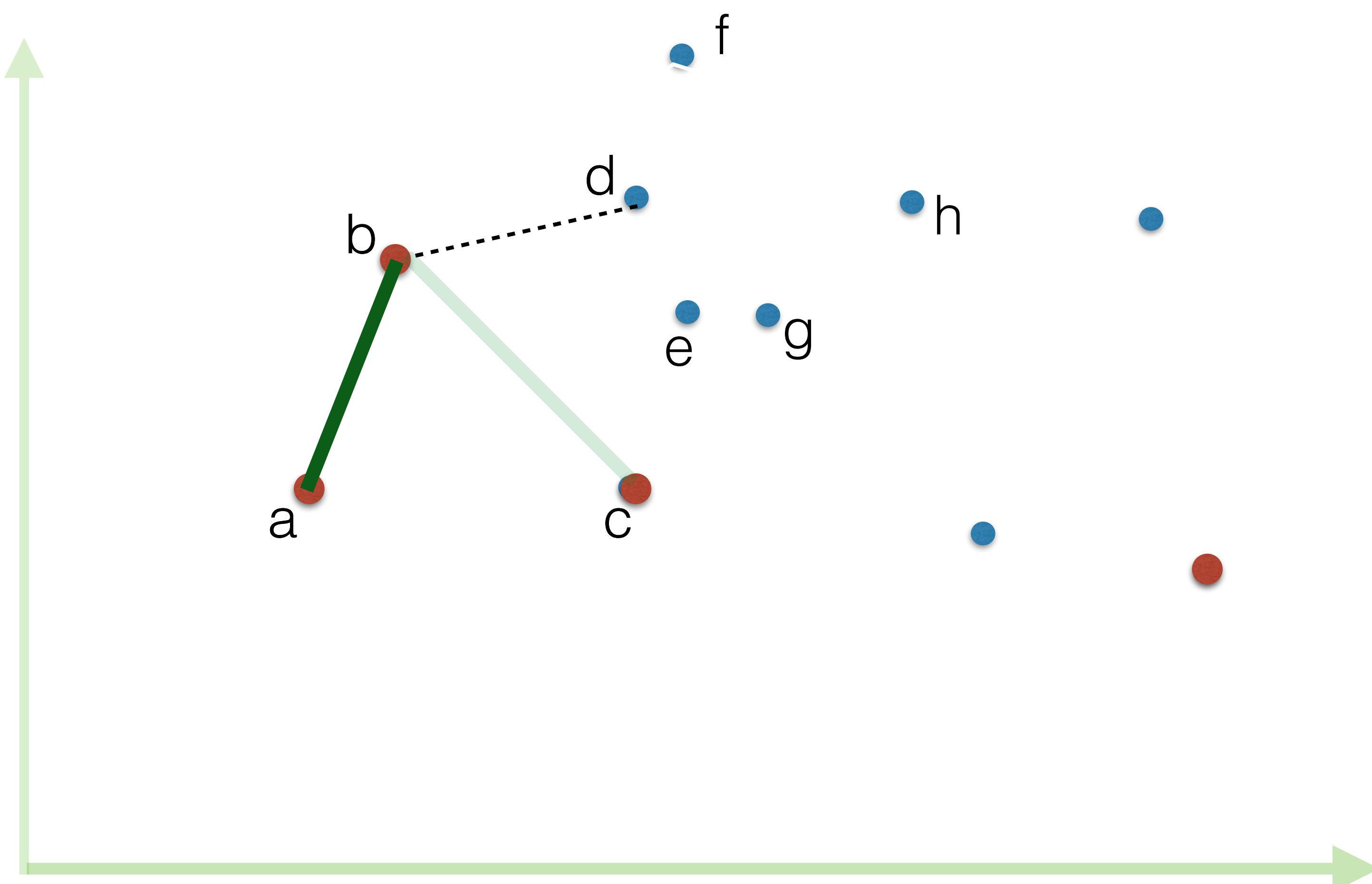
Finding the upper hull of P1



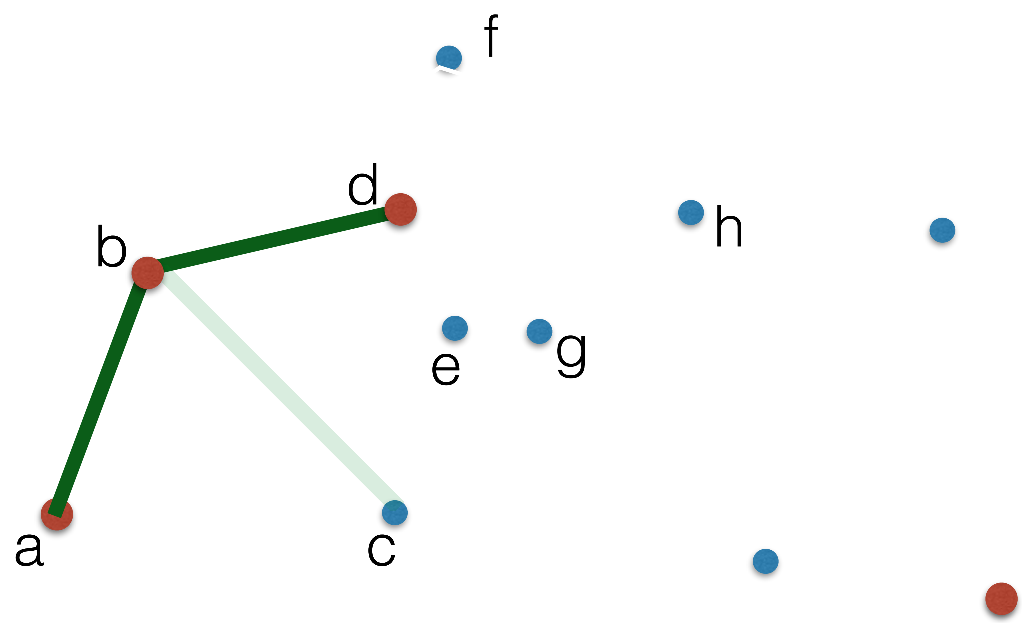
Finding the upper hull of P1



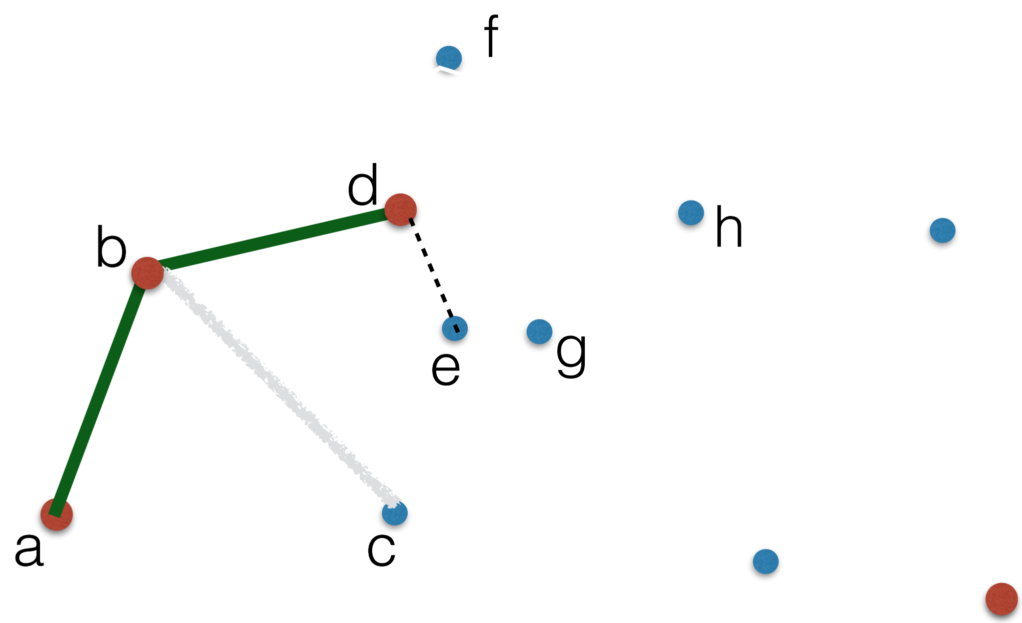
Finding the upper hull of P1



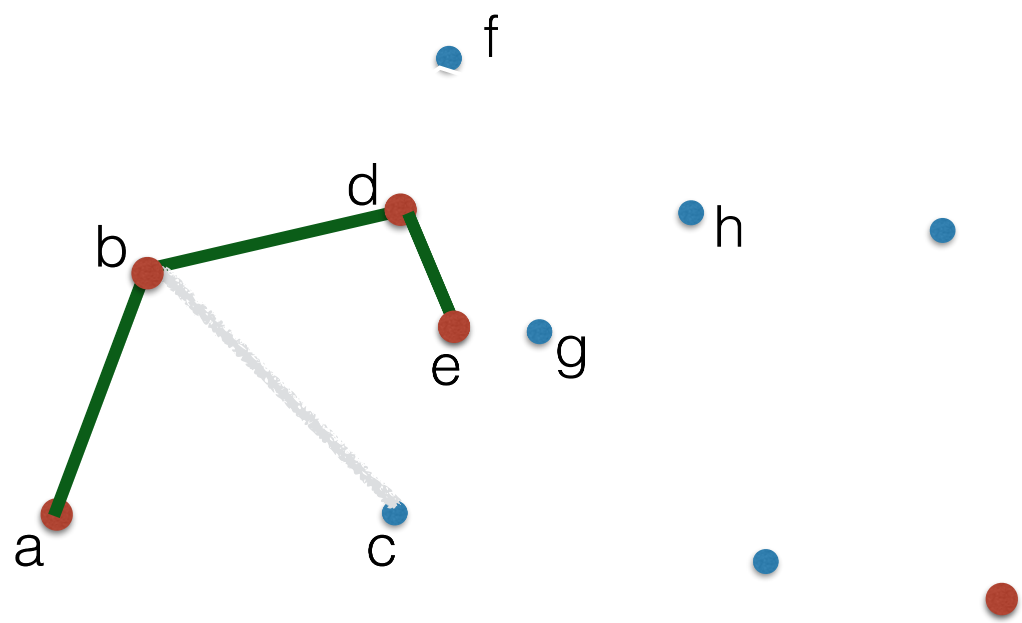
Finding the upper hull of P1



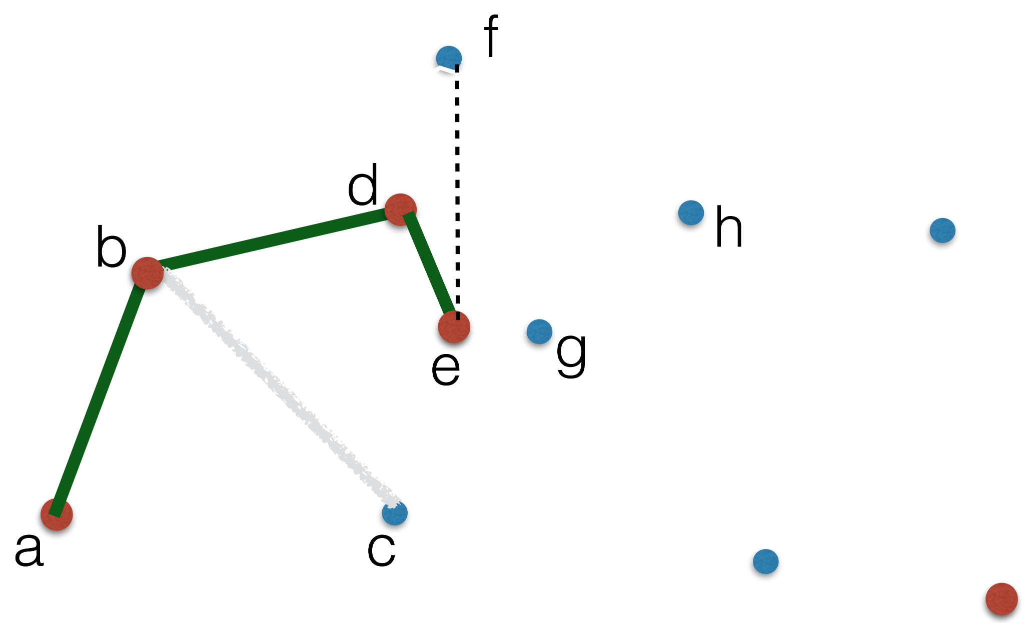
Finding the upper hull of P1



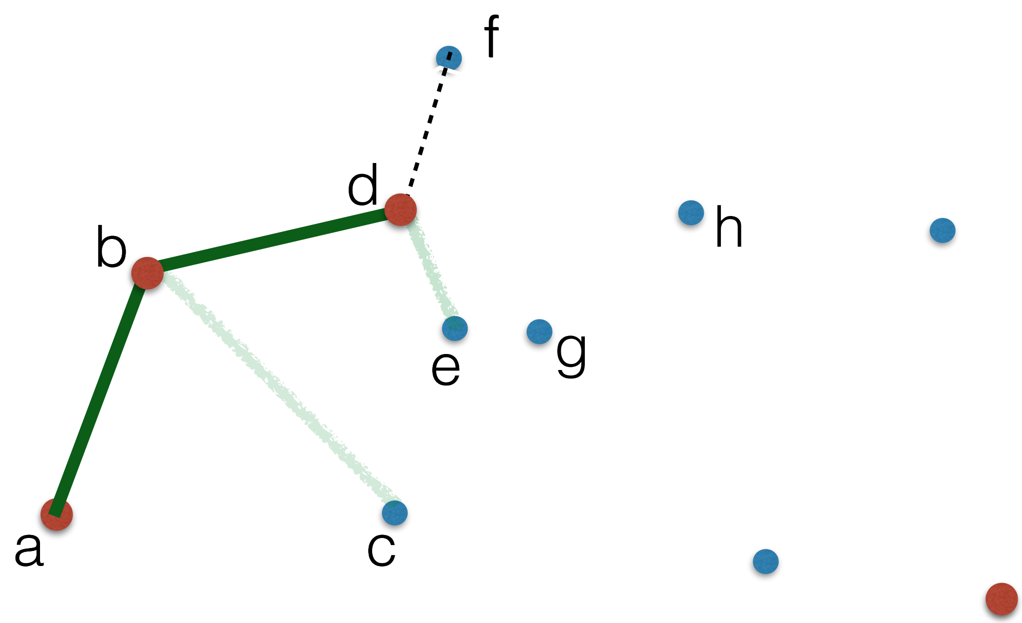
Finding the upper hull of P1



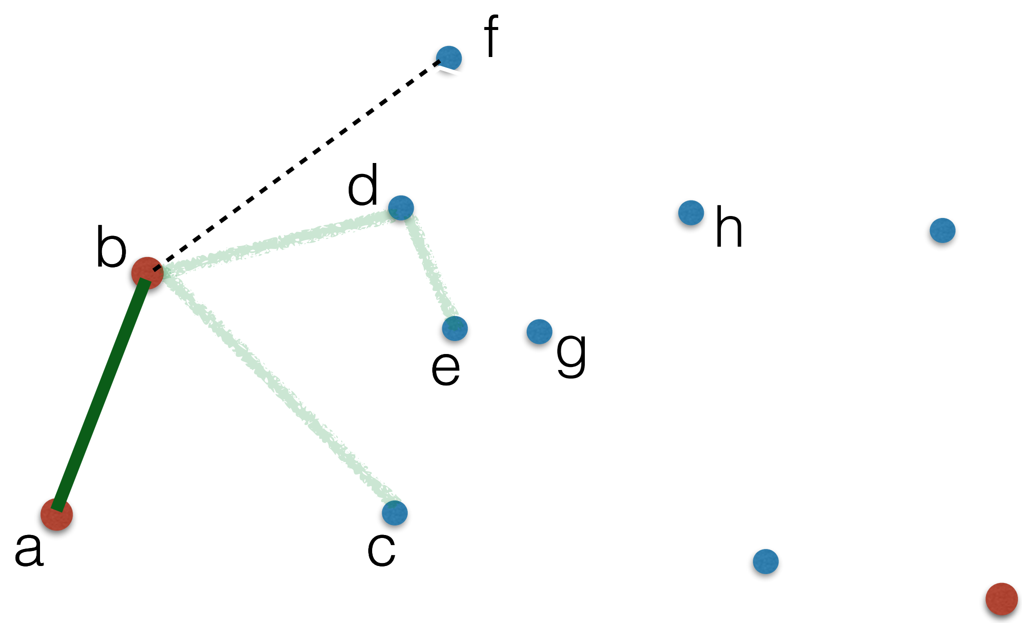
Finding the upper hull of P1



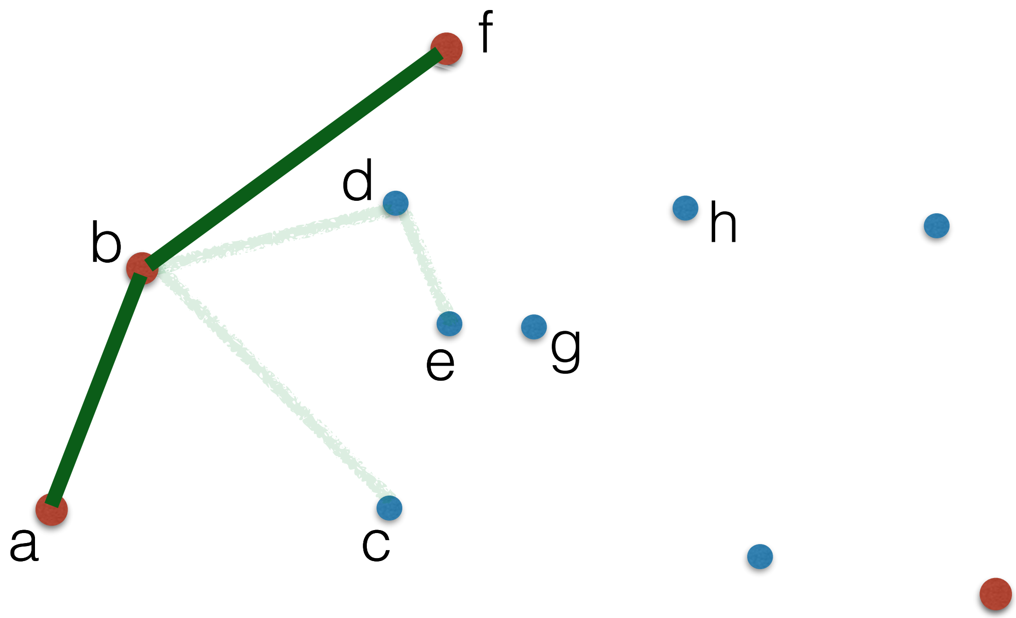
Finding the upper hull of P1



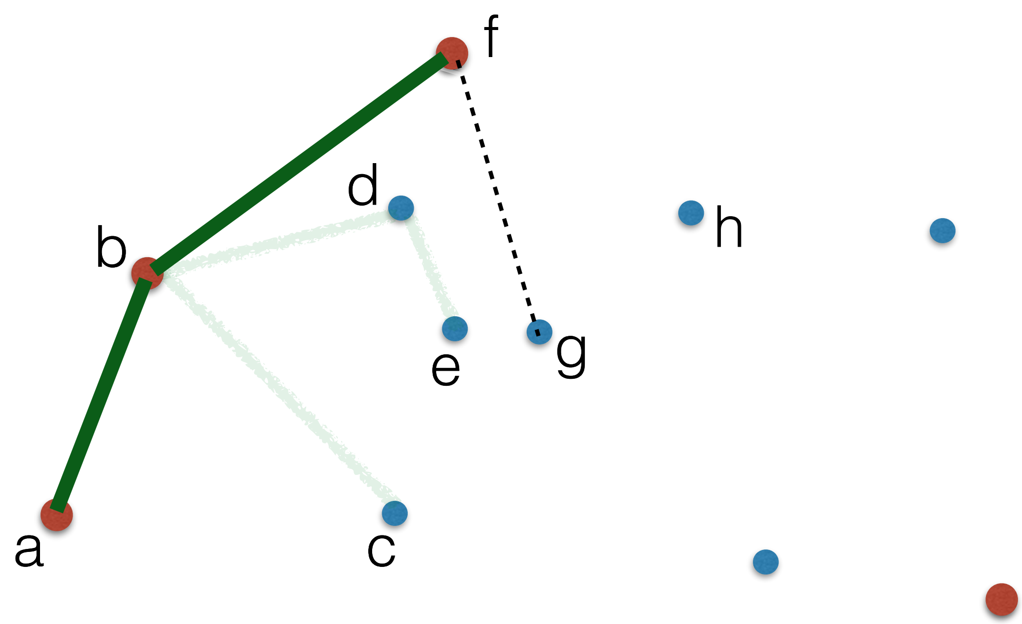
Finding the upper hull of P1



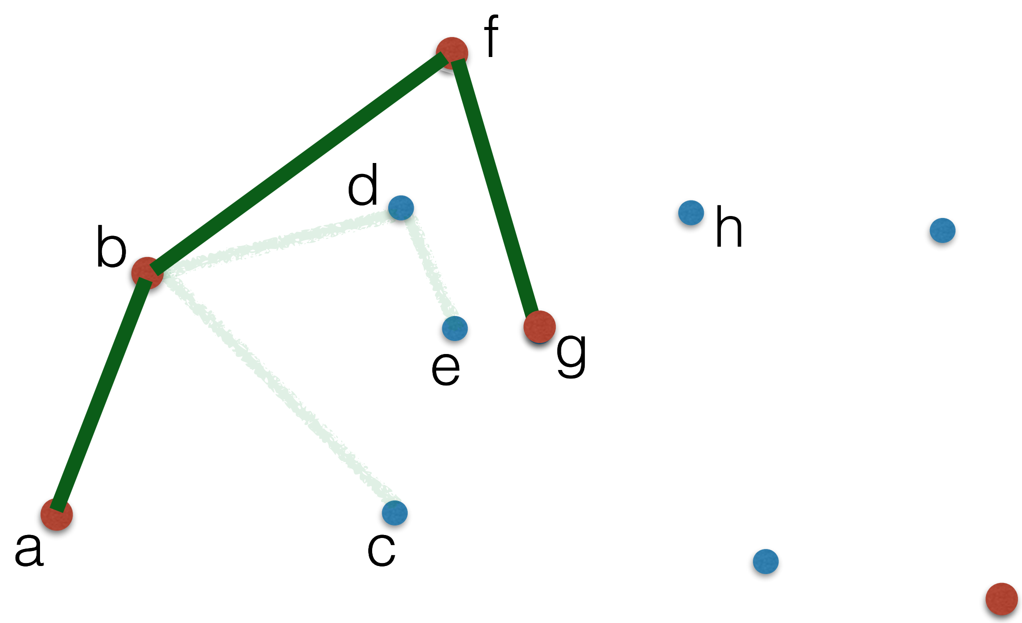
Finding the upper hull of P1



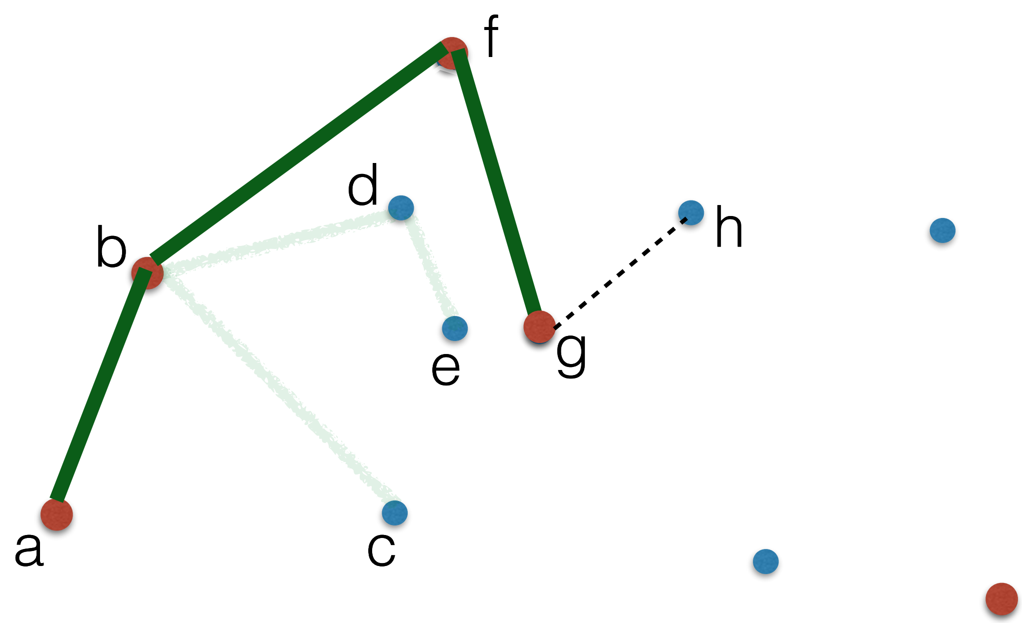
Finding the upper hull of P1



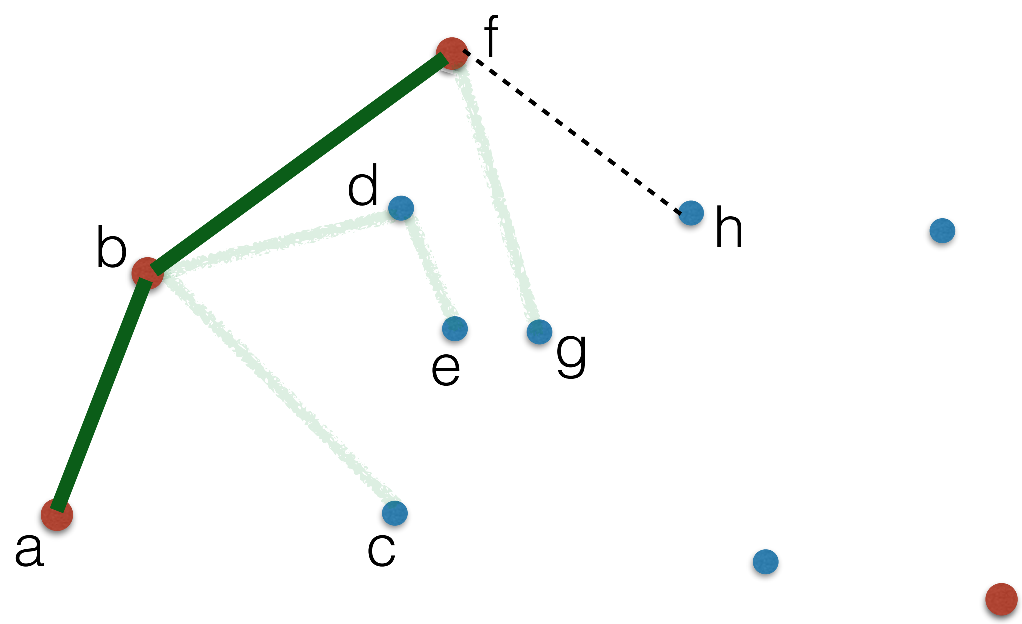
Finding the upper hull of P1



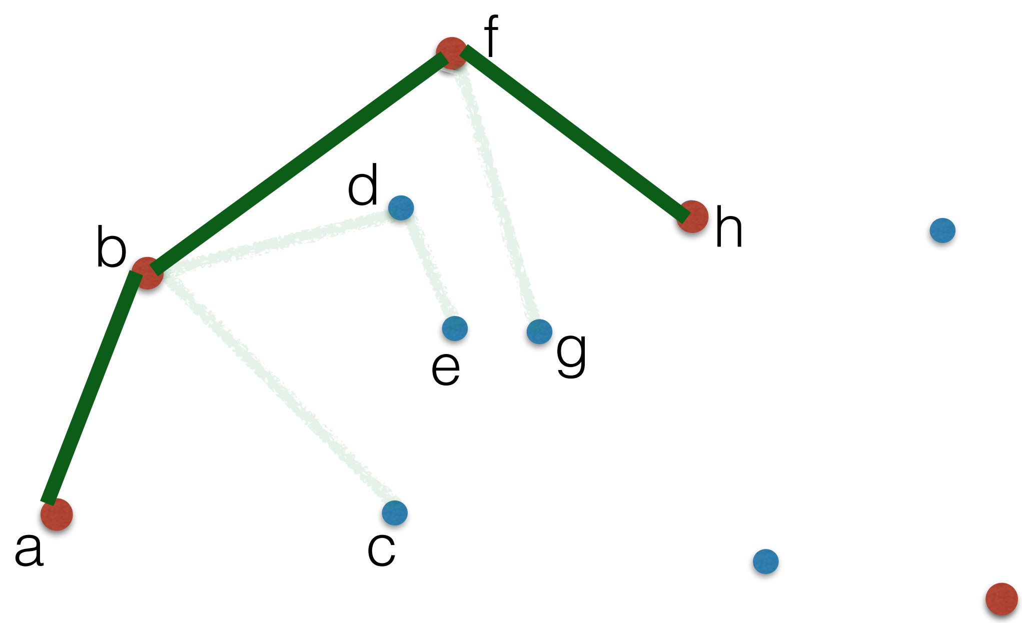
Finding the upper hull of P1



Finding the upper hull of P1



Finding the upper hull of P1



and so on..

Andrew's Monotone Chain Algorithm

- Alternative to Graham's scan
- Same running time
- Sorting by (x,y) is faster (in practice) than sorting radially

Convex hull: summary

Naive	$O(n^3)$
Gift wrapping	$O(h \cdot n)$
Quickhull	$O(n^2)$
Graham scan	$O(n \lg n)$
Andrew monotone chain	$O(n \lg n)$

Can we do better?

Lower bound

What is a lower bound?

- Given an algorithm A, its **worst-case running time** is the **largest** running time on any input of size n

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

- A lower bound $L(n)$ for a problem is a lower bound on the worst-case running time of any algorithm that solves that problem

$$T_A(n) = \Omega(L(n)), \text{ for all algorithms A that solve the problem}$$

- We could say that Convex hull has a lower bound $L(n) = \Omega(1)$ (trivial). We could also say that $L(n) = \Omega(n)$, also trivial.
- We want larger lower bounds (and lower upper bounds!)
- When the best-known worst-case $T(n)$ of an algorithm, matches the best-known lower bound for that problem, the problem is considered "solved". An algorithm that matches the lower bound is optimal!

Proving lower bounds

- Lower bounds depend on the machine model.
  - The standard model is the decision tree (comparison) model
- We can prove lower bounds directly
  - Theorem: Any sorting algorithm that uses only comparisons uses at least $\Omega(n \lg n)$ comparisons in the worst case.
  - Proof: We saw this in Algorithms..
- Or, via **reduction** from a problem known to have a lower bound
  - aka: $n \lg n < A$ and $A < B \Longrightarrow n \lg n < B$

Lower bounds by reduction

- We know that $\Omega(n \lg n) \leq$ Sorting
- If we could show that ConvexHull is at least as hard as Sorting

Sorting $\leq$ Convex hull

This would imply that ConvexHull is $\Omega(n \lg n)$

How do we show Sorting $\leq$ Convex hull ?

- We'll show that we can use ConvexHull to Sort:
  - Let P be a set of values that need to be sorted. We'll show that there exists some instance of the CH problem that sorts P, and we can build this instance in O(n) time

sortViaCH (array P of n real values)

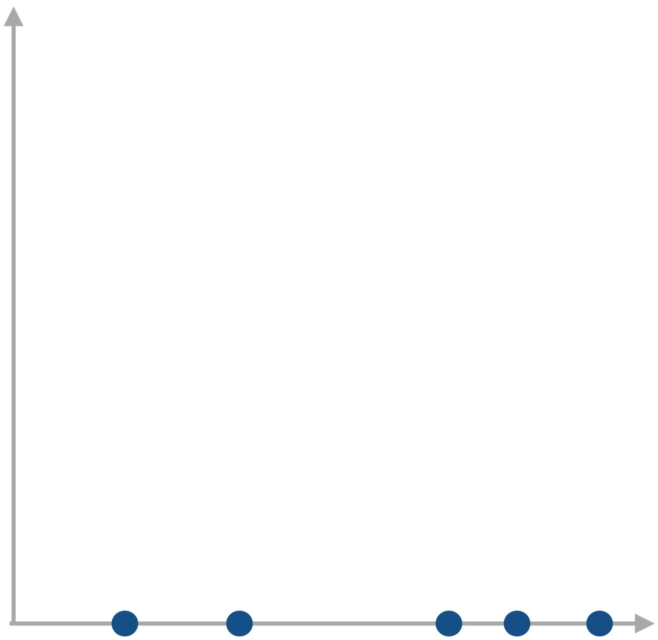
- create a set P' of points from P ← O(n)
- **findConvexHull(P')**
- use the convex hull to infer sorted order of P ← O(n)

Running time of sortViaCH: O(n) + O(findConvexHull))

- If we could find the CH faster than $\Theta(n \lg n)$ in the worst case, we could use it to sort faster than $\Theta(n \lg n)$ in the worst case, which we know is impossible!

Sorting via ConvexHull

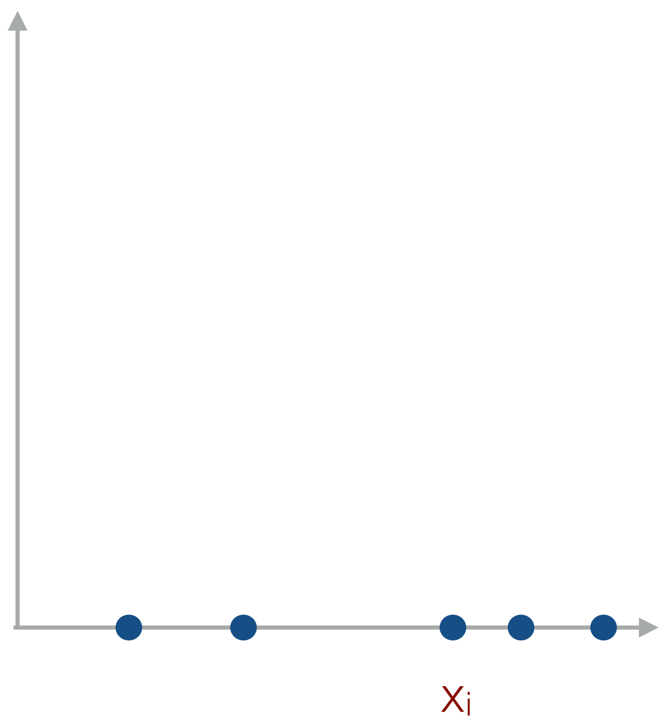
- Let P: array of real values x_1, x_2, …x_n. to sort

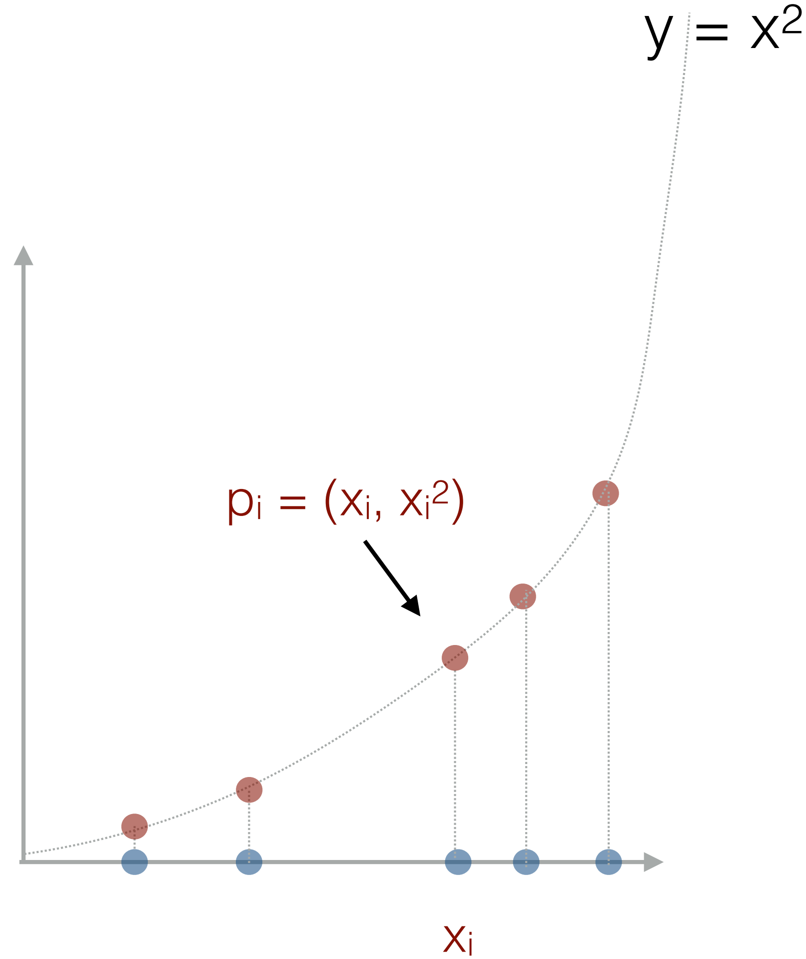


We want to find an instance of a convex hull problem that sorts P.

Sorting via ConvexHull

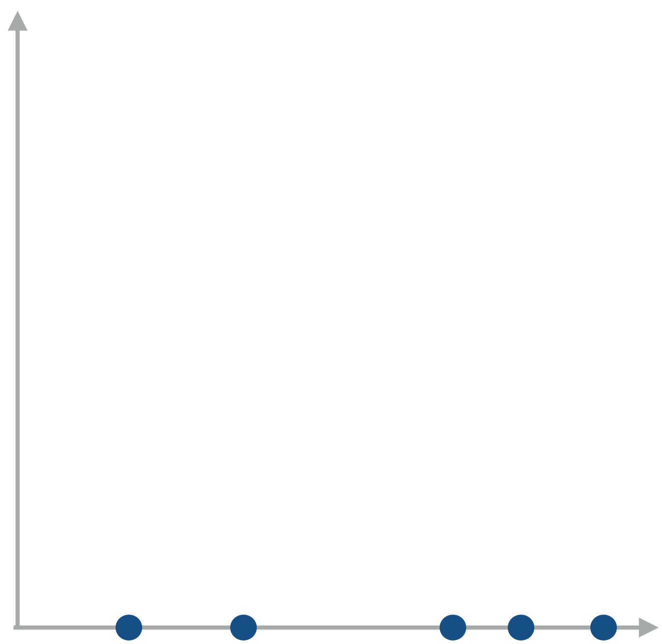
- Let P: array of real values $x_1, x_2, \ldots x_n$ to sort
- Let P': set points $\{ p_i = (x_i, x_i^2)\}$



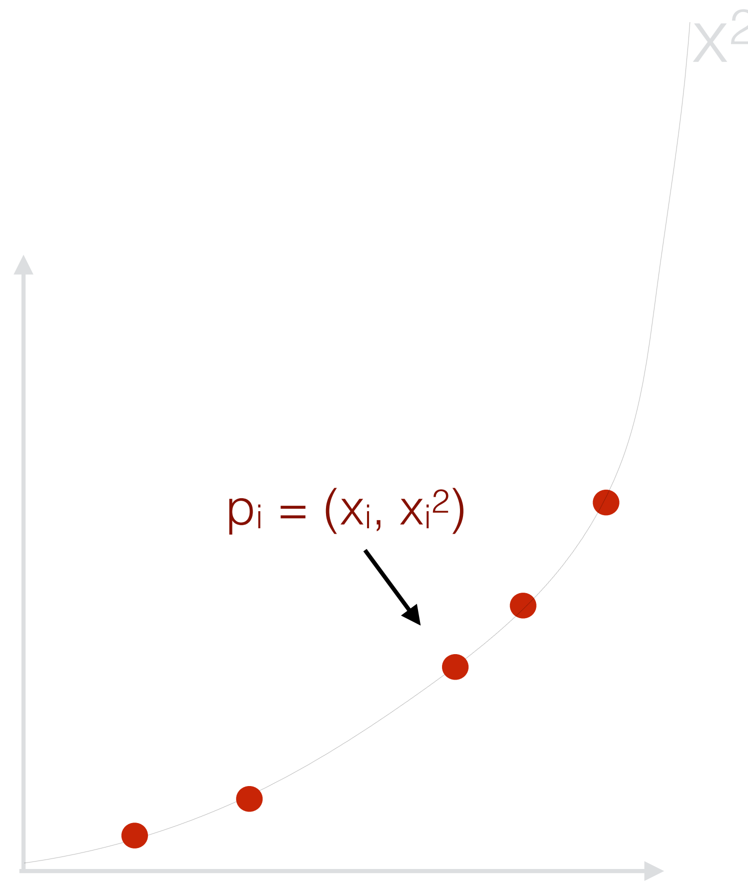


Sorting via ConvexHull

- Let P: set of values $x_1, x_2, \ldots x_n$. to sort

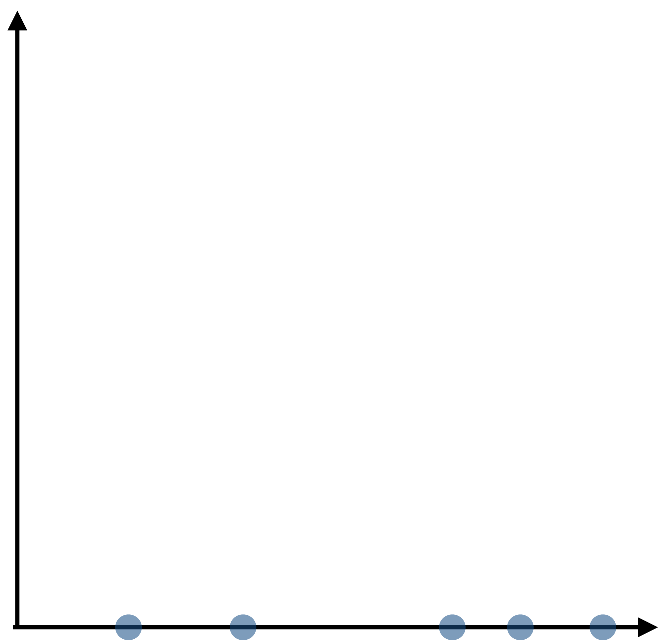


- Let P': set points $\{ p_i = (x_i, x_i^2) \}$

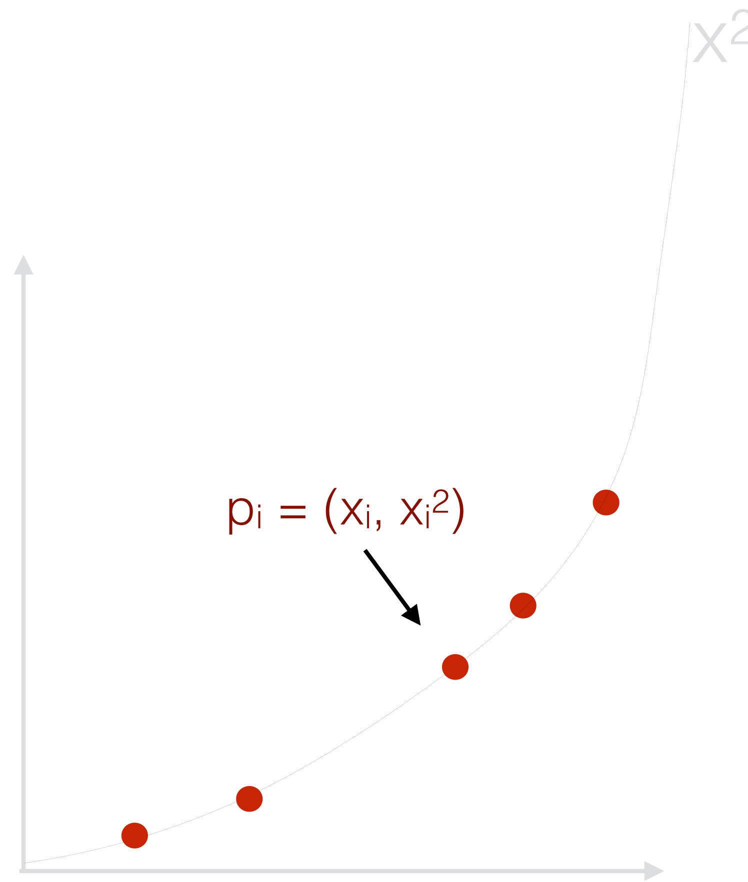


Sorting via ConvexHull

- Let P: set of values $x_1, x_2, \ldots x_n$. to sort

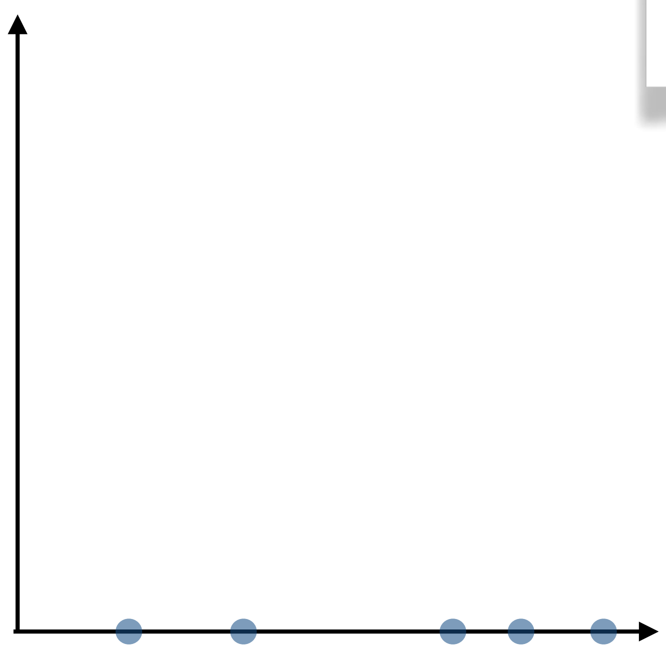


- Let P': set points $\{ p_i = (x_i, x_i^2)\}$
- Run CH(P') to find their convex hull



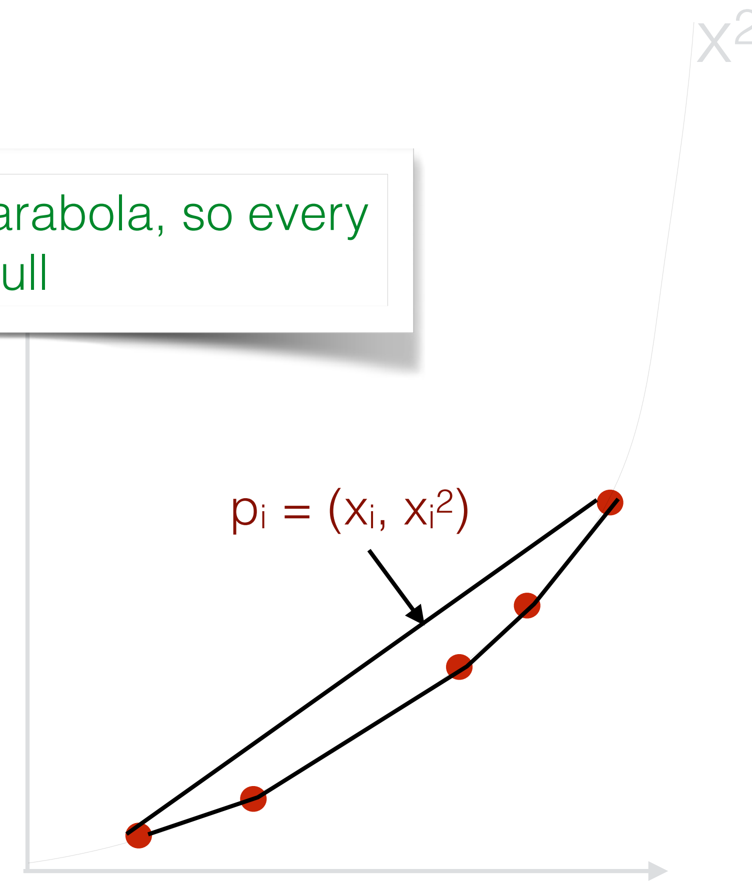
Sorting via ConvexHull

- Let P: set of values $x_1, x_2, \ldots x_n$. to sort



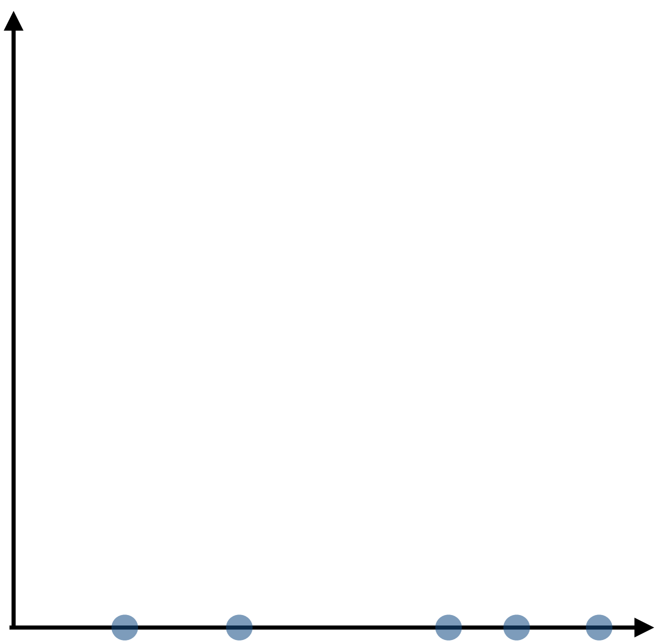
- Let P': set points { $p_i = (x_i, x_i^2)$}
- Run CH(P') to find their convex hull

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

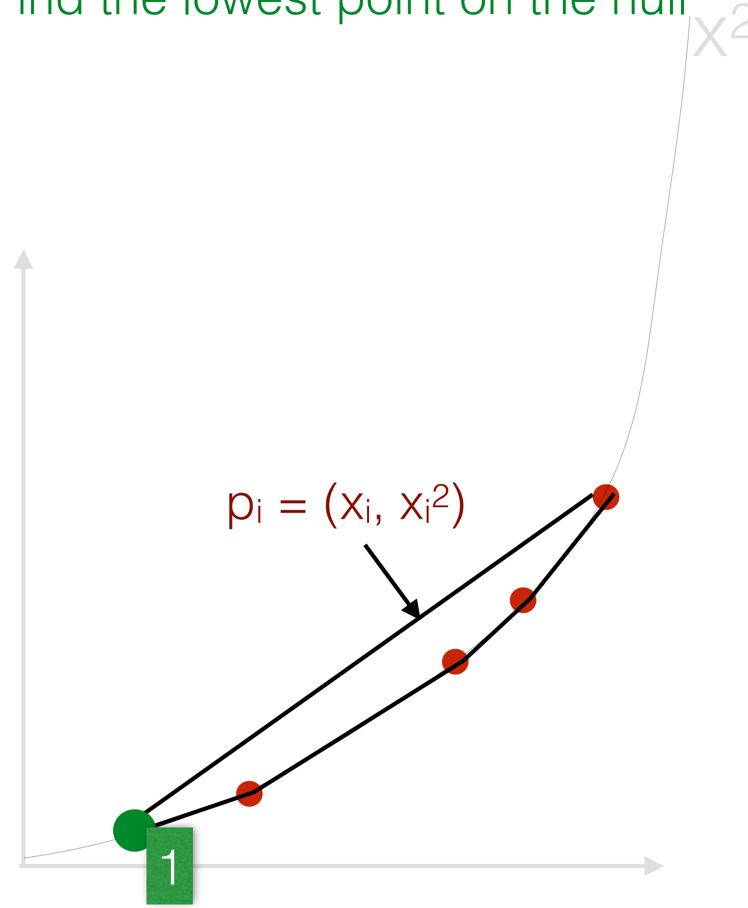


Sorting via ConvexHull

- Let P: set of values x_1, x_2, …x_n. to sort

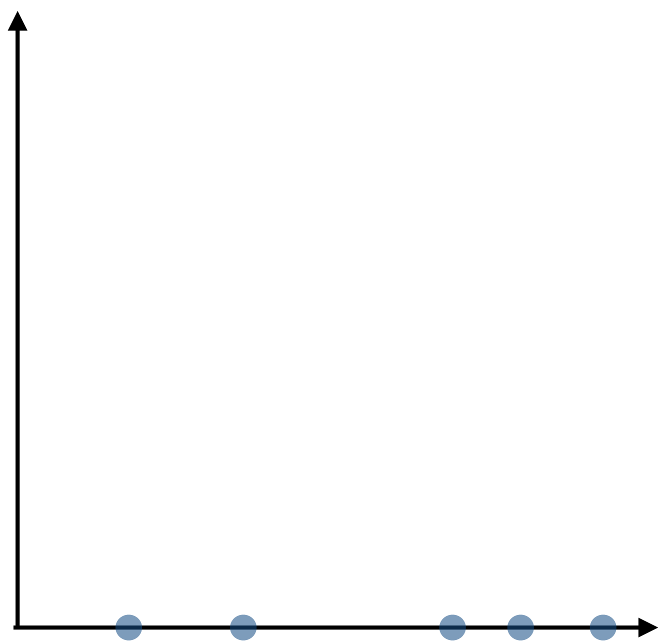


- Let P': set points { $p_i = (x_i, x_i^2)$}
- Run CH(P') to find their convex hull
- Find the lowest point on the hull

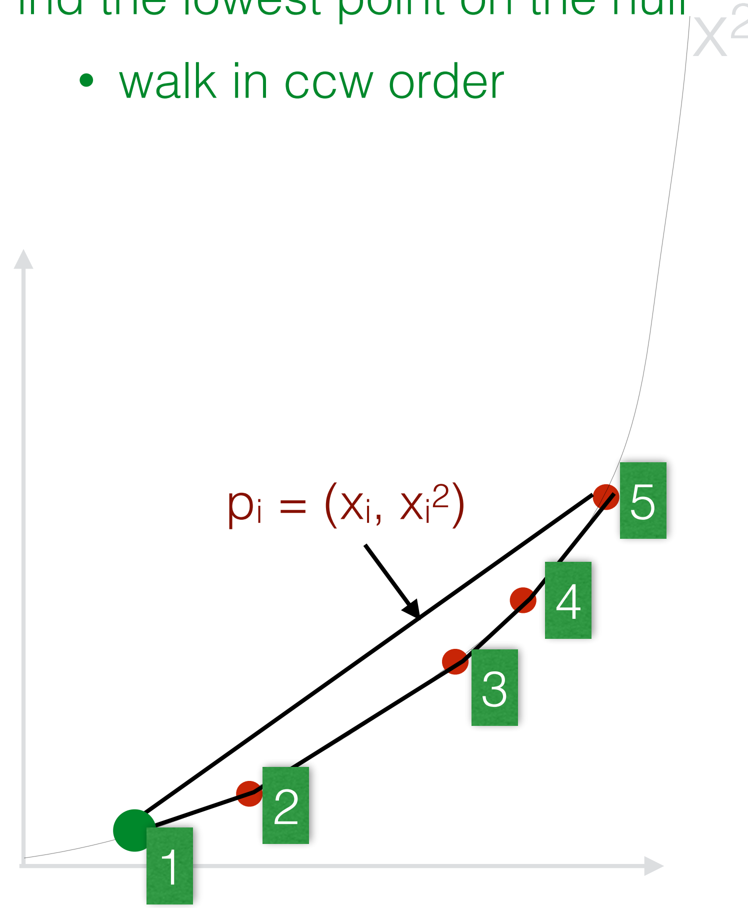


Sorting via ConvexHull

- Let P: set of values x_1, x_2, …x_n. to sort

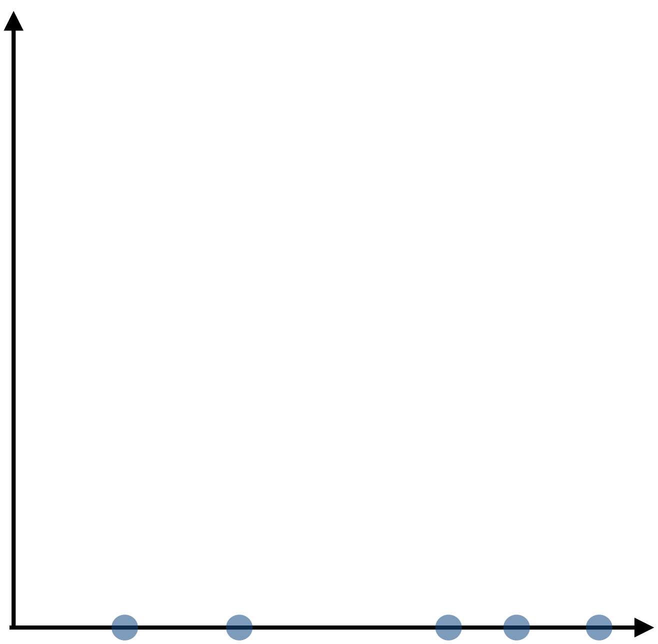


- Let P’: set points { $p_i = (x_i, x_i^2)$}
- Run CH(P’) to find their convex hull
- Find the lowest point on the hull
  - walk in ccw order



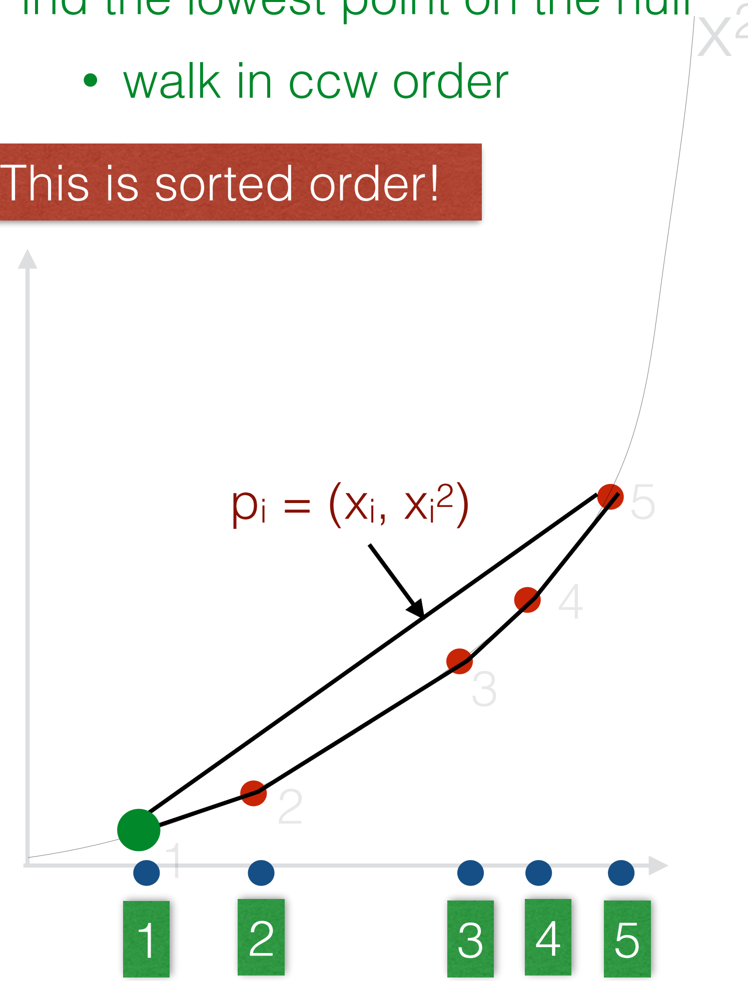
Sorting via ConvexHull

- Let P: set of values $x_1, x_2, \ldots x_n$ to sort



- Let P': set points $\{ p_i = (x_i, x_i^2)\}$
- Run CH(P') to find their convex hull
- Find the lowest point on the hull
  - walk in ccw order

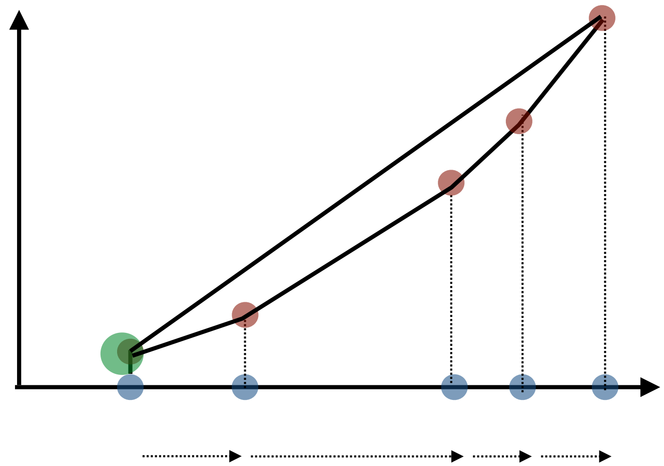
This is sorted order!



Sorting $\leq$ Convex hull

Sorting via ConvexHull

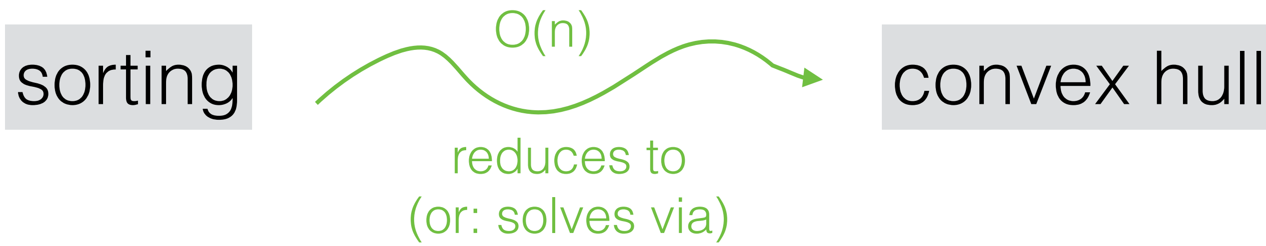
- Input: set of points $x_1, x_2, \ldots x_n$
  - Create a set of 2D points (x_i, x_i^2).
  - Run the CH algorithm to construct their convex hull.
  - Find the lowest point on the hull, and walk from in ccw order. This is sorted order!



Analysis: We can sort in O(CH(n)) + O(n)

- CH is an upper bound for sorting, or Sorting $\leq$ ConvexHull
- If we could find the CH faster than $\Theta(n \lg n)$, we could use it to sort faster than $\Theta(n \lg n)$, which is impossible!

Summary



sorting is $\Omega(n \lg n)$

CH must be $\Omega(n \lg n)$

Sorting reduces to CH

- 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.
- 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 (quite recently) that a convex hull algorithm, even if it does not produce the boundary in order, still needs $\Omega(n \lg n)$ in the worst case

Convex hull: summary

Naive	$O(n^3)$
Gift wrapping	$O(h \cdot n)$
Quickhull	$O(n^2)$
Graham scan	$O(n \lg n)$
Andrew monotone chain	$O(n \lg n)$

Can we do better than $\Theta(n \lg n)$ worst case?

No

- Yes, Graham scan is the ultimate CH algorithm but…
  - not output sensitive
  - does not extend to 3D
- The (re)search continues

An incremental algorithm for CH

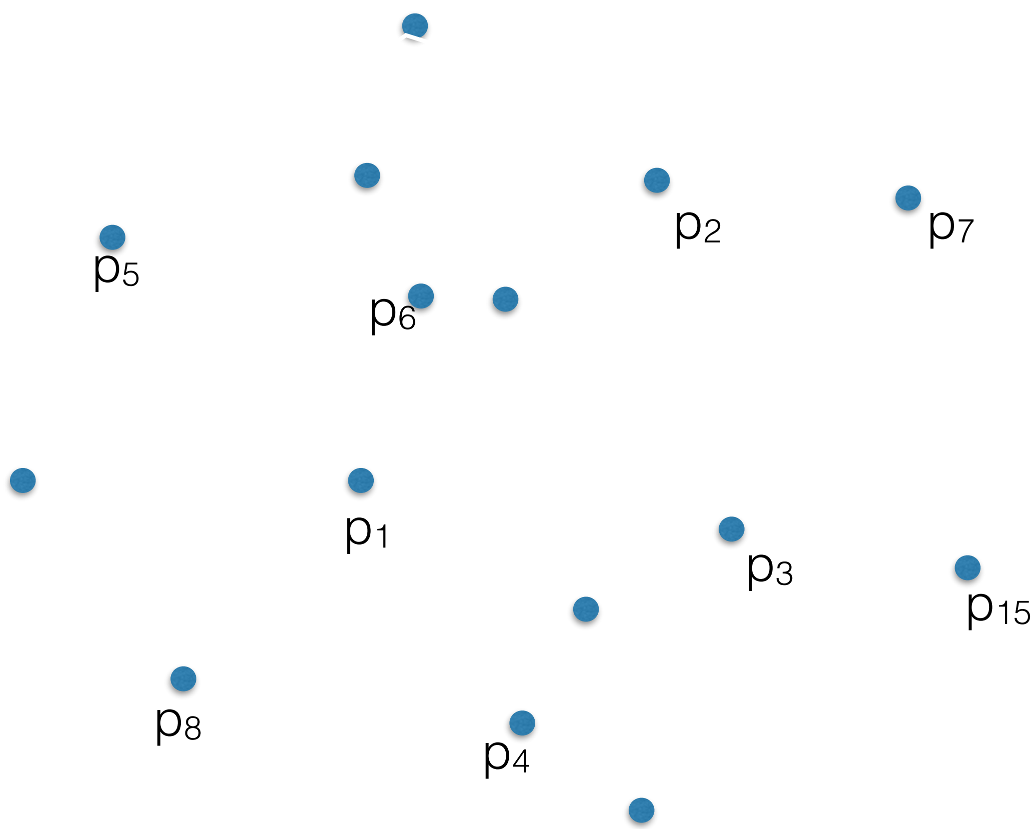
Incremental algorithms

- Idea: Traverse the points one at a time and solve the problem for the points seen so far

- Incremental Algorithm
    - initialize solution S
    - for i=1 to n
        - //S represents solution of $p_1 \ldots\ldots p_{i-1}$
        - update S to represent solution of $p_1 \ldots\ldots p_{i-1}$ p_i

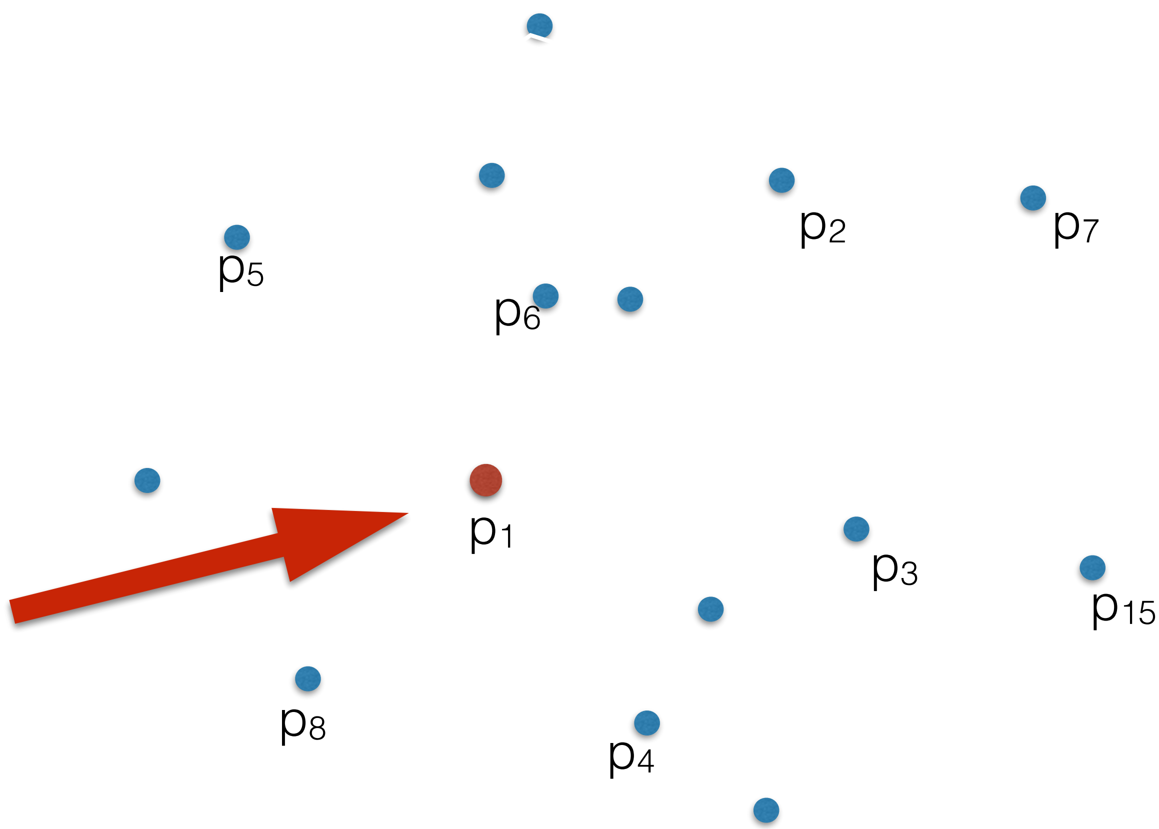
Incremental algo for CH

- CH = {}
- for i=1 to n
  - //CH represents the CH of $p_1..p_{i-1}$
  - update CH to represent the CH of $p_1..p_i$



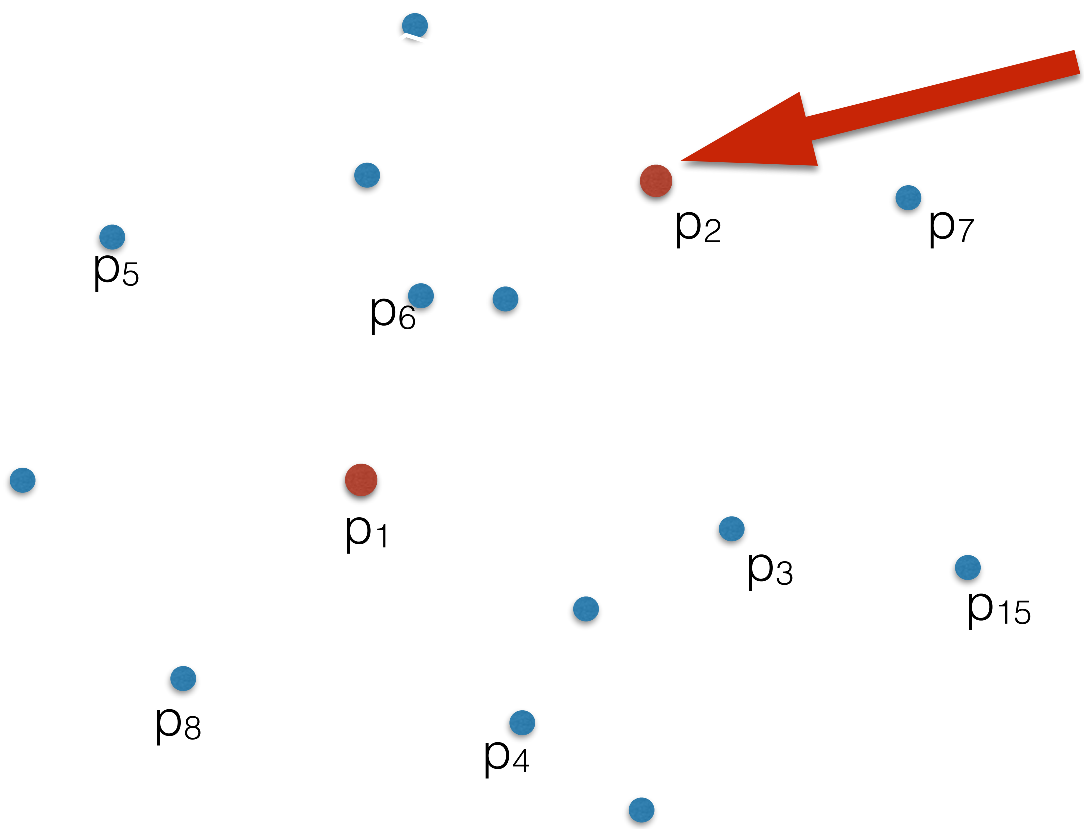
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



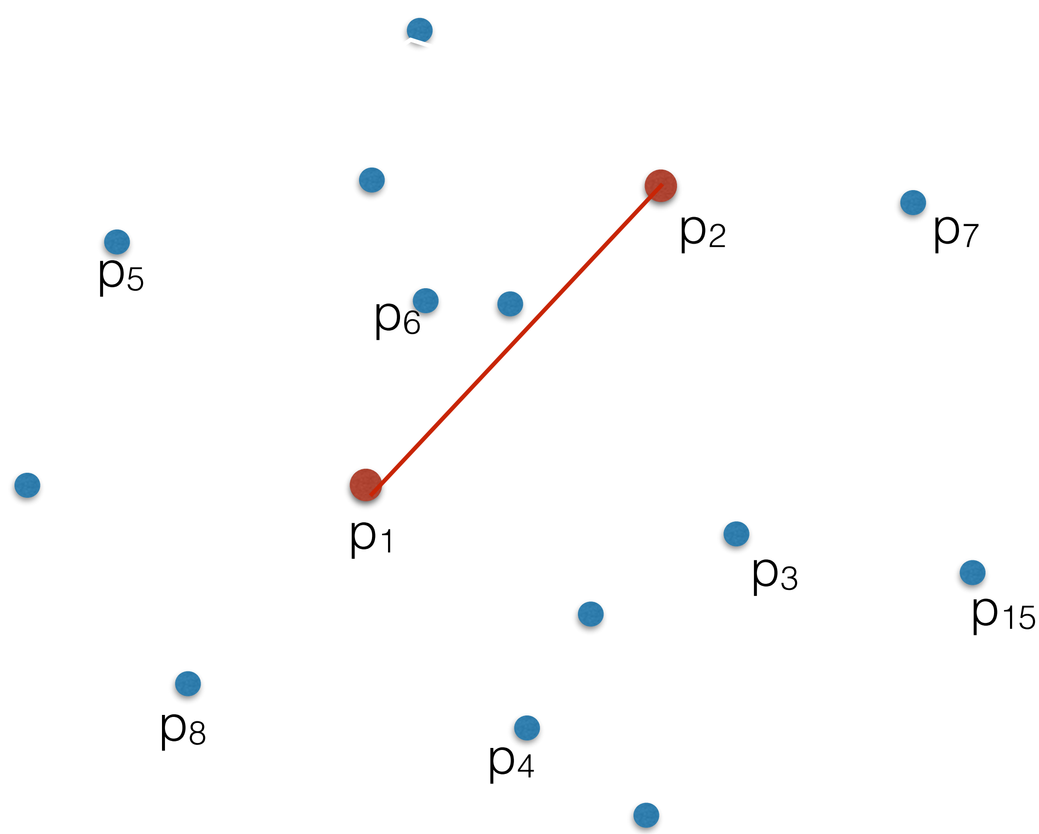
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



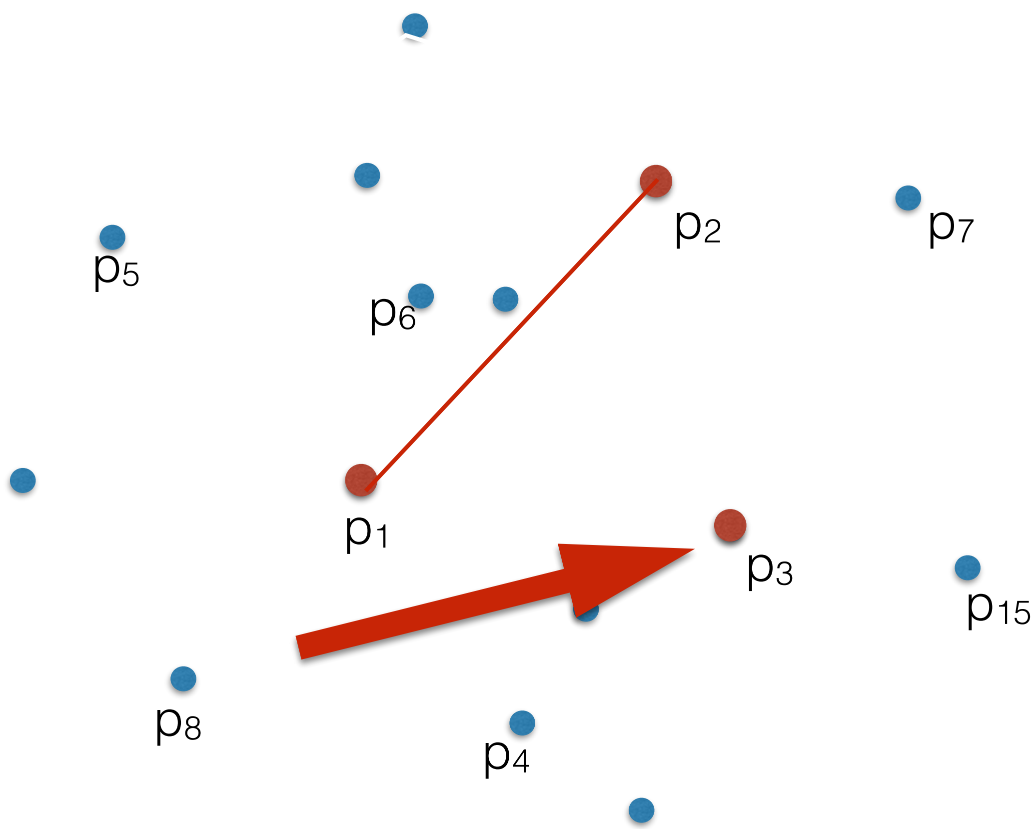
Incremental algo for CH

- CH = {}
- for i=1 to n
  - //CH represents the CH of $p_1..p_{i-1}$
  - update CH to represent the CH of $p_1..p_i$



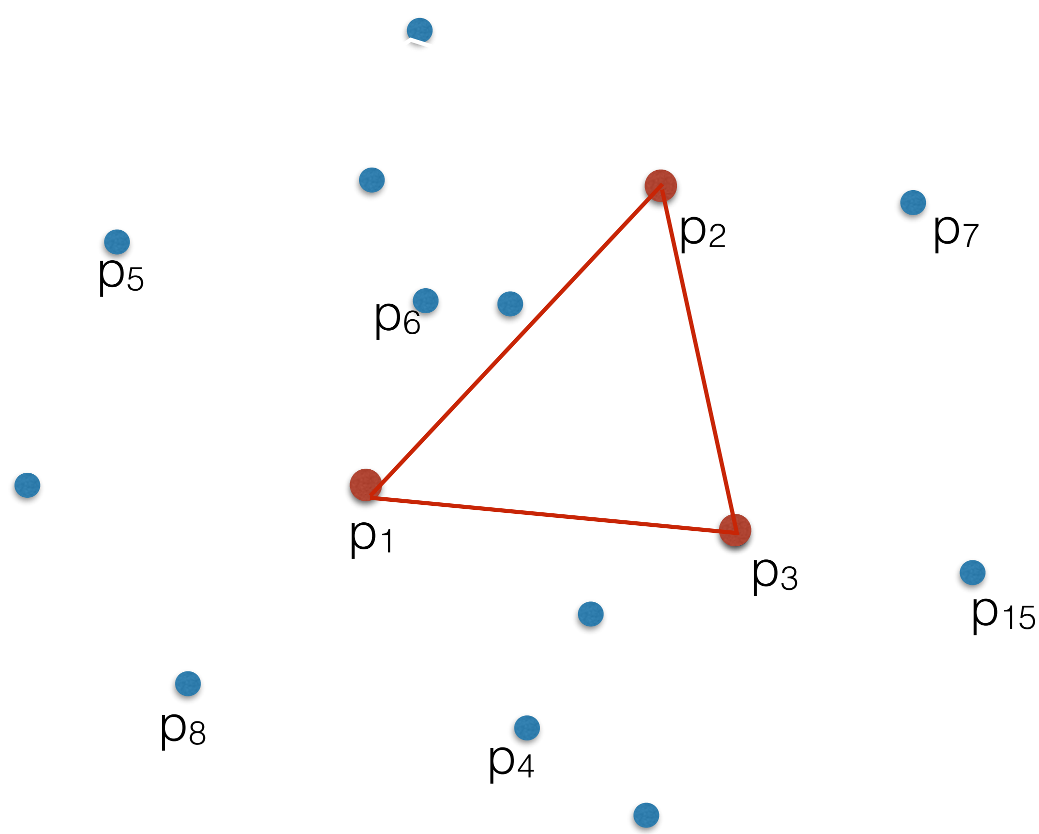
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



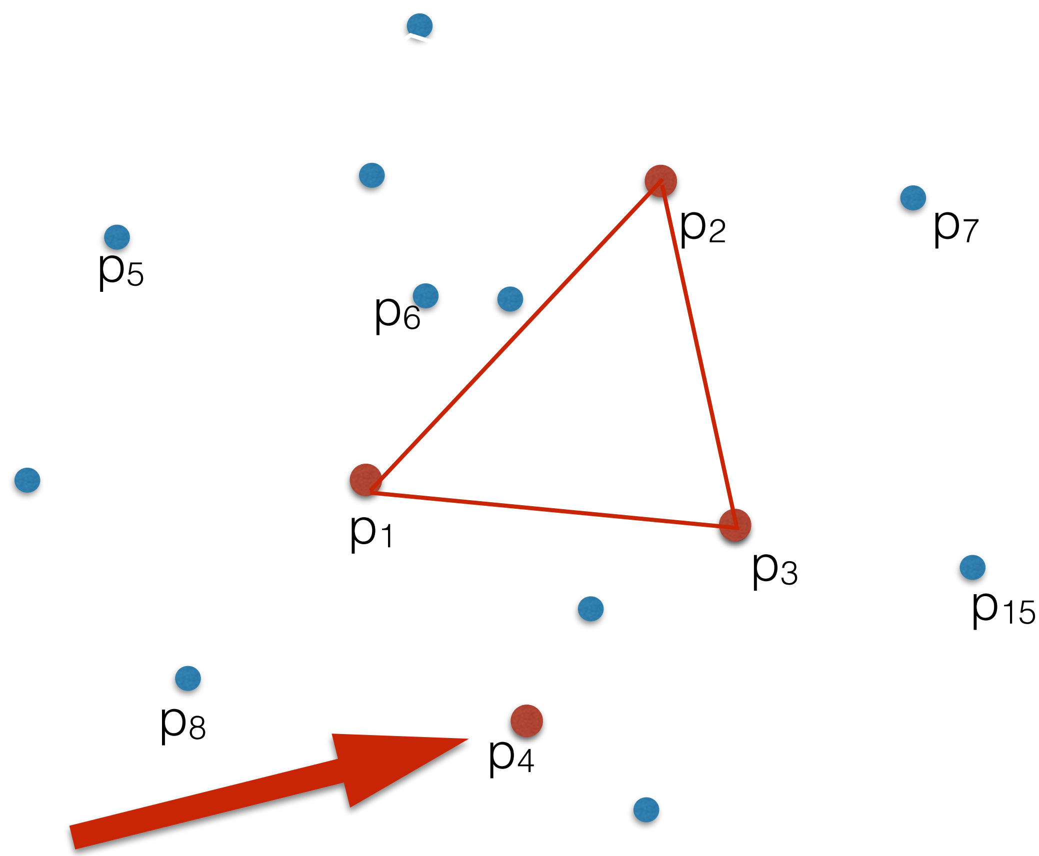
Incremental algo for CH

- CH = {}
- for i=1 to n
  - //CH represents the CH of $p_1..p_{i-1}$
  - update CH to represent the CH of $p_1..p_i$



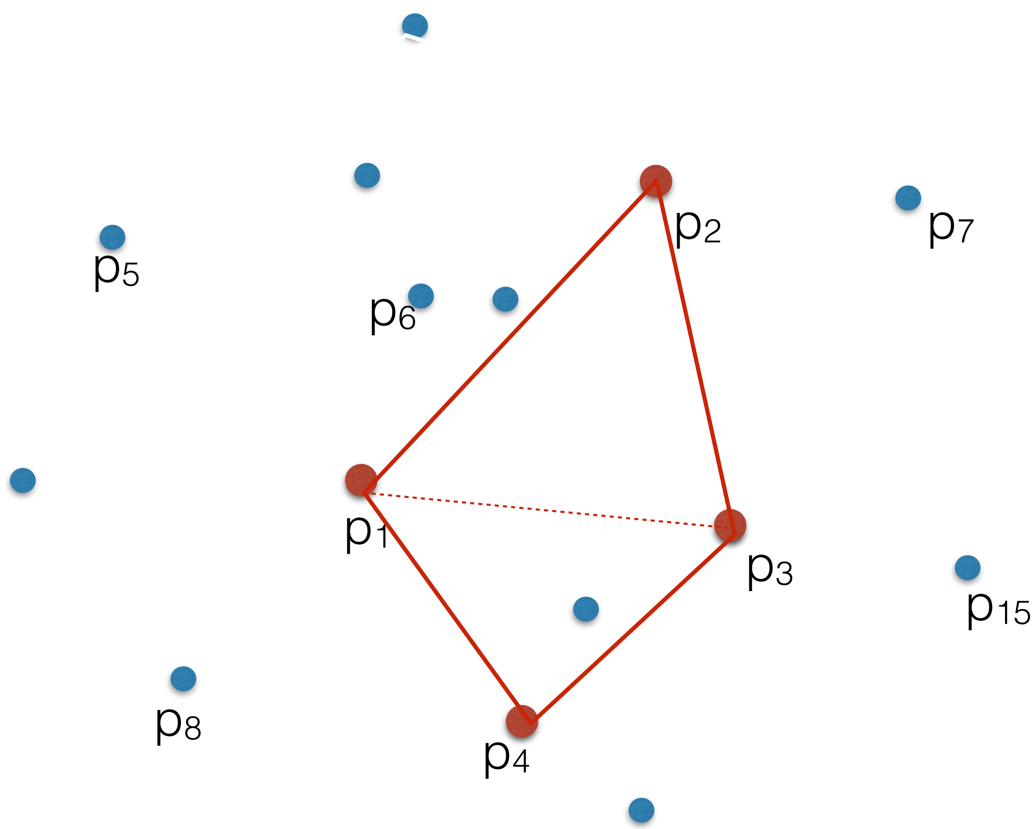
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



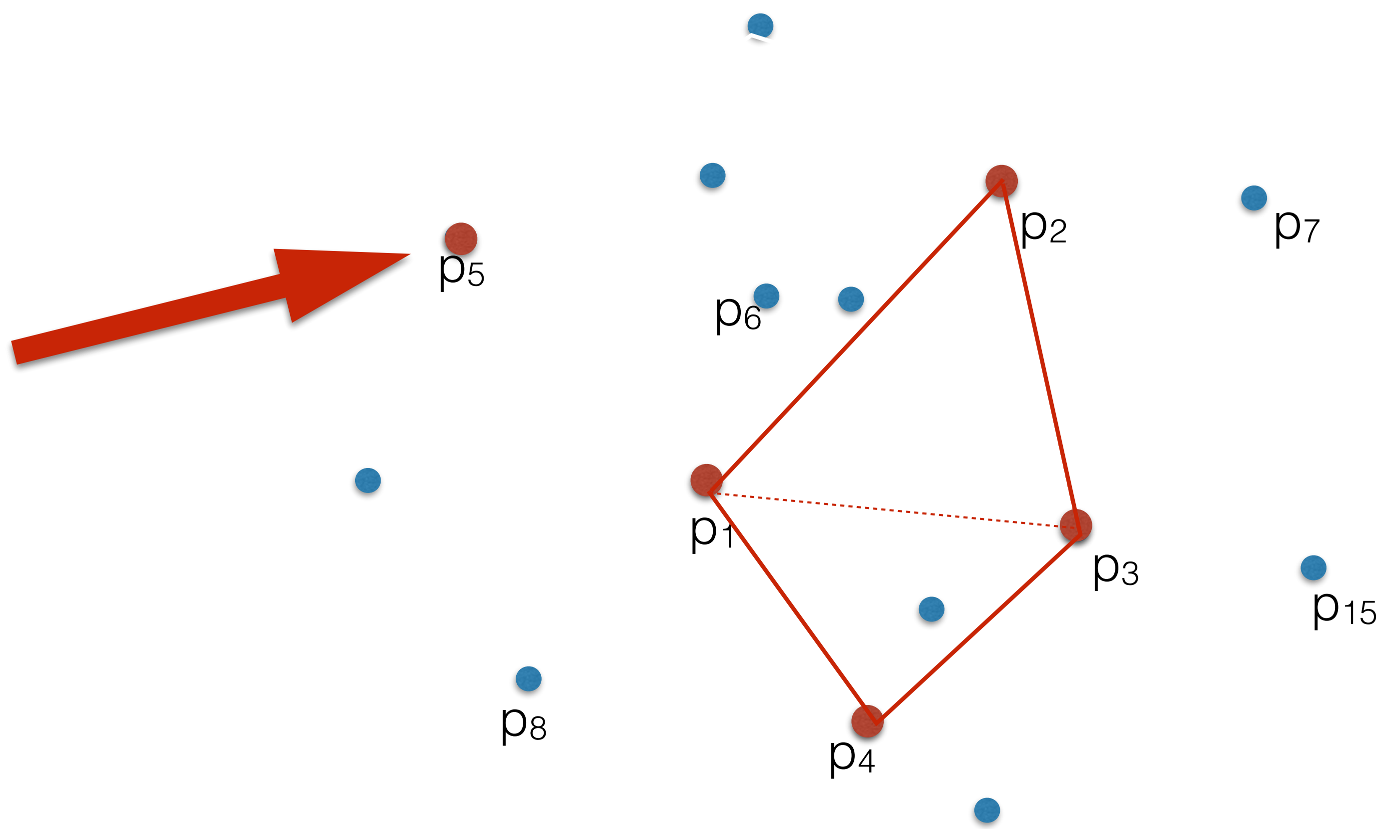
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



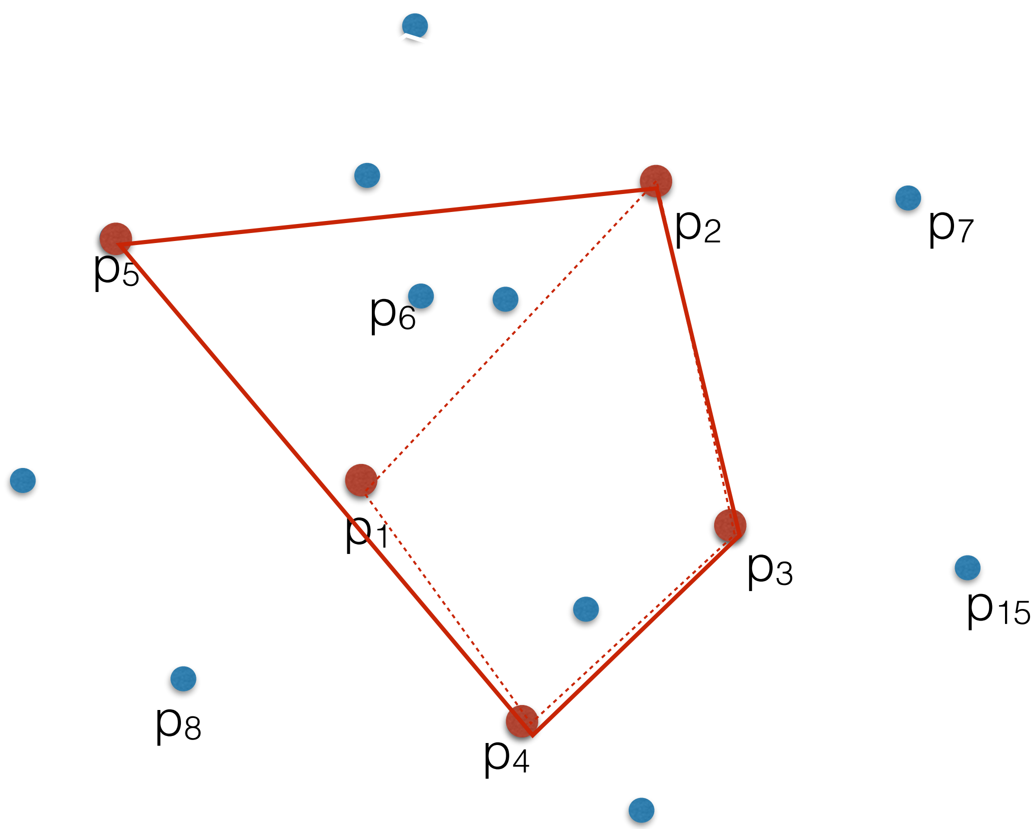
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



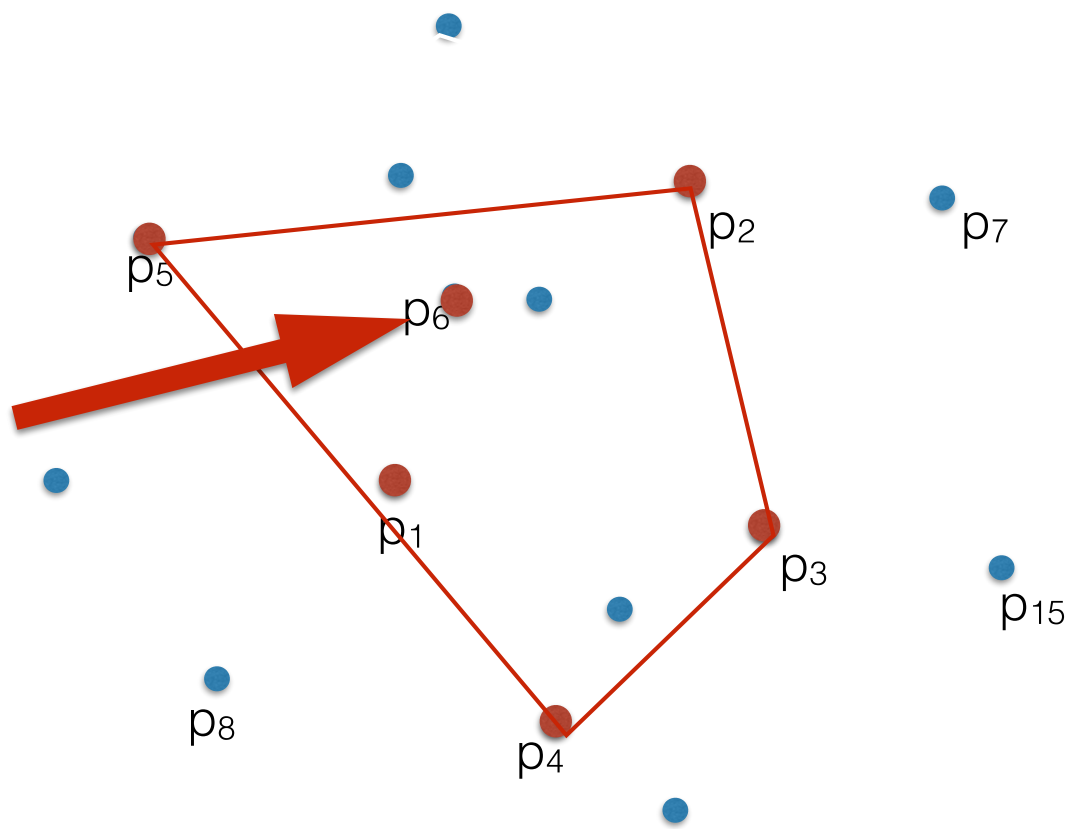
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



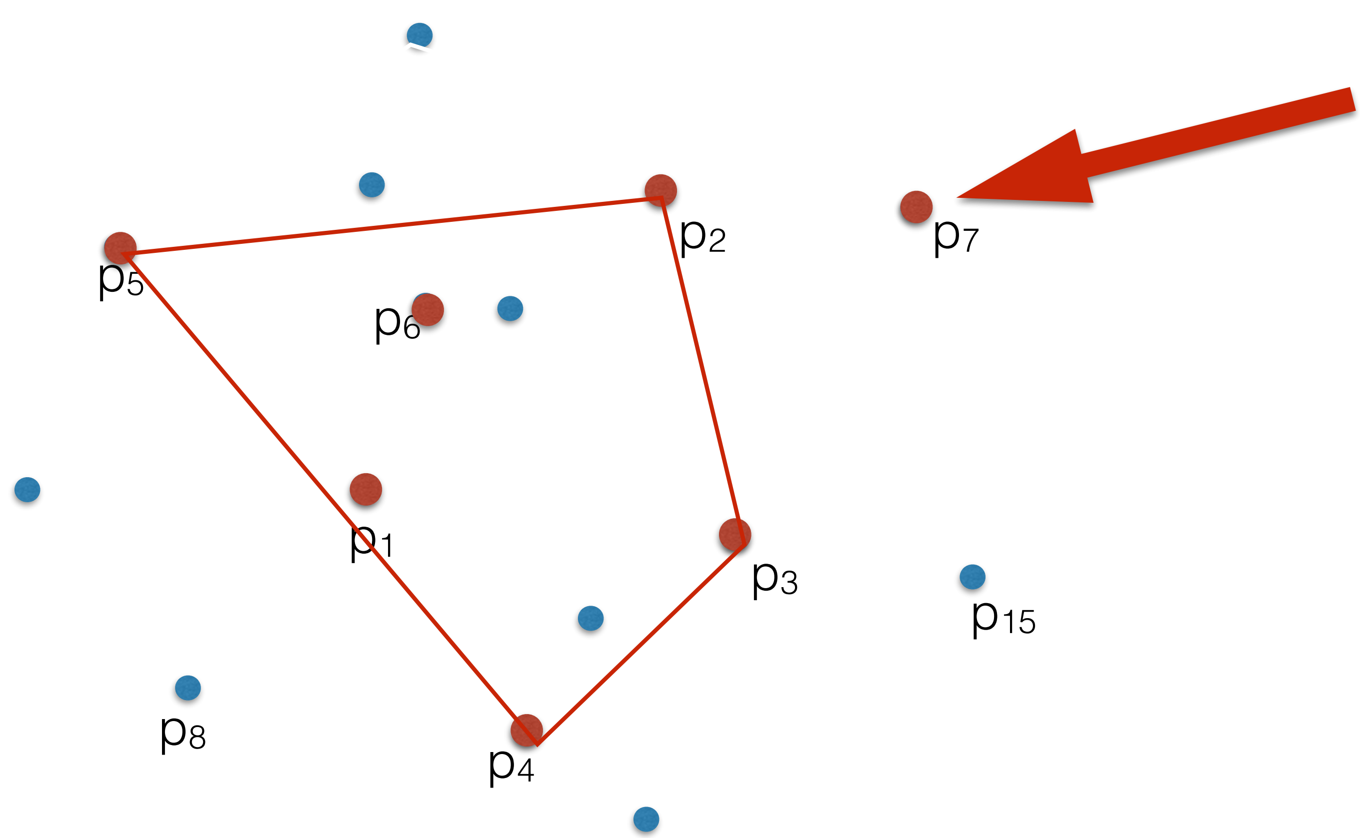
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



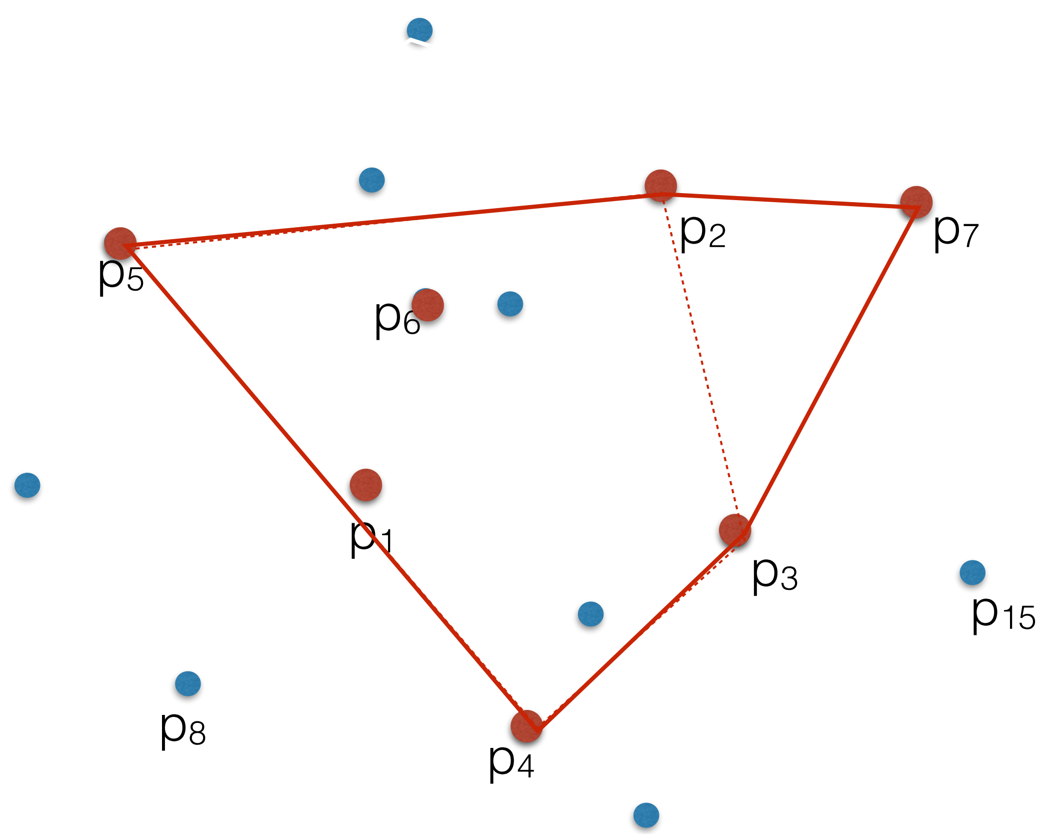
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



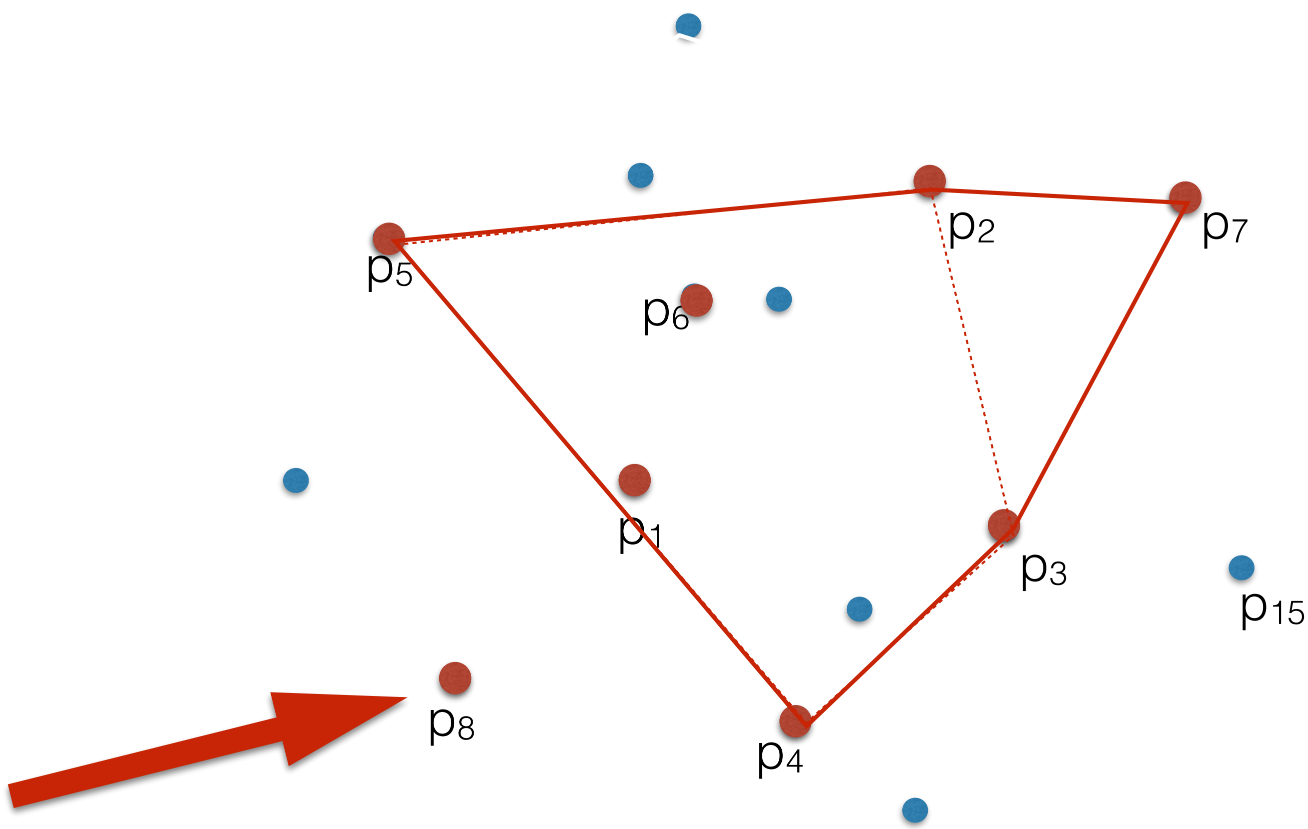
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



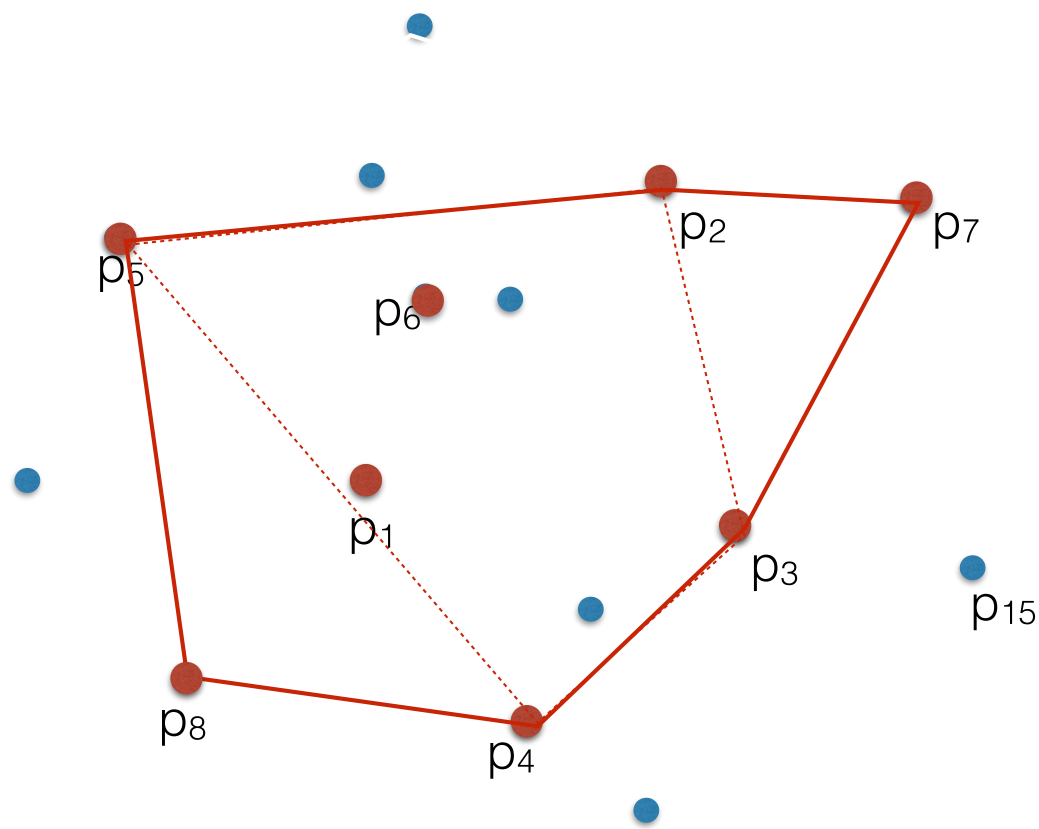
Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



Incremental algo for CH

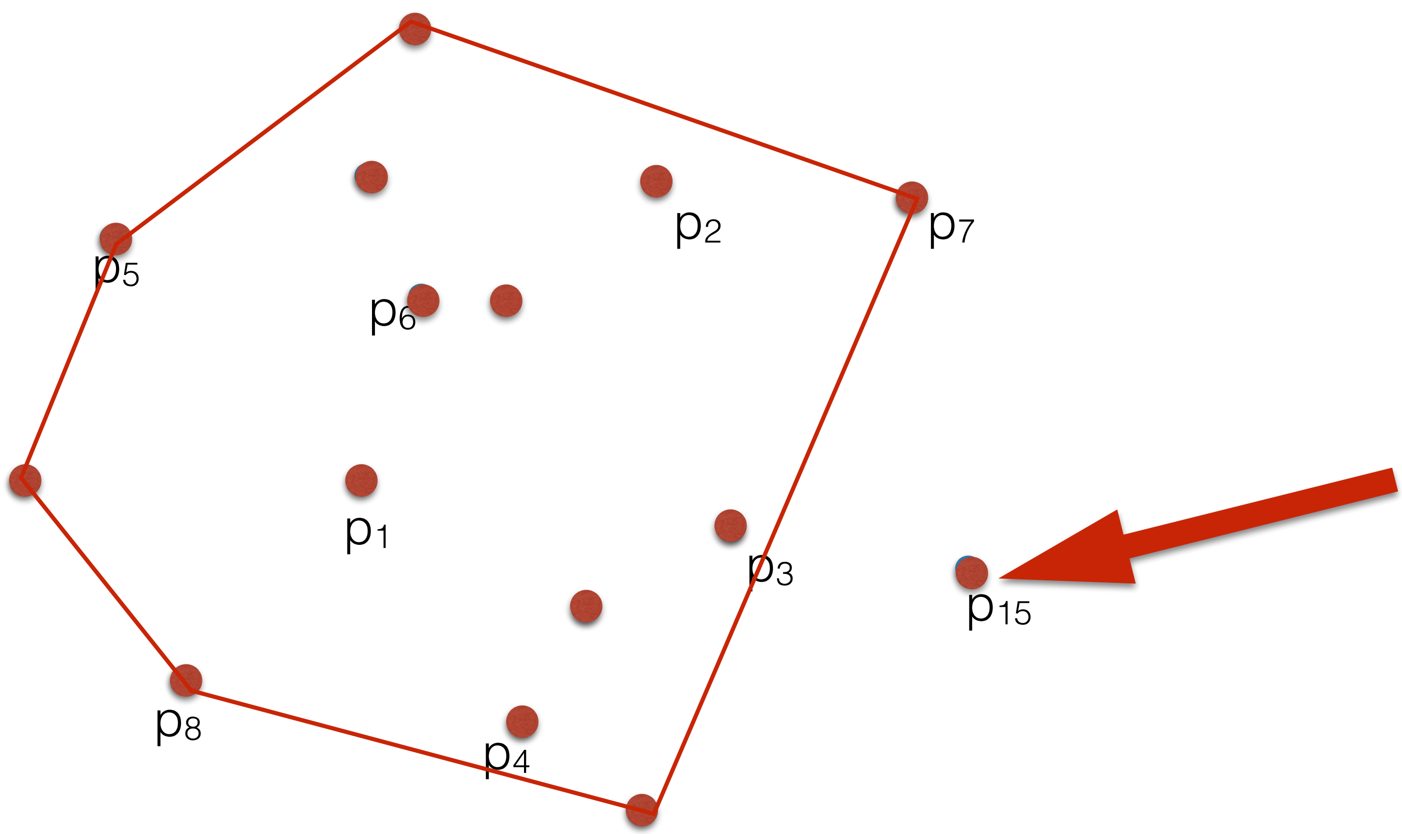
- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$



Incremental algo for CH

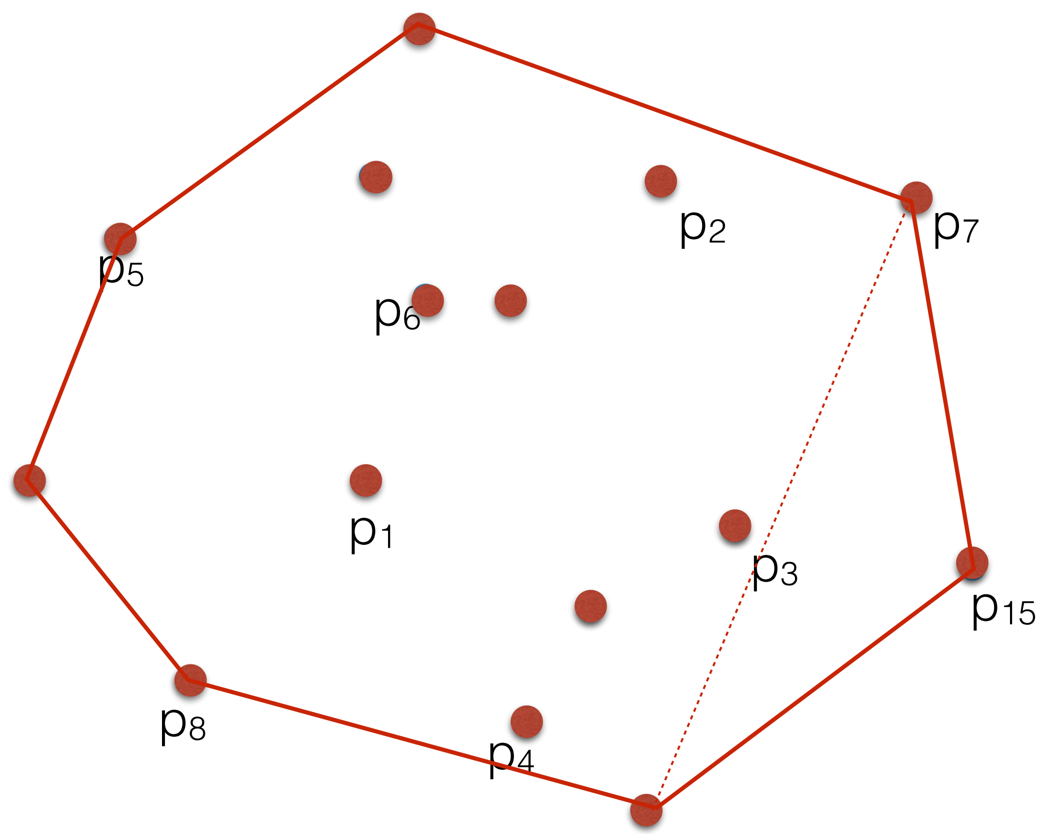
- CH = {}
- for i=1 to n
  - //CH represents the CH of $p_1..p_{i-1}$
  - update CH to represent the CH of $p_1..p_i$

and so on



Incremental algo for CH

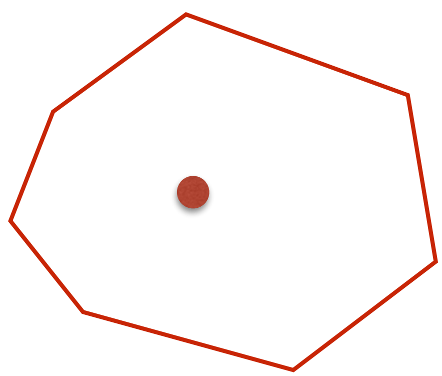
- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$

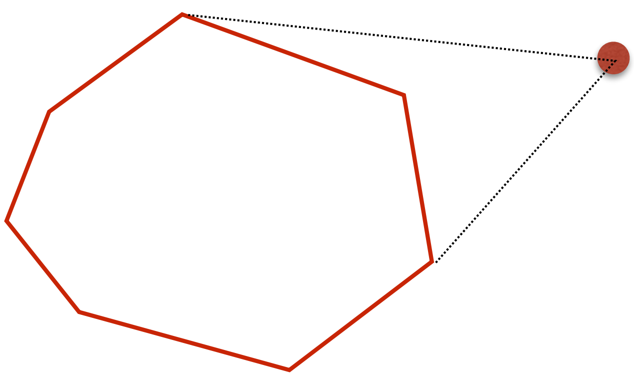


Incremental algo for CH

- CH = {}
- for i=1 to n
    - //CH represents the CH of $p_1..p_{i-1}$
    - update CH to represent the CH of $p_1..p_i$

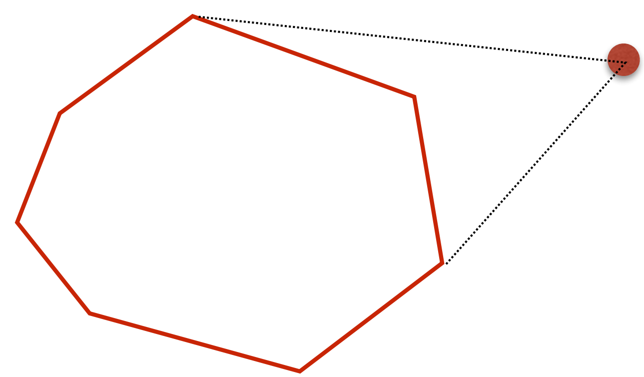
- The basic operation is adding a point to a convex polygon
    - CASE 1: p is in polygon
    - CASE 2: p outside polygon





Incremental algo for CH

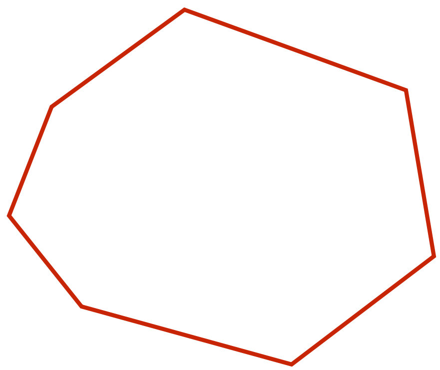
- Issues to solve
  - What's a good representation for a (convex) polygon?
  - We need a point-in-convex-polygon test
  - How to handle CASE 2 ?



Representing a polygon

A polygon is represented as a list of vertices in boundary order.

(the convention is counter-clockwise order)

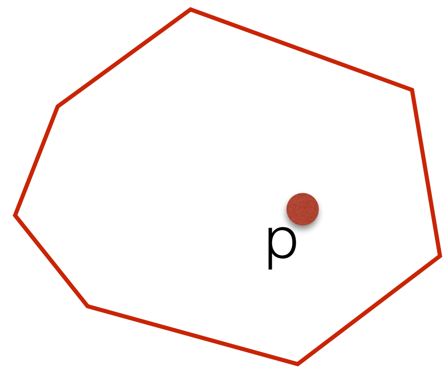


```
typedef struct _polygon{
      int k; //number of vertices
      Point* vertices; //the vertices, ccw in boundary order
  } Polygon;

  or

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

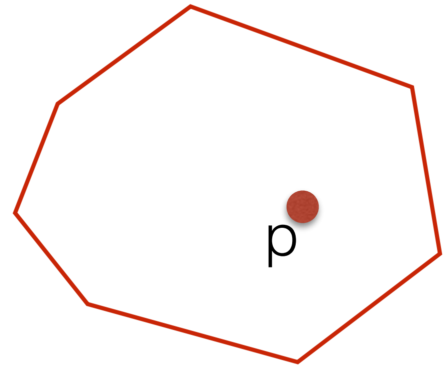
Point in convex polygon




```
//return TRUE iff p on the boundary or inside H; H is convex a polygon
bool point_in_polygon(point p, polygon H)
```

What has to be true in order for p to be inside?

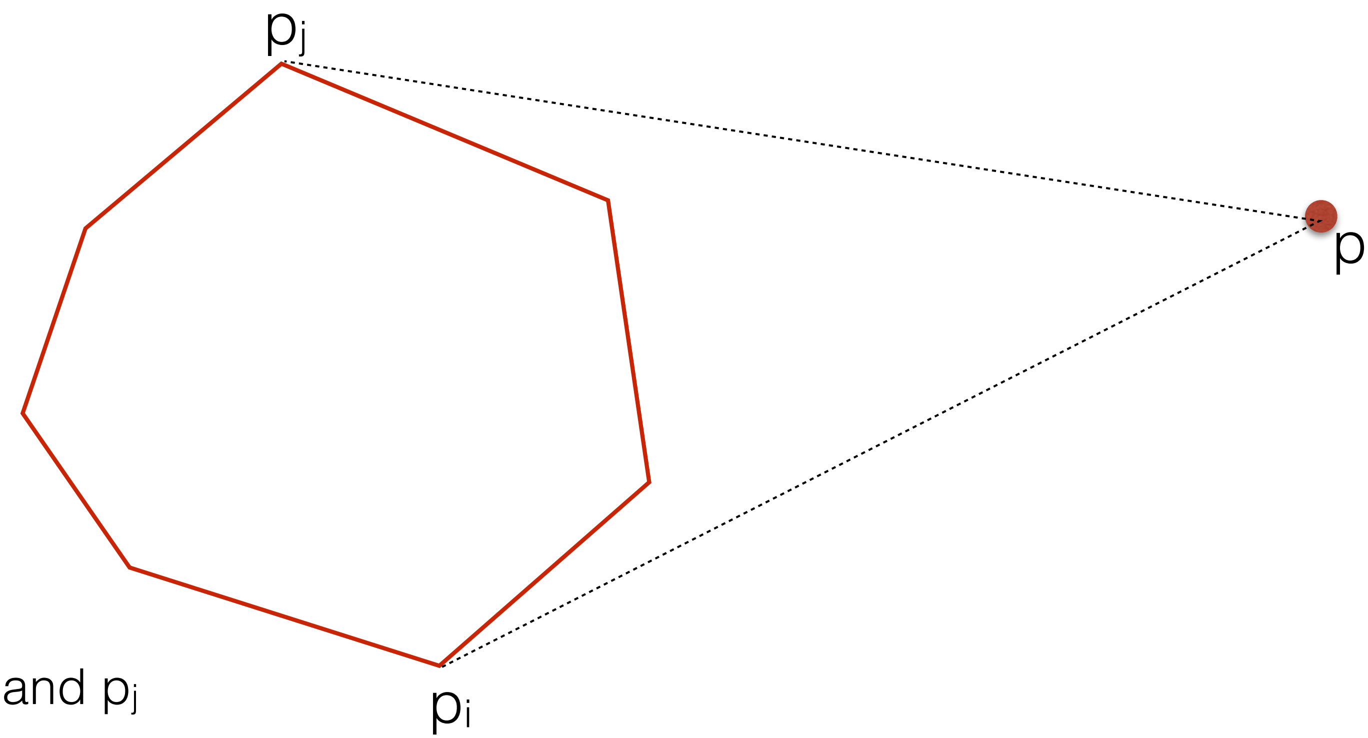
Point in convex polygon




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

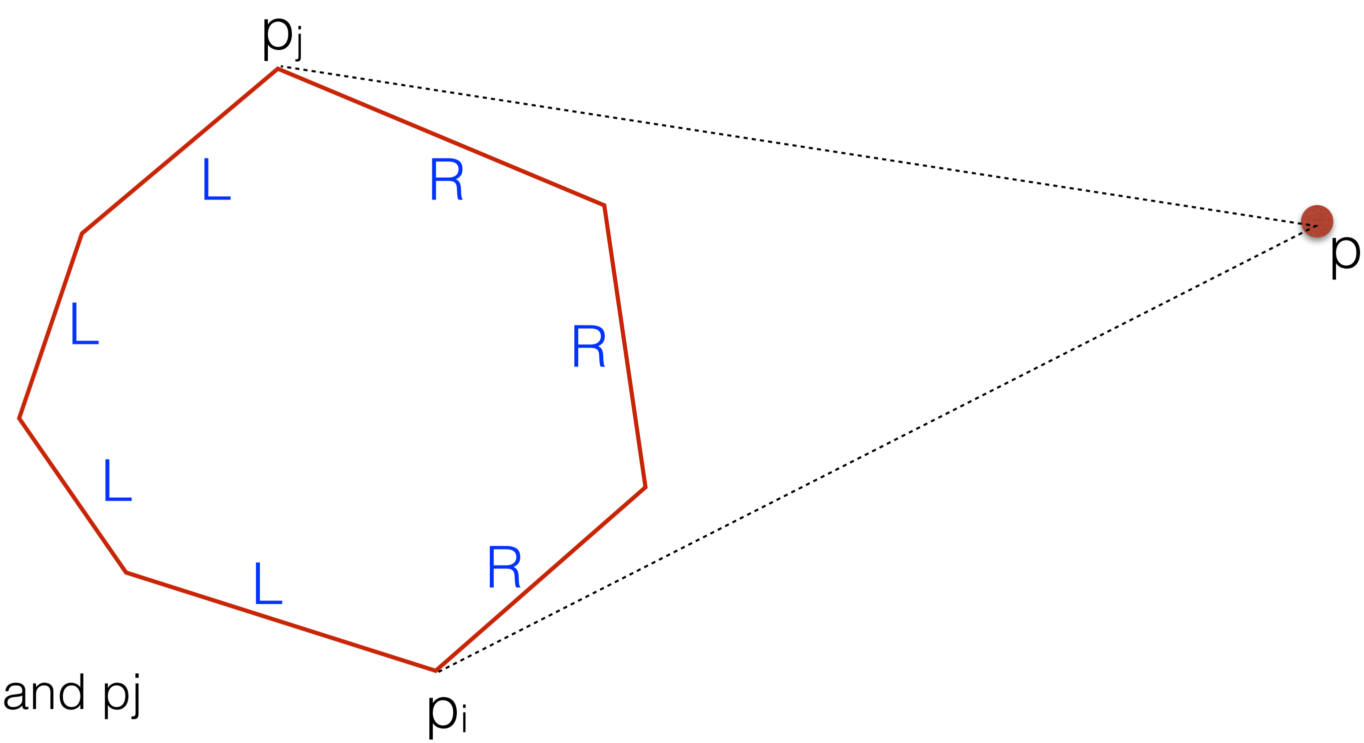
Case 2:



We want to find p_i and p_j

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

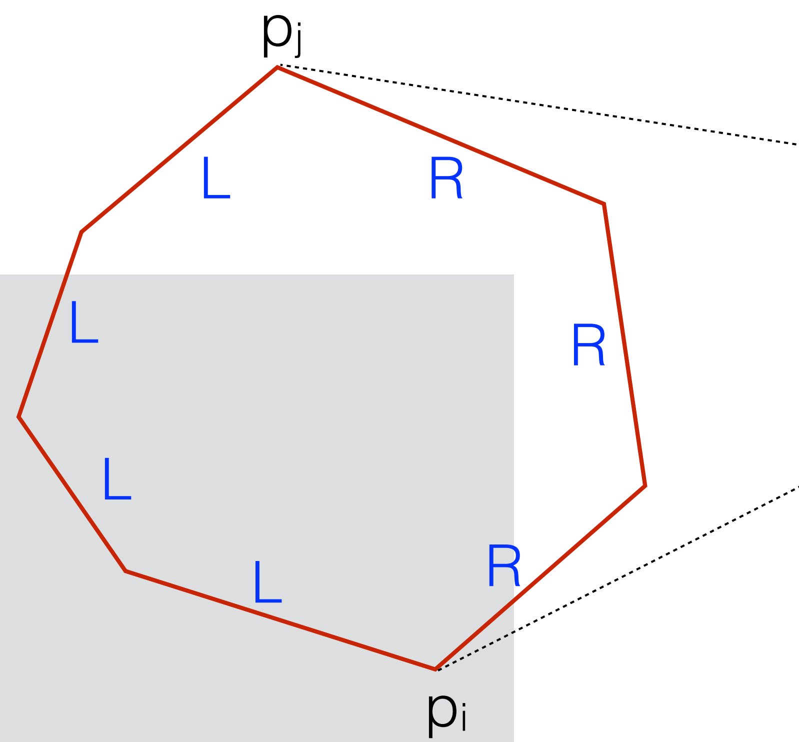
Case 2:



We want to find pi and pj

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

Finding tangent points



Input: point p outside H

polygon H = $[p_0, p_1, \ldots, p_{k-1}]$ convex

- for i=0 to k-1 do
  - prev = ((i == 0)? k-1: i-1);
  - next = (i==k-1)? 0; i+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

Putting it all together

Incremental CH

- H = [p1, p2, p3]
- for i=4 to n do
    - //add p_i to H
    - if point_in_polygon(pi, H)
        - //do nothing
    - else
        - find p_k the tangent point where orientation changes from L to R
        - find p_j the tangent point where orientation changes from R to L
        - delete the part from p_k to p_j in H (note: p_k not necessarily before p_j in the vertex array of H. view H as wrapping around)

Incremental CH

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

    ← O(i)
    - delete the part from p_k to p_j in H (note: p_k not necessarily before p_j in the vertex array of H. view H as wrapping around)

Analysis: $\sum_i O(i) = \Theta(n^2)$

Incremental CH, improved

- Pre-sort the points by their x-coordinates and add them in this order. Then
    - point p_i is to the right of p_{i-1}, so it will be **outside** $CH(p_1, p_2, \ldots, p_{i-1})$
    - No need to check if p_i is inside the CH!

- pre-sort the points by their x-coordinates. Initialize H = [p1, p2, p3]
- for i=4 to n do
    - find p_k the tangent point where orientation changes from L to R $\longleftarrow O(i)$
    - find p_j the tangent point where orientation changes from R to L
    - delete the part from p_k to p_j in H

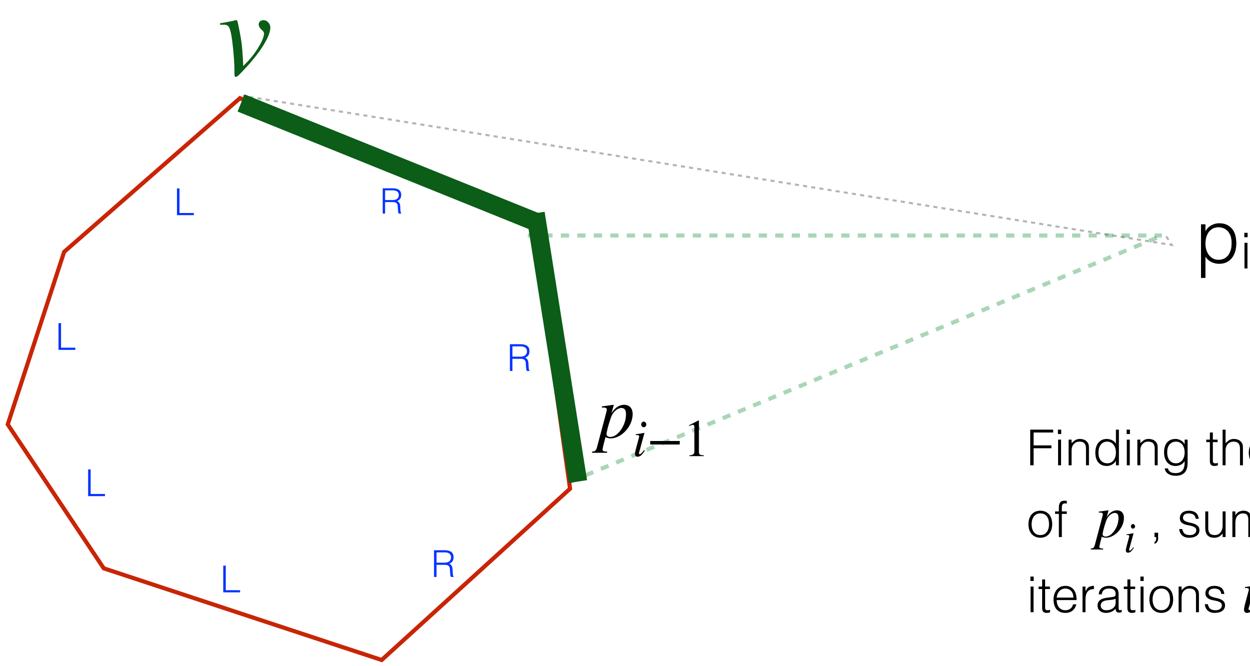
Analysis: however, this is still $\sum_i O(i) = \Theta(n^2)$

But, we can finesse finding the tangent to run in $O(n)$ total, overall all n points

Finding the UPPER tangent point of p_i to the hull H of $\{p_1, p_2, \ldots, p_{i-1}\}$

- find vertex p_{i-1} on H
- $v = p_{i-1}$
- while point p_i lies to the right of $(v, succ(v))$: $v = succ(v)$

//claim: v is the upper tangent point



Finding the upper tangent of p_i , summed over all iterations i, takes $O(n)$

Theorem: Incremental CH (in 2D) runs in $O(n \lg n)$ to sort the points followed by $O(n)$ to construct the convex hull.

A divide-and-conquer algorithm for CH

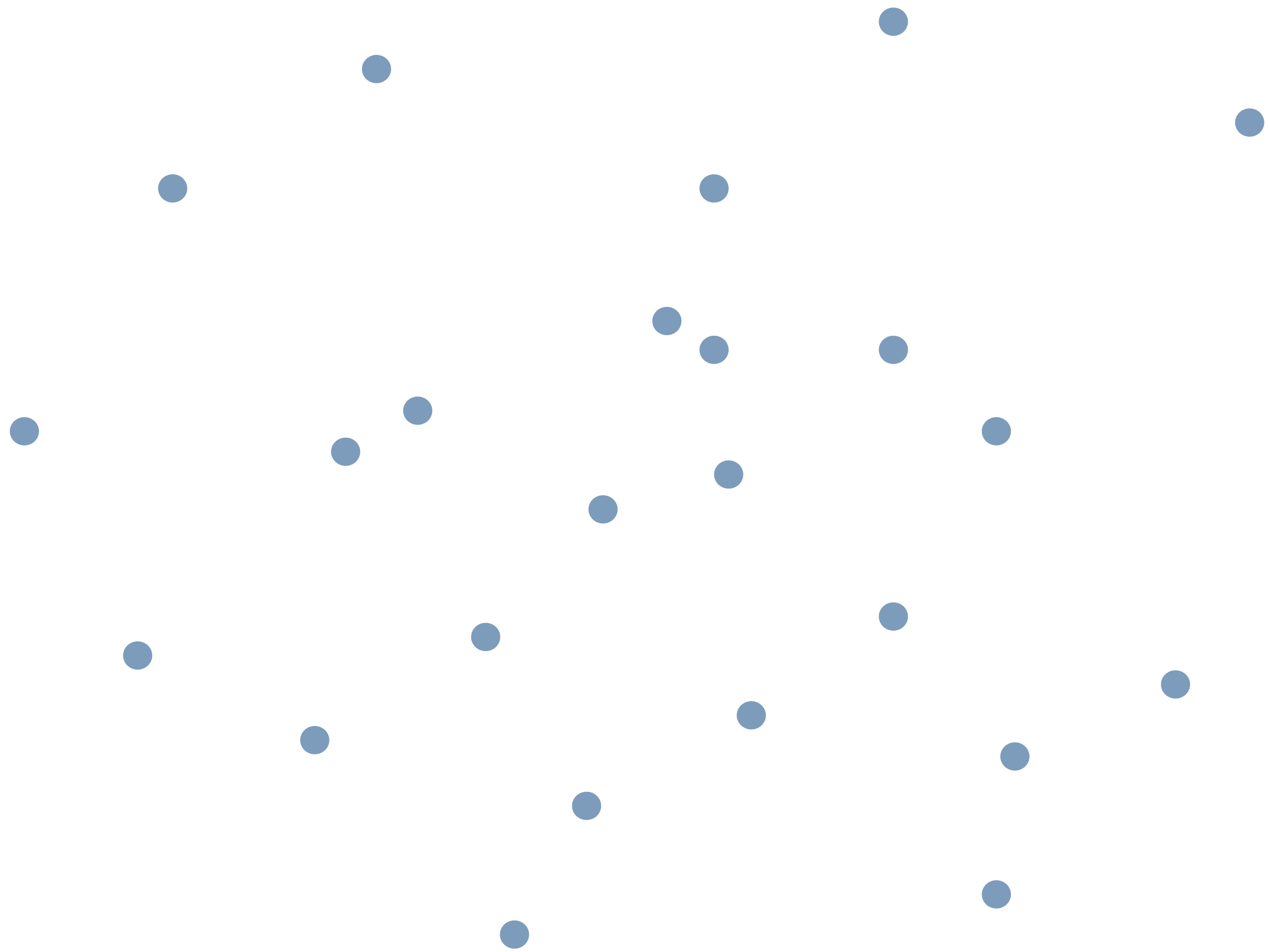
Divide-and-conquer framework

```
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
    result=figure_out_result_for_P _from_result1_and_result2
    return result
```

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

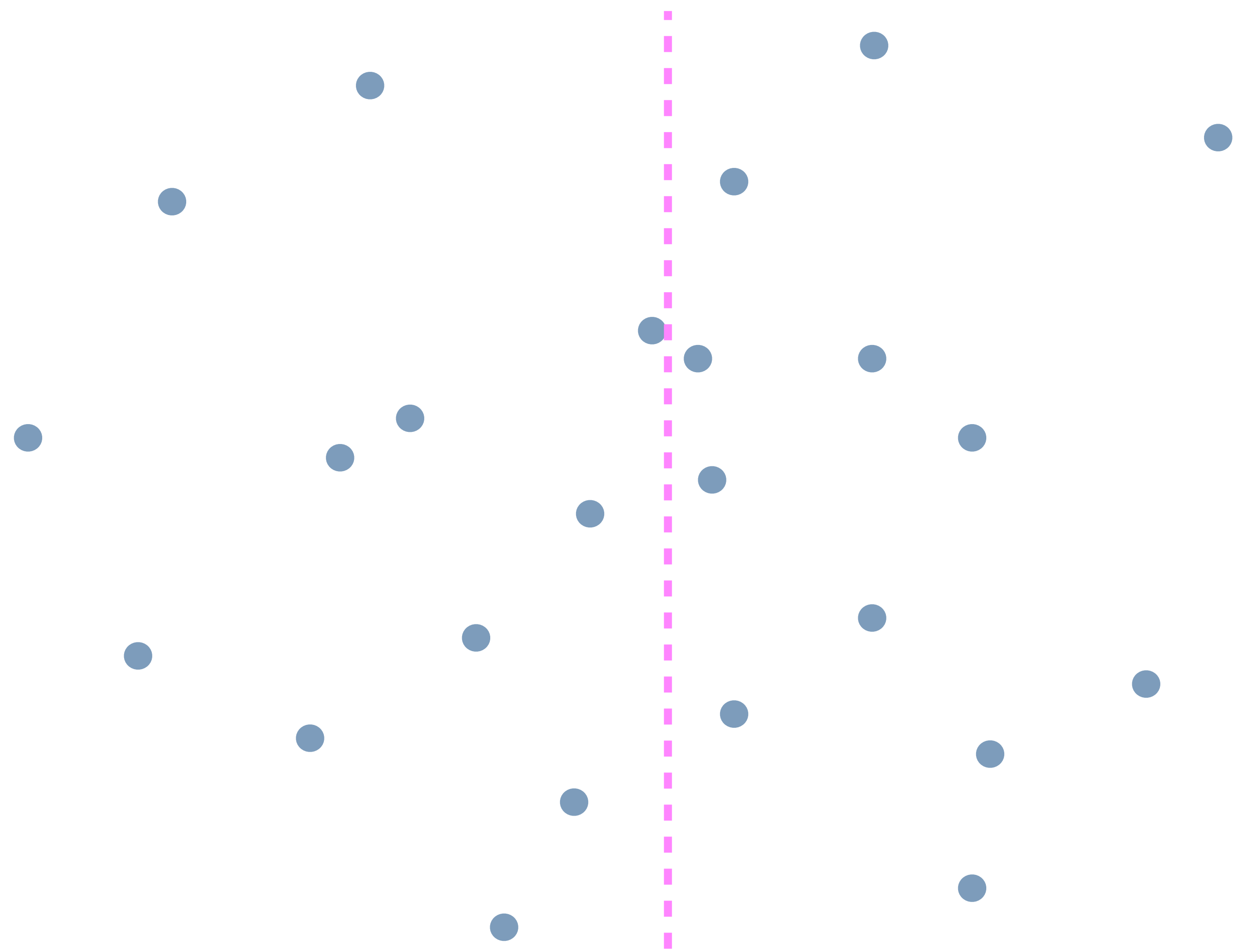
- if merge phase is $O(n)$: $T(n) = 2T(n/2) + O(n)$ => $O(n \lg n)$

CH via divide-and-conquer



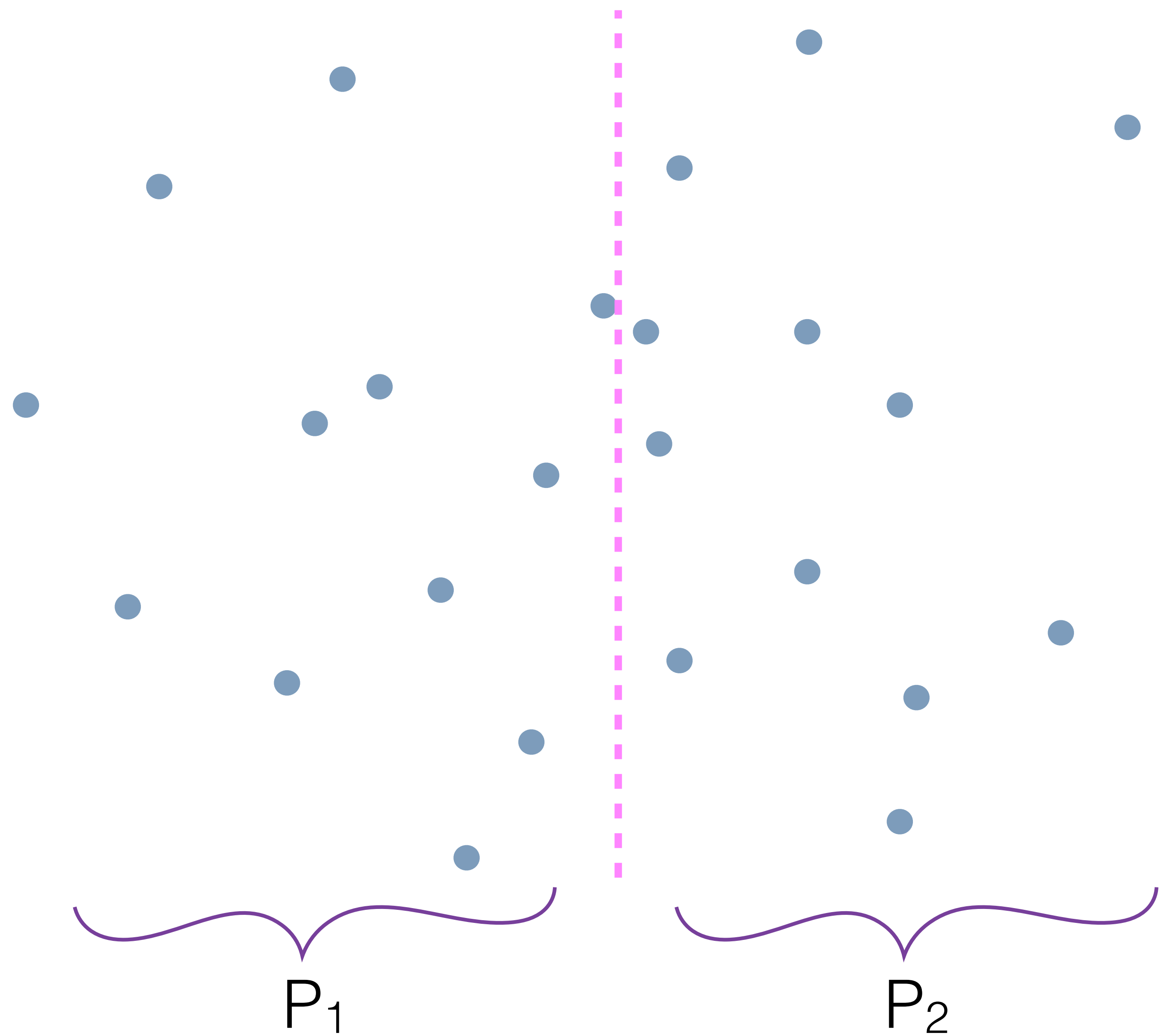
CH via divide-and-conquer

- find vertical line that splits P in half



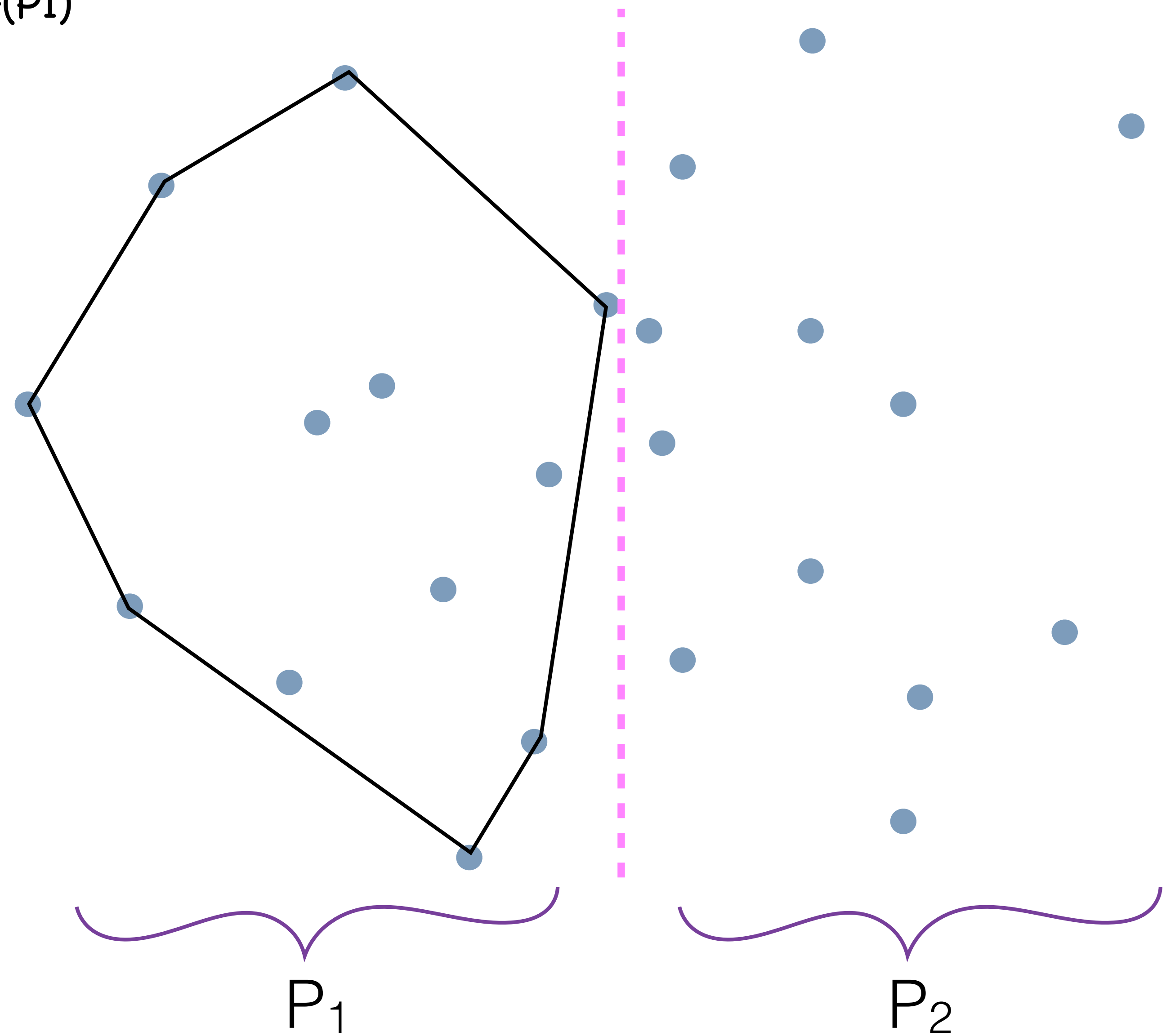
CH via divide-and-conquer

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



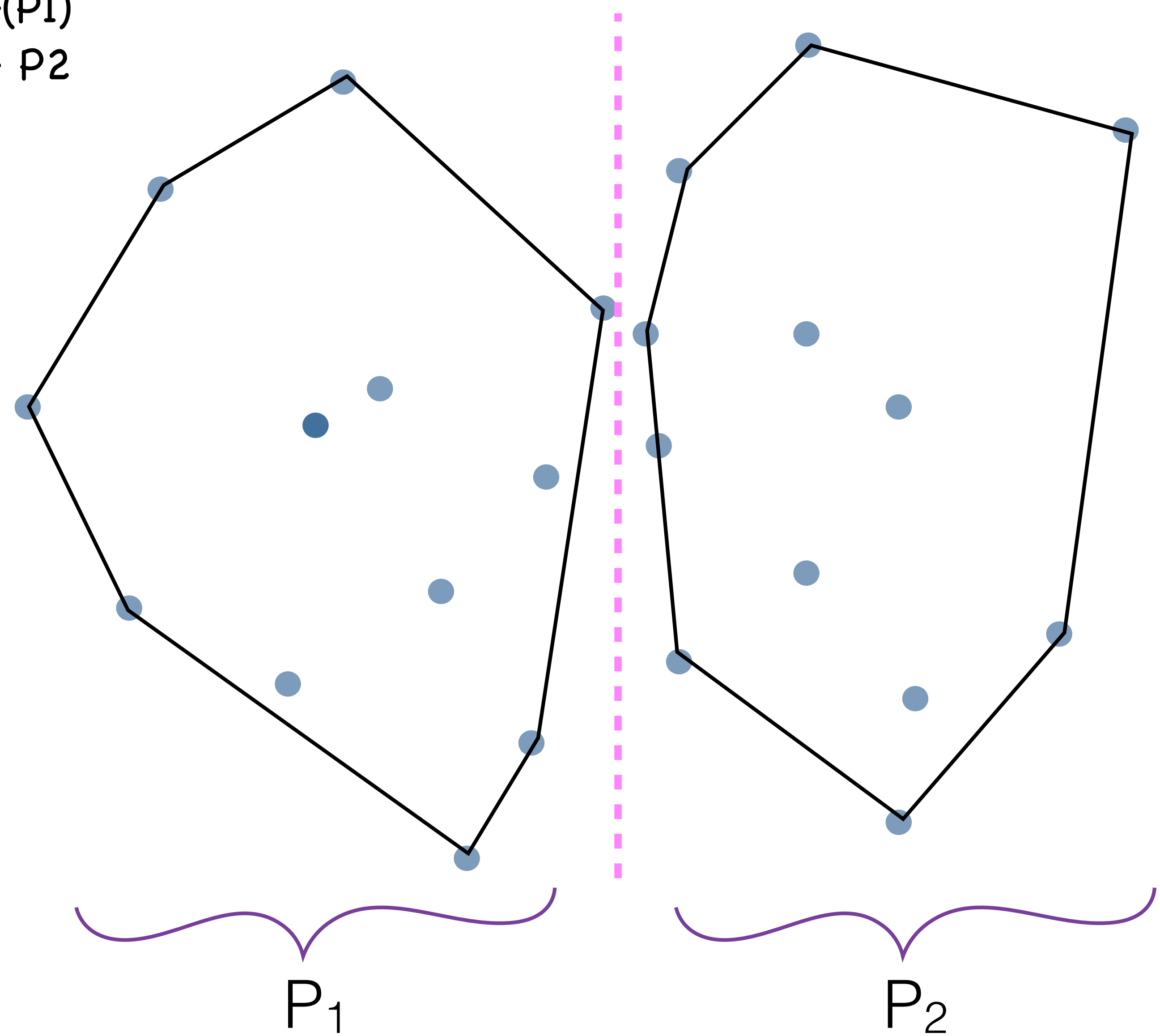
CH via divide-and-conquer

- 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

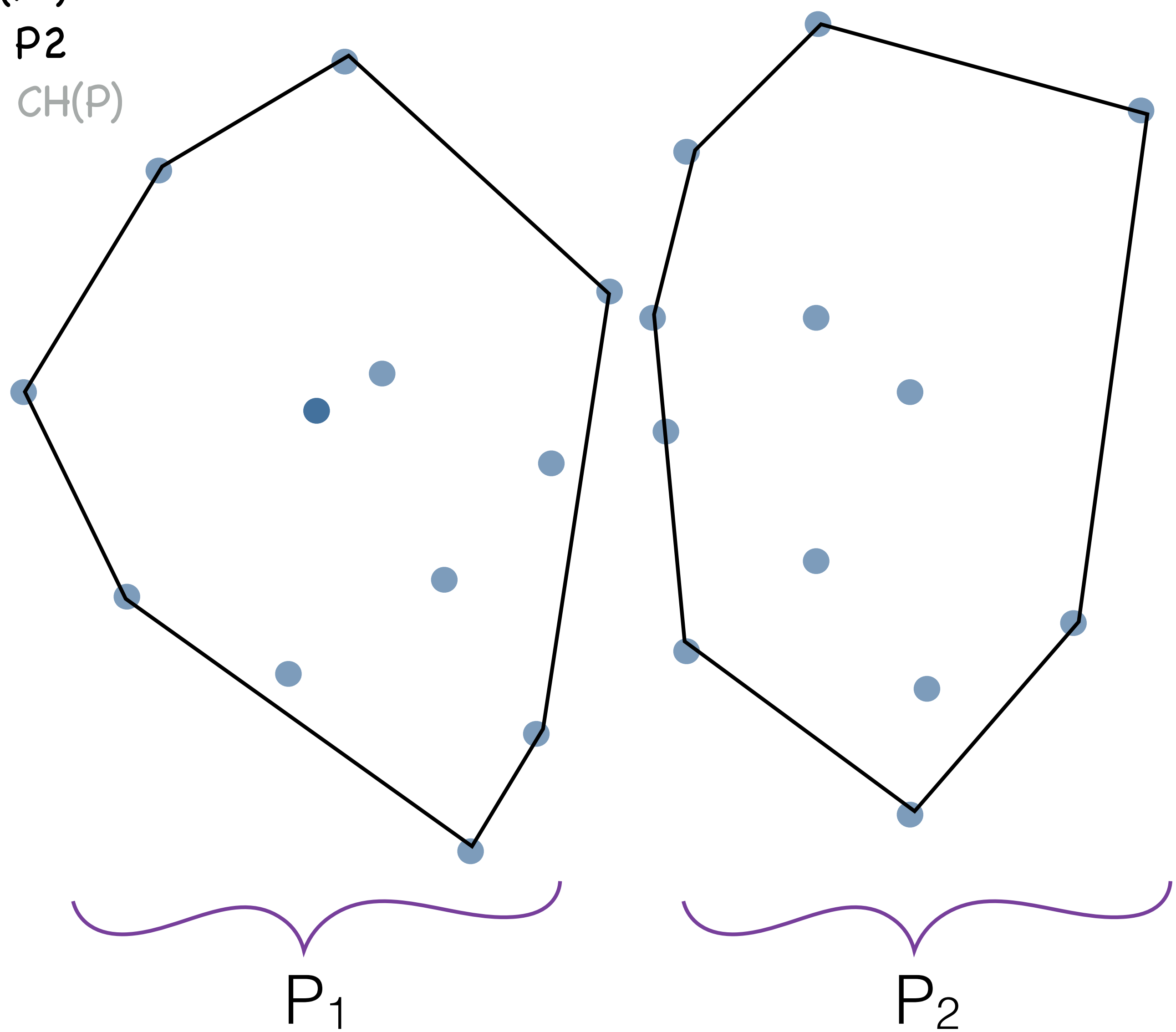
- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find CH(P1)
- recursively find CH P2



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

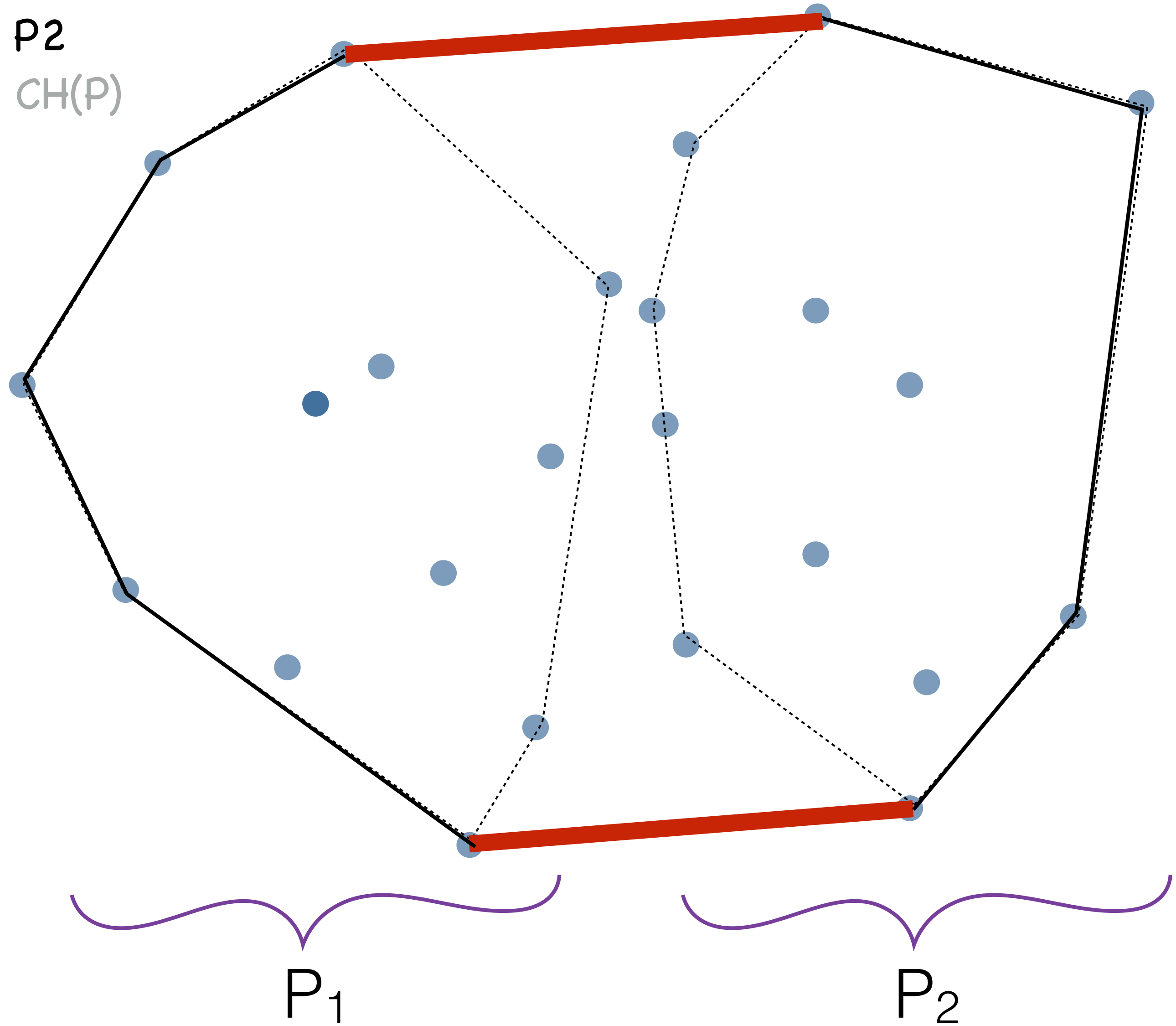
//now get somehow CH(P)



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

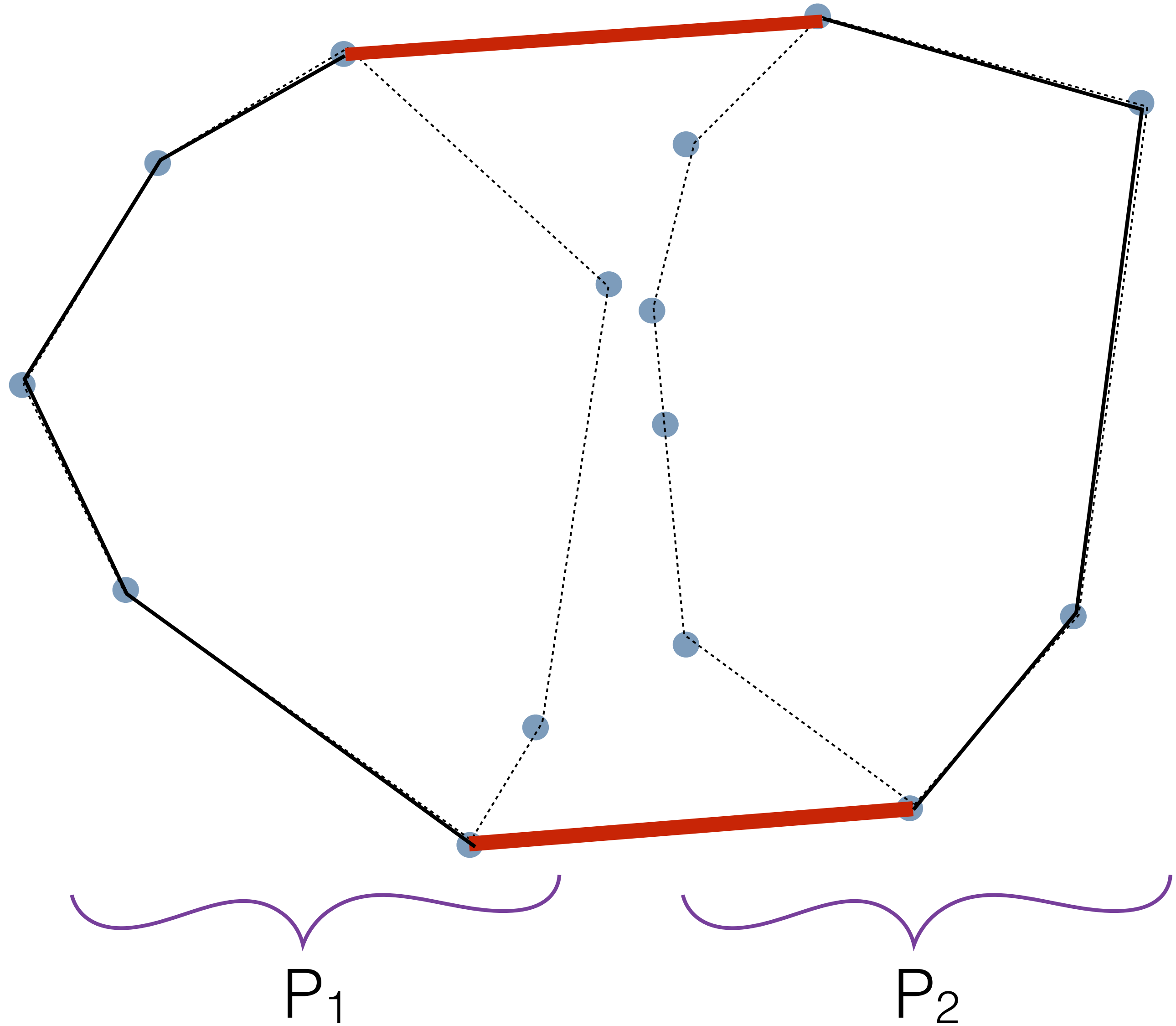
//now get somehow CH(P)



Merging two hulls in linear time

- Here it looks like the upper tangent is between the **top** points in P_1 and P_2
- Is this always true?

- Need to find the two "tangents" (or "bridges")

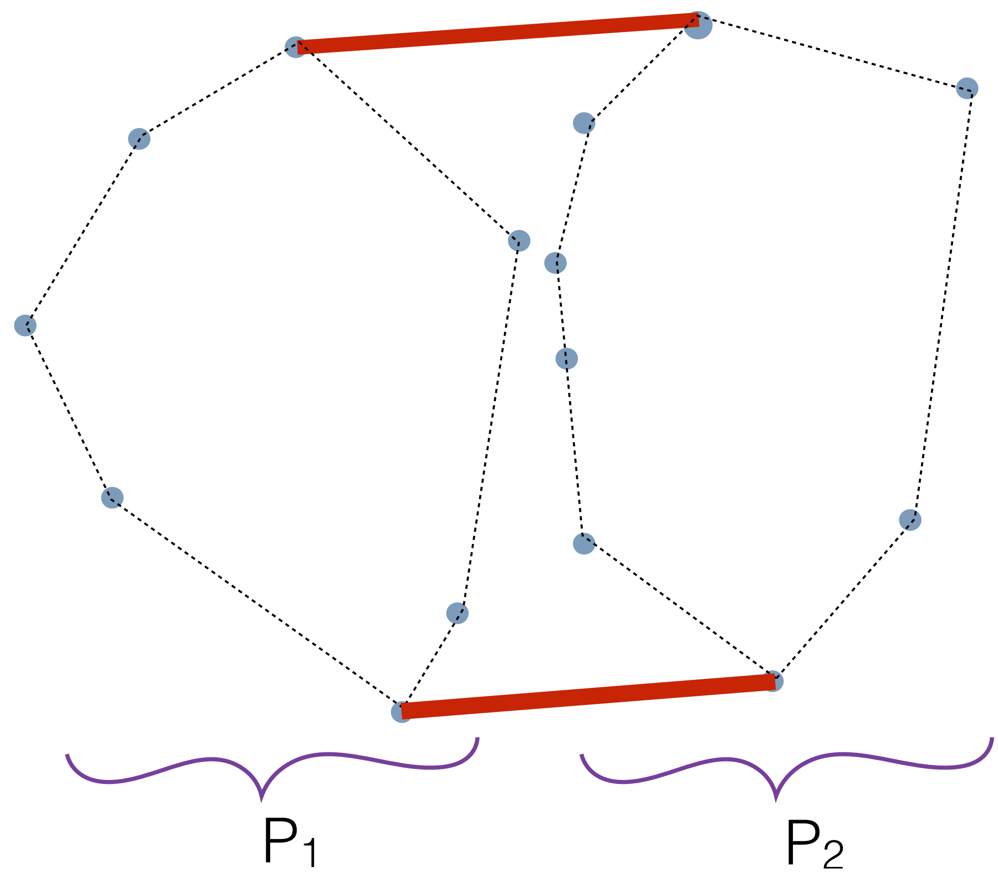


Merging two hulls in linear time

- Here it looks like the upper tangent is between the **top** points in P_1 and P_2
- Is this always true?

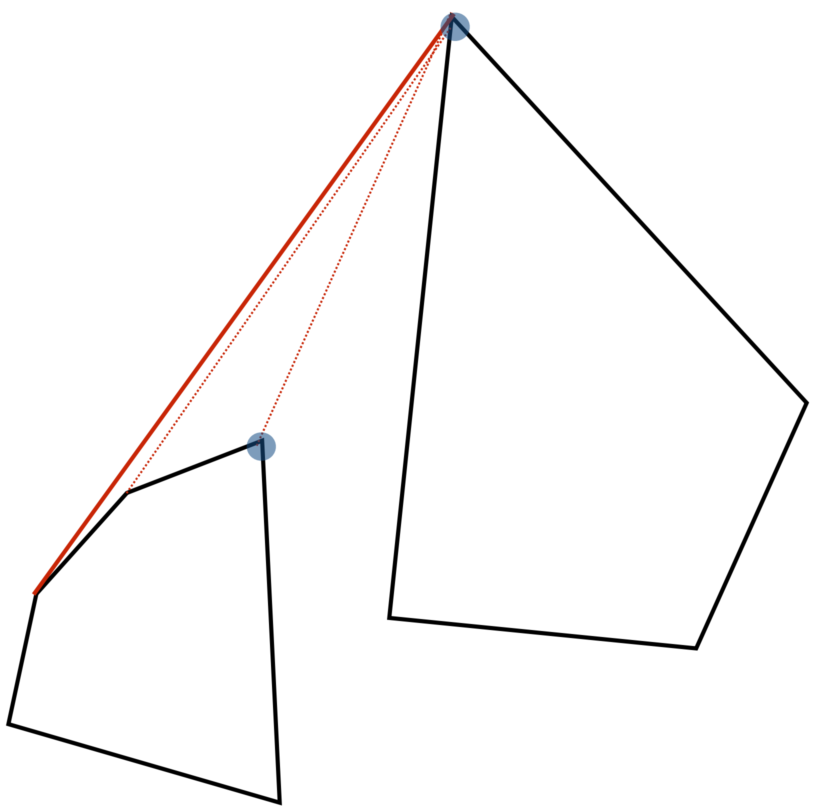
- Need to find the two "tangents" (or "bridges")
- Naive algorithm: try all segments (a,b) with a in H_1 and b in H_2

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

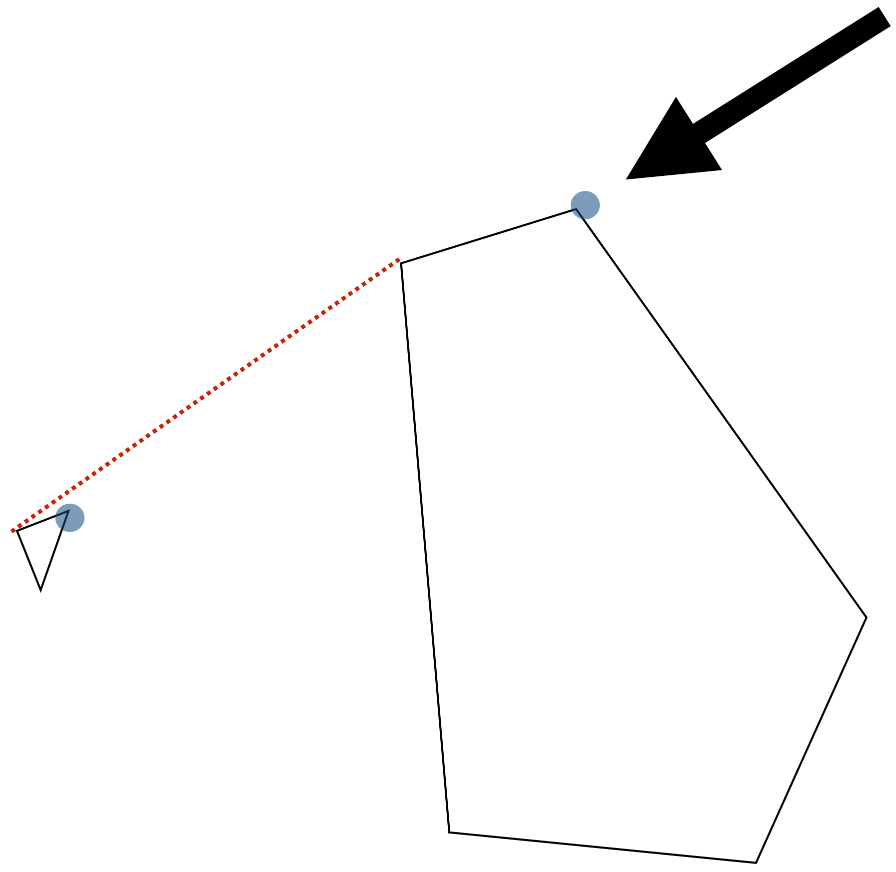


Is the upper tangent guaranteed to connect the **top** points in P_1 and P_2 ?

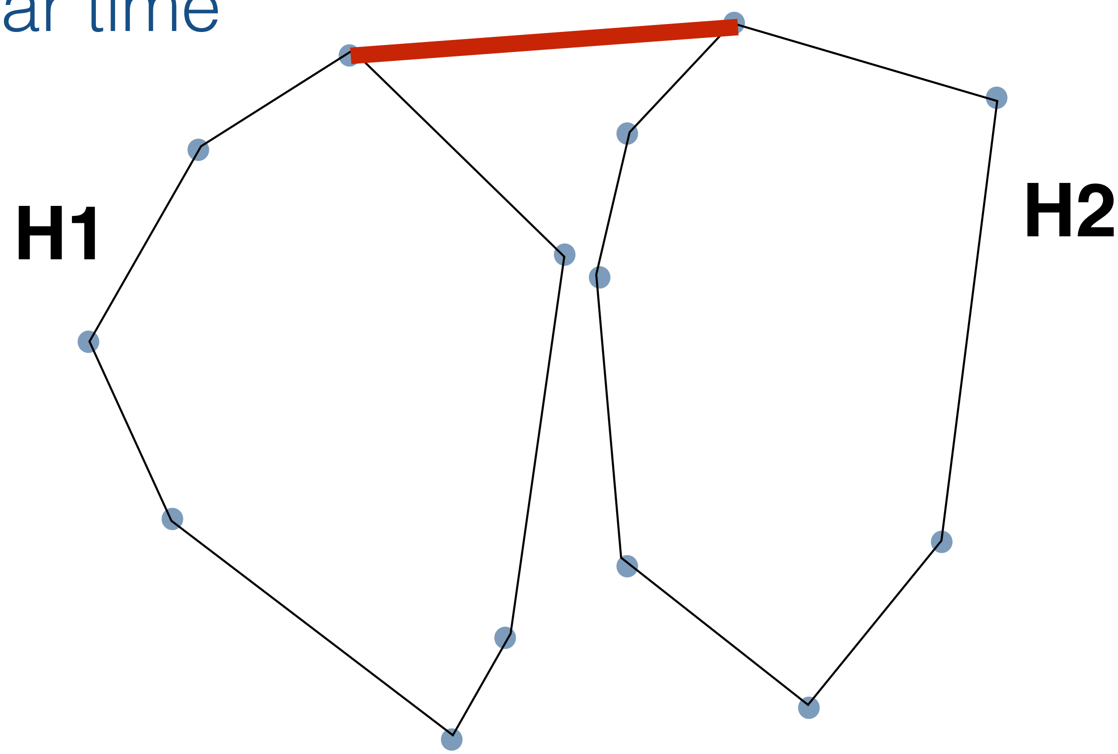
Not necessarily…



The top-most point **overall** is on the CH, but not necessarily on the upper tangent



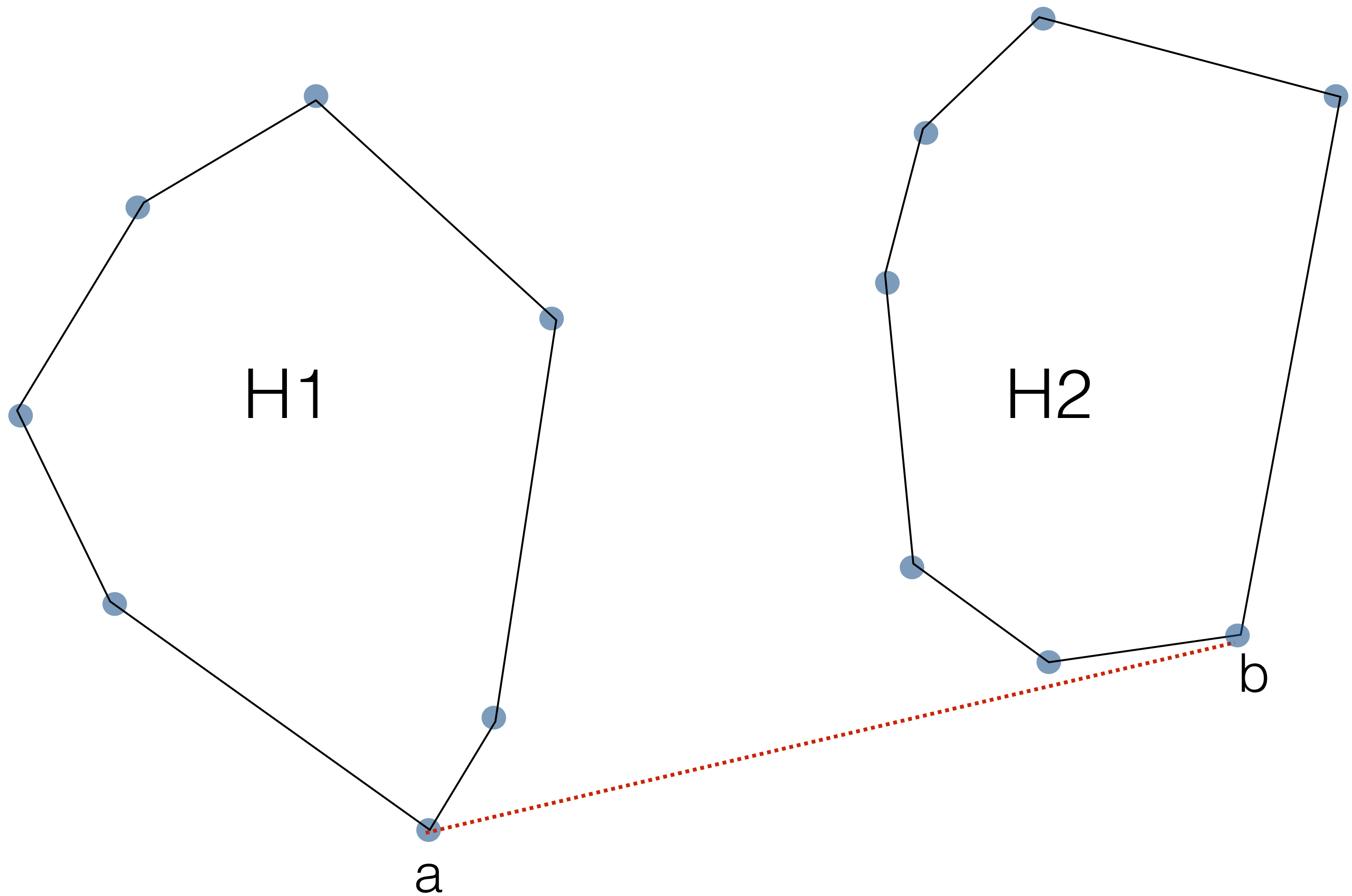
Merging two hulls in linear time



- To find the upper bridge:
  - a = right most point of P1
  - b = left most point of P2
  - while one of succ(a) and pred(b) lies above line ab do:
    - if succ(a) lies above ab then set a = succ(a)
    - else : set b = pred(b)
  - return ab as the upper bridge

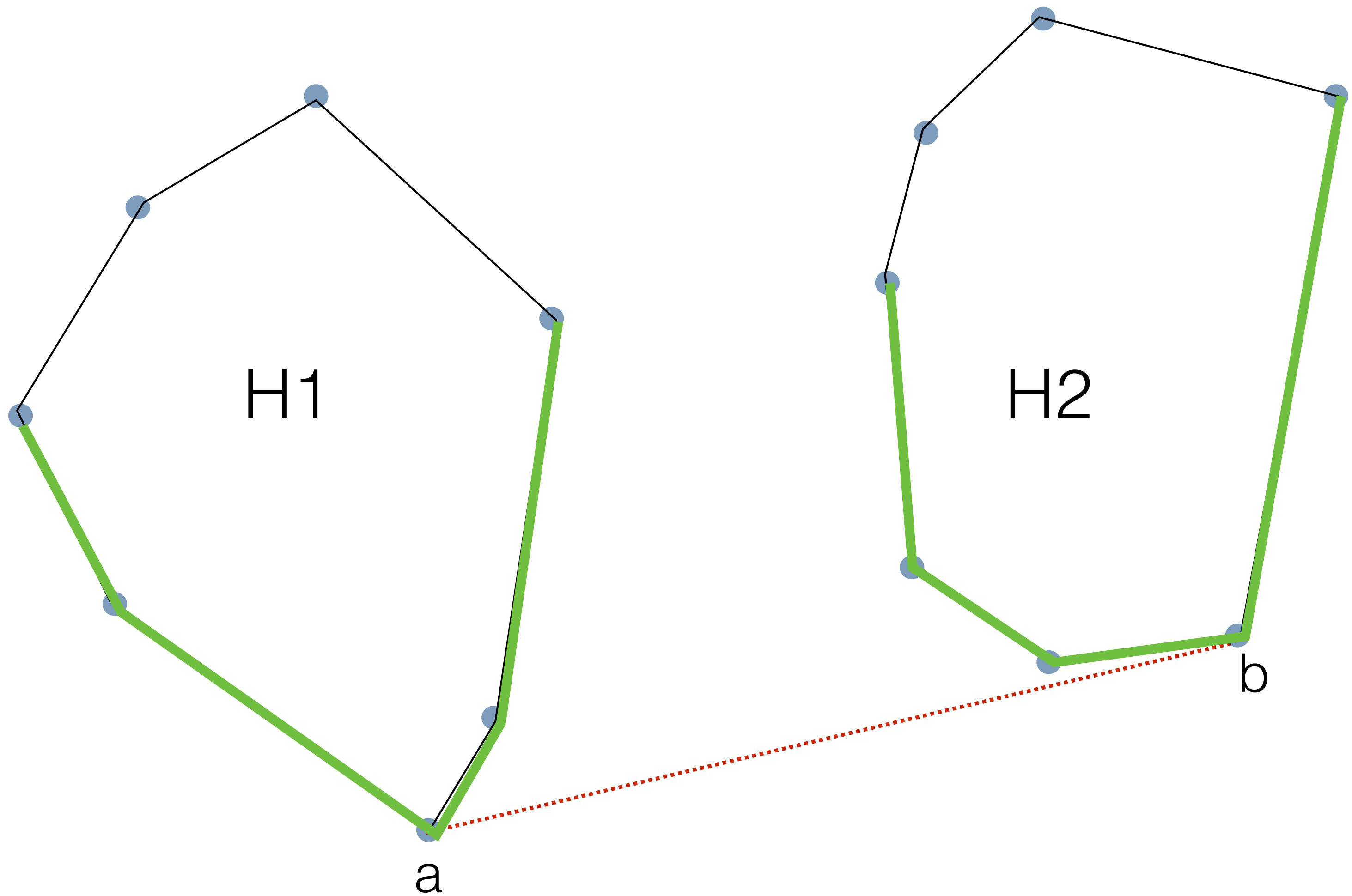
Finding the lower tangent

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



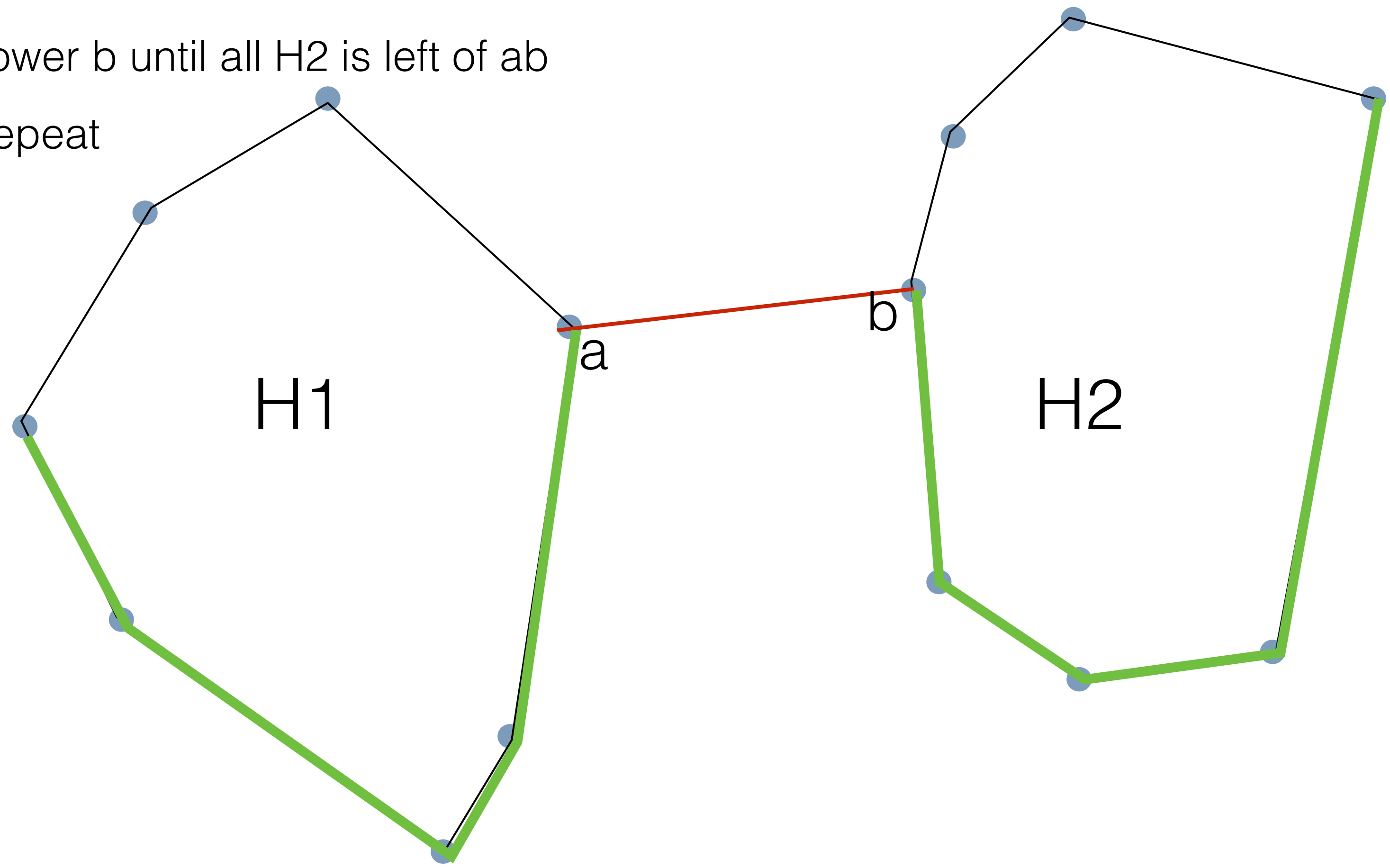
Finding the lower tangent

- Claim: Points a,b are on the lower hulls of H1 and H2, respectively.



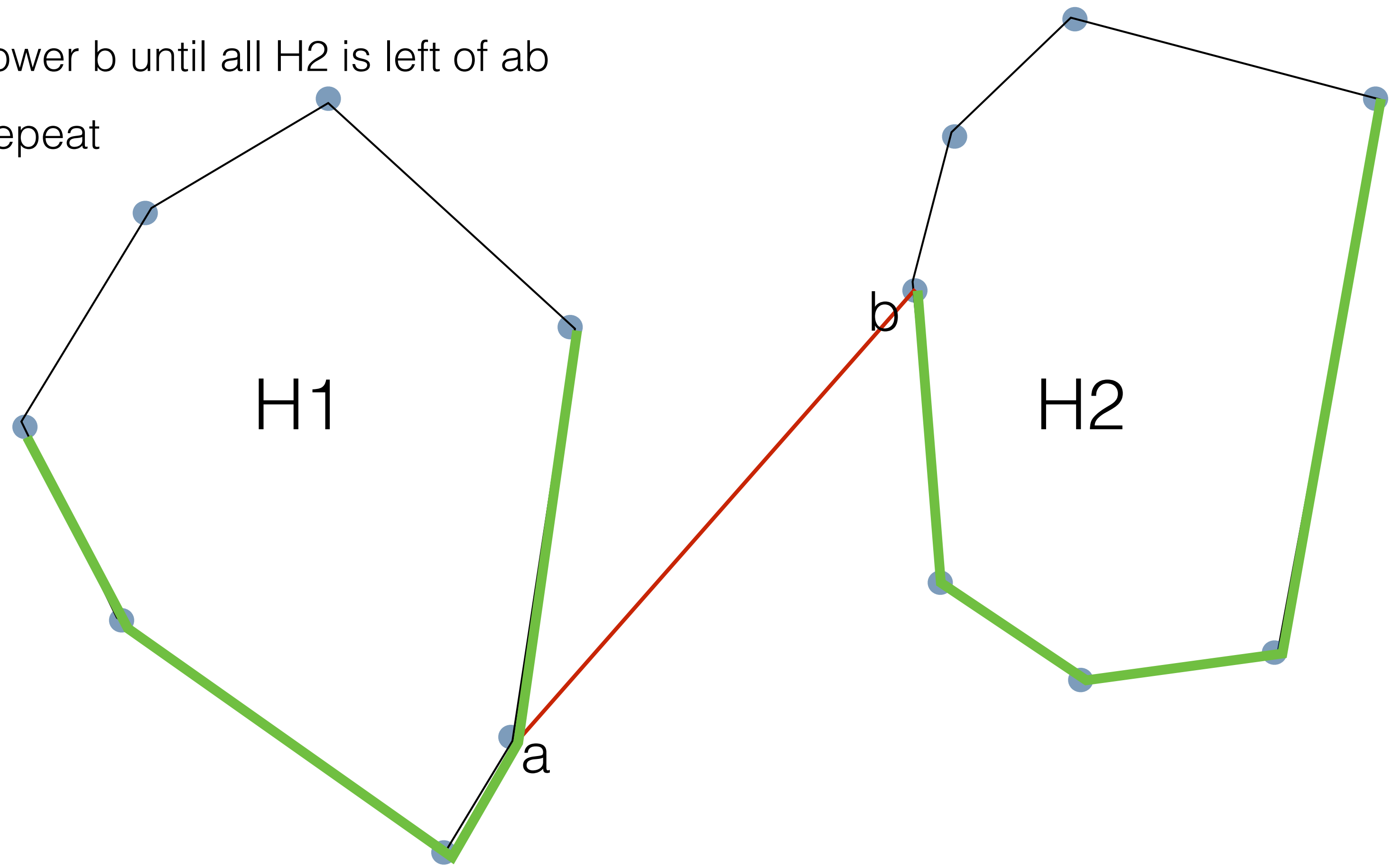
Finding the lower tangent

- 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



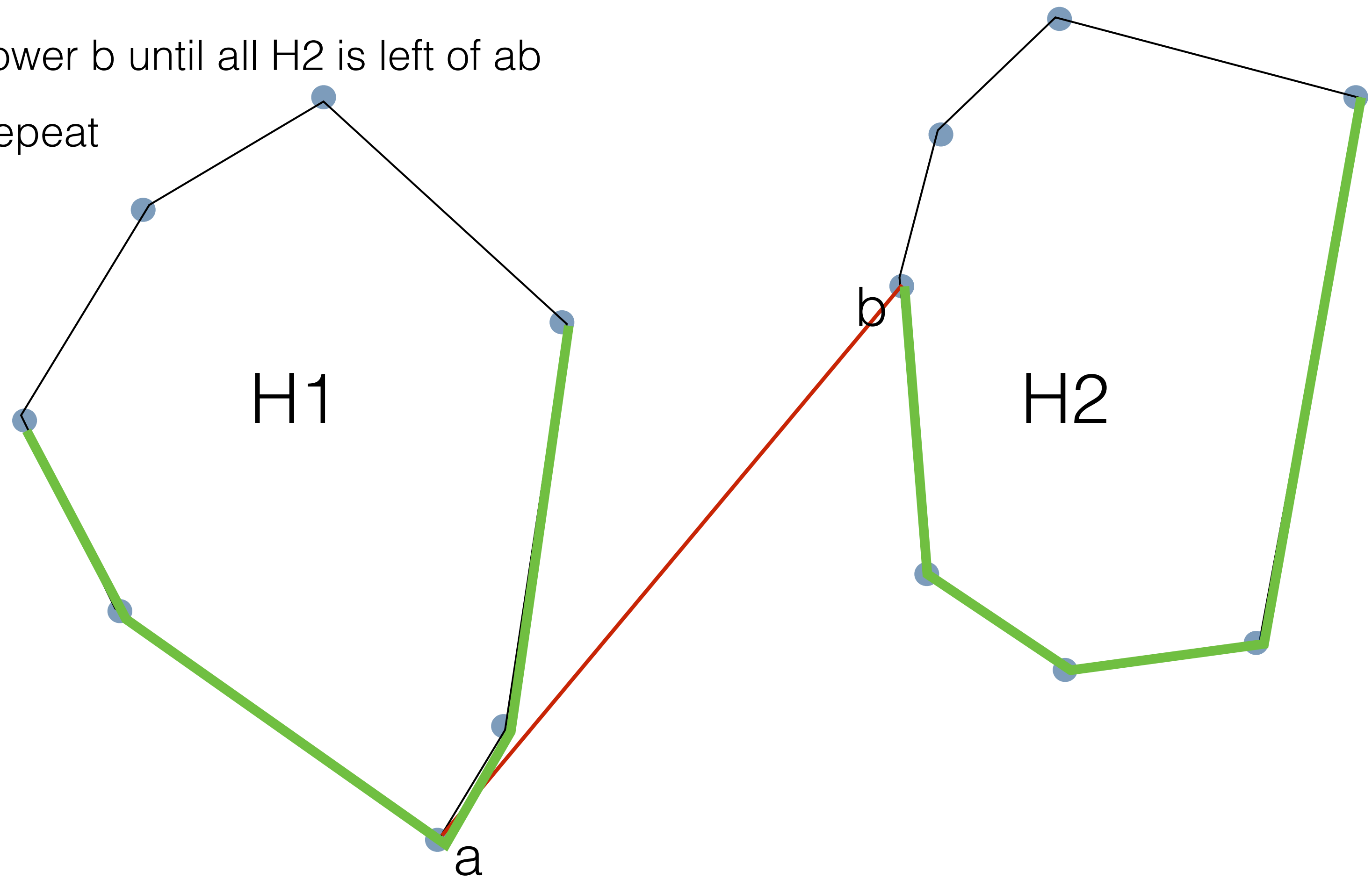
Finding the lower tangent

- 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



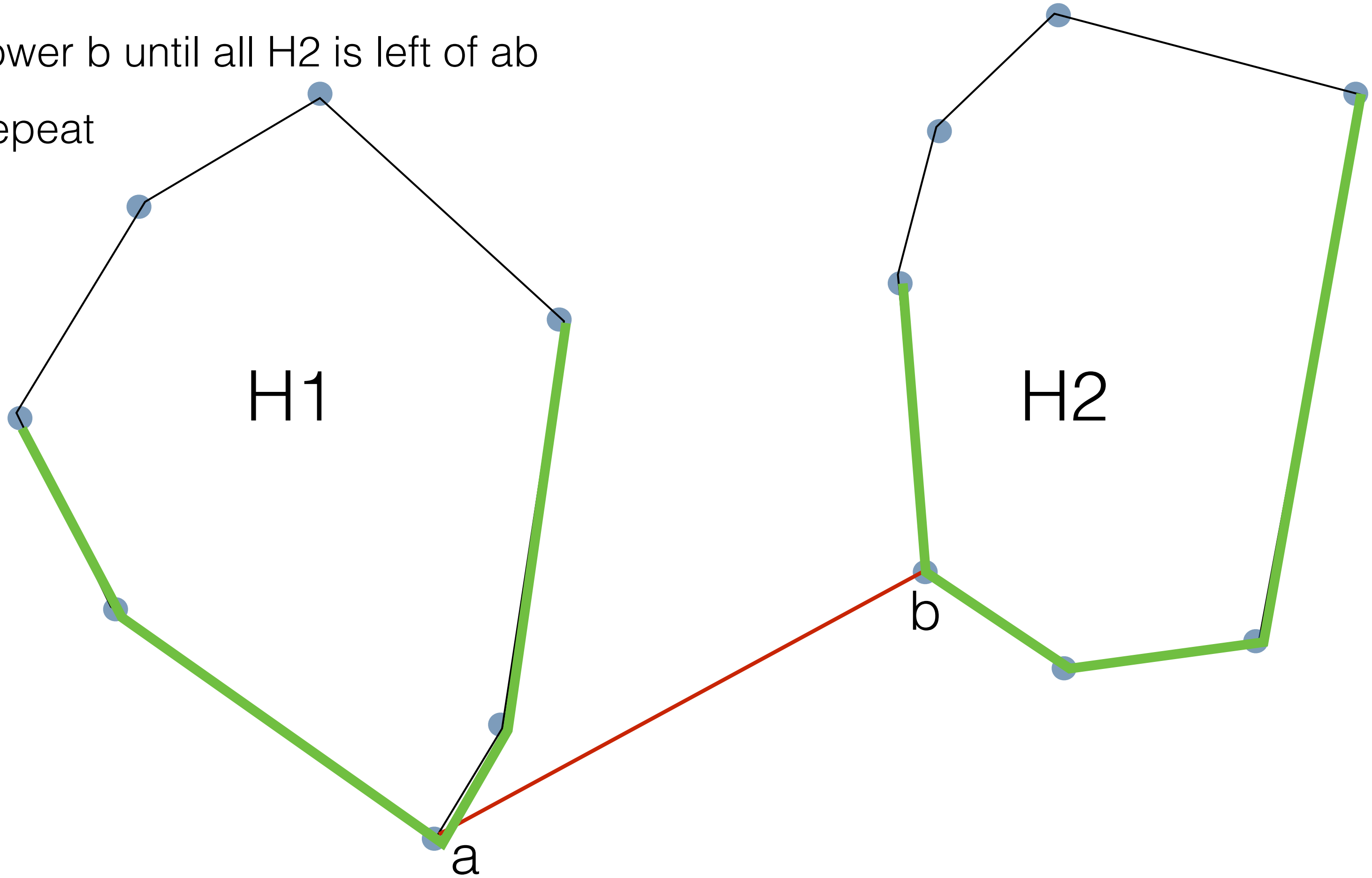
Finding the lower tangent

- 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



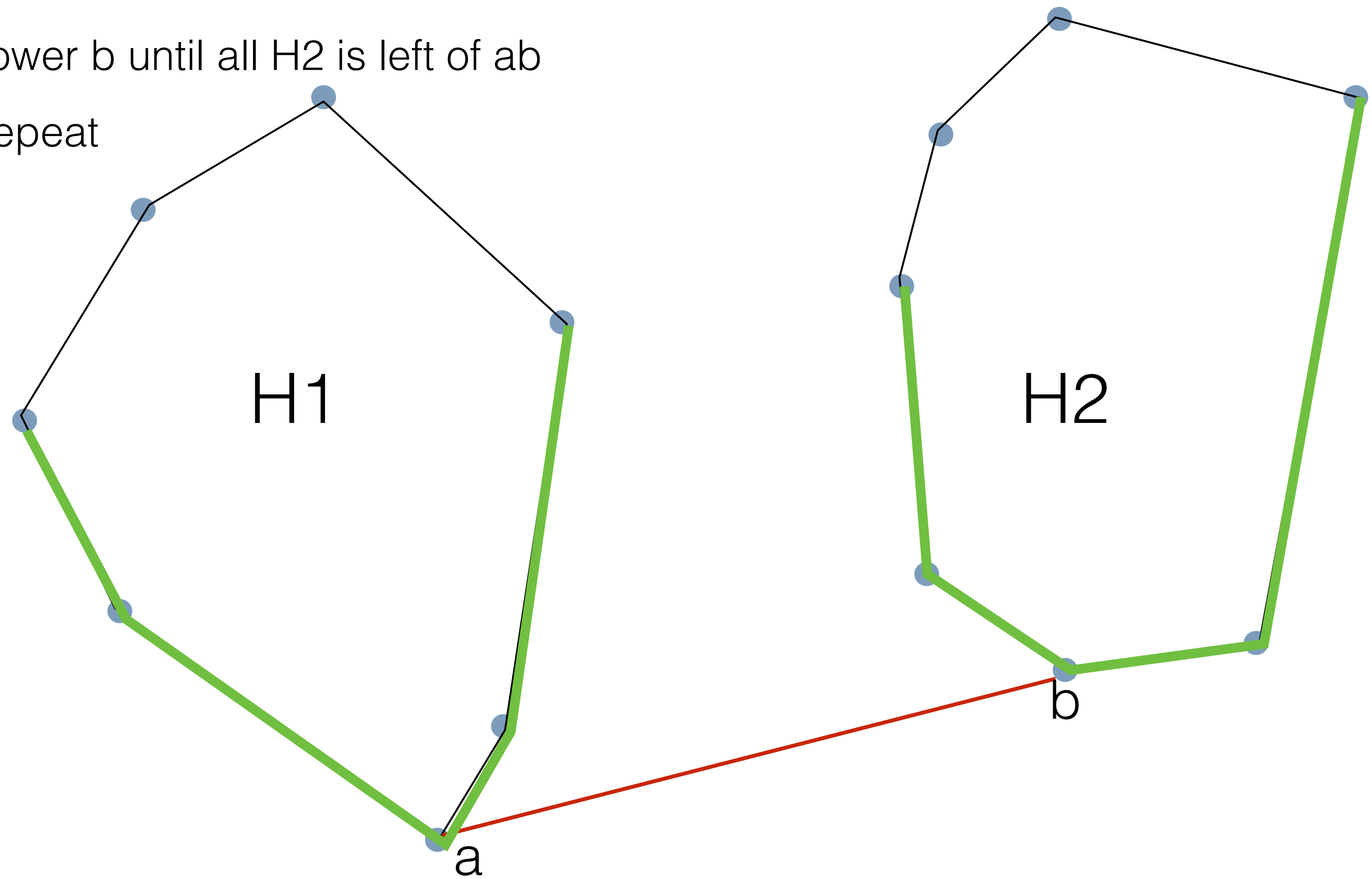
Finding the lower tangent

- 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



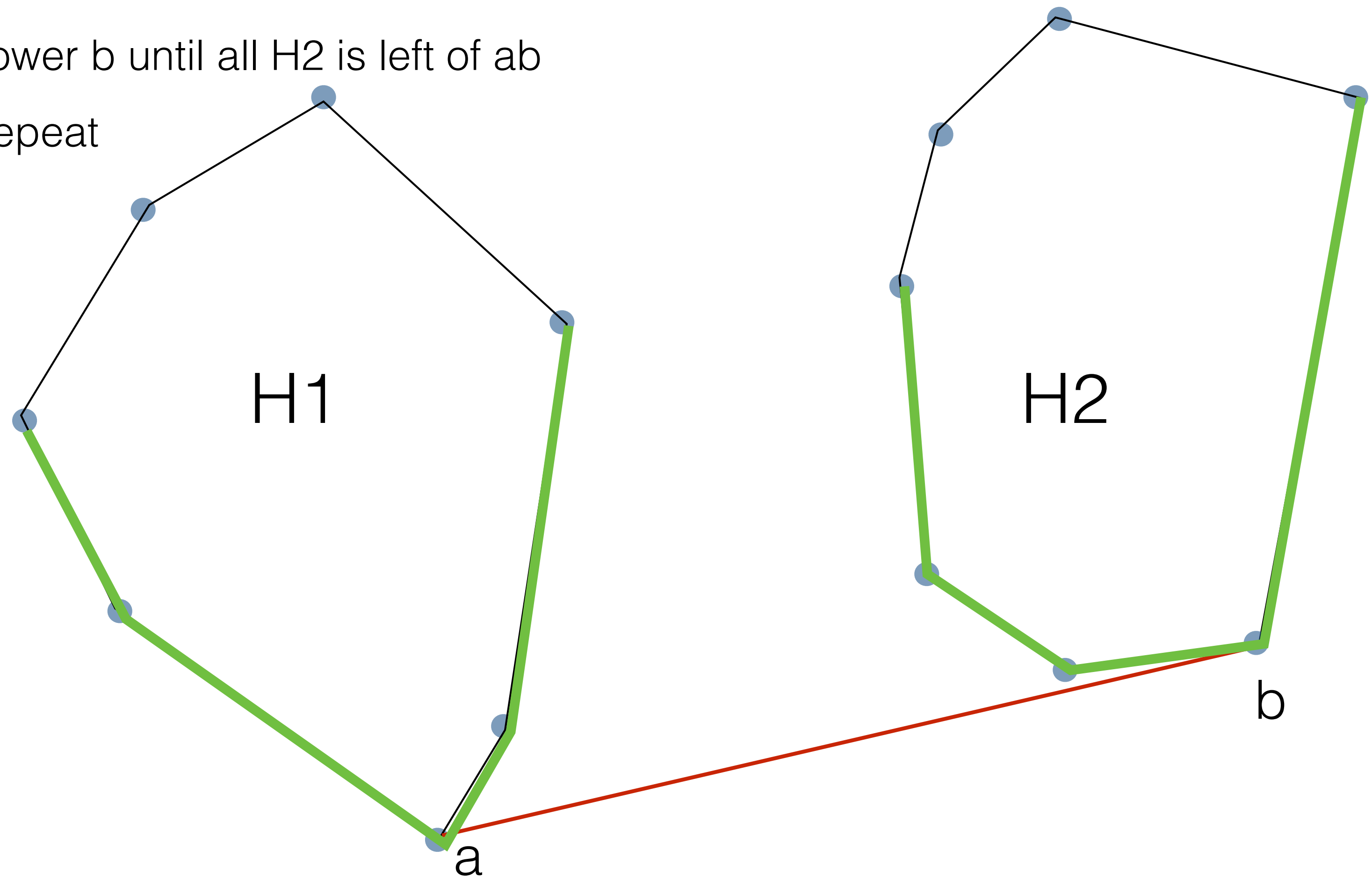
Finding the lower tangent

- 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



Finding the lower tangent

- 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



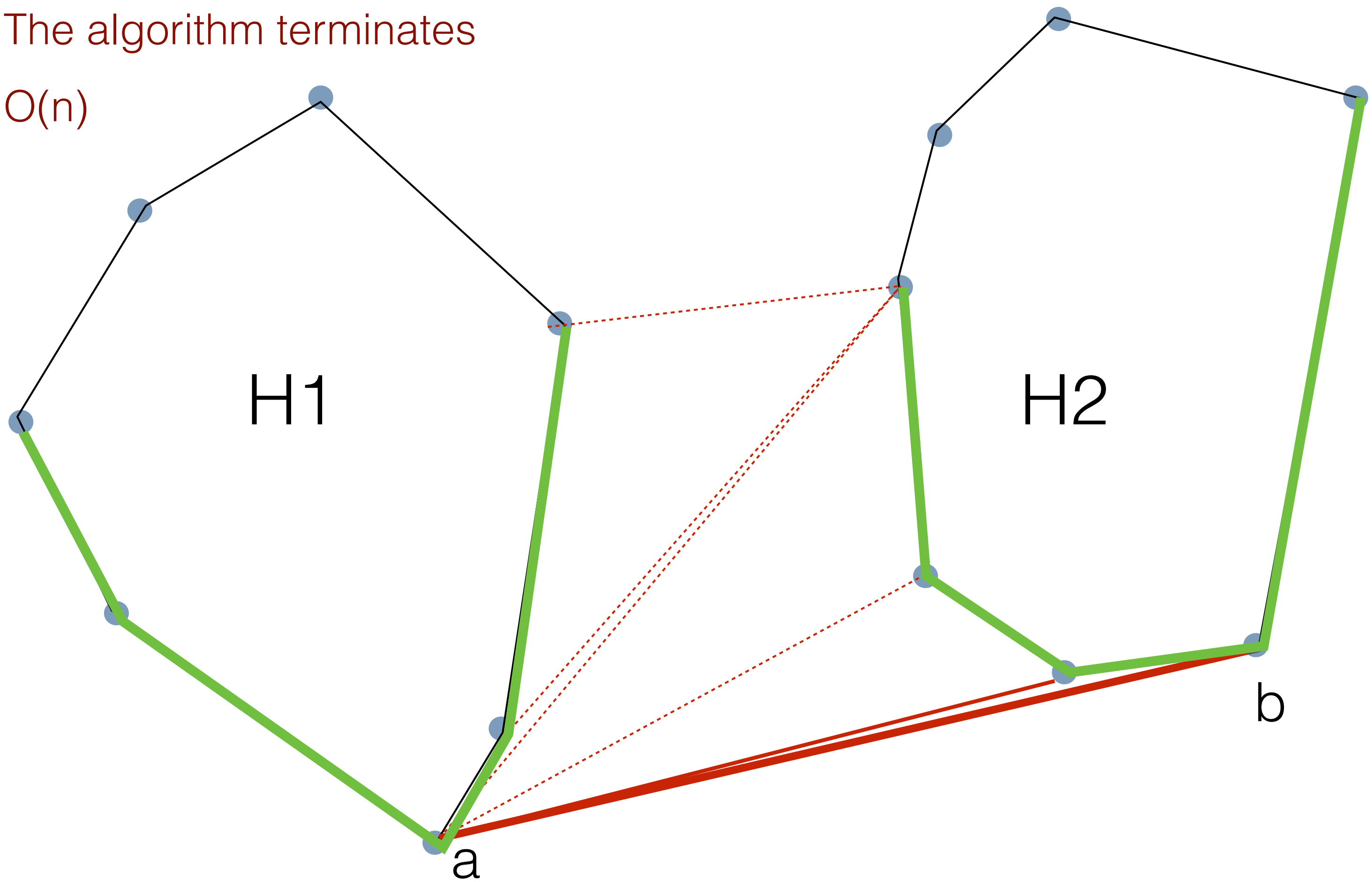
(why) does this work?

Claim: At any point during the algorithm, segment ab cannot intersect the interior of the polygons

==> a cannot move into the upper hull of P1, b cannot move into the upper hull of P2

==> The algorithm terminates

==> O(n)



CH via divide-and-conquer

- Yet another illustration of the divide-and-conquer paradigm
- $O(n \lg n)$
- Extends nicely to 3D

Convex hull in 2D: Summary

- $\Omega(n \lg n)$ lower bound
- Gift wrapping: $O(h \cdot n)$
    - output-size sensitive
    - ✦ by Chand and Kapur [1970]. Extends to 3D and to arbitrary dimensions; for many years was the primary algorithm for higher dimensions
- Graham scan, Andrew's monotone chain: $O(n \lg n)$, but
    - not output-sensitive
    - does not transfer to 3d
- Quickhull: $O(n^2)$
- Incremental CH : $O(n \lg n)$
    - extends to 3D
- Divide-and-conquer CH: $O(n \lg n)$
    - extends to 3D

Convex hull: summary

Naive	$O(n^3)$
Gift wrapping	$O(h \cdot n)$
Quickhull	$O(n^2)$
Graham scan	$O(n \lg n)$
Andrew monotone chain	$O(n \lg n)$