DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

CONVEX HULLS

PETR FELKEL

FEL CTU PRAGUE

felkel@fel.cvut.cz

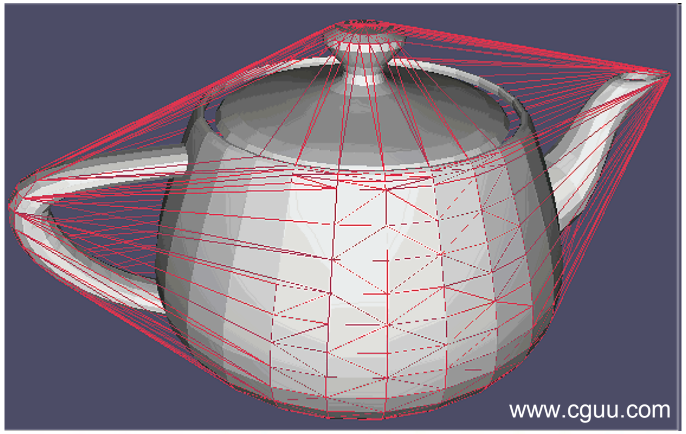
https://cw.felk.cvut.cz/doku.php/courses/a4m39vg/start

Based on [Berg] and [Mount]

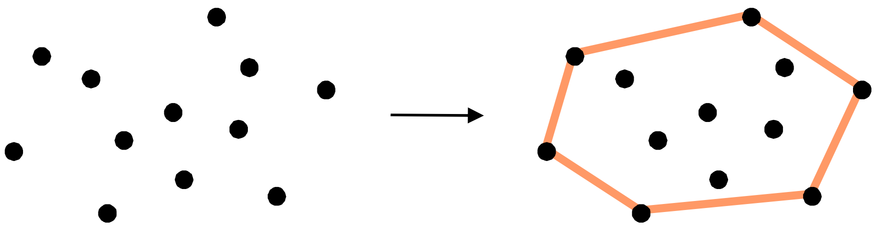
Version from 30.10.2019

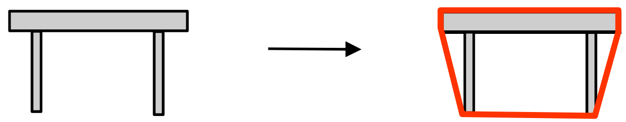
Talk overview

- Motivation and Definitions
- Graham's scan – incremental algorithm
- Divide & Conquer
- Quick hull
- Jarvis's March – selection by gift wrapping
- Chan's algorithm – optimal algorithm



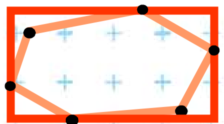
Convex hull (CH) – why to deal with it?



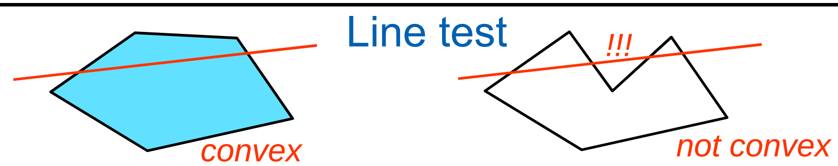


- *Shape approximation* of a point set or complex shapes (other common approximations include: minimal area enclosing rectangle, circle, and ellipse,…) – e.g., for collision detection
- *Initial stage* of many algorithms to filter out irrelevant points, e.g.:
  - diameter of a point set

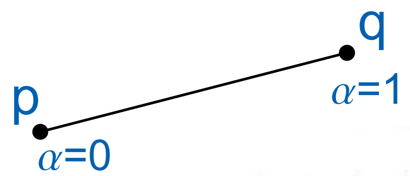
  - minimum enclosing convex shapes (such as rectangle, circle, and ellipse) depend only on points on CH



Convexity



- A set S is *convex*
  - if for any points p,q $\in S$ the line segment $\overline{pq} \subseteq S$, or
  - if any convex combination of p and q is in S
- Convex combination of points p, q is any point that can be expressed as
  $(1 - \alpha)\, p + \alpha q$, where $0 \leq \alpha \leq 1$

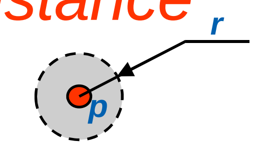


- *Convex hull* CH(S) of set S – is (similar definitions)
  - the smallest set that contains S *(convex)*
  - or: intersection of all convex sets that contain S
  - Or in 2D for points: the smallest convex polygon containing all given points

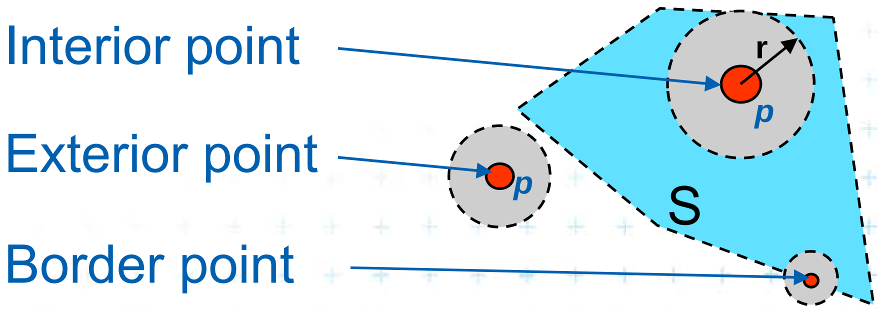
DCGI

Definitions from topology in metric spaces

- *Metric space* – each two of points have defined a *distance*
- *r-neighborhood* of a point *p* and radius *r > 0*
  = set of points whose distance to *p* is strictly less than *r*
  (open ball of diameter *r* centered about *p*)



- Given set S, point *p* is
  - Interior point of S – if $\exists r, r > 0$, (r-neighborhood about *p*) $\subset$ S
  - Exterior point – if it lies in interior of the complement of S
  - Border point – is neither interior neither exterior

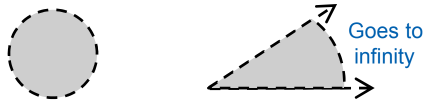


Definitions from topology in metric spaces

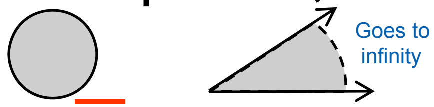
Are border points $p \in S$?

Goes to infinity?

- **Set S is Open** (otevřená)

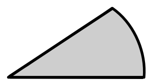


  - $\forall p \in S \;\; \exists$ (r-neighborhood about p of radius r) $\subseteq$ S
  - it contains only interior points, none of its border points
- **Closed** (uzavřená)

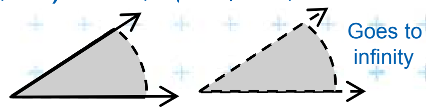


  - If it is equal to its closure $\overline{S}$ (uzávěr = smallest closed set containing S in topol. space)
    $\forall$(r-neighborhood about p of radius r) $\cap$ S $\neq \emptyset$)
- **Clopen** (otevřená i uzavřená) – Ex.: empty set $\emptyset$, or finite set of disjoint components
  - if it is both closed and open
    (S= all positive rational numbers whose square is bigger than 2)

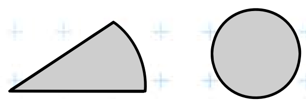
    space Q = rational numbers
    S = ($\sqrt{2}$, ∞) in Q, $\sqrt{2} \notin$ Q, S = $\overline{S}$
- **Bounded** (ohraničená)



Unbounded



  - if it can be enclosed in a ball of finite radius
- **Compact** (kompaktní)



  - if it is both closed and bounded

Clopen (otevřená i uzavřená) example

If it is both closed and open => clopen

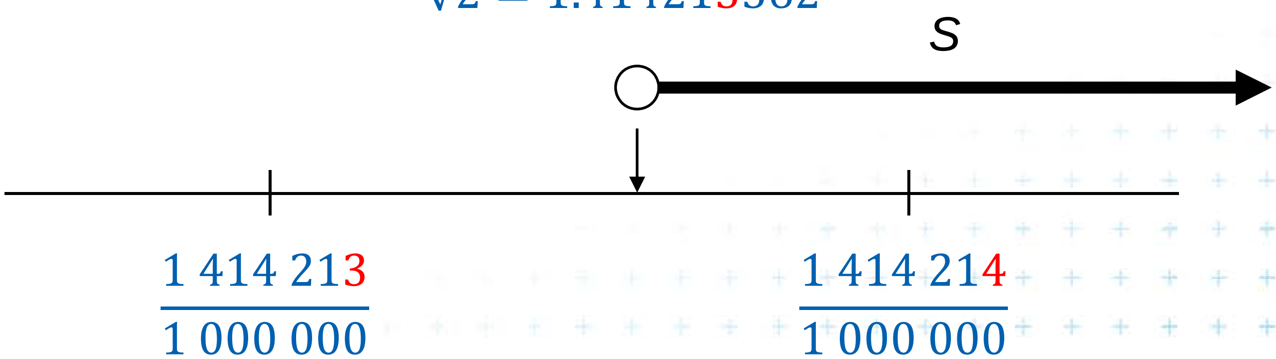
Space Q: rational numbers

Set S: all positive rational numbers whose square is bigger than 2

$S = (\sqrt{2}, \infty)$ in Q

$$\left.\begin{array}{l} \sqrt{2} \notin Q \;\Rightarrow\; \text{open (does not contain the border)} \\ S = \bar{S} \;\Rightarrow\; \text{closed (equal to its closure } \bar{S}\text{)} \end{array}\right\} \Rightarrow \text{clopen}$$

$\sqrt{2} = 1.414213562$

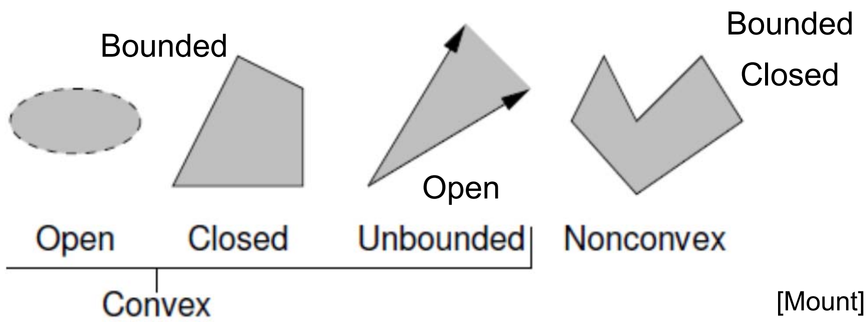


Note: in R: $\sqrt{2} \notin Q \Rightarrow \text{open}$

$S \neq \bar{S} \Rightarrow \text{not closed}, \quad \bar{S} = \langle \sqrt{2}, \infty)$

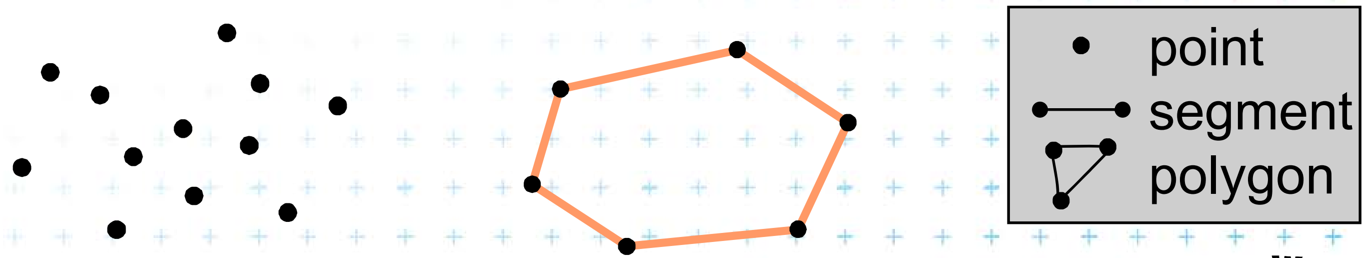
Definitions from topology in metric spaces

- *Convex set S may be bounded or unbounded*



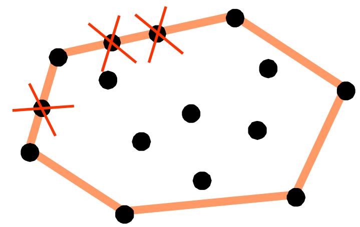
- *Convex hull* CH(S) of a finite set *S* of points in the plane

  = Bounded, closed, (= compact) convex polygon



Convex hull representation

- CCW enumeration of vertices
- Contains only the extreme points ("endpoints" of collinear points)



- Simplification for the whole semester:
  Assume the input points are in general position,
  - no two points have the same *x*-coordinates and
  - no three points are collinear

  -> We avoid problem with non-extreme points on *x*
  (solution may be simple – e.g. lexicographic ordering)

Online x offline algorithms

- Incremental algorithm
  - Proceeds one element at a time (step-by-step)
- Online algorithm (must be incremental)
  - is started on a partial (or empty) input and
  - continues its processing as additional input data becomes available (comes online, thus the name).
  - Ex.: insertion sort
- Offline algorithm (may be incremental)
  - requires the entire input data from the beginning
  - than it can start
  - Ex.: selection sort (any algorithm using sort)

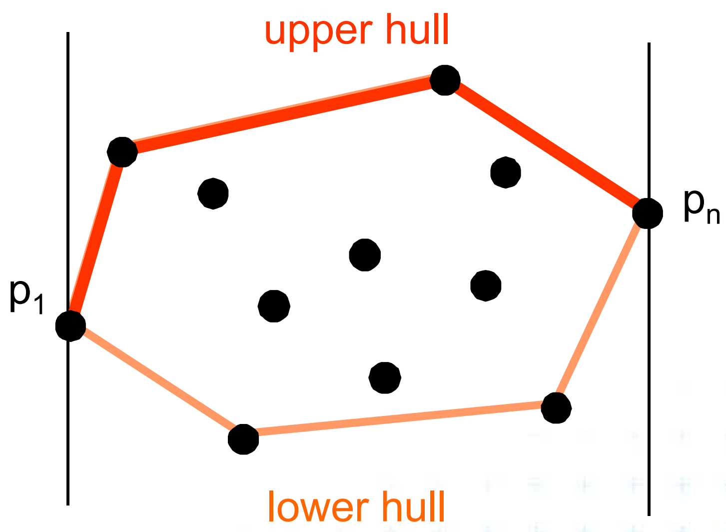
Graham's scan

- Incremental O(*n* log *n*) algorithm
- Objects (points) are added one at a time
- Order of insertion is important
  1. Random insertion
     –> we need to test: *is-point-inside-the-hull(p)* 
  2. Ordered insertion
     Find the point p with the smallest *y* coordinate first
     a) Sort points p_i according to *increasing angles* around the point p (angle of pp_i and x axis)
     b) Andrew's modification: sort points p_i according to *x* and add them left to right (construct upper & lower hull)

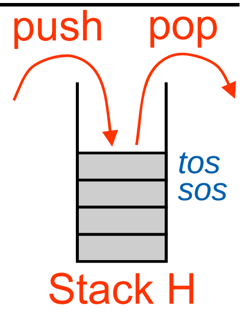
  Sorting *x-coordinates* is simpler to implement than sorting of angles

Graham’s scan – b) modification by Andrew

- O(*n* log *n*) for unsorted points, O(*n*) for sorted pts.
- Upper hull, then lower hull. Merge.
- Minimum and maximum on *x* belong to CH



Graham's scan – incremental algorithm

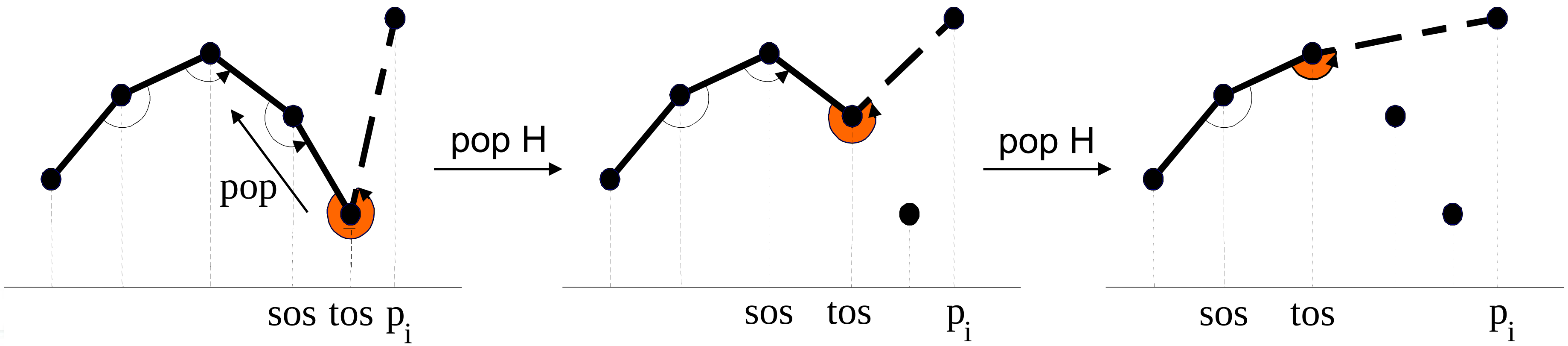


GrahamsScan(points p)

Input: points p

Output: CCW points on the convex hull

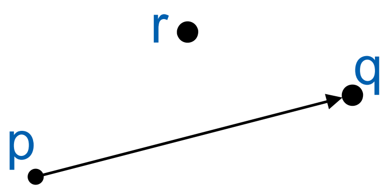
1. sort points according to increasing x-coord -> $\{p_1, p_2, \ldots, p_n\}$
2. push(p_1, H), push(p_2, H) **upper hull**
3. **for** *i = 3* ***to*** *n* **do**
4. **while**(size(H) $\geq$ 2 and orient(*sos*, *tos*, p_i) $\geq$ 0) // skip left turns
5. pop H // (back-tracking)
6. push(p_i, H) // store right turn
7. store H to the output (in reverse order) // upper hull
8. Symmetrically the lower hull

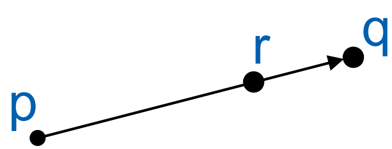


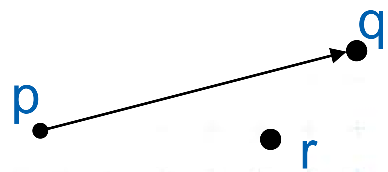
Position of point in relation to segment

$$\text{orient}(\,p,\,q,\,r\,)\begin{cases} > 0 & r \text{ is left from } pq\text{, CCW orient} \\ = 0 & \text{if } (\,p,\,q,\,r\,) \text{ are collinear} \\ < 0 & r \text{ is right from } pq\text{, CW orient} \end{cases}$$

Point *r* is: **left from *pq*** **on segment *pq*** **right from *pq***

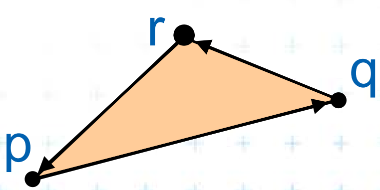


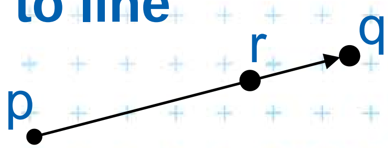


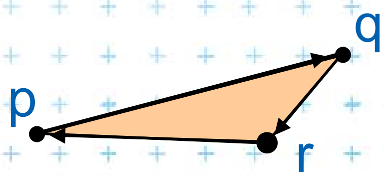


Convex polygon with edges pq and qr or

Triangle *pqr*: is CCW oriented **degenerated to line** **is CW oriented**



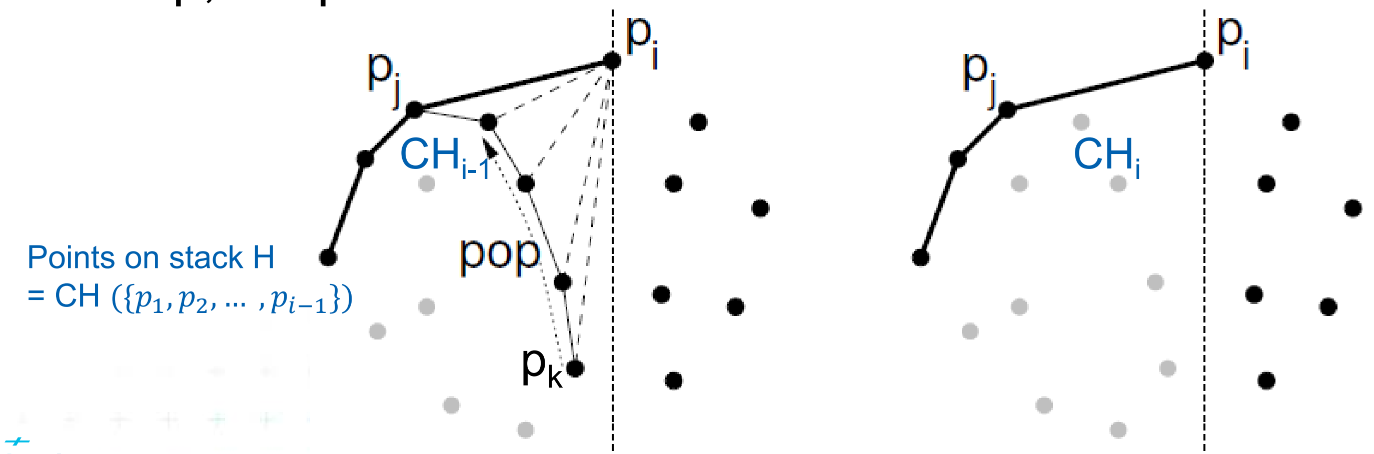




Is Graham's scan correct?

Stack H at any stage contains upper hull of the points $\{p_1, \dots, p_j, p_i\}$, processed so far

- For induction basis $H = \{p_1, p_2\}$ … true
- p_i = last added point to CH, p_j = its predecessor on CH
- Each point p_k that lies between p_j and p_i lies below $p_j p_i$ and should not be part of UH after addition of p_i => is removed before push p_i. [orient$(p_j, p_k, p_i) > 0$, p_k is right from $p_j p_i \Rightarrow p_k$ is removed from UH]
- Stop, if 2 points in the stack or after construction of the upper hull



[Mount]

DCGI

Complexity of Graham's scan

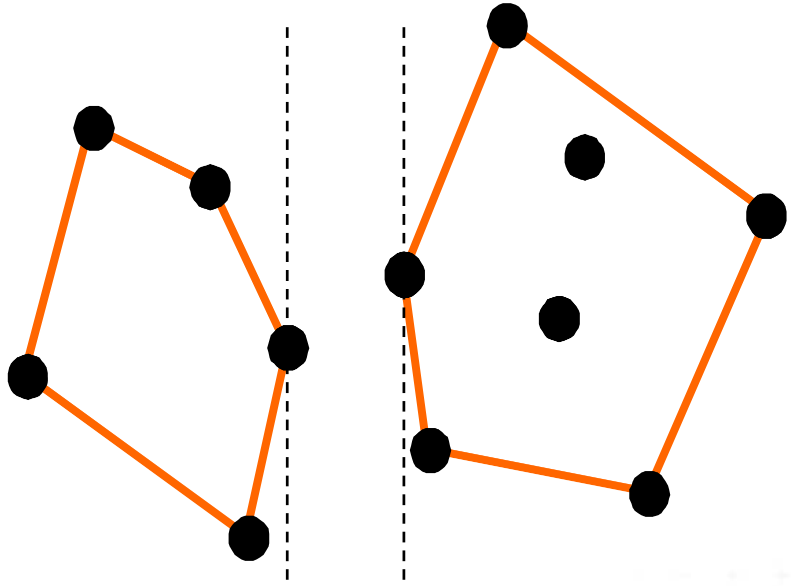
- Sorting according x – $O(n \log n)$
- Each point pushed once – $O(n)$
- Some ($d_i \leq n$) points deleted while processing p_i – $O(n)$
- The same for lower hull – $O(n)$

- Total $O(n \log n)$ for unsorted points
  $O(n)$ for sorted points

Divide & Conquer

- $\Theta(n \log(n))$ algorithm
- Extension of mergesort
- Principle
  - Sort points according to x-coordinate,
  - recursively partition the points and solve CH.

Convex hull by D&C



ConvexHullD&C(points **P)**

Input: points p

Output: CCW points on the convex hull

Sort points P according to *x*

return hull(P)

hull(points P)

    if $|P| \leq 3$ then

        compute CH by brute force,

        return

    Partition P into two sets L and R (with lower & higher coords *x*)

    Recursively compute H_L = hull(L), H_R = hull(R)

    H = Merge hulls(H_L, H_R) by computing

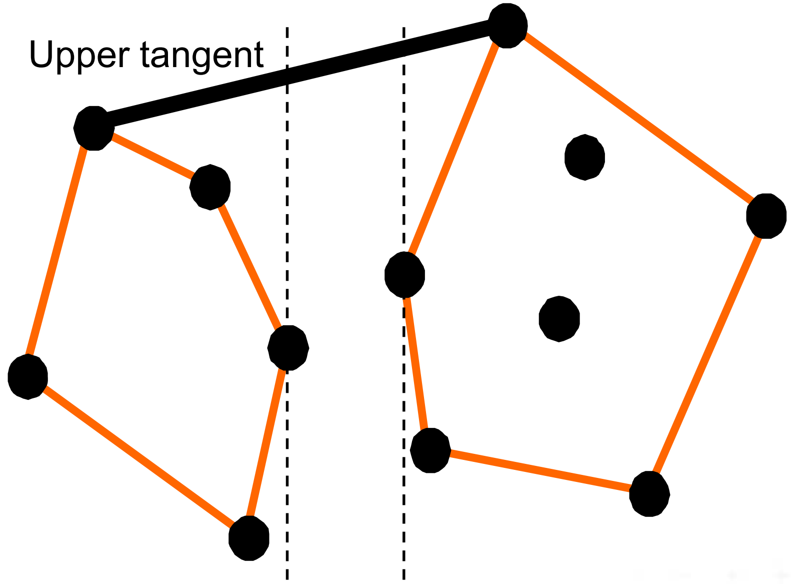
        Upper_tangent(H_L, H_R) // find nearest points, H_L CCW, H_R CW

        Lower_tangent(H_L, H_R) // (H_L CW, H_R CCW)

        discard points between these two tangents

    return H

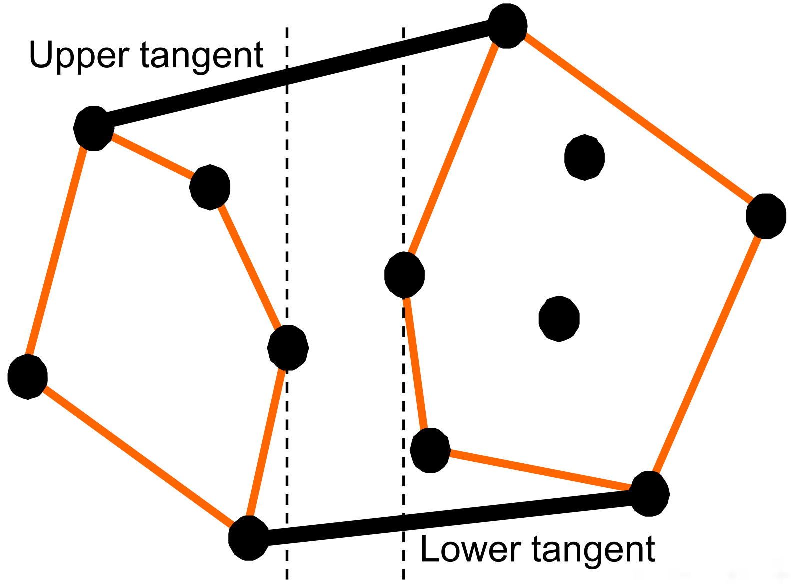
Convex hull by D&C



ConvexHullD&C(points **P)**
Input: points p
Output: CCW points on the convex hull
Sort points P according to *x*
return hull(P)

hull(points P)
    if $|P| \leq 3$ then
        compute CH by brute force,
        return
    Partition P into two sets L and R (with lower & higher coords *x*)
    Recursively compute H_L = hull(L), H_R = hull(R)
    H = Merge hulls(H_L, H_R) by computing
        Upper_tangent(H_L, H_R) // find nearest points, H_L CCW, H_R CW
        Lower_tangent(H_L, H_R) // (H_L CW, H_R CCW)
        discard points between these two tangents
    return H

Convex hull by D&C



ConvexHullD&C(points **P)**

Input: points p

Output: CCW points on the convex hull

Sort points P according to *x*

return hull(P)

hull(points P)

    if $|P| \leq 3$ then

        compute CH by brute force,

        return

    Partition P into two sets L and R (with lower & higher coords *x*)

    Recursively compute H_L = hull(L), H_R = hull(R)

    H = Merge hulls(H_L, H_R) by computing

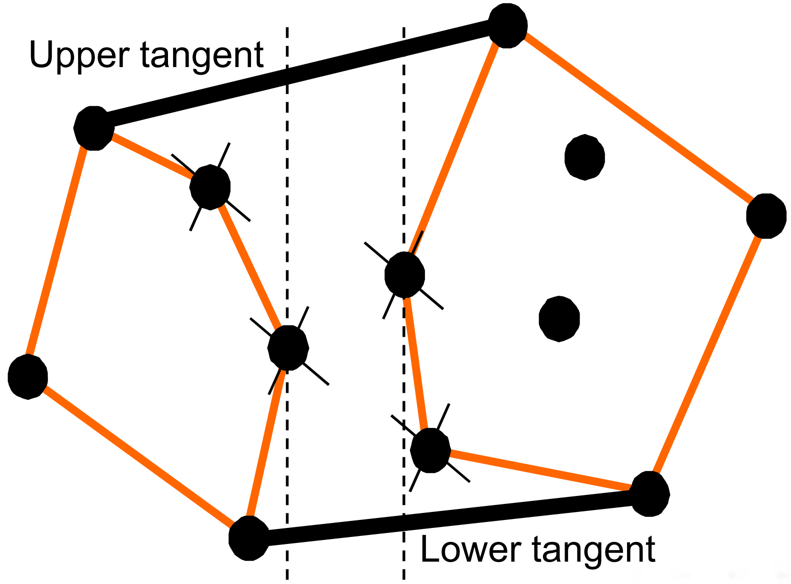
        Upper_tangent(H_L, H_R) // find nearest points, H_L CCW, H_R CW

        Lower_tangent(H_L, H_R) // (H_L CW, H_R CCW)

        discard points between these two tangents

    return H

Convex hull by D&C



ConvexHullD&C(points **P)**

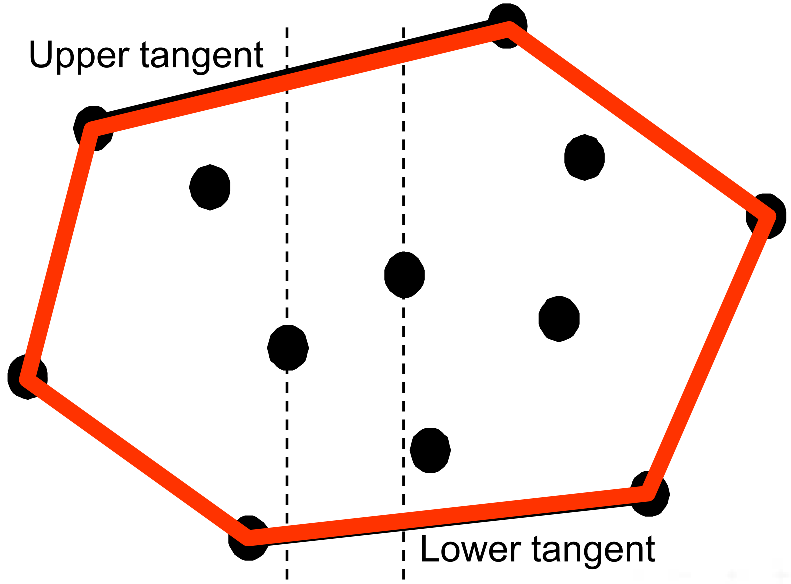
Input: points p

Output: CCW points on the convex hull

1. Sort points P according to *x*
2. return hull(P)

3. **hull(points P)**
4. if $|P| \leq 3$ then
5. compute CH by brute force,
6. return
7. Partition P into two sets L and R (with lower & higher coords *x*)
8. Recursively compute H_L = hull(L), H_R = hull(R)
9. H = Merge hulls(H_L, H_R) by computing
10. Upper_tangent(H_L, H_R) // find nearest points, H_L CCW, H_R CW
11. Lower_tangent(H_L, H_R) // (H_L CW, H_R CCW)
12. discard points between these two tangents
13. return H

Convex hull by D&C



ConvexHullD&C(points **P)**

Input: points p

Output: CCW points on the convex hull

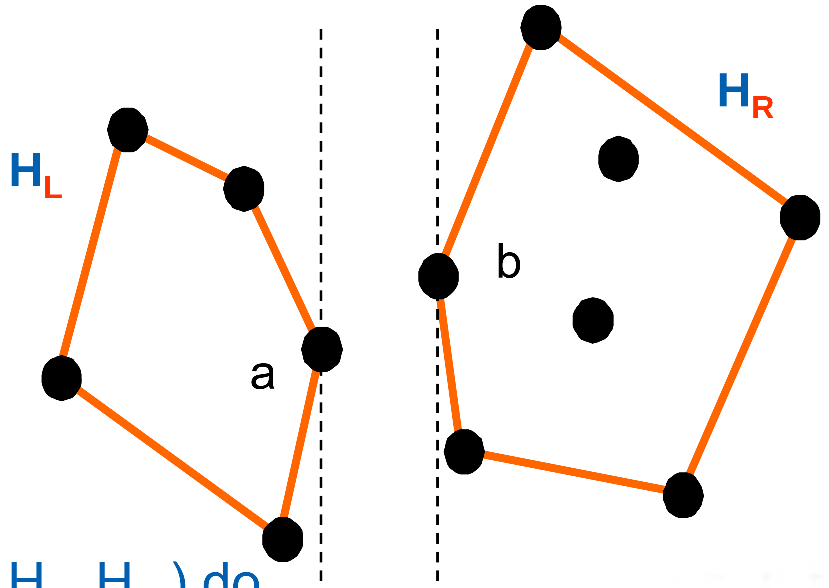
1. Sort points P according to *x*
2. return hull(P)

3. hull(points P)

4. if |P| ≤ 3 then
5. compute CH by brute force,
6. return
7. Partition P into two sets L and R (with lower & higher coords *x*)
8. Recursively compute H_L = hull(L), H_R = hull(R)
9. H = Merge hulls(H_L, H_R) by computing
10. Upper_tangent(H_L, H_R) // find nearest points, H_L CCW, H_R CW
11. Lower_tangent(H_L, H_R) // (H_L CW, H_R CCW)
12. discard points between these two tangents
13. return H

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



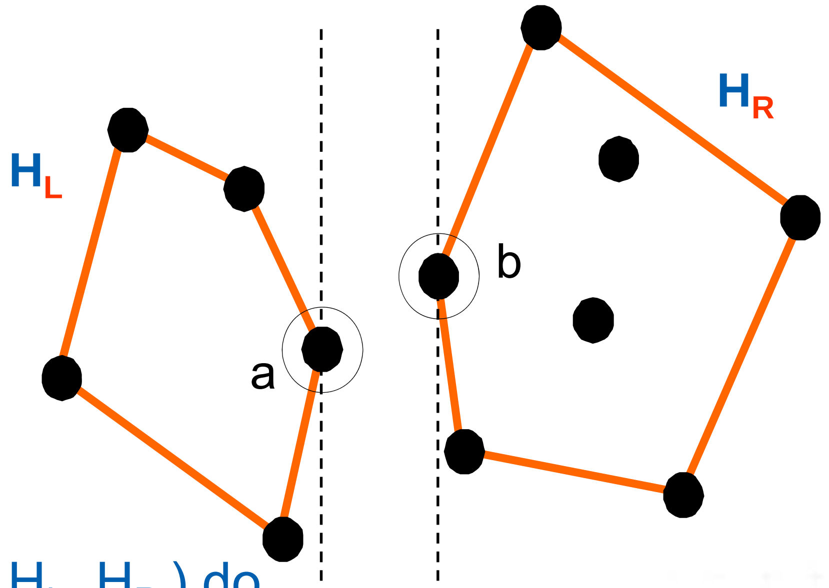
1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) ≥ 0
which means *a.succ* is left from line *ab*

$$m = |H_L| + |H_R| \leq |L| + |R| \quad \Rightarrow \text{Upper Tangent: } O(m) = O(n)$$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



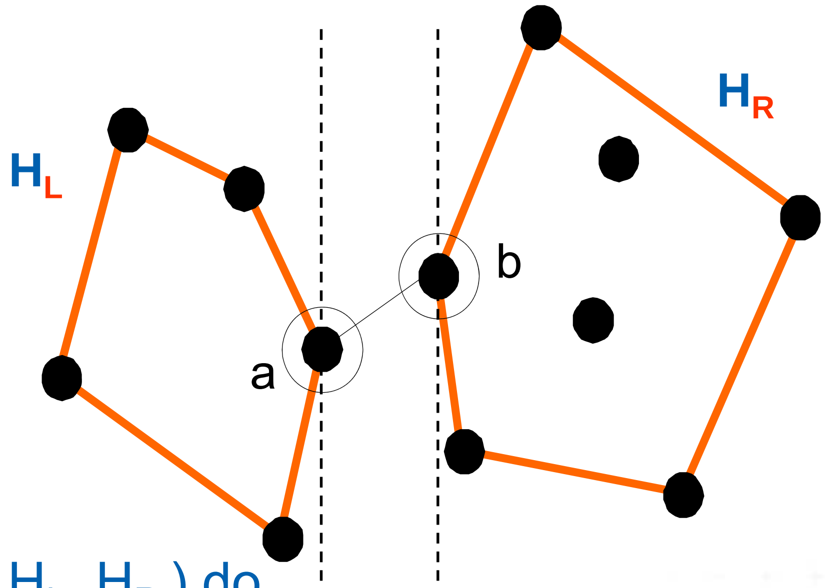
1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) ≥ 0
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



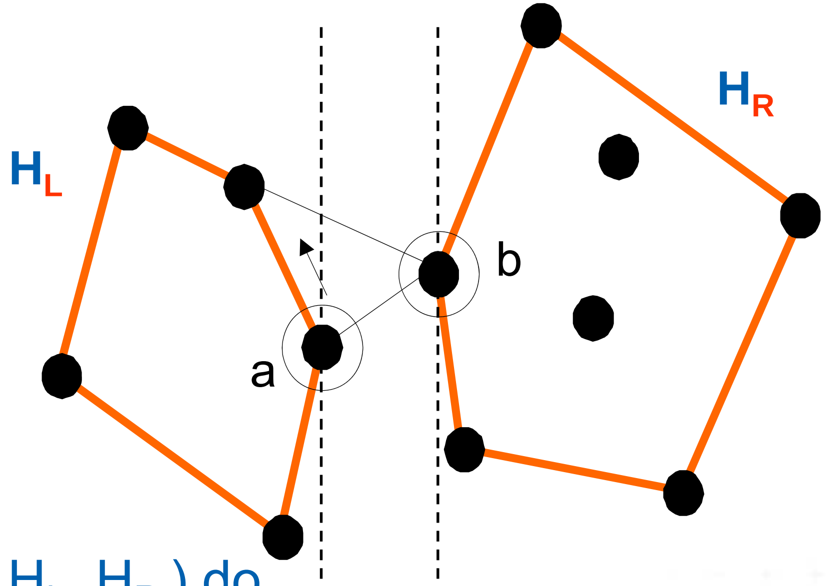
1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) $\geq$ 0
which means *a.succ* is left from line *ab*

$$m = |H_L| + |H_R| \leq |L| + |R| \quad \Rightarrow \text{Upper Tangent: } O(m) = O(n)$$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



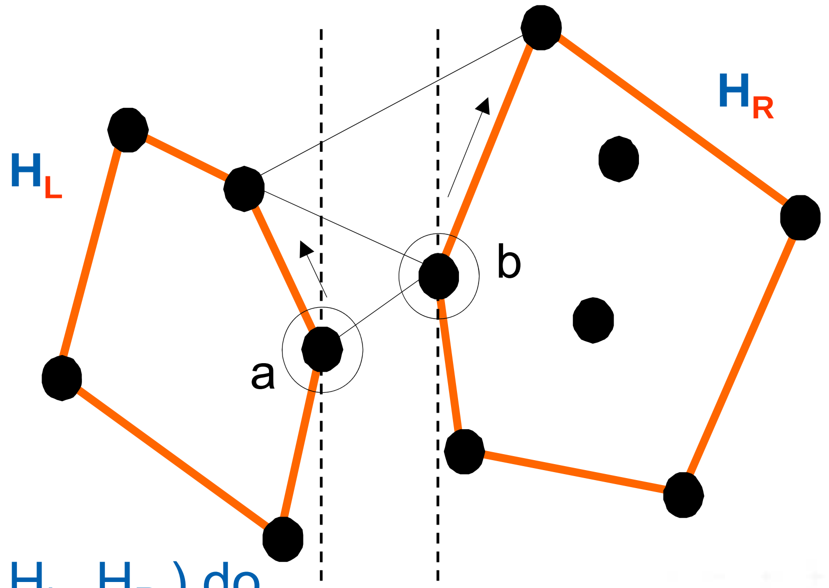
1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) $\geq$ 0
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



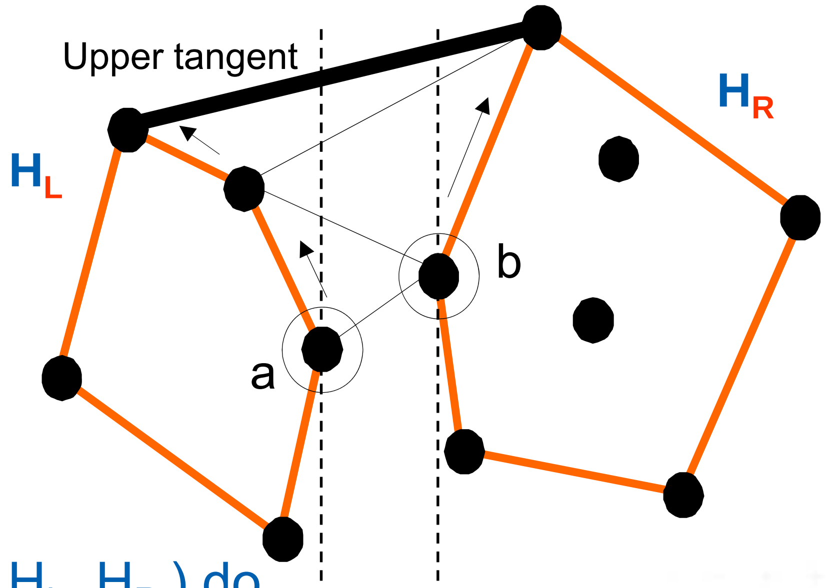
1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient$(a, b, a.succ) \geq 0$
which means *a.succ* is left from line *ab*

$$m = |H_L| + |H_R| \leq |L| + |R| \quad \Rightarrow \text{Upper Tangent: } O(m) = O(n)$$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*

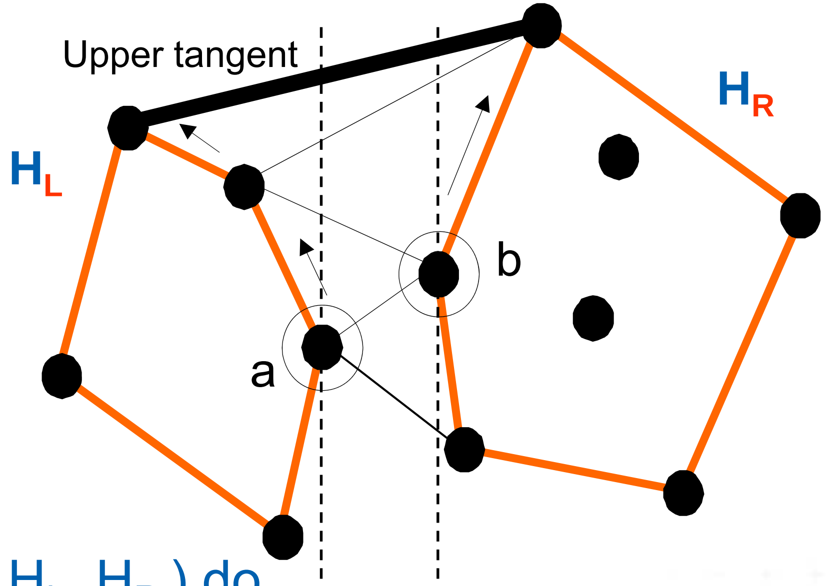


1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) ≥ 0
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Search for upper tangent (lower is symmetrical)



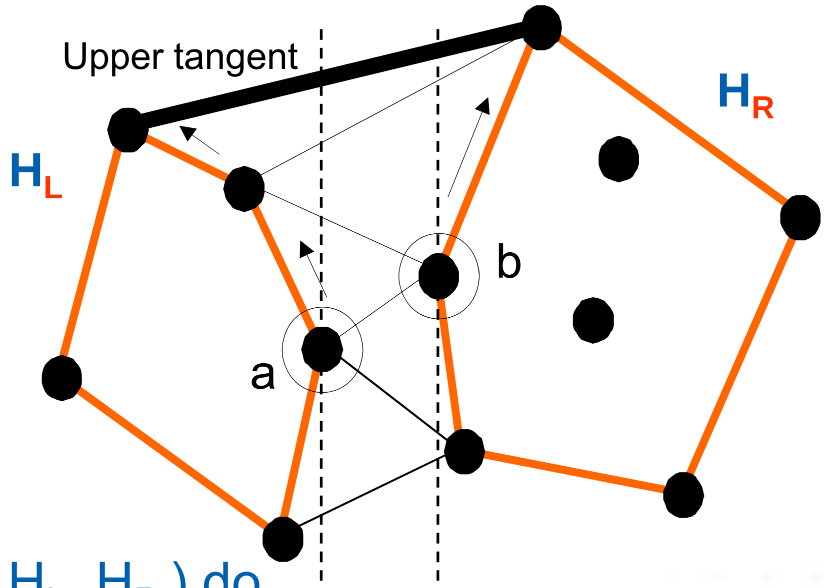
Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*

1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) ≥ 0
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Search for upper tangent (lower is symmetrical)



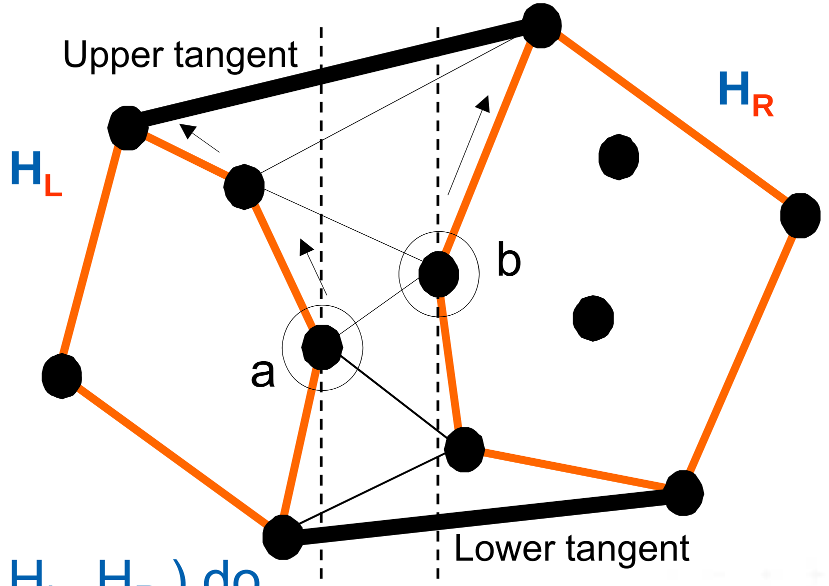
Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*

1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) ≥ 0
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: O(*m*) = O(*n*)

Search for upper tangent (lower is symmetrical)



Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*

1. a = rightmost H_L
2. b = leftmost H_R
3. while(ab is not the upper tangent for H_L, H_R) do
4. while(ab is not the upper tangent for H_L) *a = a.succ* // move CCW
5. while(ab is not the upper tangent for H_R) *b = b.pred* // move CW
6. Return *ab*

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) ≥ 0
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Convex hull by D&C complexity

- Initial sort O(n log(n))
- Function hull()
  - Upper and lower tangent O(n)
  - Merge hulls O(1)
  - Discard points between tangents O(n)

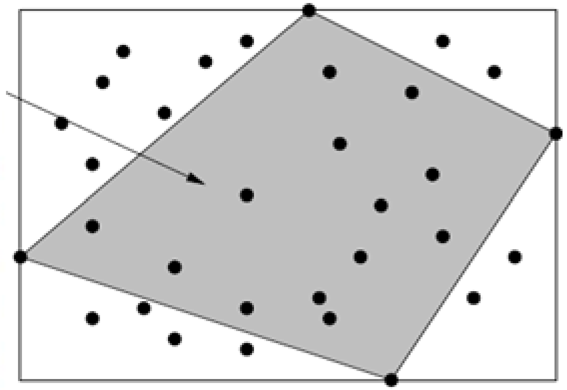
  $\left.\begin{array}{l} O(n) \\ O(1) \\ O(n) \end{array}\right\} O(n)$
- Overall complexity
  - Recursion

$$T(n) = \begin{cases} 1 & \ldots \text{ if } n \leq 3 \\ 2T(n/2) + O(n) & \ldots \text{ otherwise} \end{cases}$$

  - Overall complexity of CH by D&C: => O(n log(n))

Quick hull

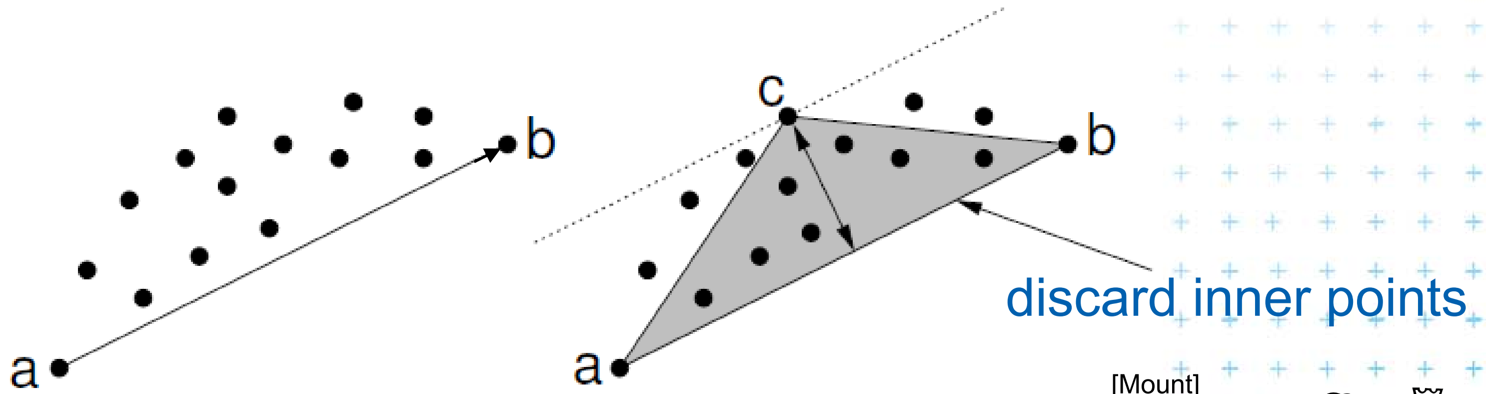
- A variant of Quick Sort
- O(n log n) expected time, max O(n^2)
- Principle
  - in praxis, most of the points lie in the interior of CH
  - E.g., for uniformly distributed points in unit square, we expect only O(log n) points on CH
- Find extreme points (parts of CH) quadrilateral, discard inner points
  - Add 4 edges to temp hull T
  - Process points outside 4 edges



[Mount]

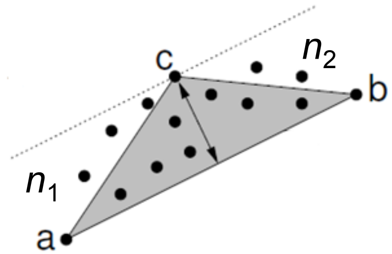
Process each of four groups of points outside

- For points outside *ab* (left from *ab* for clockwise CH)
  - Find point *c* on the hull – max. perpend. distance to *ab*
  - Discard points inside triangle *abc* (right from the edges)
  - Split points into two subsets
    - outside *ac* (left from *ac*) and outside *cb* (left from *cb*)
  - Replace edge *ab* in *T* by edges *ac* and *cb*
  - Process points outside *ac* and *cb* recursively



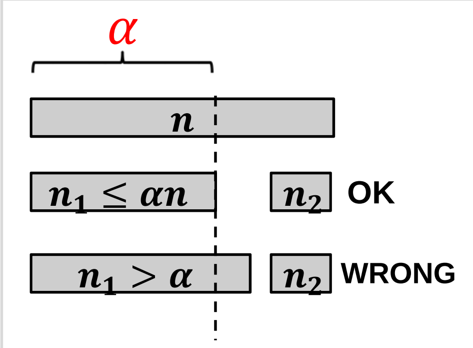
[Mount]

Quick hull complexity



- *n* points remain outside the hull
- $T(n)$ = running time for such *n* points outside
  - O(*n*) - selection of splitting point *c*
  - O(*n*) - point classification to inside & (n_1+n_2) outside
  - $n_1+n_2 \leq n$
  - The running time is given by recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n_1) + T(n_2) & \text{where } n_1+n_2 \leq n \end{cases}$$

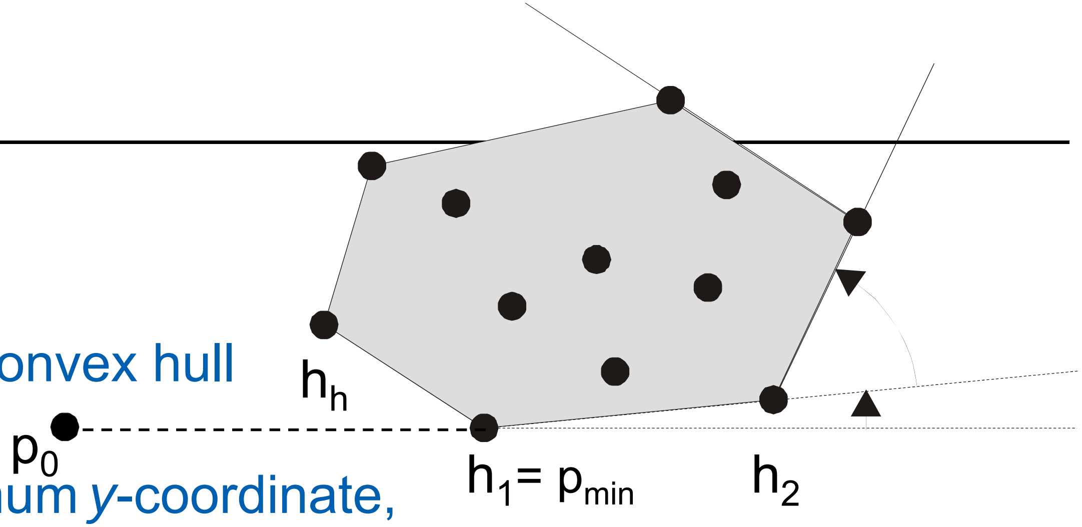


  - If evenly distributed that $\max(n_1, n_2) \leq \alpha n, 0 < \alpha < 1$ then solves as Quicksort to O(c*n* log *n*) where c=f(α) else O(n^2) for unbalanced splits
  - Output sensitive algorithm

Jarvis's March – selection by gift wrapping

- Variant of $O(n^2)$ selection sort
- Output sensitive algorithm
- $O(nh)$ … *h = number of points on convex hull*

Jarvis's March

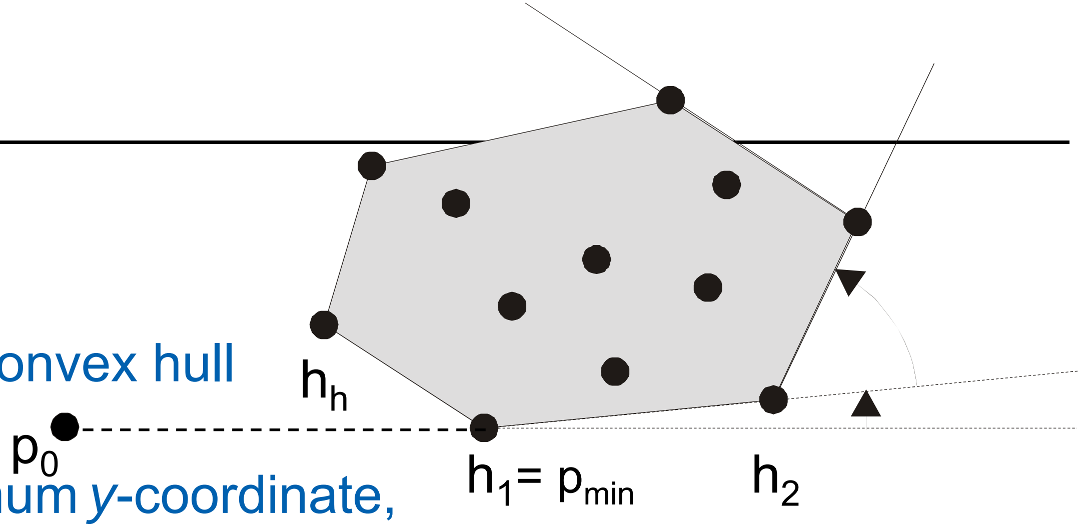


JarvisCH(points P)
Input: points p
Output: CCW points on the convex hull

1. Take point p_{min} with minimum *y*-coordinate,
   // p_{min} will be the first point in the hull – append it to the hull as h_1
2. Take a horizontal line, i.e., create temporary point $p_0 = (-\infty, h_1.y)$
3. j = 1
4. repeat
5. Rotate the line around h_j until it bounces to the nearest point q = p_q
   // compute the smallest angle by the "smallest orient(h_{j-1} , h_j, q)"
6. j++
   append the bounced nearest point q to the hull as next h_j
7. until ($q \neq p_{min}$)

Complexity: O(*n*) + O(*n*) * *h* => O(*h***n*)
good for low number of points on convex hull

Jarvis's March



JarvisCH(points P)

Input: points p

Output: CCW points on the convex hull

1. Take point p_{min} with minimum y-coordinate,
   // p_{min} will be the first point in the hull – append it to the hull as h_1
2. Take a horizontal line, i.e., create temporary point $p_0 = (-\infty, h_1.y)$
3. j = 1
4. repeat
5. Rotate the line around h_j until it bounces to the nearest point q = p_q
   // compute the smallest angle by the "smallest orient(h_{j-1} , h_j, q)"
6. j++
   append the bounced nearest point q to the hull as next h_j
7. until ($q \neq p_{min}$)

Output sensitive algorithm

Complexity: O(n) + O(n) * h => O(h*n)

good for low number of points on convex hull

Output sensitive algorithm

- Worst case complexity analysis analyzes the worst case data
  - Presumes, that all (const. fraction of) points lie **on** the CH
  - The points are ordered along CH
    => We need sorting => $\Omega(n \log n)$ of CH algorithm
- Such assumption is rare
  - usually only much less of points are on CH
- Output sensitive algorithms
  - Depend on: input size n and the size of the output h
  - Are more efficient for small output sizes
- Reasonable time for CH is $O(n \log h)$

h = Number of points on the CH

Chan's algorithm

Cleverly combines Graham's scan and Jarvis's march algorithms

Goal is O(n log h) running time

– We cannot afford sorting of all points - $\Omega(n \log n)$

=> Idea: work on parts, limit the part sizes to polynomial h^c
the complexity does not change => $\log h^c = \log h$

– h is unknown – we get the estimation later

– Use estimation m, better not too high => $h \leq m \leq h^2$

1. Partition points P into r-groups of size m, $r = n/m$

– Each group take $O(m \log m)$ time - sort + Graham

– r-groups take $O(r\, m \log m) = O(n \log m)$ - Jarvis

Chan's algorithm

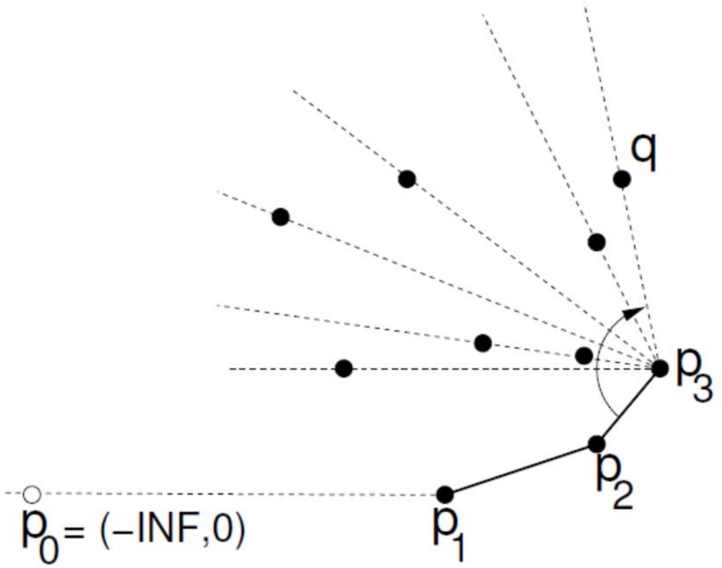
1. Partition points P into r-groups of size m, $r = n/m$
   - Each group take $O(m \log m)$ time - sort + Graham
   - r-groups take $O(r\, m \log m) = O(n \log m)$ – Jarvis

$$h \leq m \leq h^2$$

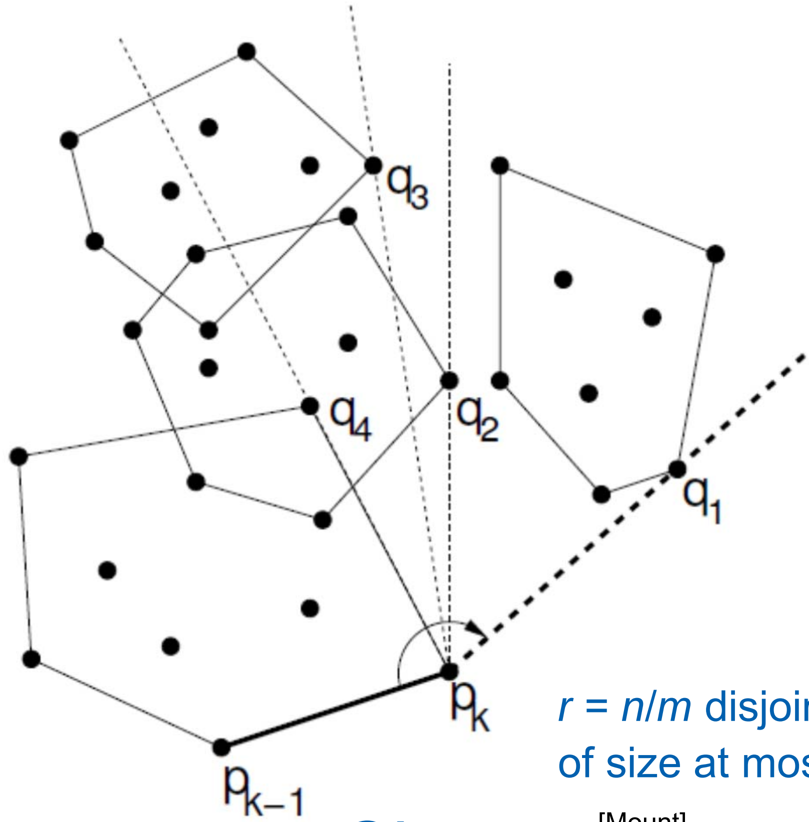
goal $O(n \log h)$

Merging of *m* parts in Chan’s algorithm

2. Merge *r*-group CHs as “fat points”
   - Tangents to convex *m*-gon can be found in O(log *m*) by binary search



Jarvis [Mount]



r = *n*/*m* disjoint subsets of size at most *m*

Chan [Mount]

Chan's algorithm complexity

- h points on the final convex hull

  => at most h steps in the Jarvis march algorithm

  – each step computes r-tangents, O(log m) each

  – merging together O(hr log m)

  r-groups of size m, $r = n/m$

- Complete algorithm O(n log h)

  – Graham's scan on partitions $O(r\ m \log m) = O(n \log m)$

  – Jarvis Merging: O(hr log m) $= O(h\ n/m \log m)$, …4a)

  $h \leq m \leq h^2$ $= O(n \log m)$

  – Altogether $\underline{O(n \log m)}$

  – How to guess m? *Wait!*

Chan's algorithm complexity

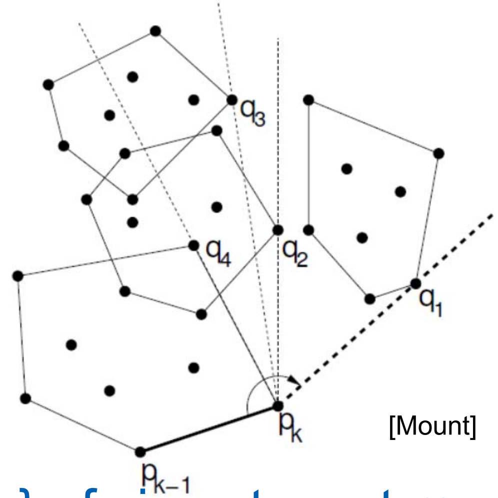
- *h* points on the final convex hull
  => at most *h* steps in the Jarvis march algorithm
  - each step computes *r*-tangents, O(log *m*) each
  - merging together O(*hr* log *m*)

r-groups of size *m*, $r = n/m$

- Complete algorithm O(*n* log *h*)
  - Graham's scan on partitions $O(r\, m \log m) = O(n \log m)$
  - Jarvis Merging: O(*hr* log *m*) $= O(h\, n/m \log m)$, …4a)
    $h \leq m \leq h^2$ $= O(n \log m)$
  - Altogether $\underline{O(n \log m)}$
  - How to guess *m*? *Wait!*

1) use m as an estimation of h 2) if it fails, increase m

Chan's algorithm for known *m*



[Mount]

PartialHull(*P*, *m*)
Input: points P
Output: group of size *m*

1. Partition *P* into $r = \lceil n/m \rceil$ disjoint subsets $\{p_1, p_2, \ldots, p_r\}$ of size at most *m*
2. for *i=1 to r do*
   a) Convex hull by GrahamsScan(P_i), store vertices in ordered array
3. let p_1 = the bottom most point of P and $p_0 = (-\infty, p_1.y)$
4. for *k* = 1 to *m* do // compute merged hull points
   a) for *i* = 1 to *r* do // angle to all *r* subsets => points q_i **O(log *m*)**
      Compute the point $q_i \in P$ that maximizes the angle $\angle\, p_{k-1}, p_k, q_i$
   b) let p_{k+1} be the point $q \in \{q_1, q_2, \ldots, q_r\}$ that maximizes $\angle\, p_{k-1}, p_k, q$
      (p_{k+1} is the new point in CH)
   c) if $p_{k+1} = p_1$ then return $\{p_1, p_2, \ldots, p_k\}$
5. return "Fail, *m* was too small"

Jarvis

Chan's algorithm – estimation of *m*

ChansHull
Input: points P
Output: convex hull $p_1 \ldots p_k$

1. for t = 1, 2, … , $\lceil \lg \lg h \rceil$ do {
   a) let $m = \min(2^{2^t}, n)$
   b) L = PartialHull(P, m)
   c) if $L \neq$ "Fail, *m* was too small" then return L

   }

Sequence of choices of *m* are { 4, 16, 256,…, 2^{2^t} ,…, *n* } … squares

Example: for h = 23 points on convex hull of n = 57 points, the algorithm will try this sequence of choices of *m* { 4, 16, ~~256~~, 57 }

1. 4 and 16 will fail
2. 256 will be replaced by *n=57*

Complexity of Chan's Convex Hull?

- The worst case: Compute all t iterations
- t^{th} iteration takes $O(\, n \log \overset{m}{2^{2^{\wedge}t}}) = O(n\, 2^t)$
- Algorithm stops when $2^{2^{\wedge}t} \geq h$ => $t = \lceil \lg \lg h \rceil$
- All $t = \lceil \lg \lg h \rceil$ iterations take:

Using the fact that $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

$$\sum_{t=1}^{\lg \lg h} n2^t = n \sum_{t=1}^{\lg \lg h} 2^t \leq n2^{1+\lg \lg h} = 2n \lg h = O(n \log h)$$

2x more work in the worst case

Complexity of Chan's Convex Hull?

- The worst case: Compute all t iterations
- t^{th} iteration takes $O(n \log 2^{\overset{m}{2^{\wedge}t}}) = O(n\, 2^t)$
- Algorithm stops when $2^{2^{\wedge}t} \geq h \;\; \Rightarrow t = \lceil \lg \lg h \rceil$
- All $t = \lceil \lg \lg h \rceil$ iterations take:

Using the fact that $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

$$\sum_{t=1}^{\lg \lg h} n 2^t = n \sum_{t=1}^{\lg \lg h} 2^t \leq n 2^{1+\lg \lg h} = 2n \lg h = O(n \log h)$$

one iteration

2x more work in the worst case

Complexity of Chan's Convex Hull?

- The worst case: Compute all *t* iterations
- t^{th} iteration takes $O(\,n \log \overset{m}{2^{2^{\wedge}t}}) = O(n\, 2^t)$
- Algorithm stops when $2^{2^{\wedge}t} \geq h \;\Rightarrow t = \lceil \lg \lg h \rceil$
- All $t = \lceil \lg \lg h \rceil$ iterations take:

one iteration

Using the fact that $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

t iterations

$$\sum_{t=1}^{\lg \lg h} n2^t = n \sum_{t=1}^{\lg \lg h} 2^t \leq n2^{1+\lg \lg h} = 2n \lg h = O(n \log h)$$

2x more work in the worst case

Complexity of Chan's Convex Hull?

one iteration

- The worst case: Compute all *t* iterations
- t^{th} iteration takes O(n log $\overset{m}{2^{2^{\wedge}t}}$) = O($n\ 2^t$)
- Algorithm stops when $2^{2^{\wedge}t} \geq h$ => $t = \lceil \lg\lg h \rceil$
- All $t = \lceil \lg\lg h \rceil$ iterations take:

Using the fact that $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

t iterations

$$\sum_{t=1}^{\lg\lg h} n2^t = n\sum_{t=1}^{\lg\lg h} 2^t \leq n2^{\overset{1+k}{1+\lg\lg h}} = 2n\lg h = O(n\log h)$$

2x more work in the worst case

Conclusion in 2D

- Graham's scan: $O(n \log n)$, $O(n)$ for sorted pts
- Divide & Conquer: $O(n \log n)$
- Quick hull: $O(n \log n)$, max $O(n^2)$ ~ distrib.
- Jarvis's march: $O(hn)$, max $O(n^2)$ ~ pts on CH
- Chan's alg.: $O(n \log h)$ ~ pts on CH

asymptotically optimal

but

constants are too high to be useful

References

- [Berg] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: Computational Geometry: *Algorithms and Applications*, Springer-Verlag, 3rd rev. ed. 2008. 386 pages, 370 fig. ISBN: 978-3-540-77973-5, Chapter 5, http://www.cs.uu.nl/geobook/
- [Mount] Mount, D.: *Computational Geometry Lecture Notes for Fall 2016*, University of Maryland, Lectures 3 and 4. http://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf
- [Chan] Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions., *Discrete and Computational Geometry*, 16, 1996, 361-368. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.389