Outline

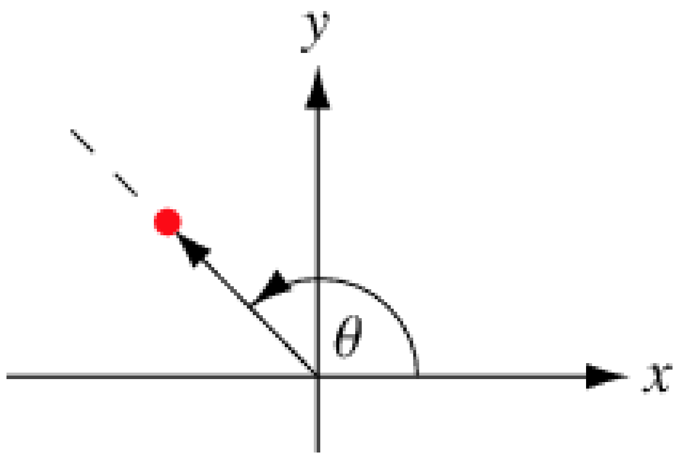
- Introduction: Background & Definition of convex hull
- Three algorithms
  - Graham’s Scan
  - Jarvis March
  - Chan’s algorithm
- Proof of these algorithms
- Application

Introduction

Shen Shiqi

Polar angle

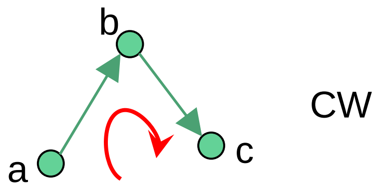
In the plane, the polar angle θ is the counterclockwise angle from x-axis at which a point in the x-y plane lies.

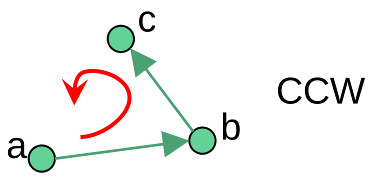


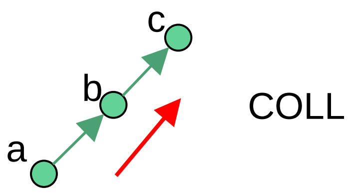
Orientation

- Calculating Orientation
  - **Three kinds of orientation for three points (a, b, c)**
    - Clockwise (CW): right turn
    - Counterclockwise (CCW): left turn
    - Collinear (COLL): no turn
  - **The orientation can be characterized by the sign of the determinant △(a,b,c)**
    - If △(a,b,c)<0 ⇒ **clockwise**
    - If △(a,b,c)=0 ⇒ **collinear**
    - If △(a,b,c)>0 ⇒ **counterclockwise**

$$\Delta(a,b,c)=\begin{vmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ 1 & 1 & 1 \end{vmatrix}$$





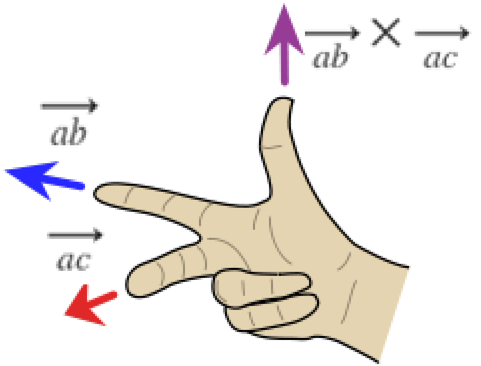


Why?

- Cross product
  - **Direction**: right hand rule
  - **Magnitude**: the area of the parallelogram that the vectors span

$$\Delta(a,b,c)=\begin{vmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ 1 & 1 & 1 \end{vmatrix}=\begin{vmatrix} x_b-x_a & x_c-x_a \\ y_b-y_a & y_c-y_a \end{vmatrix}=\|\overrightarrow{ab}\|\,\|\overrightarrow{ac}\|\sin\theta$$

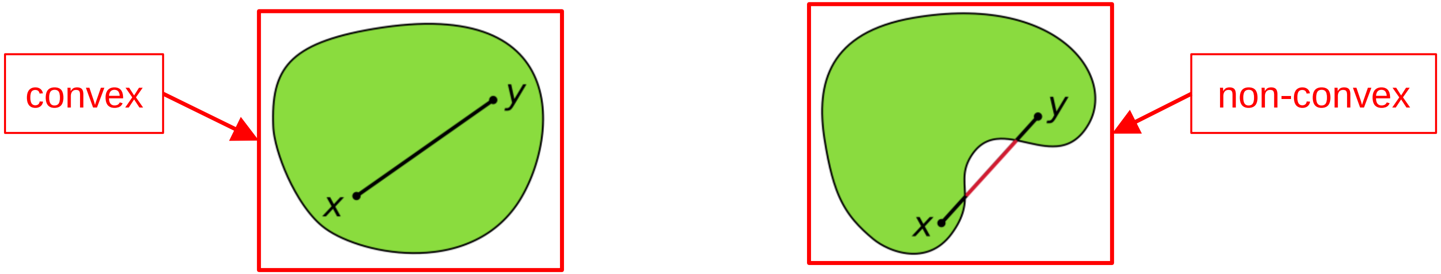
$\|\overrightarrow{ab}\|\,\|\overrightarrow{ac}\|\sin\theta$ is the scalar of $\overrightarrow{ab}\times\overrightarrow{ac}$



Convexity

A shape or set is convex :

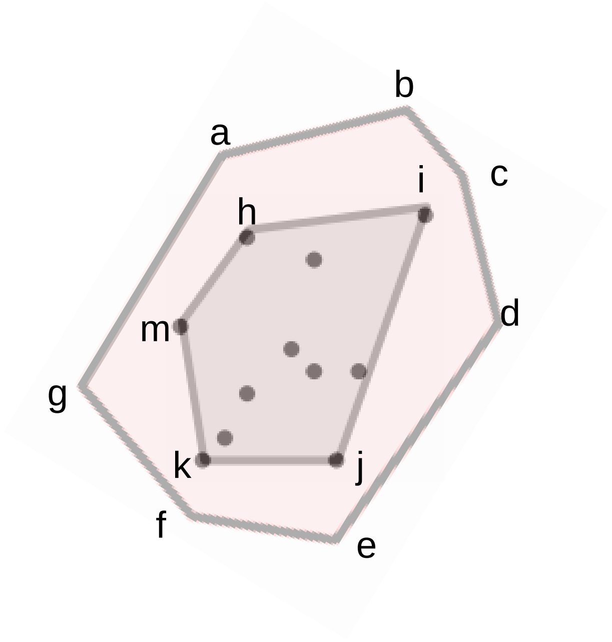
If for any two points that are part of the shape, the whole connecting line segment is also part of the shape.



Convex hull

Convex hull of a point set P, CH(P):

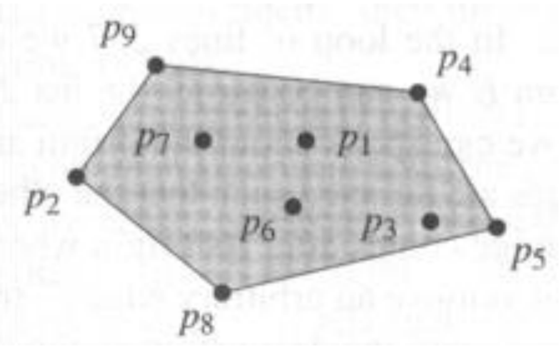
- Smallest convex set containing P
- Intersection of all convex sets containing P



Convex hull problem

Give an algorithm that computes the convex hull of any given set of n points in the plane efficiently.

- Inputs: location of n points
- Outputs: a convex polygon ⇒ a sorted sequence of the points, clockwise (CW) or counterclockwise (CCW) along the boundary



inputs= a set of point
$p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9$

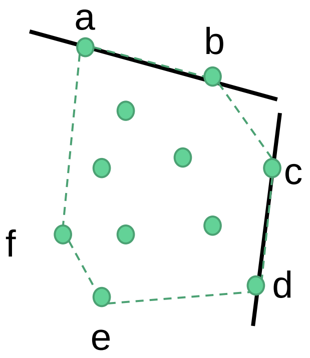
outputs= representation of the convex hull
p_4,p_5,p_8,p_2,p_9

How to develop an algorithm?

Properties:

- The vertices of the convex hull are always points from the input

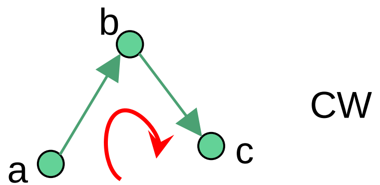
- The supporting line of any convex hull edge has all input points to one side

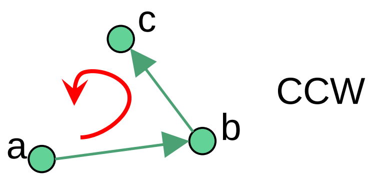


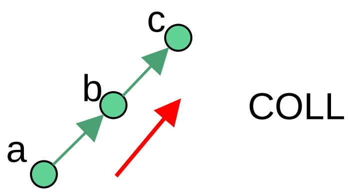
Orientation

- Calculating Orientation
  - **Three kinds of orientation for three points (a, b, c)**
    - Clockwise (CW): right turn
    - Counterclockwise (CCW): left turn
    - Collinear (COLL): no turn
  - **The orientation can be characterized by the sign of the determinant △(a,b,c)**
    - If △(a,b,c)<0 ⇒ **clockwise**
    - If △(a,b,c)=0 ⇒ **collinear**
    - If △(a,b,c)>0 ⇒ **counterclockwise**

$$\Delta(a,b,c)=\begin{vmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ 1 & 1 & 1 \end{vmatrix}$$







Divide & Conquer

hull(S) = smallest convex set that contains S (points are stored in ccw order)

1. Sort all points of S with increasing x-coordinates
2. Algorithm conv(S, n)

   if n < 4, then trivially solved

   else

   DIVIDE: S_l & S_r

   RECUR: conv(S_l, n/2), conv(S_r, n/2)

   MERGE: combine hull(S_l) and hull(S_r)

Divide & Conquer

Algorithm conv(S, n)

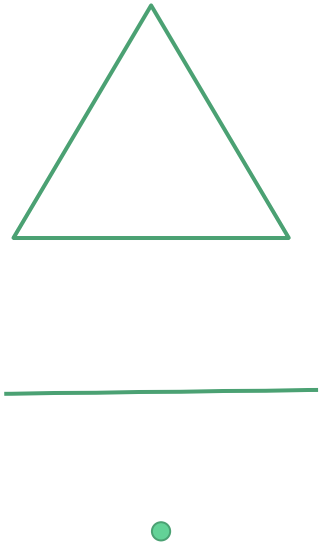
if $n < 4$, then trivially solved

else

DIVIDE: S_l & S_r

RECUR: conv(S_l, n/2), conv(S_r, n/2)

MERGE: combine hull(S_l) and hull(S_r)



Divide & Conquer

Algorithm conv(S, n)

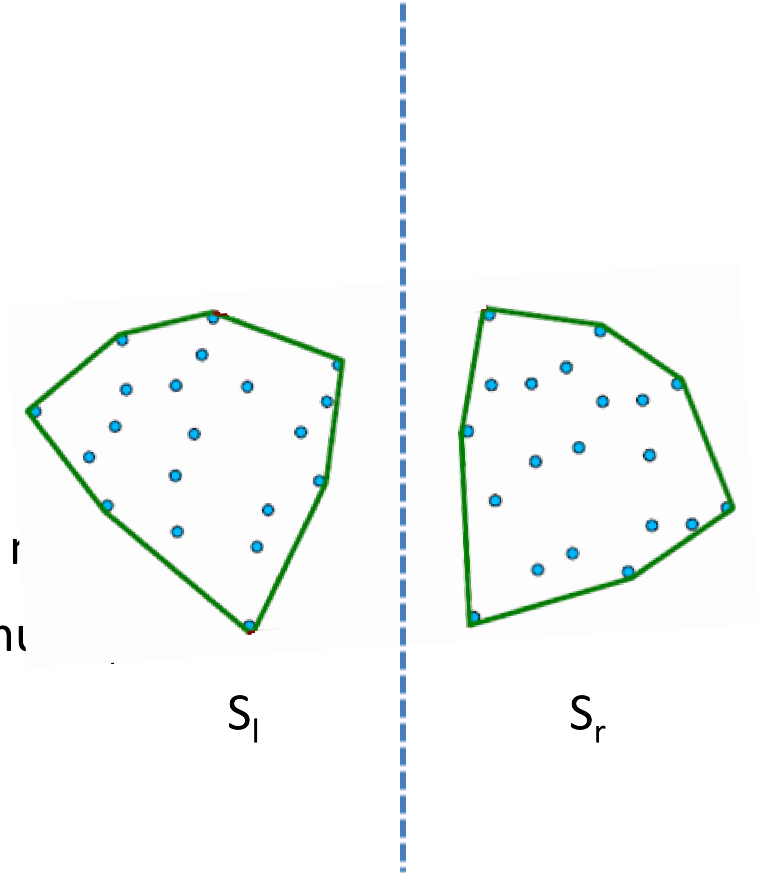
if n < 4, then trivially solved

else

DIVIDE: S_l & S_r

RECUR: conv(S_l, n/2), conv(S_r, n

MERGE: combine hull(S_l) and hu



Divide & Conquer

Algorithm conv(S, n)

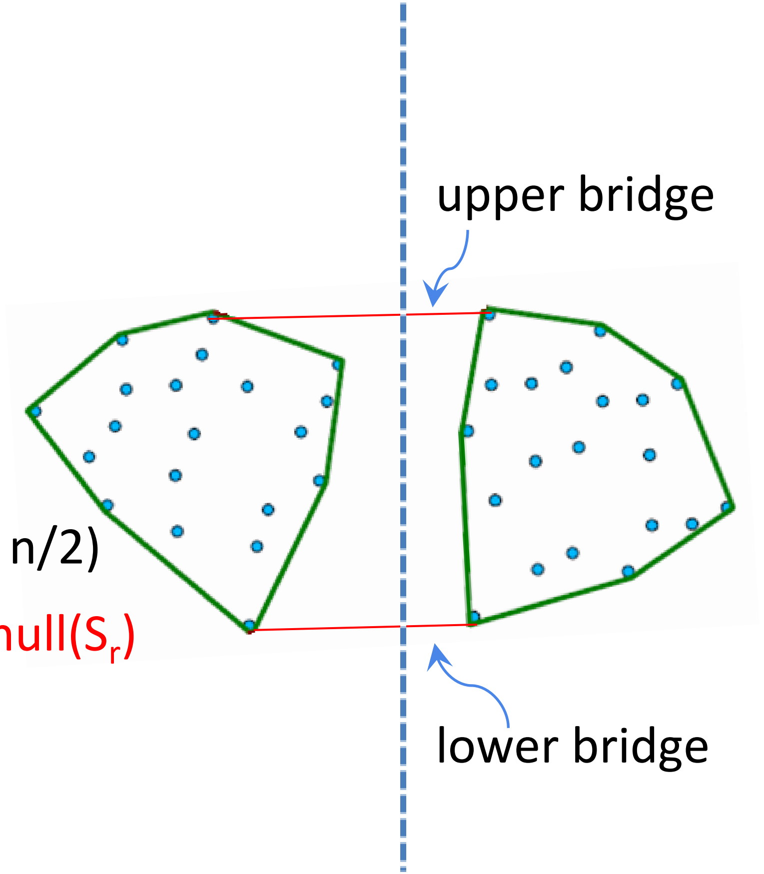
if n < 4, then trivially solved

else

DIVIDE: S_l & S_r

RECUR: conv(S_l, n/2), conv(S_r, n/2)

MERGE: combine hull(S_l) and hull(S_r)



Divide & Conquer

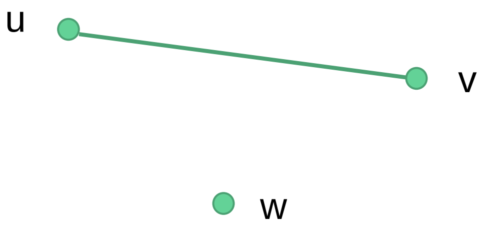
u : rightmost of hull(S_l)

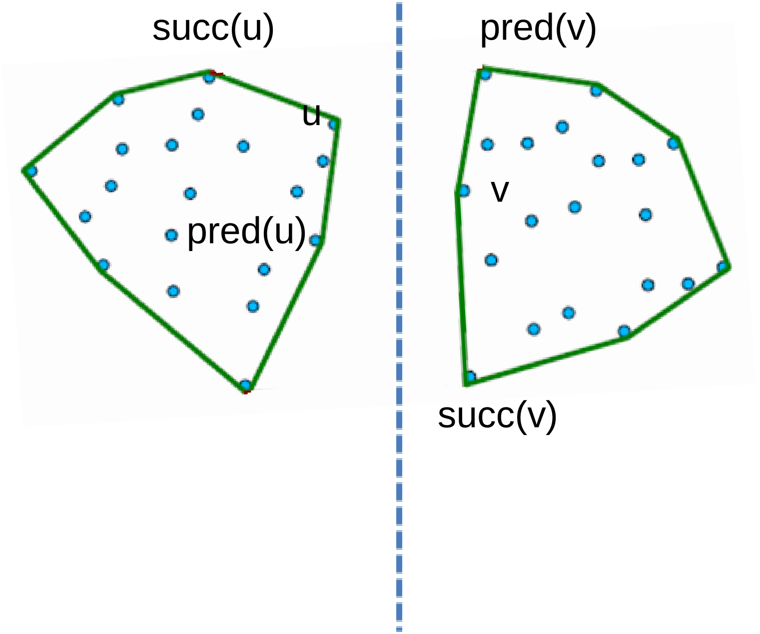
v : leftmost of hull(S_r)

succ : next point in ccw

prec : next point in cw

w lies below uv means cw(uvw)





Divide & Conquer

Procedure Find-lower-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

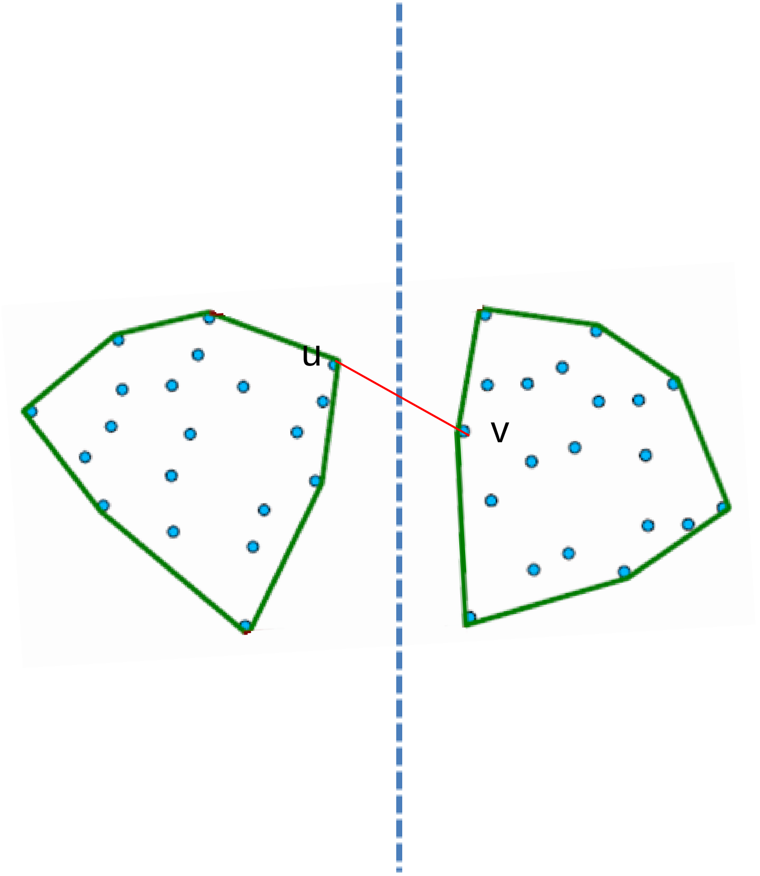
while either pred(u) **or** succ(v) lies below uv

if pred(u) lies below then u:=pred(u)

else v = succ(v)

endwhile

return uv



Divide & Conquer

Procedure Find-lower-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

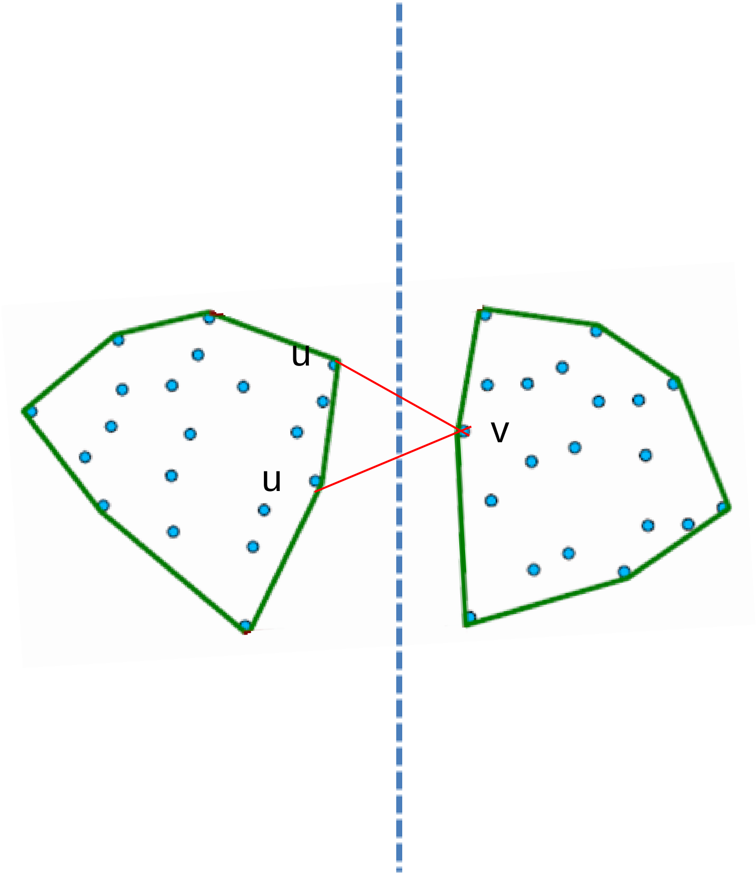
while either pred(u) **or** succ(v) lies below uv

  if pred(u) lies below then u:=pred(u)

  else v = succ(v)

endwhile

return uv



Divide & Conquer

Procedure Find-lower-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

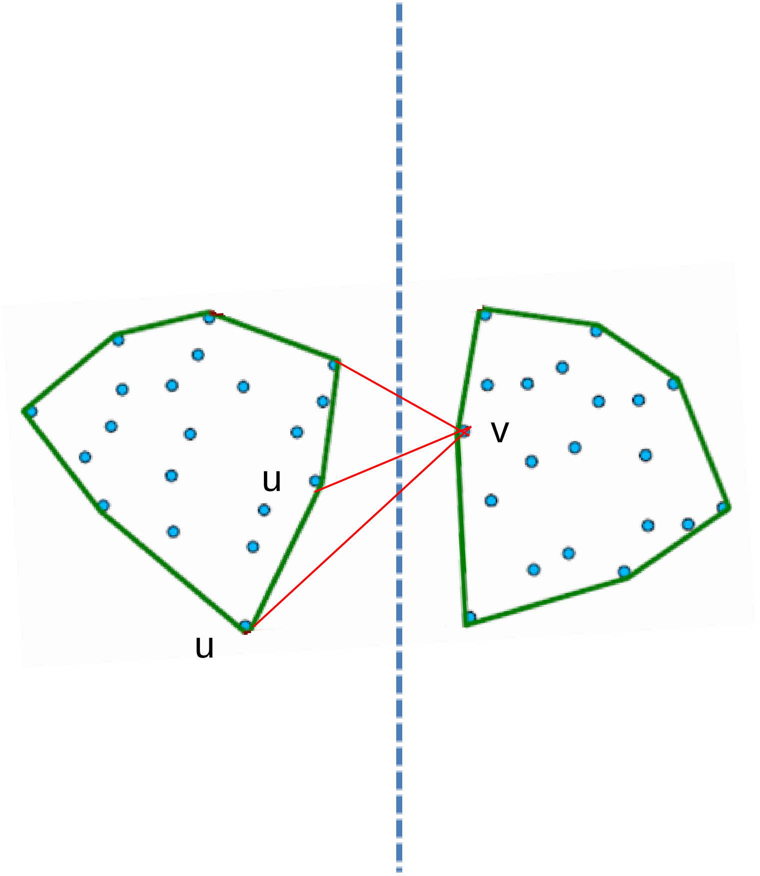
while either pred(u) **or** succ(v) lies below uv

if pred(u) lies below then u:=pred(u)

else v = succ(v)

endwhile

return uv



Divide & Conquer

Procedure Find-lower-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

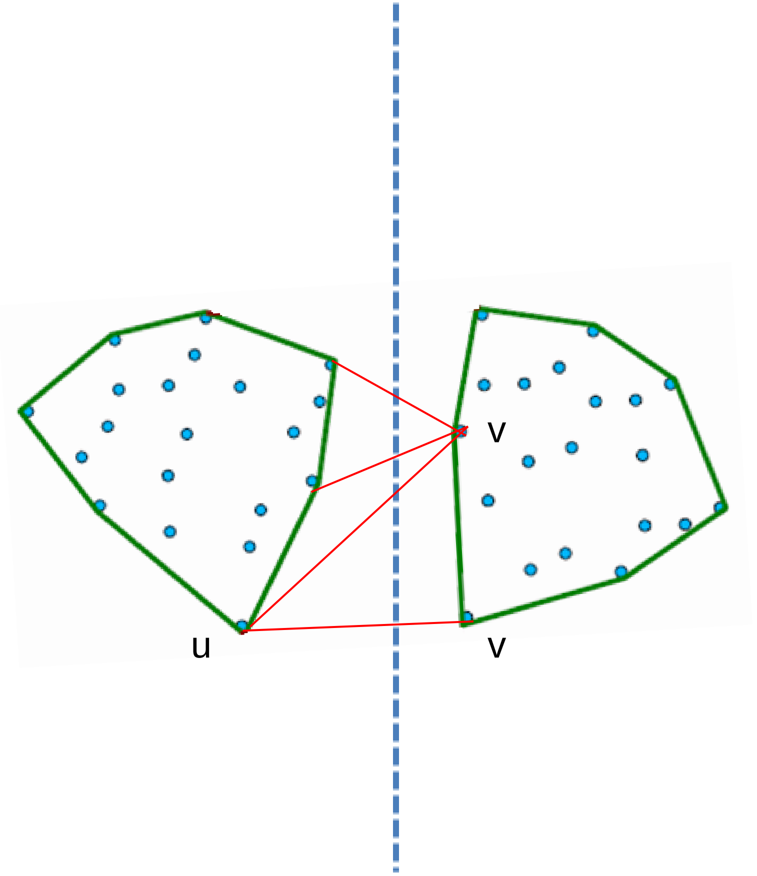
while either pred(u) **or** succ(v) lies below uv

  if pred(u) lies below then u:=pred(u)

  else v = succ(v)

endwhile

return uv



Divide & Conquer

Procedure Find-lower-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

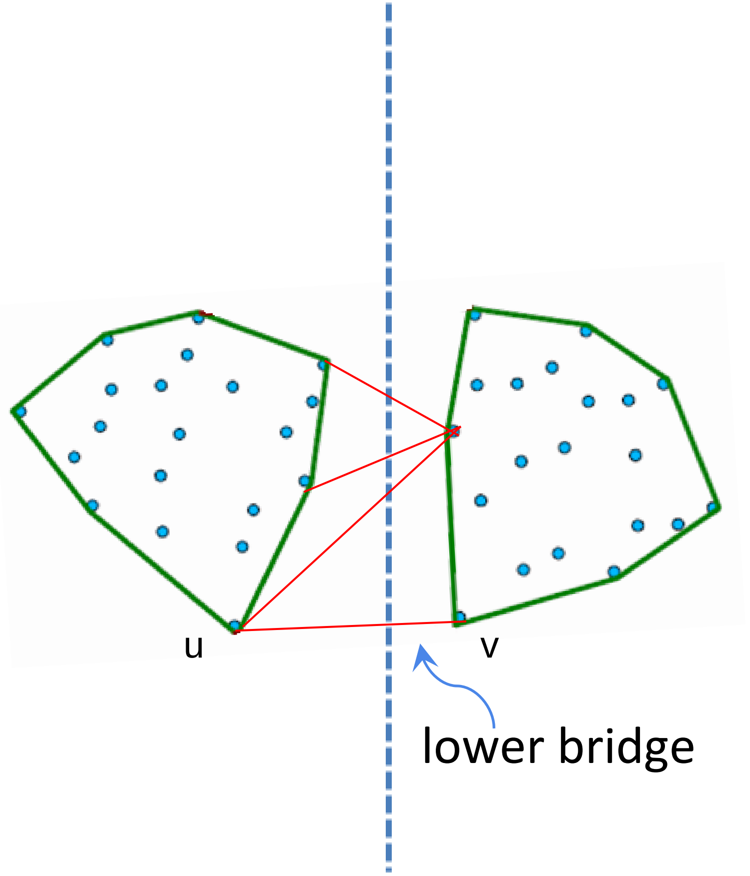
while either pred(u) **or** succ(v) lies below uv

  if pred(u) lies below then u:=pred(u)

  else v = succ(v)

endwhile

return uv



Divide & Conquer

Procedure Find-upper-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

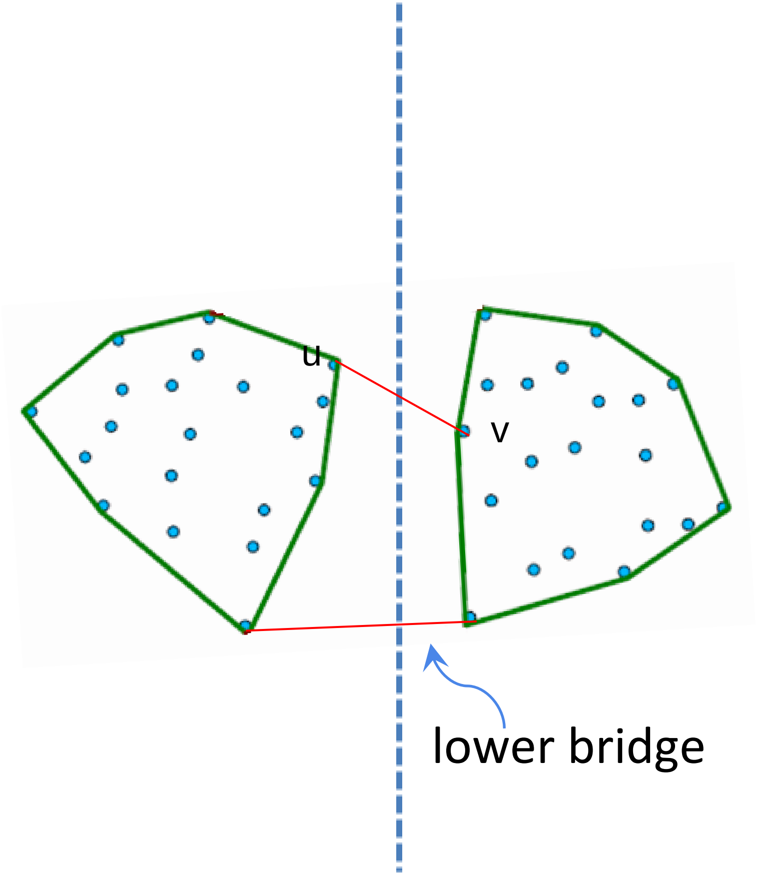
while either succ(u) **or** pred(v) lies below uv

  if succ(u) lies below then u:=succ(u)

  else v = prev(v)

endwhile

return uv



Divide & Conquer

Procedure Find-upper-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

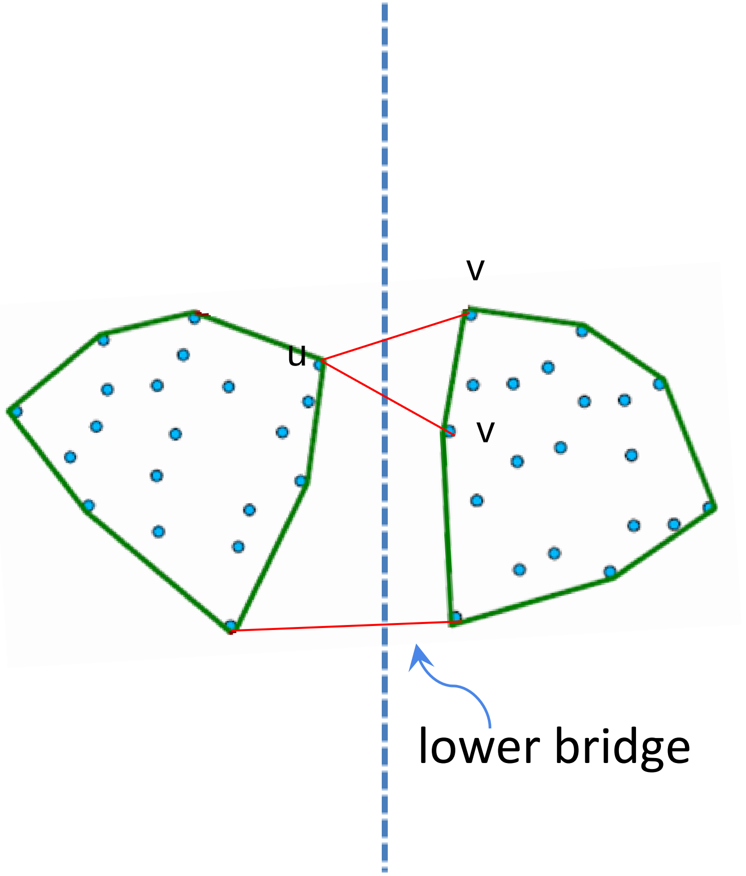
while either succ(u) **or** pred(v) lies below uv

if succ(u) lies below then u:=succ(u)

else v = prev(v)

endwhile

return uv



Divide & Conquer

Procedure Find-upper-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

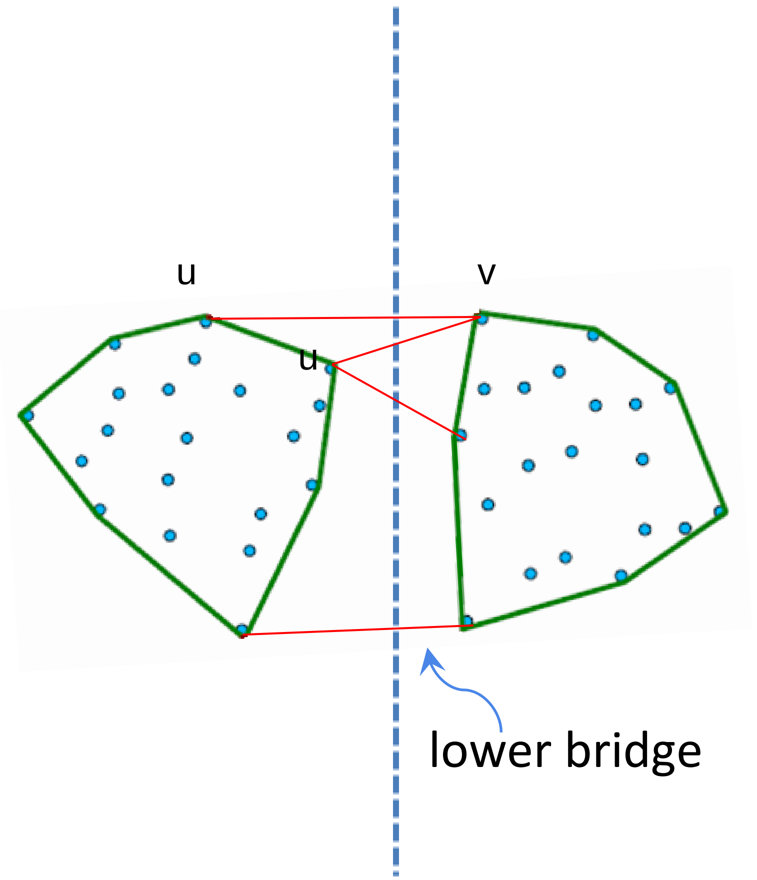
while either succ(u) **or** pred(v) lies below uv

if succ(u) lies below then u:=succ(u)

else v = prev(v)

endwhile

return uv



Divide & Conquer

Procedure Find-upper-Bridge

u : rightmost of hull(S_l)

v : leftmost of hull(S_r)

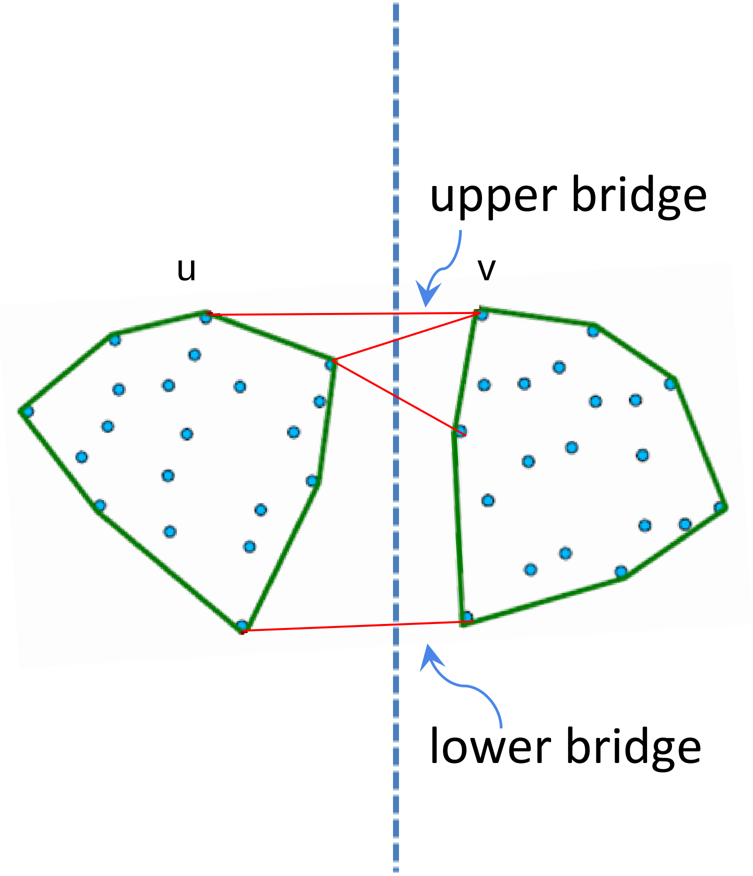
while either succ(u) **or** pred(v) lies below uv

    if succ(u) lies below then u:=succ(u)

    else v = prev(v)

endwhile

return uv



Divide & Conquer

Algorithm conv(S, n)

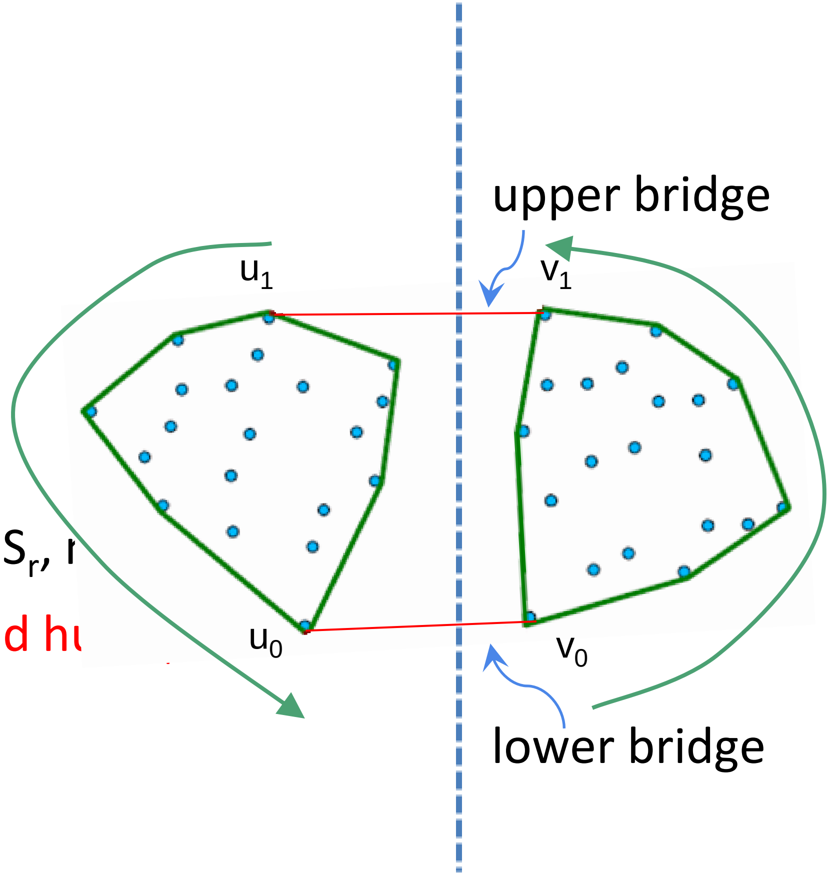
if n < 4, then trivially solved

else

DIVIDE: S_l & S_r

RECUR: conv(S_l, n/2), conv(S_r,

MERGE: combine hull(S_l) and hu



Divide & Conquer

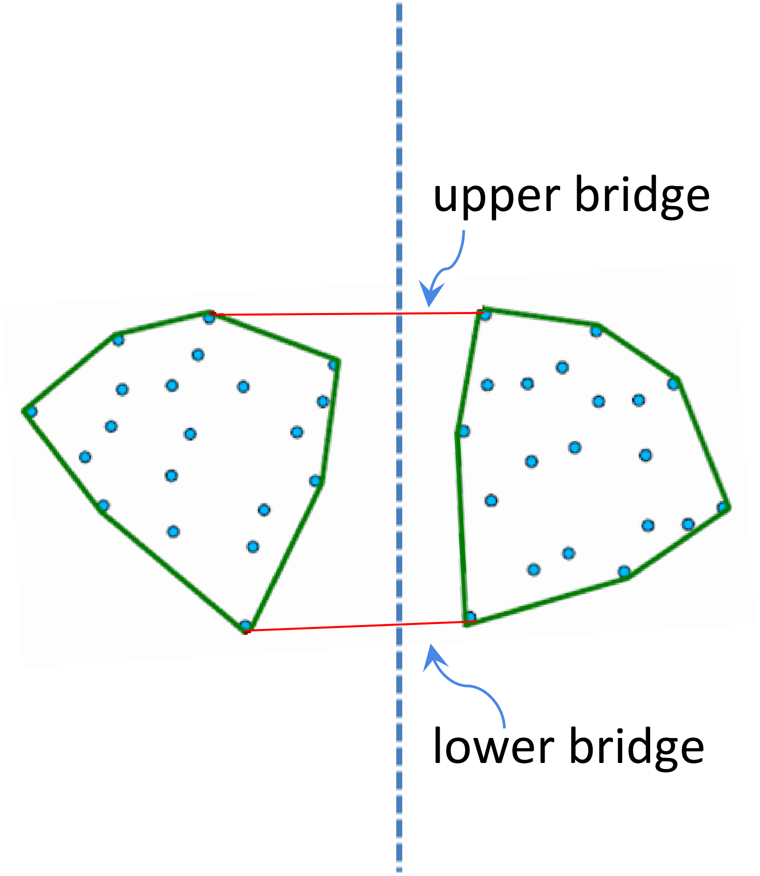
$T(n) = 2T(n/2) + cn$

$2T(n/2)$: Recursion

cn: Combine conv(S_l) and conv(S_r)

Master theorem

Easily calculate the running time without doing an expansion



Divide & Conquer

$T(n) = 2T(n/2) + cn$

$a=2, b=2$

$f(n) = cn \in \Theta(n)$

$d=1$

$T(n)=\Theta(n \log n)$ $(a=b^d=2)$

Theorem (Master Theorem)

Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Applications

- Fundamental content to computational geometry;
  (e.g. Voronoi diagrams)
- Computer visualization, ray tracing;
  (e.g. video games, replacement of bounding boxes)
- Path finding;
  (e.g. embedded AI of Mars mission rovers)
- Visual pattern matching;
  (e.g. detecting car license plates)
- Verification methods;
  (e.g. bounding of Number Decision Diagrams)

How to perform the binary search in Chan's

```
i = 0
j = t/2 {t is the number of vertices of CH(P_i)}
if isMax(i) then
  return x_i
end if
if isMax(j) then
  return x_j
end if
while true do
  if NOT containsMax(i, j) then
    swap(i, j)
  end if
  k = index of the middle vertex in {x_i, ..., x_j}
  if isMax(k) then
    return x_k
  end if
  if containsMax(i, k) then
    j = k
  else
    i = k
  end if
end while
```

Intuition of binary search in Chan's

So the above algorithm first checks whether it is the clockwise interval $\{x_i, ..., x_j\}$ or the clockwise interval $\{x_j, ..., x_i\}$ that contains the vertex that makes the maximum angle with $p_k p_{k-1}$ before it takes the midpoint of the interval to do a binary search.

The function $ismax(i)$ is simple to implement: check that x_i makes a larger angle with $p_k p_{k-1}$ than both x_{i-1} and x_{i+1}.

We need to implement the function $containsMax(i, j)$. By $x_a < x_b$, we mean the angle made by x_a with $p_k p_{k-1}$ is less than the angle made by x_b with $p_k p_{k-1}$.

Intuition of binary search in Chan's

If $x_{i-1} < x_i < x_{i+1}$ and $x_{j-1} > x_j > x_{j+1}$, then we know that the vertex that makes the largest angle in between x_{i+1} and x_{j-1} (in the clockwise boundary of P). Hence, $containsMax(i, j)$ is true in this case. (On the other hand, if $x_{i-1} > x_i > x_{i+1}$ and $x_{j-1} < x_j < x_{j+1}$, then $containsMax(i, j)$ is false.)
Suppose it was the case that both $x_{i-1} > x_i > x_{i+1}$ and $x_{j-1} > x_j > x_{j+1}$ (see the figure). How can we distinguish between the case when the interval (i, j) contains the maximum or the interval (j, i) contains the maximum? (Note that whichever interval contains the maximum contains the minimum also.)
So we need to distinguish between the case (i) when the interval (i, j) is monotonically decreasing (and so (i, j) does not contain the maximum) and the case (ii) when both the maximum and its minimum were attained in the interval (i, j). But case (ii) corresponds exactly to the case that the interval (j, i) is monotonically decreasing.

Intuition of binary search in Chan's

So we have to distinguish between the case when (i, j) is monotonically decreasing ($containsMax(i, j)$ is false) and the case when (j, i) is monotonically decreasing ($containsMax(i, j)$ is true). This is easy to do. In the first case $x_i > x_j$ and in the second case $x_i < x_j$.
So, $containsMax(i, j)$ is true in this case if $x_i < x_j$. By a symmetrical argument, if it was the case that both $x_{i-1} < x_i < x_{i+1}$ and $x_{j-1} < x_j < x_{j+1}$, then $containsMax(i, j)$ is true in this case if $x_i > x_j$.
There are 2 other cases corresponding to x_i was the minimum (i.e., $x_{i-1} > x_i < x_{i+1}$) and x_j was the minimum (i.e., $x_{j-1} > x_j < x_{j+1}$). In the first case, $containsMax(i, j)$ is true if $x_{j-1} > x_j > x_{j+1}$. In the second case, $containsMax(i, j)$ is true if $x_{i-1} < x_i < x_{i+1}$.
So, there are totally 5 cases to be considered and if any one of them is true, then $containsMax(i, j)$ is true, else $containsMax(i, j)$ is false.
It is easy to see that the running time of this algorithm is $O(\log t)$ and since $t \leq m$, it is $O(m)$.