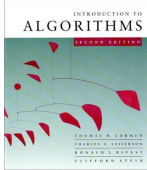


CS 445



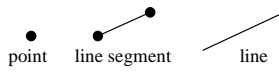
Computational Geometry

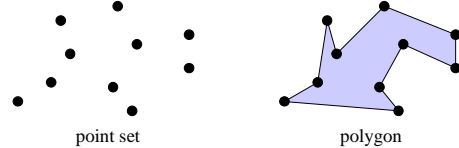
Alon Efrat

Slides courtesy of Charles E. Leiserson,
Erik Demaine Carola Wenk

Computational geometry

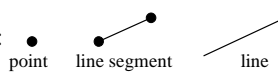
Algorithms for solving “geometric problems” in 2D and higher.

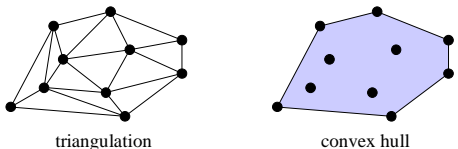
Fundamental objects:  point line segment line

Basic structures:  point set polygon

Computational geometry

Algorithms for solving “geometric problems” in 2D and higher.

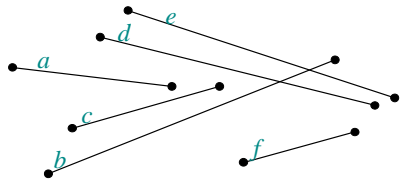
Fundamental objects:  point line segment line

Basic structures:  triangulation convex hull

Line-segment intersection

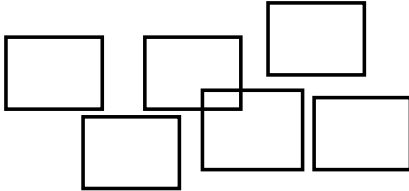
Given n line segments,

- does any pair intersect?
- Report all intersections pairs (not in this course)
 - Require a slight massaging of the algorithm,



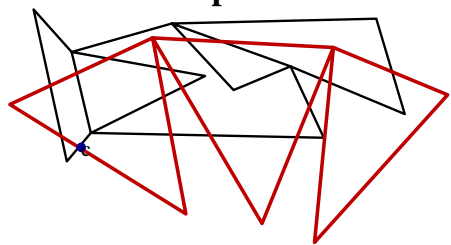
Obvious algorithm: $O(n^2)$ – checking all pairs.
Line-sweep – $O(n \log n)$ to decide if there is an intersection.

Applications 1: Checking VLSI design correctness



Need to decide if two components intersect (bug in design)

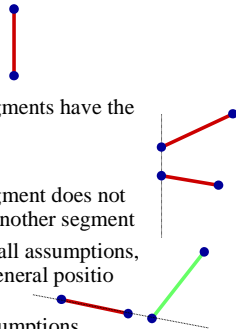
Applications 2: computing all intersection points



Applications: Map overlaying: compute all points at which a road crosses a river.

Assumptions-general position

- No vertical segment
- No two endpoints of segments have the same y-coordinate.
- The line containing a segment does not contain the endpoint of another segment
- If the segments satisfies all assumptions, we say that they are in general position



If the segments satisfies all assumptions, we say that they are in a **general position**.

Leftover from Data-structures

Given a set S of numbers, a standard balanced search tree supports $\text{insert}(x,S)$, $\text{delete}(x,S)$, $\text{find}(x,S)$

Each in $O(\log n)$ time (where $n=|S|$)

It can also support the operation $\text{succ}(x,S)$, defined as finding the smallest element of S which is strictly larger than S .

Examples $S=\{10,20,30\}$; $\text{succ}(-30,S)=10$,

$\text{succ}(10,S)=\text{succ}(12,S)=20$, $\text{succ}(30,S)=\text{succ}(40,S)=\text{UNDEFINED}$.

```
Succ(pNODE p, float x) { /*Returning node containing Succ(S, x) */
    if (p==NULL) then return UNDEFINED ;
    if (p->key ≤ x) then return Succ(p->right, x) ;
    else /*if (p->key > x) */ {
        if (p->left ≠NULL ) return Succ(p->left, x) ;
        else return p ;
    }
}
```

Leftover from Data-structures

Given a set S of numbers, a standard balanced search tree supports $\text{insert}(x,S)$, $\text{delete}(x,S)$, $\text{find}(x,S)$

Each in $O(\log n)$ time (where $n=|S|$)

It can also support the operation $\text{succ}(x,S)$, defined as finding the smallest element of S which is strictly larger than S .

Examples $S=\{10,20,30\}$; $\text{succ}(-30,S)=10$,

$\text{succ}(10,S)=\text{succ}(12,S)=20$, $\text{succ}(30,S)=\text{succ}(40,S)=\text{UNDEFINED}$.

Leftover from Data-structures

```
Succ(pNODE p, float x) {
```

```
    p = root;
```

```
    x_tmp = INFINITY ; /* x_tmp - temporally value */
```

```
    while( p ≠ NULL ) {
```

```
        if ( p->key ≤ x ) p = p->right;
```

```
        else {
```

```
            x_tmp = min(x_tmp, p->key) ;
```

```
            p = p->left ;
```

```
        }
```

```
    }
    return x_tmp;
```

```
}
```

2-segment intersection

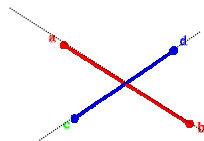
- Test whether segments (a,b) and (c,d) intersect. **How do we do it?**
- Method 1: Write down the **equations of the lines through the segments:**

- Express the line through the first segment as $y=mx+n$

- Express the line through the second segment as $y=m'x+n'$

- Find intersection point between the lines.

- Check if the intersection point lies on the segments. $O(1)$ time.



Sweep-line algorithm

Sweep a vertical line from left to right

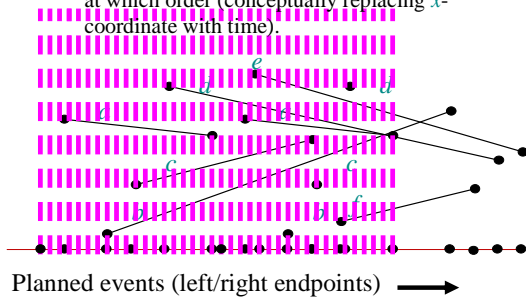
The line "knows" which segment it intersect and at which order (conceptually replacing x -coordinate with time).



Planned events (left/right endpoints) →

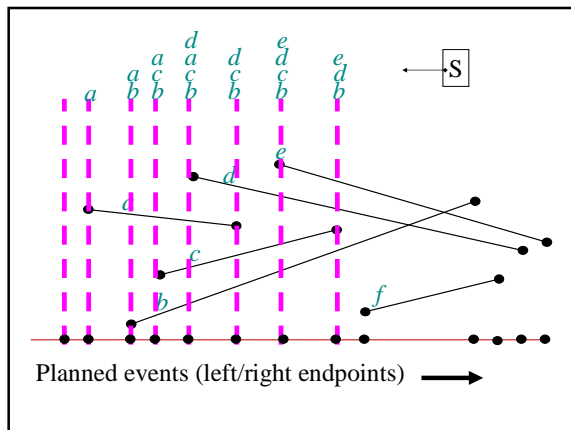
Sweep-line algorithm

Sweep a vertical line from left to right
The line “knows” which segment it intersect and at which order (conceptually replacing x -coordinate with time).

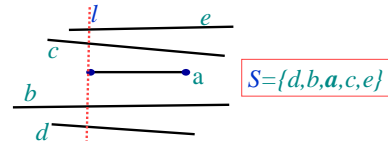


Sweep-line algorithm

- Sweep a vertical line from left to right (conceptually replacing x -coordinate with time).
- Maintain the **status** - a dynamic set S of the segments that intersect the sweep line, ordered (tentatively) by y -coordinate of intersection.
 - (so the lowest segment appears first one the list)
- The status is changed only when
 - new segment is encountered (**left** endpoints),
 - existing segment finishes (**right** endpoint)
 - **Event points** are therefore segment endpoints.



The status of the linesweep



The **status** S is the list of segments that linesweep l intersects (in the order from bottom to top).

Definition: an **event** happens when l start/stop intersecting a segment.

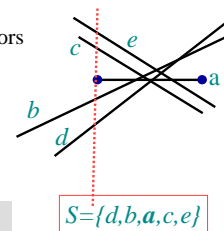
Note: the status is not changed between events, so l can jump from an event to the next event.

Algorithm - overview

- Sweep with vertical line l from left to right (I.e. scan the endpoints in increasing ordered of x -coordinates)
- Each time that l meets an endpoint –
 1. Update the status
 2. Check segments intersection as described in the next slides.

Left endpoint event:

- For a **left** endpoint of segment s :
 - Add segment s to the status S .
 - Check for intersection between s and its **neighbors** in S .
- (Will later explain how the neighbors are found)



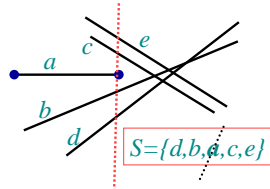
Example: a is checked for intersection with c and intersection with b .

Right endpoint event

For a **right** endpoint of segment s :

- Delete segment s from dynamic set S .
- Check for intersection between neighbors s and its **neighbors** in S .

Example: c is checked for intersection with b .



Theorem: If there is an intersection point, the algorithm finds it.

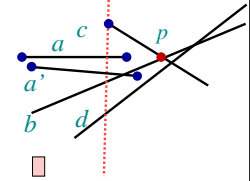
Proof: Let p be the leftmost intersection point.

Consider the last event before (to the left of) p , at which c or b are born

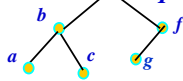
If they are not neighbors on l , it is because another segment, say a separates between them.

But then either a has a right endpoint to the left of p and then c and b become neighbors.

Or a' intersects b or c at a point to the left of p , contradiction to p be the leftmost point.



Maintaining the status S



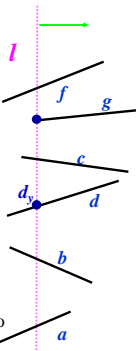
• We maintain S - the list of segments that intersect l in a sorted search tree T , sorted by the order they appear along l .

• When we insert a new segment g , we compare y' , the y -coordinates of intersections of segments with l .

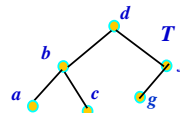
• Example: Since d is the root T , we compute the intersection point of l with d . call it d_y .

• We compare y' and d_y and deduce that g is above d , so it should be inserted into the right subtree.

• Continue recursively.



Operations on the tree



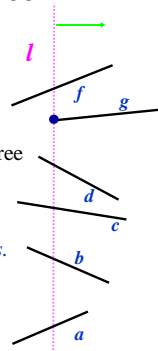
Insert(T, s) - Insert the segment s into the Tree T .

Delete(T, s)

Above(T, s) - Find the segment just below s .

Example **Above(T, b)** = c .
(successor operation in T)

Below(T, s) - Analogous operation (predecessor)



AnySegmentsIntersect(S) - pseudocode

Create an empty set T

Sort endpoints of the segments in S from left to right.

for each endpoint p in the sorted list of endpoints do {
 if p is the left endpoint of a segment s then Insert(T, s);
 if (Above(T, s) exists and intersects s) or
 (Below(T, s) exists and intersects s)
 then return TRUE /*found intersection*/
}

Else { /* p is the right endpoint of s */
 if both Above(T, s) and Below(T, s) exist then
 if Above(T, s) intersects Below(T, s) then return TRUE
 Delete(T, s)
}

Return "no two segments intersect"

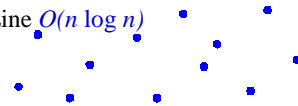
Running time

- There are $2n$ endpoints - $O(n \log n)$ time for sorting
- Each left endpoint event involved
 - Insertion into the tree $O(\log n)$
 - Finding successor/predecessor $O(\log n)$
 - Checking intersection with Above/Below - $O(1)$
- Each right endpoint event involved
 - Deletion from the tree $O(\log n)$.
 - Finding successor/predecessor $O(\log n)$.
 - Checking intersection between Above/Below - $O(1)$
- Total - $O(n \log n)$

- **And Now for Something Completely Different**

Closest Pair Problem

- Given a set P of n points, find $p, q \in P$, such that the distance $d(p, q)$ is minimum
- Algorithms for determining the closest pair:
 - Brute Force $O(n^2)$
 - Divide and Conquer $O(n \log n)$
 - Sweep-Line $O(n \log n)$



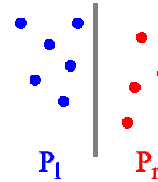
Brute Force

- Compute all the distances $d(p, q)$ and select the minimum distance
- Running time $O(n^2)$

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

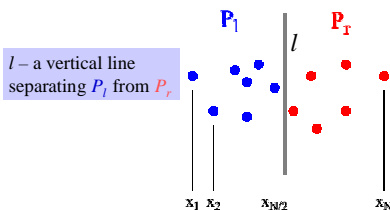
Divide and Conquer

- **Sort points** on the x -coordinate and **divide them in half**
- Closest pair is either
 - Both elements are in P_l
 - Both elements are in P_r
 - One elements is in P_l and one elements is in P_r



Divide and Conquer (2)

- **Phase 1:** Sort the points by their x -coordinate
- $P_1 P_2 \dots P_{n/2} \dots P_{n/2+1} \dots P_n$

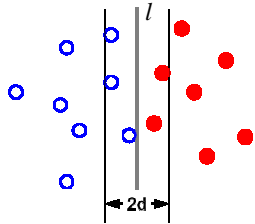


Divide and Conquer (3)

- **Phase 2:** Recursively compute the distance of the closest pairs d_l in P_l and the distance d_r of the closest pair of P_r
- Let $d = \min(d_l, d_r)$
- Find the closest pair and closest distance in central strip of width $2d$ centred at l ,
- in other words...

Divide and Conquer (4)

- Find the closest (○,●) pair in a strip of width $2d$, knowing that no (○,○) or (●,●) pair is closer than d



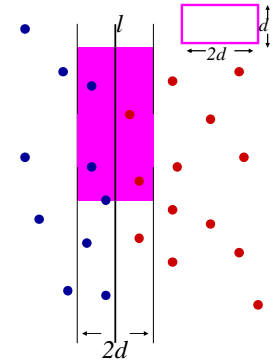
Justification: If the closest (○,●) pair has distance $< d$, they must lie in the strip.

Idea: Sweep the strip with a window

Sweep the strip using a window (width $2d$, height d),

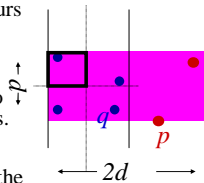
Events: Each time a point enters/exits the window.

Each time a point enters, check its distance to **all** the opposite points in the window



More formally...

- For each point p in the strip, check distances $d(p, q)$, to all points q where
 - p and q are of different colours
 - q is in the strip.
 - $p.y \geq q.y \geq p.y + d$
- Claim:** for each p , there are no more than four such blue points.
- Proof:** Divide the blue side of the window into 4 congruent squares
 - each can contain **at most** one blue points.



Running Time

- Sorting by the y-coordinate at each conquering step yields the following recurrence

$$\begin{aligned} T(n) &= 2T(n/2) + n \log n \\ T(1) &= 1 \end{aligned}$$

Solution $O(n \log^2 n)$

$$\begin{aligned} T(n) &= 2T(n/2) + n \log n \\ &= 4T(n/4) + 2(n/2) \log(n/2) + n \log n \\ &= 4T(n/4) + n(\log n - 1) + n \log n \\ &\dots \\ &= 2^k T(n/2^k) + n(\log n + (\log n - 1) + \dots + (\log n - k + 1)) \\ &\dots \\ &\text{stop when } n/2^k = 1; k = \log n \\ &= n + n(1 + 2 + 3 + \dots + \log n) \\ &= n + n(\log n + 1) \log n / 2 \end{aligned}$$

Improved Algorithm

- The idea: **Sort** all the points by x and y coordinate **once**
- Before recursive calls, **partition the sorted lists** into two sorted sublists for the left and right halves
- When combining, run through the y -sorted list once and select all points that are in a $2d$ strip around l .

Running Time

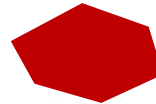
- Phase 1:
 - Sort by x and y coordinate: $O(n \log n)$
- Phase 2:
 - Partition: $O(n)$
 - Recur: $2T(n/2)$
 - Combine: $O(n)$
- $T(n) = 2T(n/2) + n = O(n \log n)$
- Total Time: $O(n \log n)$**

And Now for Something Completely Different:

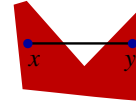
Convex Hull.

Convexity

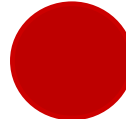
- A set S is **convex** if $x \in S$ and $y \in S$ implies that the segment xy lies inside S



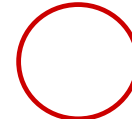
Convex



Non-Convex



Convex

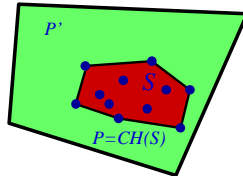


Non-Convex

Convex Hull

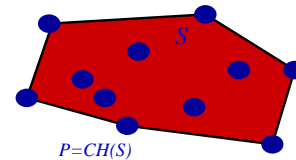
- Let S be a set of n points in the plane.
- The Convex Hull of S (denoted $CH(S)$), is the smallest convex set P that encloses S .
- \Rightarrow there is no other convex set P' such that $S \subseteq P' \subset P$
- Intersection of all convex sets containing S .

A convex polygon containing S , but not the smallest one.



Convex Hull

- Informal definition: Convex hull of a set of points in plane is the shape taken by a rubber band stretched around the nails pounded into the plane at each point

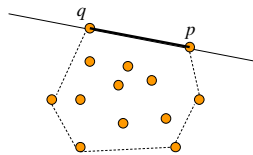


Edges of a Convex Hull

If S is discrete (only n points), then $CH(S)$ is a polygon P

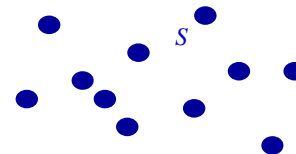
Edge of $CH(S)$:

For every edge both endpoints $p, q \in P$.
All other points in P lie to the same side of the line passing through p and q



Problem: Given S , compute $CH(S)$

One of many applications: Given S find the farthest pair of points.



Furthest pair - cont

Claim 1: The farthest pair must be between vertices of $CH(S)$.

So what ? There are many pairs of vertices !

Claim 2: We can rotate $CH(S)$ so this pair is the leftmost and rightmost vertices.

Farther point - cont

Algorithm

- Compute $P=CH(S)$
- Rotate P , and maintain the distance between the leftmost and rightmost vertices.

• Only $O(n)$ pairs . .

An $O(n \log n)$ for computing $CH(S)$ Graham scan

To compute $CH(S)$ we need to know two things

1. Which points of S are vertices of $CH(S)$.
2. The order that they appear along the boundary of $CH(S)$

Note: on a convex set, when driving from v_i to v_{i+1} to v_{i+2} , we are turning the steering wheel **left** at v_{i+1} .

Computing $CH(S)$: Orientation test

Crossproduct $v_1 \times v_2 = x_1 y_2 - y_1 x_2 = |v_1| |v_2| \sin \theta$.

Thus, $\text{sign}(v_1 \times v_2) = \text{sign}(\sin \theta) > 0$ if θ convex,
 < 0 if θ reflex,
 $= 0$ if θ borderline.

$(p_2 - p_1) \times (p_3 - p_1)$
 > 0 if ccw
 < 0 if cw
 $= 0$ if collinear

Driving from p_1 to p_2 to p_3 , are we turning left or right.

Concave and convex vertices

A **convex vertex** – the angle inside P is < 180 .
 When driving cck along the polygon, we turn **left** in the vertex (see vertex 3)

A **concave vertex** – the angle inside $P < 180$.
 When driving cck along the polygon, we turn **right** one the vertex (see vertex 6)

Toward an $O(n \log n)$ for computing $CH(S)$

Two stages algorithm:

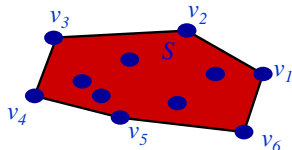
- 1) Find a polygon that does not intersect itself, its vertices are points of S , and contains S in its interior.
- 2) Repeat – Find a concave vertex v . Obviously it is not a vertex of $CH(S)$. Delete it, and connect its neighbors in P . Stop when all vertices are convex.

The order of accessing the concave vertices is not important, but it turns out that some orders are more convenient than others.

An $O(n \log n)$ for computing $CH(S)$ Graham scan

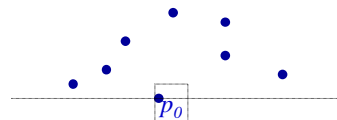
To compute $CH(S)$ we need to know two things

1. Which points of S are **vertices** of $CH(S)$.
2. The order that they appear along the boundary of $CH(S)$



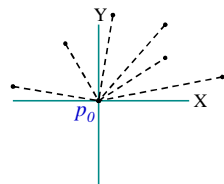
Step 1

- Find the point p_0 of S , with smallest Y-coordinate.
Note - p_0 must be a vertex of $CH(S)$.



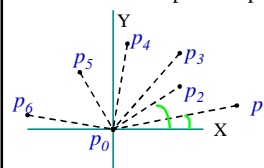
Step 2

- Translate the interior point so p_0 would be the origin.



Step 3

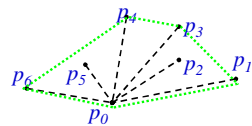
- Sort the points of S according to the orientations of the segments connecting them to p_0 . Let $S = \{p_1 \dots p_{n-1}\}$ be the sorted set.
- **Def:** The orientation of a segment is the CCW angles between the positive part of the X-axis and the segment.



So when we stand on p_0 looking south, and rotating the direction we watch, we first see p_1 , then p_2 etc.

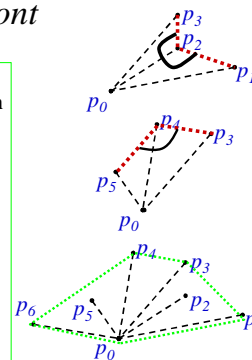
Some observations

- The vertices of $CH(S)$ appear in the same order they appear in (the sorted) S .
- Basic idea for computing $CH(S)$: delete the points which are not vertices of $CH(S)$.
- Note that to verify that p_5 is **not** a vertex of $CH(S)$, it is enough to look at the triple $p_4 p_5 p_6$



Basic idea - cont

- Define the angle of point p_i to be the inner angle between the segment $p_{i-1}p_i$ and $p_i p_{i+1}$.
- **Idea:** find a point p_i whose angle $> 180^\circ$, (*concave point*) delete it, and repeat.
- Once we are left only with points whose angle $< 180^\circ$, then each of them is a vertex of $CH(S)$.
- **Question:** At which order should we access the points?



The Graham Scan Algorithm

Let p_0 be the point of S with *minimum* y-coordinate
 Let $\langle p_1, p_2, \dots, p_{n-1} \rangle$ be the remaining points in P
 sorted in counterclockwise order by polar angle around p_0 .

Create an empty stack *Stack*.

Push(p_0 , *Stack*)

Push(p_1 , *Stack*)

Push(p_2 , *Stack*)

for $i \leftarrow 3$ to $n - 1$

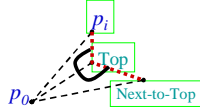
do while (Top(*Stack*) is concave)

/*creates concave angle with p_i and Next-to-Top(*Stack*)*

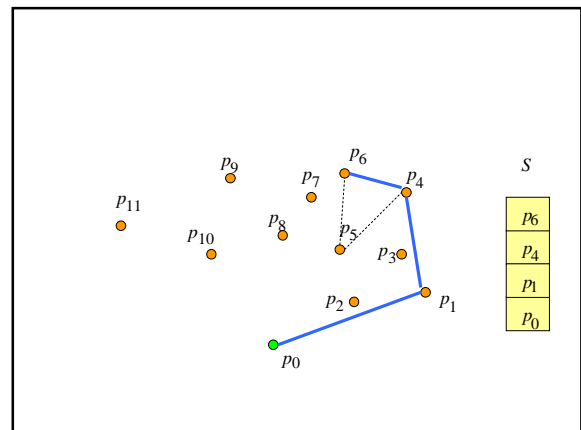
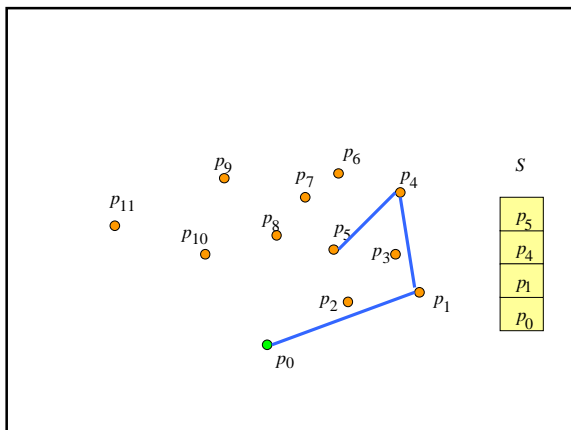
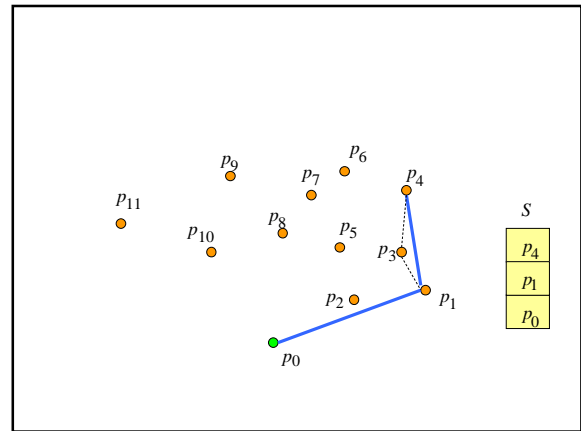
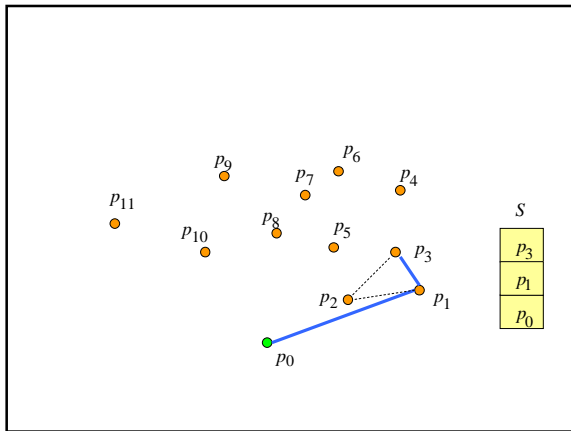
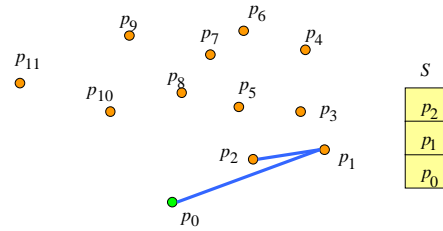
do Pop(*Stack*)

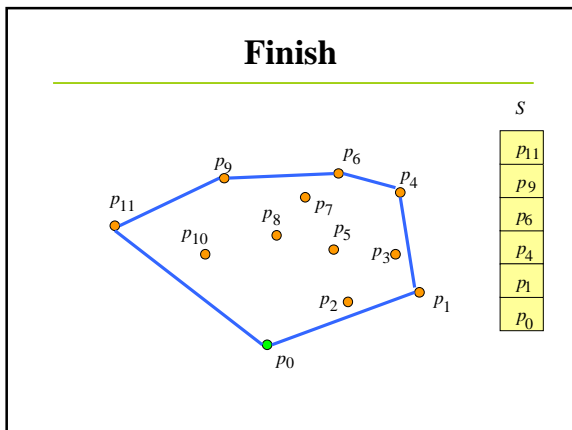
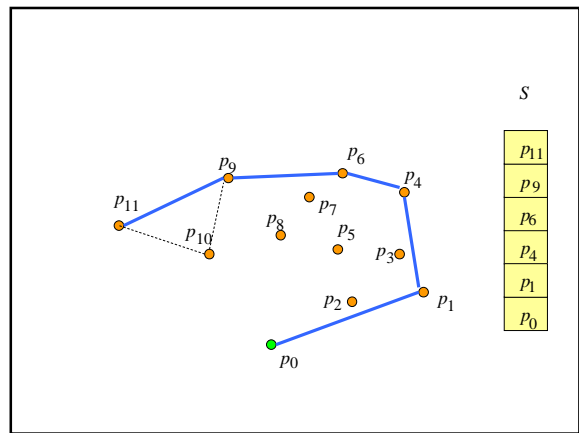
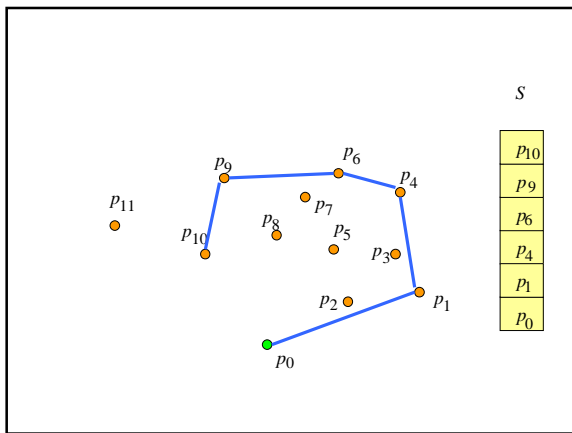
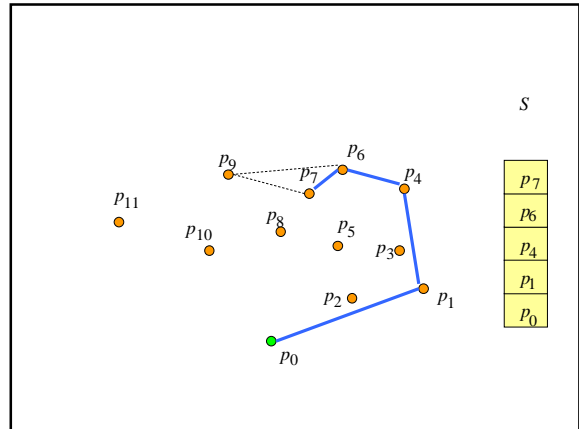
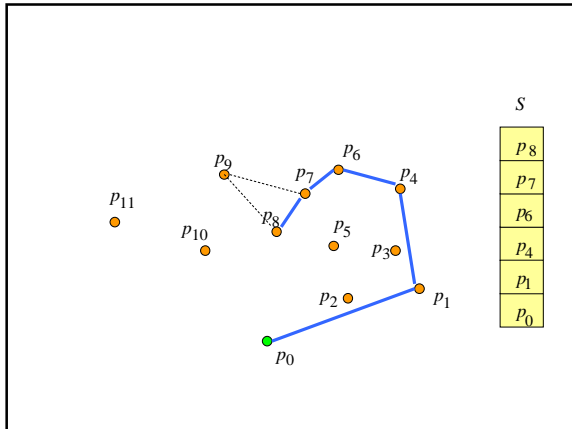
Push(p_i , *Stack*)

return S



Stack Initialization





Efficiency

- Assume n is the number of points in S .
 - Sorting takes time $O(n \log n)$
 - A point can be pushed into the stack and pop at most once – total $O(n)$

Total time – $O(n \log n)$