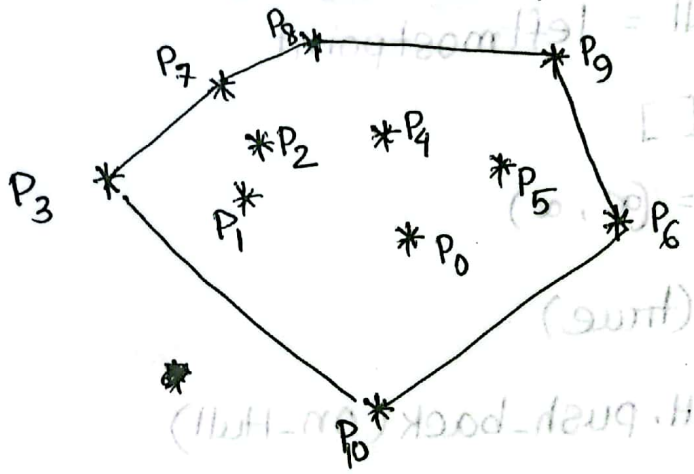
Convex Hull

Lec-17

draw a convex polygon so that, the convex polygon contains all the points inside



$CP = [P_3, P_7, P_8, P_9, P_6, P_{10}]$

Convex Hull: Smallest convex polygon that can contain all the points of the set within it

Smallest polygon: minimum number of vertex or minimum number of edges so that the polygon still remain convex.

Algorithm:

Jarvich_march (points[], n)

1. leftmostpoint = get_leftmost_point (points[], n)
2. On_Hull = leftmostpoint
3. CH = []
4. CH[0] = (∝, ∝)
5. while (true)
   - 5.a. CH.push_back (On_Hull)
   - 5.b. next_point = points[0]
   - 5.c. for i = 0 to n-1

     if (On_Hull == next_point ||
     ccw (On_Hull, next_point, points[i]) > 0)

     next_point = points[i];

     else if (ccw (On_Hull, next_point, points[i] =
     && get_dist (On_Hull, next_point) <
     get_dist (On_Hull, points[i]))

     next_point = points[i];

   - 5.d. On Hull = next_point

6. if (On_Hull == CH[1]) break;

Time complexity = $O(n*h)$

Best case = $O(3n)$ [for triangle]

Worst case = $O(n*n) = O(n^2)$ [for circle]

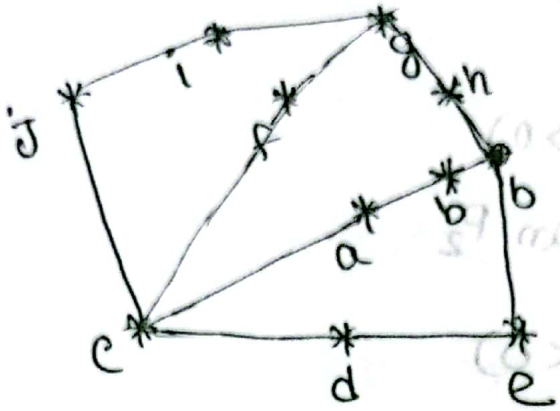
Graham Scan Algorithm:

```
Graham_scan_CH(points[], n)
1. bottom_most_P = get_bottom_most_point(points[], n);
2. swap(points[0], points[b_m_P]
3. sort(P, P+n, compare)
4. remove_colinear_points(P[0], P[], n)
5. CH = [], CH.push(P[0]), CH.push(P[1])
6. for i=2 to n-1
   while (true)
   {
     top = CH.pop()
     next_top = CH.pop()
     if (ccw(next_top, top, P[i])>0)
     {
       CH.push(next_top);
       CH.push(top);
       CH.push(P[i]);
       break;
     }
   else
   {
     CH.push(next_top);
} } }
```

Example:

```
Compare (P0, P1, P2)
{
  if (ccw (P0, P1, P2) > 0)
  P1 come first than P2
  if (ccw (P0, P1, P2) < 0)
  P2 comes first than P1
  if (ccw (P0, P1, P2) == 0)
  {
    if (get_Dist(P0, P1) > get_Dist (P0, P2))
    P2 come first than P1
    else
    P1 come first than P2
  }
```

Example:



Step-I

get_bottom_most_point,

b_m_p = c

Step-II

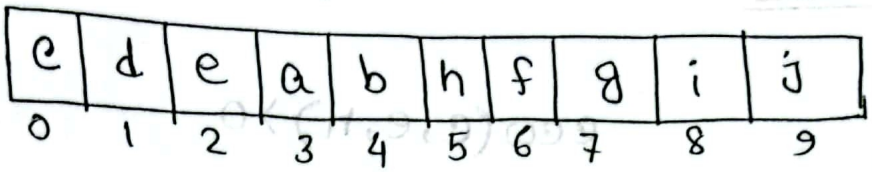
swap (b_m_p, P[0])

a	b	c	d	e	f	g	h	i	j
0	1	2	3	4	5	6	7	8	9

After swapping,

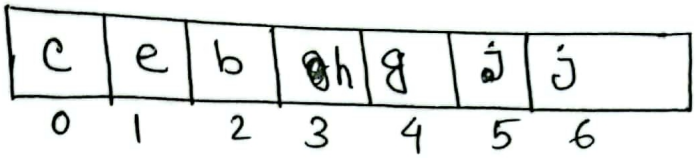
c	b	a	d	e	f	g	h	i	j
0	1	2	3	4	5	6	7	8	9

sort (P, P+n, compare)

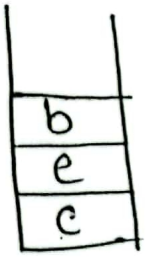


Step-03

remove_colinear_points (P[ ], n)



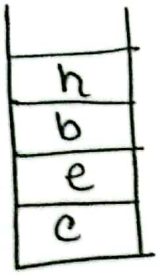
iteration : 01



ccw (next_top, top, P[2]) > 0

P[2] ∈ CH

iteration : 02

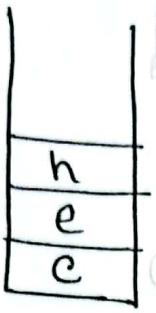


ccw (e, b, P[3]) ≯ 0

P[3] ∉ CH

remove top element

iteration : 03



ccw(c, e, h) > 0

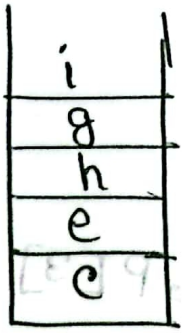
CH ∈ h

iteration : 04

ccw(e, h, g) > 0

CH ∈ g

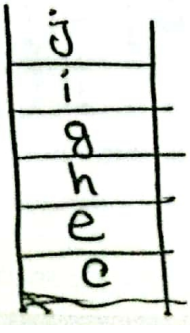
iteration : 05



ccw(h, g, i) > 0

CH ∈ i

iteration : 06



ccw(g, i, j) > 0

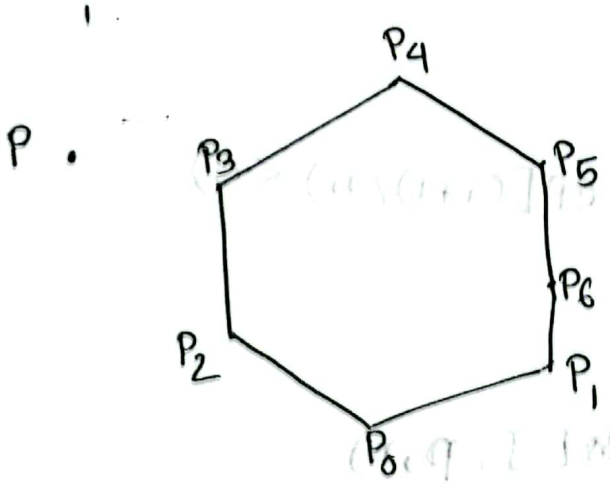
CH ∈ j

Q.

Lec -18

given a convex polygon and a point how many tangent can be drawn,

=> 2 tangent (atmost)



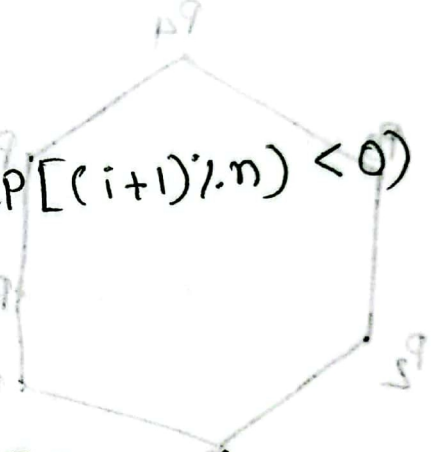
We will search for ~~P~~ ~~until~~

Algorithm will be build where have to search for ~~left~~ upper tangent and lower tangent

Algorithm:

Upper_tangent (CP[], P, n)

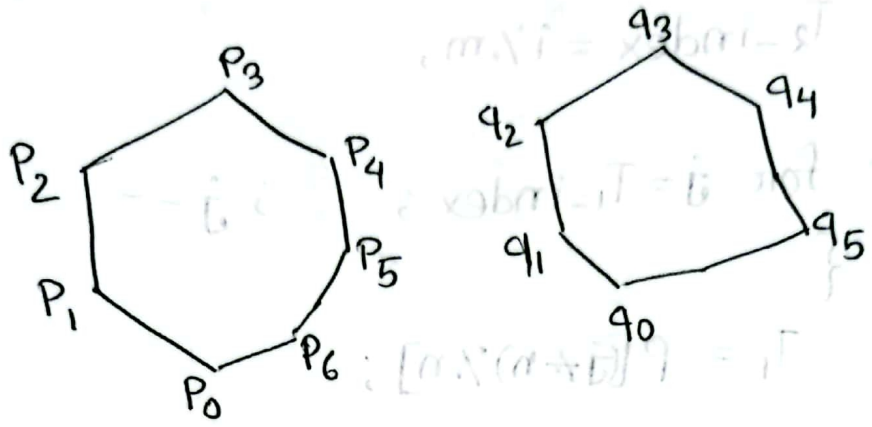
1. T_index = get_leftmost_point (points[], n)
2. for i = T_index to n-1; i++
   {
   T = CP[i%n]
   if (ccw (P, T, CP[(i+1)%n) < 0)
   break;
   }



lower_tangent (points[], p, n)

1. T_index = get_leftmost_point (CP[], n)
2. for i = T_index; ; i--;
   {
   T = CP[(n+i)%n]
   if (ccw (P, T, CP[(n+i-1)%n] > 0)
   break;
   }

Q Given two convex polygon, Find the tangent,



Algorithm:

Upper_tangent (P, n, Q, m)

1. T_1_index = get_rightmost_point (P[], n)

2. T_2_index = get_left_most_point (Q[], m)

3. T_1 = P[T_1_index]

   T_2 = Q[T_2_index]

4. for i = T_2_index ; ; i++ // Keeping T_1 Fixed and trying to draw tangent to Q

   {

   T_2 = Q[i%m]

   if (ccw (T_1, T_2, Q[(i+1)%m]) < 0)

   break;

   }

5. $T_2 = Q[i \% m]$;

$T_2\text{-index} = i \% m$;

6. for $j = T_1\text{-index}$; ; $j--$

{

$T_1 = P[(j+n) \% n]$;

if (ccw (T_2, T_1, $P[(n+j-1) \% n]$) > 0)

break;

}

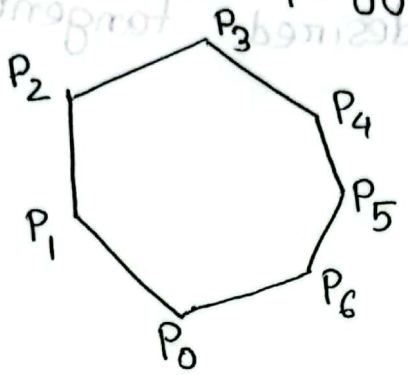
7. $T_1 = P[(j+n) \% n]$

$T_1\text{-index} = (j+n) \% n$

Example: Lec-19

given a convex polygon and a point to left of the polygon

Q.



Solve:

Step-1 leftmost_point = ls = get_leftmost_point (P[ ], n)

ls = P_1

Step-2 T = P_1

iteration-1

ccw (Q, T, P[2]) > 0

PP_1 is not the desired tangent

T = P_2

iteration-2

ccw (Q, T, P[3]) > 0

PP_2 is not the desired tangent

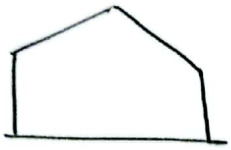
iteration-3 $T = P_3$

$ccw(P, T, P_4) < 0$

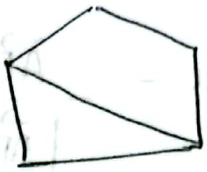
PP_3 is the desired tangent

Lec-21

min score/cost polygon triangulation:

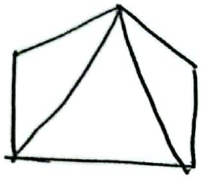


1)



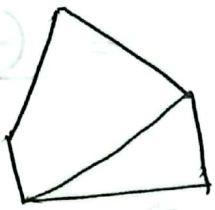
$\Delta 2 \times \Delta 4 = 1 \times 2 = 2$

2)



$\Delta 3 \times \Delta 3 = 1$

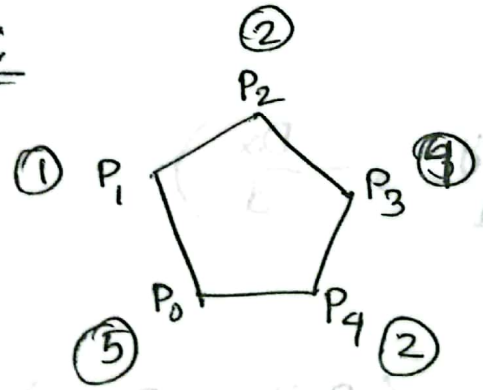
3)

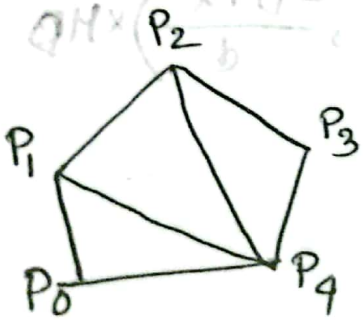


$\Delta 4 \times \Delta 2 = 2$

Total triangulation possible $= 2+1+2 = 5$

Ques:



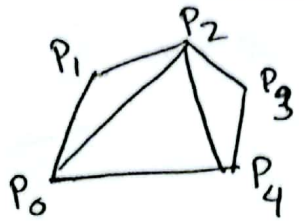


$\text{cost} = P_0P_1P_4 + P_1P_2P_4 + P_2P_3P_4$

$= (5 \times 1 \times 2) + (1 \times 4 \times 2) + (1 \times 2 \times 4)$

$= 26$

2)

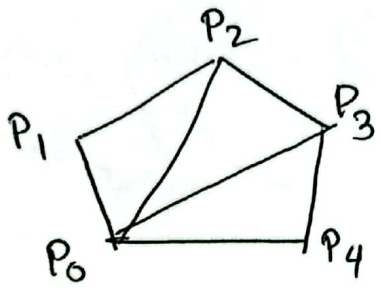


$cost = P_0P_1P_2 + P_0P_2P_4 + P_2P_3P_4$

$= (5 \times 1 \times 2) + (5 \times 2 \times 2) + (2 \times 4 \times 2)$

$= 46$

3)

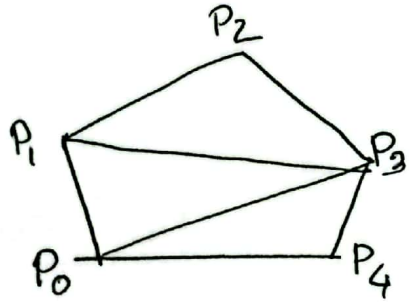


$cost = P_0P_1P_2 + P_0P_2P_3 + P_0P_3P_4$

$= (5 \times 1 \times 2) + (5 \times 2 \times 4) + (5 \times 4 \times 2)$

$= 90$

4)

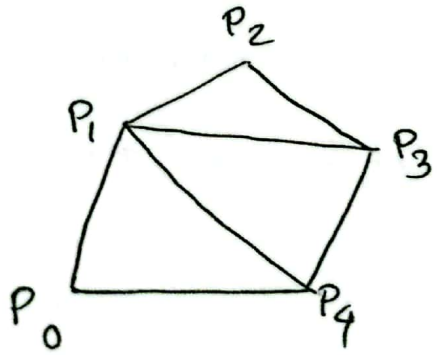


$cost = P_0P_1P_3 + P_0P_3P_4 + P_1P_2P_3$

$= (5 \times 1 \times 4) + (5 \times 4 \times 2) + (1 \times 2 \times 4)$

$= 68$

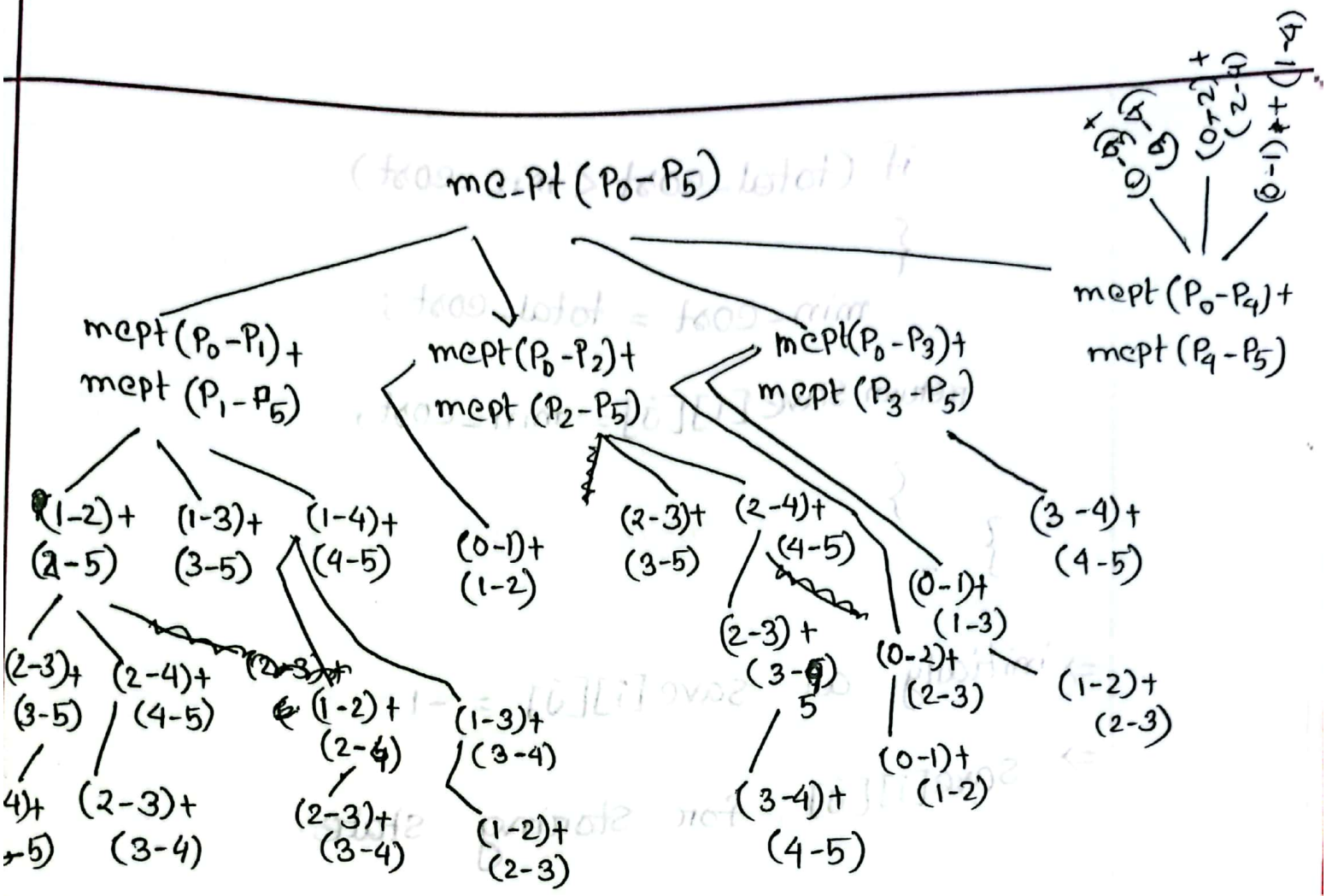
5)



$cost = P_0P_1P_4 + P_1P_2P_3 + P_1P_3P_4$

$= (5 \times 1 \times 2) + (1 \times 2 \times 4) + (1 \times 4 \times 2)$

$= 26$



Algorithm:

```
mc_pt (i, j)
if (save[i][j] != -1) return save[i][j]
if (i == j) return 0;
if (i+1 == j) return 0;
else
{
   min_cost = ∞
   for(K = i+1; K ≤ j-1; K++)
   {
      total_cost = cost(i,j,K) + mc_pt (i,K) + mc-pt(K,j
```

```
        if (total_cost < min_cost)
        {
          min_cost = total_cost;
        return save[i][j] = min_cost;
        }
    }
```

⇒ initially all save[i][j] = -1

⇒ save[i][j] for storing state

Line - sweep_Algorithm :

1. Sort (P, P+n)
2. d = ∝
3. for i = 0 to n-1

```
set < points> active_events
 x_l = lowerbound (P[i].x - d , P[i].y)
 for(j = x_l ; j <=i; j++)
 {
  active_events. insert (P[j], y , P[j].x);
  y_l = lowerbound (y-d, x)
  y_u = upperbound (y+d, x)
  for (k = y_l ; k < yu ; k++)  → // run atmost 5 times
  {
    if (get_Disk(P[i], active_events[k]) < d)
    d = get_Dist2 ( P[i] , active_events[k]);
  }
 }
```

4. return d.

Upper_convex_Hull (L,)

1. Sort all the lines in no-decreasing order of their slope
2. for i=0 to n-1
   {
   input a line L[i];
   add (L[i]);
   }

```
void add (current_L)
{
  UH.push_back (current_L)
  while( UH.size()≥3 && badlines (UH[UH.size()-1],
  {                              UH[UH.size()-2] , UH[UH.size()-3])
     UH.erase (UH.size()-2); // removing previous
                                line
  }
}
```

```
bool bad_lines (current_L, prev_L, before_prev_L)
{
  return ((prev_L.m - before_prev_L) *
          (prev_L.c - current_L.c) <=
          (current_L.m - prev_L.m) *
          (before_prev_L.c - prevL.c))
  }

Query (x)
{
    l = 0
    r = n-1
    while (l <= r)
    {
        m1 = (2l + r) / 3
        m2 = (l + 2r) / 3
        m1_y = get_y (L[m1].m, L[m1].c, x)
        m2_y = get_y (L[m2].m, L[m2].c, x)
```

```
if (m1_y > m2_y)
    l = m1 + 1
else if (m2_y > m1_y)
    r = m2 - 1
else
{
    l = m1 + 1;
    r = m2 - 1;
}
return get_y(L[m1].m, L[m1].c, x);
}

get_y(m, c, x)
{
return m * x + c;
}
```