Assignment 1

Computational Geometry CS60064

Pratyush Jaiswal | 18EE35014
Ram Niwas Sharma | 18CS10044

Question 1

Given n points on 2D-plane, propose an algorithm to construct a simple polygon P with all the given points as vertices, and only those. Provide its proof of correctness and deduce its time complexity. (A simple polygon is one in which no two edges intersect each other excepting possibly at their endpoints.

Answer:

ALGORITHM

Here we can choose to get the final result as a list of points to get joined in Counter Clockwise or Clockwise direction to form a simple polygon.

1. Assign any random point as starting point. However we will be considering the polar coordinates but for getting the angle, trigonometric ratios calculations would be used. So to skip that we can choose the **leftmost bottom** point to start with because using that all the angles will lie between $-\frac{\pi}{2}$ and $+\frac{\pi}{2}$, and we can just map the angle with slope because in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$, $\tan\theta$ is always increasing.

2. Sort the points (excluding the origin) in increasing order of their (θ, r), i.e. compare the angle subtended first and if it is same then sort based on the value of radial distance. We can use **Orientation concept** instead of calculating θ because of lack of accuracy while computing. Considering the starting point say O, if we have to consider two points say A and B (A with less θ than B), to go to then according to the comparision of θ we will be going to the one with less θ which is A.

   Using orientation test,
   $Orientation(O, A, B) > 0 => CCW$ Since theta subtended by B is greater than A
   $Orientation(O, A, B) < 0 => CW$

So we can sort on the basis of orientation test of two points and decide which to sort before or after. Here we will be joining in Counter Clockwise manner so the sorting will be based on $(-Orientation, r)$. Negative of orientation is taken because CCW will give positive orientation value and that will be preferred while sorting.

3. The order of points we get appended with the assumed starting point are the coordinates of expected Simple polygon **P**

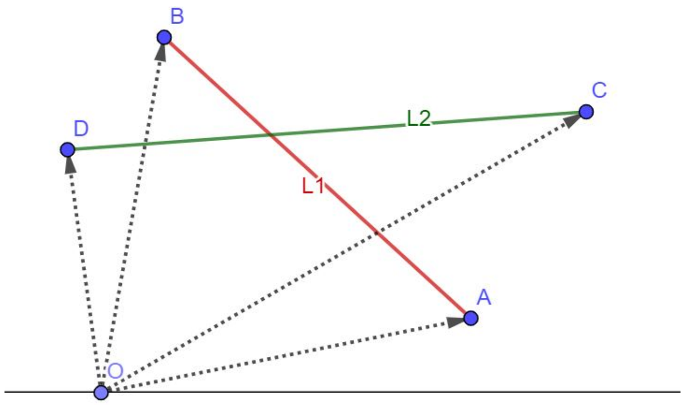
PROOF of CORRECTNESS

For now, we can assume that this algorithm fails and two edges intersect. Let the two edges be $L1$, joining the points A and B and $L2$ joining the points C and D.

The angles subtended by A, B, C and D with the horizontal axis are θ_A, θ_B, θ_C and θ_D respectively, where $\theta_A < \theta_B$ and $\theta_C < \theta_D$.

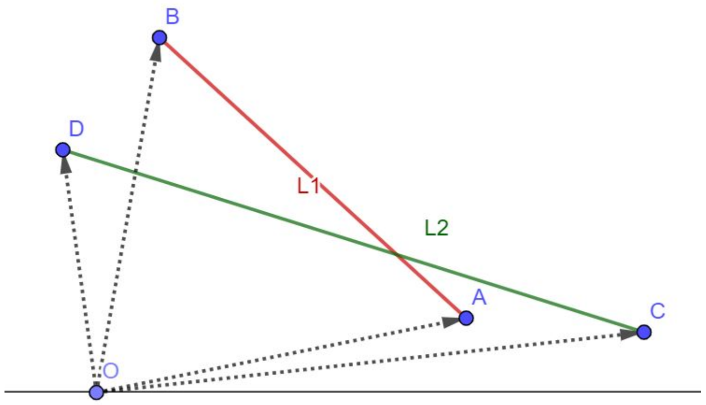
Then there can be four cases of intersecting the lines.

1. $\theta_A \leq \theta_C$ *and* $\theta_B \leq \theta_D$



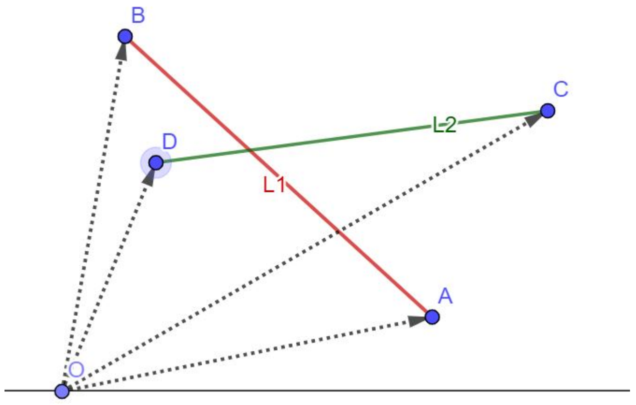
Here according to our alogrithm, C should have been connected to B (in the increasing order of their angle from O) and A should have been connected to C. So it contradicts the assumption.

2. $\theta_A \geq \theta_C$ *and* $\theta_B \leq \theta_D$



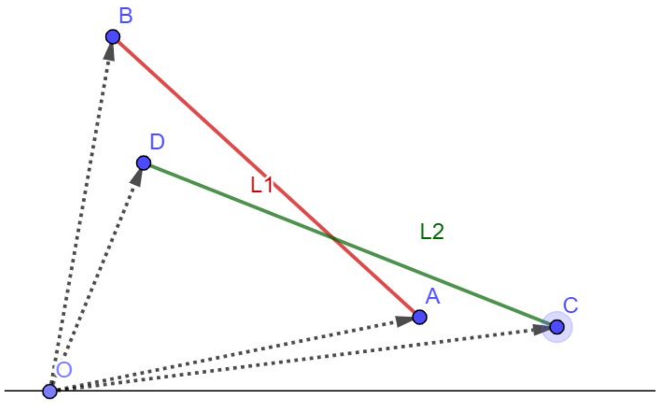
Here according to our alogrithm, C should have been connected to A. So it contradicts the assumption.

3. $\theta_A \leq \theta_C$ *and* $\theta_B \geq \theta_D$



Here according to our alogrithm, A should have been connected to C and D should have been connected to B. So it contradicts the assumption.

4. $\theta_A \geq \theta_C$ *and* $\theta_B \geq \theta_D$



Here according to our alogrithm, C should have been connected to A and A should have been connected to D. So it contradicts the assumption.

So here we can see that every scenarios of intersection of two edges are not possible according to our algorithm which proves that in general this algorithm will always generate simple polygons with non intersecting lines.

TIME COMPLEXITY ANALYSIS

Steps in algorithms

1. Searching for the leftmost bottom point will take $O(n)$
2. Sorting the points based on $(orientation, r)$: This takes $(n \log(n))$

3. Joining the points to construct the polygon : Since there are 2n points it also takes $O(n)$ time

Overall time complexity : $O(n \log(n))$

CODE SAMPLE

```
# Using Python 3.7
import random
from functools import cmp_to_key
import matplotlib.pyplot as plt

class Point:
    def __init__(self, x=None, y=None):
        self.x = x
        self.y = y

    def __add__(self, pt):
        x = self.x + pt.x
        y = self.y + pt.y
        return Point(x, y)

    def __sub__(self, pt):
        x = self.x - pt.x
        y = self.y - pt.y
        return Point(x, y)

    def cross(self, pt):
        return self.x*pt.y - self.y*pt.x

    def dot(self, pt):
        return self.x*pt.x + self.y*pt.y

    def orientation(self, pt1, pt2):
        # Orientation Test for points (self, pt1, pt2)
        return (pt1-self).cross(pt2-pt1)

    def __len__(self):
        return self.dot(self)

    def __str__(self):
        return "{} {}".format(self.x, self.y)

min_x = 0
min_y = 0

max_x = 2000
max_y = 2000

num_points = 30

x_coords = random.sample(range(min_x, max_x), num_points)
y_coords = random.sample(range(min_y, max_y), num_points)
```

```python
points = [[x_coords[i], y_coords[i]] for i in range(num_points)]

# Converting points list having (x, y) to list of class Point
for i in range(num_points):
    points[i] = Point(points[i][0], points[i][1])

# Finding leftmost bottom point for the starting point
starting_point_ind = 0
for i in range(1, num_points):
    if((points[i].x<points[starting_point_ind].x) or (points[i].x==points[
   starting_point_ind].x and points[i].y<=points[starting_point_ind].y)):
        starting_point_ind = i

starting_point = points[starting_point_ind]

def cmp_orientation(point1, point2):
    # Comparing function for the orientation of (starting point, point1,
   point2)

    # Point1 will be joined before Point2 if the orientation comes out to be
   CCW (CounterClockWise)
    # else Point2 will be preffered to Point1
    # If the orientation comes out to be zero then the nearest point will be
   joined first
    orientation = starting_point.orientation(point1, point2)
    if(orientation==0):
        return len(point1-starting_point) - len(point2-starting_point)
    return -orientation

# Eliminating Starting Point from the list and sorting based on the
   orientation
polar_coords = [points[i] for i in range(num_points) if i!=starting_point_ind]
polar_coords.sort(key=cmp_to_key(cmp_orientation))

# Inserting starting point at the end for joining the last point to the
   starting point to close the polygon
polar_coords.append(points[starting_point_ind])

# Joining the points
curr_point = starting_point
for i in range(num_points):
    plt.plot([curr_point.x, polar_coords[i].x], [curr_point.y, polar_coords[i
   ].y])
    curr_point = polar_coords[i]

# Showing the plot
plt.show()
```

Question 2

2A

(a) A convex polygon P is given as counter-clockwise ordered sequence of n vertices, in general positions, whose locations are supplied as (x, y) co-ordinates on the x-y plane. Given a query point q, propose an algorithm to determine in $O(\log n)$ time and $O(n)$ space, including pre-processing, if any, whether or not P includes q.

Answer:

ALGORITHM

Let's pick the point with smallest x coordinate and if there are many, choose the one with smallest y coordinate. Let's call it P_0.

Now arrange all other points P_1, P_2, ..., P_{n-1} in counter clockwise direction such that P_0 is the starting point.

First check if the query point (let's say Q) lies between P_1 and P_{n-1}. If not the point is already outside the polygon. There will be one special case when Q lies on the edge P_0P_1 or P_0P_{n-1}.

If the query point lies inside the polygon then it must lie inside a triangle, $\Delta P_0P_iP_{i+1}$ or it may lie in more than one if it lies on the boundary of the triangles.

Now for all points $P_j \ \forall j > i$, the query point will be oriented in counterclockwise direction and for rest of the points, orientation will be clockwise.

Using binary search, we can find that maximum i till which the $orientation(P_0, P_i, Q) >= 0$.

IMPLEMENTATION

Given an array of convex polygon vertices in counter clockwise manner.

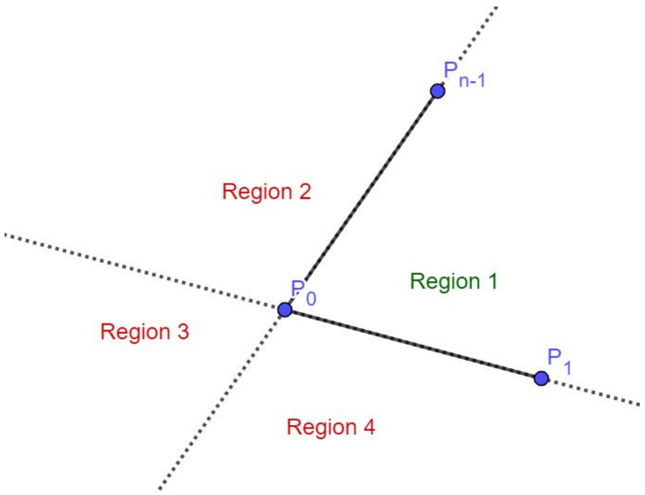
- **PRE-PROCESSING**
  1. Get the index of the leftmost bottom point for the starting point.
  2. Left rotate the array such that starting point comes to the starting of the array. This step is basically arranging the points such that starting point comes to the zeroth index keeping the arrangement in counter clockwise direction.
- **CHECKING FOR QUERY POINT**

  Given a query point Q.
  1. Check whether Q lies on the edge P_0P_1 or P_0P_{n-1}.
     Do the orientation test for P_1 and P_{n-1}.

```
If orientation(P_0P_1Q)==0:
    If dot_product(P_0Q,P_0P_1)>= 0 and length(P_0P_1)>=length(P_0Q)
   :
        Lies Inside
    Else
        Lies Outside
```

Do the same for P_{n-1}.

2. Check whether Q lies between P_1 and P_{n-1}.



Q should lie in **Region 1** else it is already outside the polygon. This can be ensured by checking if $sign(orientation(P_0P_1Q)) == sign(orientation(P_0P_1P_{n-1}))$ and $sign(orientation(P_0P_{n-1}Q)) == sign(orientation(P_0P_{n-1}P_1))$ else Q lies outside the polygon.

3. Do binary search and find maximum index i such that $orientation(P_0, P_i, Q) >= 0$.
4. Check whether Q lies inside the triangle $\Delta P_0P_iP_{i+1}$. Just check if the unsigned area of the triangle $\Delta P_0P_iP_{i+1}$ is equal to the combined unsigned area of ΔP_0QP_i, ΔP_iQP_{i+1} and $\Delta P_{i+1}QP_0$. If equal, it lies inside else lies outside.

TIME AND SPACE COMPLEXITY ANALYSIS

Steps in algorithm

- **PRE-PROCESSING**

  1. Finding the starting point will take $O(n)$ time and $O(1)$ space.
  2. Rotating an array will take $O(n)$ time and $O(n)$ space.

We can see that the overall time and space complexity for pre-processing is $O(n)$ for both time and space.

- **CHECKING FOR QUERY POINT**

1. Checking whether Q lies on the edge P_0P_1 or P_0P_{n-1} takes $O(1)$ for both space and time.
2. Checking whether Q lies between P_1 and P_{n-1} again takes $O(1)$ for both space and time.
3. Binary Search takes $O(\log n)$ time and $O(1)$ space
4. Checking whether Q lies inside triangle again takes $O(1)$ for both space and time.

We can see that the overall time and space complexity for checking the query point takes $O(\log n)$ time and $O(1)$ space complexity.

2B

(b) Write a code to implement your algorithm. Construct a convex polygon with 30 vertices, and show your results for a few internal and external points.

Answer: **CODE SAMPLE**

```python
import math
import random

num_points = 30

thetas = random.sample(range(0,360), num_points)
thetas.sort()
r = 100

points = [[int(r*math.cos((math.pi*thetas[i])/180)), int(r*math.sin((math.pi*
    thetas[i])/180))] for i in range(num_points)]

class Point:
    def __init__(self, x=None, y=None):
        self.x = x
        self.y = y

    def __add__(self, pt):
        x = self.x + pt.x
        y = self.y + pt.y
        return Point(x, y)

    def __sub__(self, pt):
        x = self.x - pt.x
        y = self.y - pt.y
        return Point(x, y)

    def cross(self, pt):
        return self.x*pt.y - self.y*pt.x

    def dot(self, pt):
        return self.x*pt.x + self.y*pt.y

    def orientation(self, pt1, pt2):
        # Orientation Test for points (self, pt1, pt2)
        return (pt1-self).cross((pt2-pt1))

    def __len__(self):
        return self.dot(self)

    def __str__(self):
        return "({} {})".format(self.x, self.y)

num_points = len(points)

# Converting points list having (x, y) to list of class Point
for i in range(num_points):
    points[i] = Point(points[i][0], points[i][1])

# Finding leftmost bottom point for the starting point
starting_point_ind = 0
for i in range(1, num_points):
    if((points[i].x<points[starting_point_ind].x) or (points[i].x==points[
   starting_point_ind].x and points[i].y<=points[starting_point_ind].y)):
        starting_point_ind = i
starting_point = points[starting_point_ind]

# Rotating left by starting_point_ind steps so that starting_point becomes
   first point in points
points = points[starting_point_ind:] + points[:starting_point_ind]

def sgn(val):
    if(val>0):
        return 1
    elif(val==0):
        return 0
    return -1

def isPointInsideTriangle(A, B, C, P):
    # If the query P lies inside the triangle then it must satisfy the below
   relation for unsigned area.
    # Area(triangle ABC) = Area(triangle APB) + Area(triangle BPC) + Area(
   triangle CPA)
    left_area = abs(A.orientation(B,C))
    right_area = abs(A.orientation(P,B)) + abs(B.orientation(P,C)) + abs(C.
   orientation(P,A))
    return right_area == left_area

def isPointInsideConvexPolygon(query):

    # Checking if the Query Point lies on left or right edge of starting_point
    if(starting_point.orientation(points[1], query)==0):
        return ((points[1]-starting_point).dot(query-starting_point)>=0) and (
   len(query-starting_point) <= len(points[1]-starting_point))
```

```
    if(starting_point.orientation(points[-1], query)==0):
        return ((points[-1]-starting_point).dot(query-starting_point)>=0) and
   (len(query-starting_point) <= len(points[-1]-starting_point))

    # Checking if the query point lies between the points just left and right
   side of the starting point
    # For the point to lie between the two points P_1 and P_{n-1}, orientation
     of P_{n-1} and Q must be same
    # for P_1 and same goes for P_{n-1} otherwise Q will lie outside.
    if(not (sgn(starting_point.orientation(points[1], query))==sgn(
   starting_point.orientation(points[1], points[-1])) and sgn(starting_point.
   orientation(points[-1], query))==sgn(starting_point.orientation(points
   [-1], points[1])))):
        return False

    pos = 0
    left = 0
    right = num_points - 1
    while(left<=right):
        mid = left + (right-left)//2
        if(starting_point.orientation(points[mid], query)>=0):
            # If the orientation of starting_point, curr_point and query is
   positive
            pos = mid
            left = mid + 1
        else:
            right = mid - 1

    # Check inside the triangle
    return isPointInsideTriangle(starting_point, points[pos], points[pos+1],
   query)

def save_svg(filename,query,result):
    # Determining the viewBox for the svg file
    min_x = float("inf")
    min_y = float("inf")
    max_x = float("-inf")
    max_y = float("-inf")
    for i in range(num_points):
        min_x = min(min_x, points[i].x)
        min_y = min(min_y, points[i].y)
        max_x = max(max_x, points[i].x)
        max_y = max(max_y, points[i].y)

    min_x = min(min_x, query.x)
    min_y = min(min_y, query.y)
    max_x = max(max_x, query.x)
    max_y = max(max_y, query.y)
    radius = 5

    file = open(filename, 'w')
    file.write("<svg xmlns=\"http://www.w3.org/2000/svg\" viewBox=\"{} {} {}
   {}\">".format(min_x-3*r, min_y-3*r, (max_x-min_x)+5*r, (max_y-min_y)+5*r))
```

```
    # Drawing all the vertices
    for i in range(num_points):
            file.write("<circle cx=\""+ str(points[i].x) +"\" cy=\""+ str(
   points[i].y) +"\" r=\""+ str(radius) +"\" stroke=\"#000000\" stroke-width
   =\"0\" fill=\"#00ffff\" fill-opacity=\"1.0\" />")

    # Joining all the vertices
    for i in range(num_points-1):
        file.write("<line x1=\""+str(points[i].x)+"\" y1=\""+str(points[i].y)+
   "\" x2=\""+str(points[i+1].x)+"\" y2=\""+str(points[i+1].y)+"\" style=\"
   stroke:rgb(255,0,0);stroke-width:2\"/>")

    # Joining the starting point and last point
    file.write("<line x1=\""+str(points[0].x)+"\" y1=\""+str(points[0].y)+"\"
   x2=\""+str(points[num_points-1].x)+"\" y2=\""+str(points[num_points-1].y)+
   "\" style=\"stroke:rgb(255,0,0);stroke-width:2\"/>")

    # Drawing the query point
    file.write("<circle cx=\""+ str(query.x) +"\" cy=\""+ str(query.y) +"\" r
   =\""+ str(radius) +"\" stroke=\"#000000\" stroke-width=\"0\" fill=\"black
   \" fill-opacity=\"1.0\" />")

    # Displaying the result whether Inside or Outside
    file.write("<text x=\"0\" y=\"0\"> {} </text>".format(result))
    file.write("</svg>")
    file.close()

# Generating random query points and checking for them
for i in range(10):
    query = [random.randint(-100,100), random.randint(-100,100)]
    # Converting query to Point Class form
    query = Point(query[0], query[1])
    filename = "fig_{}.svg".format(i)
    result = (isPointInsideConvexPolygon(query))
    if(result):
        result = "Inside"
    else:
        result = "Outside"
    save_svg(filename, query,result)
    print(result)
```