Computational Geometry (CS60064)
Homework Set 1

Bratin Mondal - 21CS10016

Question 1

Given n points on the xy-plane, design an algorithm to construct a simple polygon P such that all the given points serve as vertices of P, and no other points are included as vertices. Provide a proof of correctness for your algorithm and deduce its time complexity. (A *simple polygon* is defined as one in which no two edges intersect, except possibly at their endpoints.)

Input

Let us denote the set of n points as $S = \{s_1, s_2, \ldots, s_n\}$.

Description of the Algorithm

1. From the set S, select the leftmost point with the minimum x-coordinate. If there are multiple points with the same minimum x-coordinate, choose the one with the minimum y-coordinate. Similarly, select the rightmost point with the maximum x-coordinate. In case of ties, choose the point with the minimum y-coordinate. Denote these points as s_{left} and s_{right}, respectively.

2. Define a set $S' = S \setminus \{s_{\text{left}}, s_{\text{right}}\}$ and initialize two empty sets A and B. For each point $s_i \in S'$, determine its position relative to the line joining s_{left} and s_{right}. To do this, compute the vector cross product:

$$(s_{\text{right}} - s_{\text{left}}) \times (s_i - s_{\text{left}}).$$

   If the result is positive or zero (indicating that the point lies above or on the line), add s_i to set A. Otherwise, add s_i to set B.

3. Sort the points in set A in increasing order of x-coordinates, breaking ties by y-coordinates (if two points have the same x-coordinate, the one with the smaller y-coordinate comes first). Similarly, sort the points in set B in decreasing order of x-coordinates, breaking ties by y-coordinates (if two points have the same x-coordinate, the one with the larger y-coordinate comes first).

4. Form the polygon P by concatenating the points in the following sequence: $s_{\text{left}} \to A \to s_{\text{right}} \to B$.

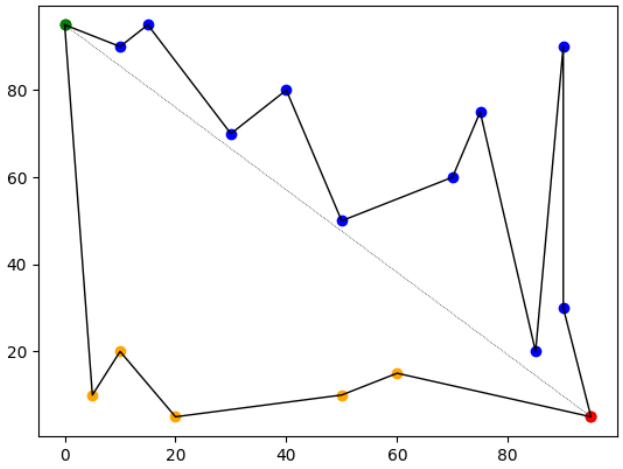

Figure 1: Construction of the simple polygon

The algorithm is illustrated in Figure 1. The green point represents s_{left}, and the red point represents s_{right}. Blue points belong to set A, and yellow points belong to set B. Points in A lie above or on the line joining s_{left} and s_{right}, while points in B lie below that line.

Proof of Correctness

Claim 1. *The polygon P constructed by the algorithm includes all points in S as its vertices, with no extra points.*

Proof. By construction, all points in S except s_{left} and s_{right} are partitioned into sets A and B. In step 4, the final polygon P is formed by concatenating s_{left}, the points in A, s_{right}, and the points in B. This ensures that all points in S are included as vertices of the polygon and no additional points are added. Hence, the claim holds. □

Claim 2. *The polygon P constructed by the algorithm is simple, meaning that no two edges intersect, except possibly at their endpoints.*

Proof. Consider each segment in the output polygon P. We will show that during the construction, any segment added to the polygon does not intersect with any other segment already added.

We begin with s_{left}. The first segment added is from s_{left} to the first point in set A. Since this is the first segment, it does not intersect with any other segment. As we proceed by increasing x-coordinates, each new segment is added to the right, while the previous segments are to the left. In the case of two consecutive points having the same x-coordinate, the point with the smaller y-coordinate comes first. This guarantees that all previously added segments are either to the left or below the current segment. Thus, as we move from s_{left} through set A to s_{right}, no intersecting segments are added. We denote these segments as L_1.

A similar argument applies when we move from s_{right} through set B to s_{left}, and we denote these segments as L_2.

Finally, segments in L_1 and L_2 cannot intersect because the segments in L_1 lie on or above the line joining s_{left} and s_{right}, while the segments in L_2 lie below that line. Therefore, the polygon P is simple. □

Time Complexity Analysis

- Step 1 requires $O(n)$ time to find s_{left} and s_{right}.
- Step 2 requires $O(n)$ time to determine the position of each point relative to the line.
- Step 3 requires $O(n \log n)$ time to sort the points in sets A and B.
- Step 4 requires $O(n)$ time to construct the final polygon.

Thus, the total time complexity of the algorithm is $O(n \log n)$.

Question 2

A convex polygon P is provided as a counter-clockwise ordered sequence of n vertices, with their locations specified as (x, y) coordinates. Given a query point q, develop an algorithm to determine whether q lies inside P in $O(\log n)$ time, using $O(n)$ space, including any necessary preprocessing. Justify the time and space complexities of your algorithm.

Input

Let us denote the convex polygon as $P = \{p_1, p_2, \ldots, p_n\}$ listed in counter-clockwise order and the query point as $q = (x_q, y_q)$.

Preprocessing

1. Identify the point with the minimum x-coordinate. If there are ties, choose the point with the maximum y-coordinate. Arrange the vertices of the polygon in counter-clockwise order starting from this point. Assume, without loss of generality, that the vertex with the minimum x-coordinate is p_1, and let the ordered vertices be $P = \{p_1, p_2, \ldots, p_n\}$.

2. For $i = 3, 4, \ldots, n-1$, draw line segments from p_1 to p_i. Since the polygon is convex, the convex angle formed by the line segment p_1p_i with p_1p_2 strictly increases as i increases. These line segments divide the polygon into $n-2$ triangles.

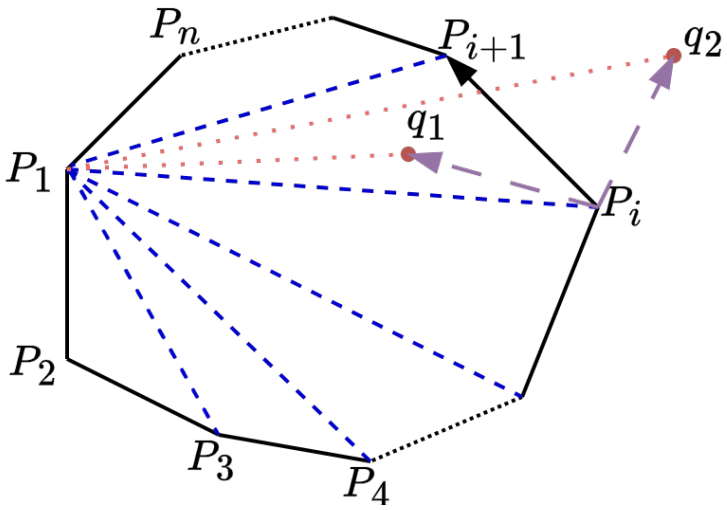

Figure 2: Illustration of locating the point q in the convex polygon

Processing a Query

1. First, check if the query point q lies inside the convex angle $p_2p_1p_n$. Any point inside the polygon must also lie inside or on one of the triangles $p_1p_ip_{i+1}$ for some $i = 2, 3, \ldots, n-1$. Additionally, the line segment p_1q must lie within the convex angle $p_2p_1p_n$.

   To verify this, compute the cross products:

   $$(p_2 - p_1) \times (q - p_1) \quad \text{and} \quad (p_n - p_1) \times (q - p_1).$$

   For the point q to lie inside the convex angle, the first cross product must be non-negative, and the second cross product must be non-positive.

   If this condition is not satisfied, q lies outside the polygon. Otherwise, proceed to the next step.

2. To find the triangle in which the query point q potentially lies, perform a binary search. Assume that q lies in the triangle $p_1p_ip_{i+1}$. In this case:

   $$(p_1p_j) \times (p_1q) \geq 0 \quad \text{for } j \leq i, \quad \text{and} \quad (p_1p_j) \times (p_1q) < 0 \quad \text{for } j > i.$$

Using this property, perform a binary search to find the maximum i such that:

$$(p_1p_i) \times (p_1q) \geq 0.$$

Denote this value as $i_{\max}$.

3. After identifying $i_{\max}$, determine if the query point lies inside the polygon.

   If $i_{\max} = n$, then p_1p_n and p_1q are collinear. To check if q lies on the line segment p_1p_n, compare the length of p_1p_n with the sum of the lengths of p_1q and qp_n. If the lengths are equal, q lies on p_1p_n and hence inside the polygon. Otherwise, q lies outside.

   If $i_{\max} < n$, then the query point potentially lies inside the triangle $p_1p_{i_{\max}}p_{i_{\max}+1}$. To verify this, compute the cross product:

   $$(p_{i_{\max}+1} - p_{i_{\max}}) \times (q - p_{i_{\max}}).$$

   If the cross product is non-negative, q lies inside the triangle. Otherwise, q lies outside the polygon. (In Figure 2, point q_1 lies inside the triangle $p_1p_ip_{i+1}$, while point q_2 lies outside the polygon.)

Time Complexity Analysis

Preprocessing

Preprocessing requires $O(n)$ time to find the minimum x-coordinate and arrange the vertices in counter-clockwise order. Joining the vertices to form triangles requires $O(n)$ time since we join $n - 3$ line segments. Note that computing the segments can either be done once during preprocessing or on the fly during query processing.

Processing a Query

- Step 1 requires $O(1)$ time to compute the cross products for determining if the query point lies inside the convex angle.
- Step 2 requires $O(\log n)$ time to perform a binary search to find $i_{\max}$.
- Step 3 requires $O(1)$ time to verify if the query point lies inside the polygon.

Since the binary search dominates the time complexity, the overall time complexity of the algorithm is $O(\log n)$.

Space Complexity Analysis

The space complexity of the algorithm is $O(n)$ for storing the vertices of the polygon. Additionally, the newly added line segments require $O(n)$ space. However, it's important to note that the segment information is redundant and can be computed on the fly. Since the input vertices are already provided, they require $O(n)$ space, and no extra space is used for storage. Therefore, the overall space complexity remains $O(n)$.