Computational Geometry (CS60064)

Homework Set 1

Vanshita Garg - 19CS10064
Ashutosh Kumar Singh - 19CS30008

Question 1

We are given a set $S = \{s_1, ..., s_n\}$ of n points on the 2D-plane. We need an algorithm to construct a simple polygon P with all the given points as its vertices, and only those.
Note: CH means convex hull.

Algorithm

- Randomly select three points $s_1, s_2, s_3 \in S$ such that no other point of S lies within $CH(\{s_1, s_2, s_3\})$. Let $S_1 := S \backslash \{s_1, s_2, s_3\}$. Now for the i-th iteration (for $1 \leq i \leq n-3$), we
  1. Randomly choose a point $s_i \in S_i$ such that no remaining point of $S_{i+1} := S_i \backslash \{s_i\}$ lies within $CH(P_{i-1} \cup \{s_i\})$.
  2. Find an edge (v_k, v_{k+1}) of P_{i-1} that is completely visible from s_i. Break this edge and replace it with the edges (v_k, s_i) and (s_i, v_{k+1}).
- The polygon P_{n-3} obtained at the end of $(n-3)$-th iteration is the desired simple polygon P with all the given points as its vertices.

Note that

- A point $s_i \in S_i$ which is suitable for Step-1 always exists as we can take the point that lies closest to $CH(P_{i-1})$.
- An edge (v_k, v_{k+1}) of P_{i-1} which is suitable for Step-2 always exists since the point s_i lies outside $CH(P_{i-1})$. This claim can be shown by induction: We first compute the supporting vertices of $CH(P_{i-1})$, Now, consider the chain from the left supporting vertex to the right one. This is the chain that faces s_i. If this chain consists of only one edge which is defined by the two supporting vertices, then it must be completely visible since both its endpoints are visible. Otherwise, if the chain consists of k edges, then consider its leftmost edge e. We are done in the case if this edge is completely visible. Otherwise, consider the leftmost edge e' which is in front of e and faces s_i. Now, the left endpoint of e' must be visible from s_i. Hence, we obtain a new chain with at most $k-1$ edges whose left and right endpoints are visible from s_i.

Time Complexity

- Selecting three points $s_1, s_2, s_3 \in S$ such that no other point of S lies within $CH(P_{i-1} \cup \{s_i\})$ takes $O(n)$ time. For this, we first randomly select a point s_1 from S. Now, selecting a point s_2 from S, that is nearest to s_1, takes $O(n)$ time since all that has to be done is to compute the distance of every point in S from s_1. Similarly, selecting the second nearest point s_3 to s_1 that does not have the same slope as $s_1 s_2$ from S, takes $O(n)$ time. Hence, overall time complexity for this step is $O(n)$.

- Selecting a point $s_i \in S_i$ in Step-1 takes $O(n^2)$ time. For this, we find the distance from all the remaining points in S to $CH(P_{i-1})$ and take the point which is nearest to $CH(P_{i-1})$. As there can be at most $O(n)$ points in S and for each point, we find its distances from at most $O(n)$ edges of $CH(P_{i-1})$, the total time complexity for this step is $O(n^2)$.

- Finding an edge (v_k, v_{k+1}) of P_{i-1} that is completely visible from s_i in Step-2 again takes $O(n^2)$ time. To find a visible edge, we iterate on all the edges of P_{i-1}. Now for the i-th edge, we connect its two end points with s_i and check whether any other edge of P_{i-1} has an intersection with any of the edges (v_k, v_{k+1}), (s_i, v_{k+1}) or (s_i, v_k). If yes, the current edge is not completely visible from s_i and is not the required edge, so we remove the edges (s_i, v_{k+1}) and (s_i, v_k) and move on to the next edge. Else if no other edge of P_{i-1} intersects with these three edges, we can say that the current edge is the desired visible edge and so we remove the edge (v_k, v_{k+1}). Since, we have to iterate on all the edges of the polygon P_{i-1} and for every current edge, we iterate on all the edges of P_{i-1} excluding the current edge, this step takes $O(n^2)$ time in the worst case.

Since we iterate on all n points of set S and for every iteration, Step-1 and Step-2 takes $O(n^2)$ time, the overall time complexity of the algorithm is $O(n^3)$.
However, we can find both the point s_i in Step-1 and a visible edge (v_k, v_{k+1}) in Step-2 in $O(n)$ time.(cf. Joe and Simpson [JS87]). And this will reduce the complexity of the above algorithm from $O(n^3)$ to $O(n^2)$.

Question 2

We are given a convex polygon P as a 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, we need an algorithm to determine in $O(log\ n)$ time and $O(n)$ space, including pre-processing, if any, whether or not the polygon P includes q.

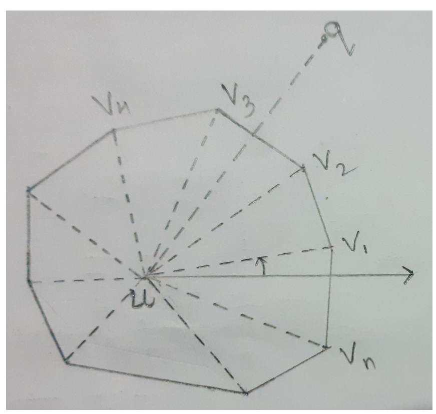

Figure 1: Example Polygon

Pre-processing

- We first choose a random point inside the polygon. We do this by taking the centroid of the triangle formed by any three consecutive vertices of the polygon. This step takes $O(1)$ time. Let us call this point u. We will treat this point as our origin.

- Suppose the sequence of points that we are given is $v_1, v_2, ..., v_n$. Now, if we consider the polar angles formed by the lines $uv_1, uv_2, ..., uv_n$, then these will form a rotated sorted array. For example, if we consider $n = 5$, then the array can look something like $[3, 4, 5, 1, 2]$. In this array, the index of the smallest element (in this example, the position of the element 1) can be found using binary search in

$O(log\ n)$ time. Note that we never create the entire array explicitly as that would take linear time. Instead, during the binary search we calculate in $O(1)$ time the polar angle value for the element at the *mid* index during the current iteration of the binary search. So, we will calculate at most $O(log\ n)$ polar angle values, and hence the time taken for this step is $O(log\ n)$.
So now we theoretically have a partition of the polar angles into two halves. In the example $[3, 4, 5, 1, 2]$, the first half would be $[3, 4, 5]$ and the second half would be $[1, 2]$. We do not store these partitions explicitly, we have the index of the smallest element, and hence we know the point of partition.

Processing each query

- First, we try to find the wedge between which the point lies. Basically, these are the two vertices of the polygon between the polar angles of which, the polar angle of the query point q lies. For example, in this diagram, we need to find the vertices v_2 and v_3.

- So, we perform a binary search over the polar angle first in the left partition to check if q lies between two vertices in the left partition. If not, then we perform another binary search over the polar angle in the right partition. We just need to handle separately the case when the polar angle of q lies between the greatest value in the right half and the smallest value in the left half. For example, consider the polar angle array as $[3, 4, 5, 1, 2]$, then this case arises when the polar angle of q is 2.6. An important point to note is that in this step too while performing the binary searches, we only compute the polar angles on the fly for those vertices which we need during that iteration of the binary search (for what we call the *mid* in a generic binary search).
Thus we get the wedge between which q lies in $O(log\ n)$ time.

- Now, we just need to perform one orientation test to determine if q lies inside, outside, or on the polygon. Since, we have the vertices in counter-clockwise order, consider the diagram where q lies between v_2 and v_3. If q lies to the left to the directed ray from v_2 to v_3, then it is inside the polygon. If it lies to the right, then q is outside the polygon. If it lies on the line joining v_2 and v_3, then it is on the polygon. This orientation test can be done in $O(1)$ time by computing the sign of the determinant $\begin{vmatrix} 1 & v_{2x} & v_{2y} \\ 1 & v_{3x} & v_{3y} \\ 1 & q_x & q_y \end{vmatrix}$. Negative means q is to the left (hence inside the polygon), positive means q is to the right (hence outside the polygon), and zero indicates that q lies on the polygon.

In this manner, we can perform the point inclusion test in $O(log\ n)$ time per query with $O(log\ n)$ pre-processing. Note that we only need $O(n)$ space because we just need to store the vertices of the polygon, others use constant space.

Some test cases to illustrate the results are on the next pages.
Points inside the polygon are red, points outside are blue, and points on the boundary are green.

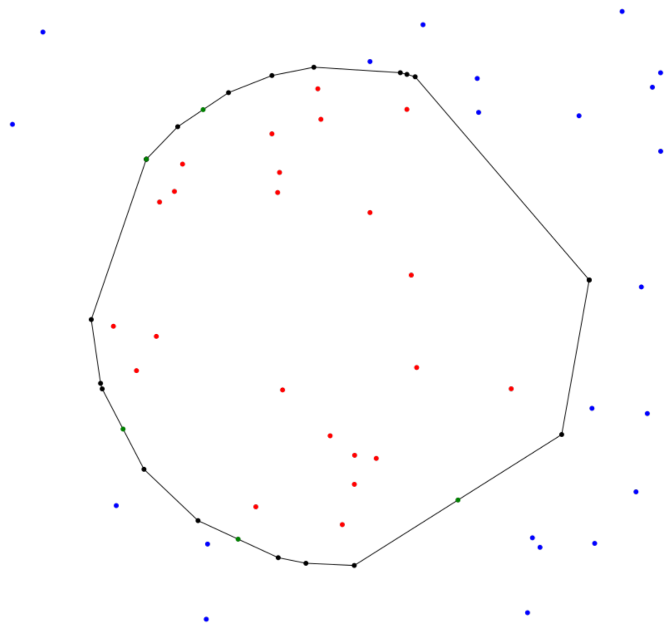

Figure 2: 20-sided polygon with 50 test points

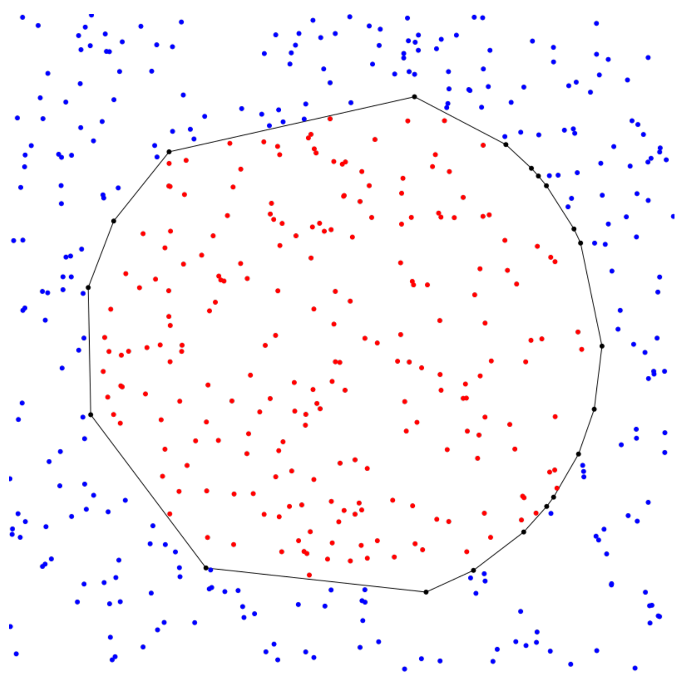

Figure 3: 20-sided polygon with 500 test points

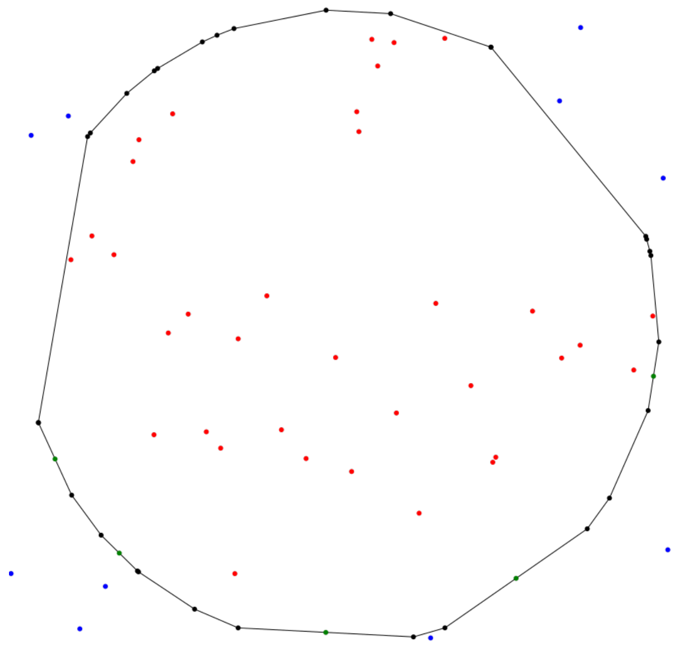

Figure 4: 30-sided polygon with 50 test points

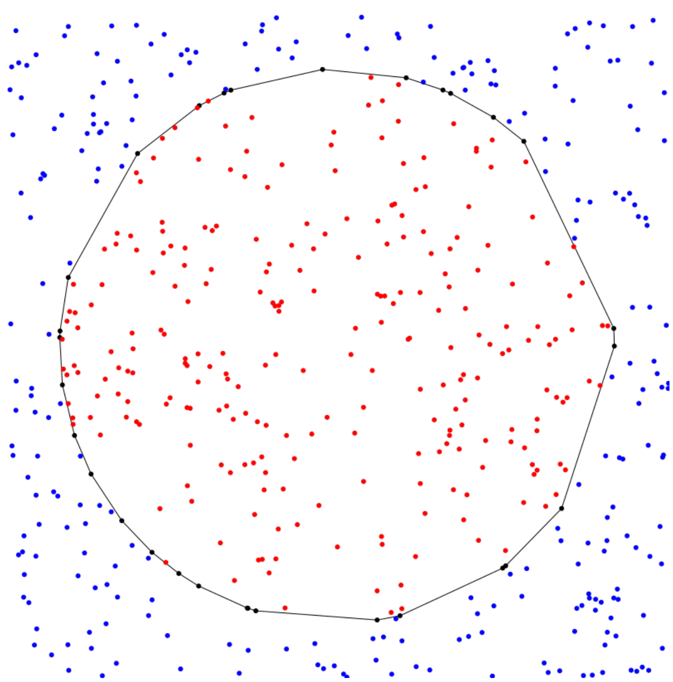

Figure 5: 30-sided polygon with 500 test points