Problem 1

Prove that the ray intersection algorithm is correct, and explain how to deal with degenerate cases. (One degenerate case is when ρ intersects an endpoint of an edge. Are there other special cases?) What is the running time of the algorithm?

> Clearly if there are 0 intersections, the point must be outside the polygon. Otherwise the polygon would be unbounded and therefore not simple. If the ray does intersect the polygon, then the final intersection it makes must leave the polygon by the same reasoning above. Note that two consecutive intersections will maintain the current state of the ray. If the ray currently is inside the polygon, two intersections means it must have left the re-entered the polygon. By similar reasoning if the ray was outside then two intersections leaves it outside. Therefore adding 2 intersection maintains the state of the ray intersections. Since we know 0 intersections corresponds to being outside the polygon, then $0 + 2n = 2n$ intersections corresponds to being outside the polygon. Similarly, 1 intersection corresponds to being in the polygon so $2n + 1$ intersections corresponds to being in the polygon. Hence even parity means being outside and odd being inside.
>
> There are two degenerate cases. One is when the ray intersects a vertex, and one when the ray intersects an edge parallel to itself. When the ray intersects a vertex, we can simply check the adjacent edges. If both are on the same side of the ray, then we dont count the intersection. If the edges are on opposite sides however, we do. In the case of a parallel edge, the same procedure is done with the adjacent edges to the parallel edge.
>
> Testing the intersection between a ray and an edge can be done in constant time by checking if the endpoints of the edge lay on opposite sides of the ray. Therefore doing this check for all $O(n)$ edges gives a running time of $O(1) \cdot O(n) = O(n)$.

Problem 2

A polygon P is called *star-shaped* if a point p in the interior of P exists such that, for any other point q in P, the line segment $\overline{pq}$ lies in P. Assume that such a point p is given with the star-shaped polygon P. As in the previous two exercises the vertices of P are given in sorted order along the boundary in an array. Show that, given a query point q, it can be tested in time $O(\log n)$ whether q lies inside P. What if P is star-shaped, but the point p is not given?

> Since P is star shaped and p can see every point of P, we know that if a ray eminated from p is swept radialy around, it must only be intersecting a single edge of P at a time. If this was not the case, then there will be some point in P not visible by p. This means we can order the vertices of P by the angle they

make with respect to p. Since we are given P in a sorted order, we can do a binary search on the angles they make with p and compare it to the angle between p and q without having to make a new list. The angles for comparisons can simply be calculate on the fly in constant time as the search is happening. This search will give two adjacent vertices in which the angle q makes with p lies between. The edge the two vertices create can then be checked for intersection against $\overline{pq}$. If there is an intersection, q lies outside of P. Otherwise q is in P. Since this is simply a binary search on the existing list of vertices, this takes $O(\log n)$ time.

If p is not given, then we can still find the region in which such a p could exist by using the prior algorithm outlined in our previous homeworks. This region can be sample for a such a point p and the above outlined algorithm can be used. Finding the region to sample p takes $O(n)$ time and so the complexity is $O(n+\log n) = O(n)$.

Problem 3

Suppose you are given an n-vertex simple polygon, P. Describe how to build an efficient data structure for determining in $O(\log n)$ time whether a query point, q, is inside of P or not. What is the space and preprocessing time for your data structure?

We will use the method of Kirkpatrick's Point Location algorithm.

Algorithm 1 BuildStructure(P: polygon)

```
1. Triangulate $P$ and mark each triangle as inside the polygon
2. Find 3 points whose triangle bounds $P$
3. Triangulate the space between this bounding triangle and $P$ are mark these
   triangles as outside the polygon.
```

This preprocessing step takes $O(n \log n)$ time overall since triangulating P can be done in $O(n \log n)$ time, and the outer triangulation only is adding 3 more vertices and hence only takes $O(n \log n)$ time as well. Since we are triangulating a simple polygon, the number of triangles in $O(n)$ meaning the space complexity is just $O(n)$. With this triangulation in hand, querying the point is as simple as using Kirkpatrick's algorithm.

Algorithm 2 QueryPoint(T: triangulation, p: point)

```
1. If $p$ does not lie in the bounding triangle, return false
2. Otherwise, run Kirkpatrick's algorithm to determine which triangle $p$ lies
   in. If it is marked inside the polygon, return true and otherwise return false.
```

Since we are using Kirkpatrick's algorithm, we still maintain space complexity of $O(n)$ and querys take $O(\log n)$ time as desired.

Problem 4

Design a *deterministic* algorithm, that is, one that doesn't make random choices, to compute the trapezoidal map of a set of non-crossing line segments. Use the plane sweep paradigm from Chapter 2. The worst-case running time of the algorithm should be $O(n \log n)$.

Algorithm 3 CreateMap(L: line segments)

```
1. Initialize both an empty queue $Q$ and status structure $T$
2. Insert all segment's endpoints in $Q$ and store the segment alongside all left
   endpoints
3. Add two special segments to $T$ to start off
     • Let $x_{\min}$ and $x_{\max}$ denote the minimum and maximum $x$-coordinates of
       any segment endpoints in $L$
     • Let $y_{\min}$ and $y_{\max}$ denote the same as above but for $y$-coordinates
     • Add the horizontal segments $(x_{\min}, y_{\min}) \rightarrow (x_{\max}, y_{\min})$ and
       $(x_{\min}, y_{\max}) \rightarrow (x_{\max}, y_{\max})$
4. While the queue is not empty
     (a) Pop the next point $p$ from $Q$
     (b) Find the line segment that is nearest to $p$ going upwards and denote it
         as $l_u$
     (c) Find the line segment that is nearest to $p$ going downwards and denote
         it as $l_d$
     (d) Create and add a vertical segment to the trapezoidal map by considering
         the vertical line eminating from $p$ and using the intersection points that
         line makes with $l_u$ and $l_d$ as the endpoints
     (e) If $p$ is a left endpoint, add its associated line segment to $T$
     (f) Otherwise remove the associated line segment to $p$ from $T$
```

For complexity, if both Q and T are some form of balanced binary search tree, then we have $O(\log n)$ processing time for the updates and deletions. Since there are $2n$ endpoints and 4 other endpoints being processed, in total there are $2n + 4$ events.

Therefore the running time of the plane sweep is $O((2n+4)\log n$ which is simply $O(n\log n)$.

Problem 5

Give a data structure for the vertical ray shooting problem for a set S of n non-crossing line segments in general position. Bound the query time and storage requirement of your data structure. What is the preprocessing time? Can you do the same when the segments are allowed to intersect each other?

We can use the trapezoidal map alongside a persistent data structure to have an efficient data structure in both terms of querying and storage.

Algorithm 4 BuildStructure(S: non-crossing line segments)

```
1. Construct a trapezoidal map $T$ from the segments in $S$
2. Build a persistent search tree using $T$ and return it
```

The complexity of making the data structure is $O(n\log n)$ since building the trapezoidal map takes $O(n\log n)$ time and building the persistent search tree takes $O(n\log n)$ time as well. Hence the preprocessing time is $O(n\log n)$. The use of a persistent search tree also means the space complexity is $O(n)$. The query algorithm is very simple:

Algorithm 5 FindIntersection(T: trapezoidal search tree, p: point)

```
1. Perform a point location query of $p$ on $T$
2. Find the trapezoid containing $p$ and return its top edge
```

Finding this edge takes $O(\log n)$ time since the point location query takes $O(\log n)$ time and determining the top edge of a trapezoid takes just $O(1)$ time since it has a bounded number of edges.

In total then, the given data structure and ray algorithm has $O(n\log n)$ preprocessing time, $O(n)$ space complexity, and $O(\log n)$ query time.