Problem set 2

Due **1pm, Thursday November** 1^{st}.

Required Part

Problem 1 [10 points]. Show how to solve the orthogonal range query problem in d dimensional space using $n^{O(d)}$ space but with $O(d \log n)$ query time.

Problem 2 [5 points]. Exercise 6.7, p. 145. A polygon $\mathcal{P}$ is called *star-shaped* if there exists a point p in $\mathcal{P}$ such that for any other point $q \in \mathcal{P}$ the segment $p - q$ belongs to $\mathcal{P}$. Assume that the vertices of $\mathcal{P}$ are given in sorted order along the boundary in an array. Show that given a query point q one can detect if $q \in \mathcal{P}$ in $O(\log n)$ time.

Problem 3 [5 points]. Exercise 7.11, p. 163. Let P be a set of n points in the plane. Give an $O(n \log n)$ time algorithm to find, for each point in P, another point in P closest to p.

Problem 4 [15 points]. Exercise 8.13, p. 182. Given a set of n lines in the plane, give an $O(n \log n)$-time algorithm to compute the maximum level of any vertex in the arrangement induced by the lines.

Problem 5 [15 points]. Give an example of a set of lines in the plane where the greedy method of constructing an autopartition (where the splitting line is chosen to minimize the number of segments cut, and ties are broken arbitrarily) results in a BSP of quadratic size.

Extra credit problems. The points obtained for the extra credit problems will be *added* to your score from other sections. E.g., if you solve the problem below, and get 100 points for the rest of the homework, you will get 150 points overall.

Problem X [50 points]. Consider a version of the greedy algorithm from Problem 5 where a tie is broken by choosing a line which results in the most balanced partition (i.e., such that the difference between the number of elements on the left and on the right side of partition line is minimized). Show that the resulting algorithm can still create a BSP of quadratic size.

Hint: At present we do not know how to solve this problem.

Optional Part

Problem A [10 points]. Suppose you are given a set of halfspaces in d dimensions, and wish to compute their intersection, using only an LP algorithm and an algorithm to compute the convex hull of a set of points (not a set of planes). Describe a duality transform suitable for this situation. (Hint: use the well-feasible LP algorithm from PS1.) Write pseudo-code for the algorithm IH (i.e., intersection of half-spaces) given the sub-routines CHP (i.e., convex hull of points) and LP (i.e., linear programming).

Problem B [15 points]. Exercise 9.11, p. 209. A Euclidean Minimum Spanning Tree (EMST) of a set P of n points in the plane is a tree of minimum total edge length connecting all the points. Show that the set of edges of a Delaunay triangulation of P contains an EMST for P and thus that one can compute EMST in $O(n \log n)$ time.

Problem C [15 points]. Exercise 8.16, p. 182. Let S be a set of n segments in the plane. A line l that intersects all segments in S is called a *stabber* for S.

1. Give an $O(n^2)$ algorithm to decide if a stabber exists for S
2. Give a (randomized) $O(n)$ algorithm for the same problem if all segments in S are vertical.

Programming Exercise.

The programming exercise this time has these objectives:

1. Gain familiarity with manipulating higher-order geometric primitives;
2. Interleave interaction, computation and display;
3. Produce an "inspection" capability for construction and display of a data structure;
4. Get some hands-on experience with an interesting geometric data structure.

The vehicle data structure we'll use for the exercise is BSP trees, but the development will use DCELs and ideas from point location and arrangement walking as well.

If you had trouble getting the basic interface working for PS1, feel free to use Jason Yang's or Daniel Vlasic's code as a starting point for PS2. There are some public-domain Java BSP implementations out there, which you might want to look at as well, but you should write the BSP code yourself.

This exercise will end up with a set of methods for incremental construction and querying of a 2D BSP tree of 2D line segments. You can design things any way you wish, of course, but here are some suggestions for how you might proceed.

Define a Vertex class to represent a 2D point (x, y).

Define a Halfspace class to represent the halfspace $ax + by + c > 0$.

Define a Segment class to represent a 2D line segment with two Vertex endpoints P and Q, and the Halfspace defined by the left side of the line directed from P to Q.

Define a Face class (after the DCEL shown in the text, and described several times in class) to represent a planar subdivision. Make your constructor initialize the Face to (the CCW contour of) a 2D bounding box.

Define a BSPCell class, which has a Face, and (if not the root) a pointer to the parent BSPCell, and (if not a leaf) a Segment upon which the Cell has been split, and two (possibly NULL) pointers to child BSPCells.

Define a colorful .Draw() method for each class above. (You can assume that a Halfspace is always accompanied by a Segment, so you know where to draw it.)

Define Segment.Clip(H), which clips the segment to a given halfspace H.

Define BSPCell.Split(S), which splits the (leaf) cell by a given segment S.

Define BSPCell.Insert(S), which inserts a given segment S into the BSP tree. There are several ways to do this. Here's a suggestion for a recursive, top-down definition of Insert():

```
Insert(BSP Cell B, Segment S)
  Clip S to B
  If it clips out, return
  if B is a leaf
    Split B by S
  else
    Insert S into B's children
```

Now initialize a root BSPCell with a Face that has been initialized to the 2D bbox (say, your working screen area). Define a segment entry method (probably the easiest is to mouse-down, sweep out the segment, then mouse up), and Insert each generated Segment into the BSP tree.

Define BSPCell.Locate(P), which identifies the cell containing a given point P. Put an eye-point icon E (say, a point from PS1) into the scene, and make it selectable and draggable by the mouse. Point-locate on E, and highlight the resulting cell.

Define BSPCell.Traverse(P), which generates a back-to-front ordering of the Segment fragments in the BSP tree. Draw the fragments, each with its render order displayed nearby as an integer. Or, assign each Segment a unique color and "render" the scene onto the view-line (analogous to the view-plane, in 3D) of the eyepoint.

Extra Credit:

Implement leaf-level insertion of S, by point-locating on one endpoint of S, then walking through the leaf cells of the BSP tree (splitting as you go) until you reach the other endpoint. This requires neighbor pointers connecting the leaf cells (possibly as half-edges in the DCEL, although this constrains neighboring cells to match in a way that you might not want), so that the segment insertion can step across them.

Define the portals on the boundary of a BSP cell as the complement of any Segments on the boundary. Represent the Portals as a list of (complement) Segments for each Cell. Render the portals. Render the connected component reachable from any query Cell through all portal sequences originating at that Cell.

Enable your GUI to select and drag each Segment. (Suggestion: if you select the segment near an endpoint, only that endpoint moves; but if you select the segment near its midpoint, the whole thing translates rigidly.) Now dynamically extract, and reinsert, the Segment as it is dragged around. (Warning: it is hard to do this efficiently.)