Amy Bui
HW 6

1 Data Structures

(a) We can use an segment tree as the primary data structure $\mathcal{T}$ to store horizontal interval segments and their y-positions. Using the segment endpoints x-coordinates, the primary tree can order the segments by the x-ordered intervals. This way, we can query which segments contain query point q_x in $O(\log^2 n + k)$ time, where k is the number of segments containing q_x. But we need to narrow the answer to those segments whose y-positions are in the range of $[\ell_y, h_y]$. So in addition to the primary data structure, the associated data structure can be a list that contain the y-position of a segment; so each node contains the segment it covers and a list of those segments' y-positions. Additional segments that the vertical query segment q may pierce can be found by traversing the segment tree as described in the book (p. 234).

The time to build this augmented tree is the same as the preprocessing time for building a regular segment tree in $O(n \log n)$ time, where n is the number of segments and the depth is $O(\log n)$. I don't think this tree needs another tree as the associated data structure at each node, since we only need the y-positions of segments that contain a point with x-coord q_x.

Likewise, storage is $O(n \log n)$ for n segments and a tree of depth $O(\log n)$. The addition of the y-position list at each node is only for those nodes that contain a segment interval, and there is only every at most $O(n)$ additional space for the y-position for the entire tree, not each not. So the space is $O(n \log n)$, rather than $O(n^2 \log n)$.

(b) As the problem suggests, we can organize the points in an interval that also uses two treaps as the associated data structures. The primary tree $\mathcal{T}$ is built such that nodes are organized by the x-values of the points. So we can traverse down the tree until we meet a median x value stored in a node that is contained by the ractangluar query's x interval. Once there, we traverse down each subtree to find the next nodes just within the query's x-range (i.e. the leftmost node in the left subtree and the right most node in the right subtree). Each node can have two treaps[1] built from the points of the enture subtree of the current node, as a treap can report the points in an open rectangluar query range in $O(\log n + k)$ time (lemma 10.8, page 230) for k solution points; one S_L treap that can report the left closed interval of points the query, and one S_R treap that can report the right closed interval of the query, where the final solution is the points contained in both query sets.

The storage for one normal BBST is $O(n)$; each node also stores treaps based on the node's subtrees' point, so with each level deeper into the tree, the associated data structure becomes smaller because the tree is balanced; but the total storage is still $O(n^2)$ because each level of the tree stores a total of $O(n)$ additional space for all associated treap on each level of the primary tree. Likewise, preprocessing would take

[1] Jake and Diane discussed this augmentation for the primary tree that works to find points in two rectangular intervals

$O(n^2 \log^2 n)$ time because there are n points for the primary tree of depth $O(\log n)$, and each node's treaps take $O(n \log n)$ to build. Query would take $O(\log n + A)$ where A is the number of solutions in the query range: it takes $O(\log n)$ time to find the median x in the node of the primary tree and the target nodes, and an additional $O(\log n + A)$ for each of the node's two treaps to report the points in the rectangular range.

2 Boundary of Intersection

(a) Given two arbitrary simple polygons of n vertices, we can compute the boundary of the intersection with a vertical line sweep. In addition to stopping at each vertec point, we must also stop at each intersection point in the set, so there are $O(n^2)$ stops. At each stop, we update along the sweep line the faces we passed and update how many polygons the current face covers (0, 1, or 2). For those stopping points that change the count from 0 to 2, 2 to 0, 1 to 2, or 2 to 1, add those points to the solution, as they indicate entering/leaving an intersection region of the two polygons. There is $O(\log n)$ work at each stopping point, so the total runtime is $O(n^2 \log n)$[2]. Storage remains $O(n)$ if the status DS is updated at each pass, removing points allready processed.

(b) We can solve a similar problem with two recilinear polygons with line sweep, as well, where the stopping points are in a segment tree DS ($O(n \log n)$ space) and we only need to stop and process the intersecting regions at each *vertical* line segment in the data structure, rather than stop at every point and intersection point. There are now only $O(n)$ stops, and we process how many polygons the current face covers and report solution points similar to how we did above: report points that make the count go from 0 to 1, 1 to 2, 2 to 1, and 1 to 0 indicate intersection regions for rectilinear polygons. With $O(\log n)$ time processing at each stop, the total runtime is $O(n \log n)$.

References

[1] Jake and Diane's office hours.

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. Computational Geometry: Algorithms and Applications (3rd ed. ed.). Springer-Verlag TELOS, Santa Clara, CA, USA.

[2]Jake went over how to determine this method's runtime in OH.