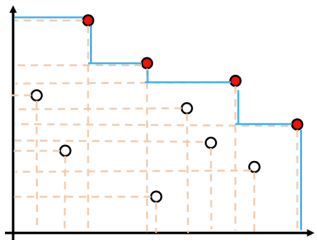
1 Maxima Finding

a. Set S with the maxima highlighted.



b. **Incremental Maxima**: Sort the n distinct (multiple points do not overlap) points of S in descending order by their x-coordinate ($O(n \log n)$). For two points $i, j \in S$, where $i \neq j$, if $i_x = j_x$, then point i comes before j if $i_y > j_y$, and vice versa (order the higher y-coordinate point first). (A) The first point v in the sorted set, S', is trivially in the set of S's maxima, because v has the largest x value (and higher y value against any points with the same x value, which v therefore dominates). Let m be the *latest* maxima found so far, and so, m_y is the current largest y-coordinate; $S'[0]$ is, of course, our first m. Checking each point in S' in order, (B) if the current p has a $p_y > m_y$ seen so far, then add p to the set of maxima and make p the new m, because our S' ordering has $p_x \leq m_x$, so m doesn't dominate p. (C) If the current $p_y \leq m_y$, reject it from the maxima set; since we ordered S' by descending x-coordinate, $p_x \leq m_x$ ($p_y < m_y$ if $p_x = m_x$), p is dominated by m. This part of the algorithm runs $O(n)$ time and correctly computes all the maxima of S. Space complexity may be $O(1)$ if we sort in place and toss out the points of S' that are not rejected. Overall runtime is $O(n \log n)$ due to preprocessing.

c. **Divide-and-Conquer Maxima**: Sort the n distinct points of S in ascending order by their x-coordinate, and call it V ($O(n \log n)$). Perform this D&C operation on V, also $O(n \log n)$:

1) If $|V| \leq 1$, then return the current set V, as it is trivially a set of maxima.
2) Otherwise, divide set V evenly into two subsets L and R, for the left and right half of the list. Run algorithm from step 1 on L to produce L' (maxima of L). Likewise, run it on R to produce R' (maxima of R).
3) Merge L' and R' to produce maxima set of V: All $r \in R'$ are maxima of V, because they are the maxima of the right subset; since the set is ordered, no $\ell \in L'$ has a $\ell_x > r_x$ for any $r \in R'$. An r cannot be dominated by an ℓ. For any $\ell \in L'$ where $\ell_y < R'[0]_y$, discard that ℓ, because $R'[0]$ (the tallest point in R') dominates ℓ. Keep only those ℓ whose $\ell_y > R'[0]_y$, because those are not dominated by any maxima on the right subset. Return the $L'' \cup R'$ as the maxima set for V.

Likewise, space complexity may be $O(1)$ if we sort and do the D&C in place. Runtime for the D&C is $O(n \log n)$, $O(\log n)$ divisions and $O(n)$ for the merging.

d. **Dynamic Maxima**[1]: Sort the n distinct points of S in ascending order by their x-coordinate ($O(n \log n)$). Let these sorted vertices be the leaves of a Balanced Binary Search Tree we shall construct from the leaf level to the root level; the root will contain the maxima of set S. At each level of the tree, starting at the leaves, compute the maxima between two adjacent nodes using the strategy described above in part c step 3 when finding the set of maxima between two sets of maxima. Store the resultant maxima set in the parent node of the two adjacent nodes you just compared, in the next level up. Stop when we've computed the root node, which contains maxima of S, and is correct using the previous strategy. Since this is a BBST, height of the tree is $O(logn)$, so computing the whole tree was $O(n \log n)$, which works in worst case where every vertex is also a maxima. Insertion and deletion from the BBST as described in class will also be $O(\log n)$ time. And space complexity is $O(n \log n)$ for worst case described previously.

e. **Marriage-Before-Conquest Maxima**[1]: Sort the n distinct points of S in ascending order by their x-coordinate ($O(n \log n)$).

   1) We can compute the median x-line to split the current set into a left L and right R subset in $O(n)$ time.
   2) On the R side, we find the max y-coordinate point in $O(n)$ time, we call that point $r_{\max}$. Using $r_{\max}$, we can toss out/prune all the points p to the left of $r_{\max}$ (i.e. all p such that $p_x < r_{\max\ x}$) if $p_y < r_{\max\ y}$, i.e. we prune points which point $r_{\max}$ can dominate, as they cannot be in the maxima we are computing. Pruning takes $O(n)$ time.
   3) Repeat from step 1 on whats left of the L and repeat it on what's left of the R side after the pruning.
   4) Combine the maxima we found for L and R, and report that as the maxima of the current set.

   Since we can have $O(\log n)$ of these recursions, the overall runtime is $O(n \log n)$. Space complexity is $O(1)$.

2 Line Sweep and Augmented Data Structures

[1] Went over dynamic and MBC algorithm with Jake in office hours, with Alex and Stephanie.

3 Convex Polygons and Monotone Polygons

a. **Convex Polygon**[2]: When the given vectors are translated to the origin (in the order of a walk around the polygon), the polygon is convex if: 1) the angle given by a vector and the next vector translated must extend in the one direction thats the same as the previous two's direction, i.e. the next vector must not cross over previous vectors added, as this would indicate a two different turns the edges of the polygons take, leading to a concave or complex polygon; and 2) the vectors should only form one rotation around the origin, and cannot continue rotating about the origin more than once, as this corresponse to a polygon that self intersects despite all directed edges forming one kind of turn, as the example in Homework 1.1. Checking this takes $O(n)$ time as we are checking each vector formed by two adjacent vertices.

b. **Monotone Polygon**[2]: The following alorithm can run in $O(n)$ time to see if polygon P with n vertices is monotone:

   1) Take an arbitrary vector starting point v_0 to point v_1, translate that vector to the origin.

   2) For each point $v_i \in P$ for $i \in [1, n]$ in order walking about the edge of P, translate vector from v_i to v_{i+1} to the origin, and increment the count in area of the plane by 1, as it gets swept over by the angle formed by vectors $\overrightarrow{v_{i-1}v_i}$ and $\overrightarrow{v_iv_{i+1}}$. I call these areas we increment on slices. This takes $O(n)$ time to translate and "sweep" all the vectors.

      i. To save space and make space complexity constant, we can toss out the slices that have a count greater than 2, as a line of monotonicity cannot lie perpendicular to anything in that area.[3]

   3) We look for these area slices with a count of only 1. If there are at least two of these slices and a straight line ℓ can intersect both slices across the origin, then ℓ is the line perpendicular to the line of monotonicity of P and so P is montone. It also takes $O(n)$ to look at all the slices, so overall runtime is $O(n)$, and space complexity is constant if we prune.

[2]Discussed verification of these using vectors with Diane and Jake at office hours.
[3]Diane discussed space efficiency about these regions at office hours Alex and Stephanie.

4 DCEL

1. Like how we learnt in class, we can compute the intersection points of two convex polygons in $O(n)$.

2. Ignore regions of P_2 that exist outside the convex hulls intersections.

3. Use an algorithm similar to one described in our text book[4]. In essence, wherever an edge of P_2 intersects P_1, we locally recompute the DCEL for P_1 to include this new local edge. Since P_1 is already triangulated, any triangle intersected by a line forms another triangle and a quadrilateral. We can ignore retriangulation, as it is not necessary. Since we are inserting new edges locally into where they intersect P_1, making these updates is constant time, and there s $O(n)$ insertions[5]. In addition to adding these edges, since the DCEL of P_1 gives us information about the faces of the polygon, we can add more attribute information to the new faces affected by the new edge insertions, namely, if a face is an overlap of an interior region of P_2 and interior region of P_1, and overlap of an interior region of P_2 and exterior region (pocket) of P_1.

4. At this point, we can conclude that we have all points of P_2 that intersects P_1. Now we can walk along the edge of P_2 and find the areas that represent the $P_1 \cap P_2$. At a point on P_2, which may be an intersection point or point of a new edge in the DCEL, start walking along the edges of P_1 where which the the face the edge is connected to is marked as being overlapped by an interior region of P_1 and P_2. If we return to the starting point, report this polygonal intersection. If there are still points on the bounds of P_2 we have not reported, go to the next one, and repeat the above procedure. Repeat until we've seen all marked intersection points on the bounds of P_2. We have report all the intersections of $P_1 \cap P_2$.

References

[1] Jake and Diane's office hours, classmates: Stephanie, Alex, Anju with homework problem discussions.

[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.

[4] Section 2.1 describes a sweep line algorithm that is used in an algorithm in Section 2.3 for recomputing the half edges of an overlay of two subdivisions, page 34-35 [2].

[5] Diane went over two versions of this solution in detail at office hours, and I try to describe a version of it here.