CS170 Problem Set 14 Out: Apr 30, 2001
Due: May 7, 2001

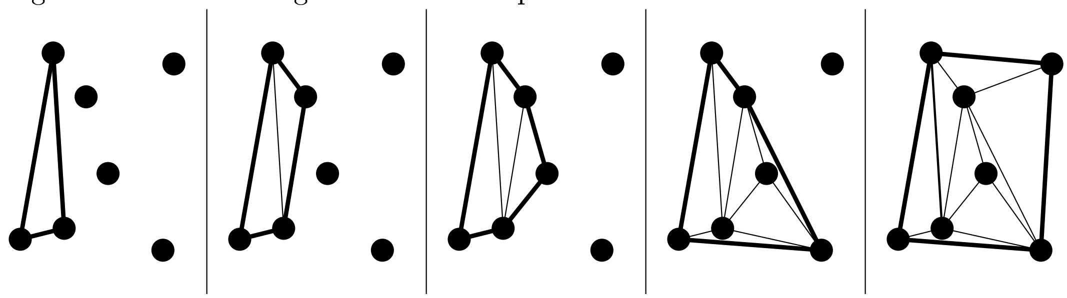
Because this homework was issued so late, it is not due until Monday, May 7 at 4pm.

1. CLR Exercise 33.1-5.
2. CLR Exercise 33.2-6.
3. CLR Exercise 33.2-9.
4. CLR Exercise 33.4-1.
5. CLR Exercise 33.5-2.
6. Describe an algorithm for sorting a set of vectors in counterclockwise order about the origin in $O(n \log n)$ time.

   The problem here is that the order is circular, so traditional sorting methods like merge-sort don't quite work. When you sort integers, you know that if $i < j$ and $j < k$, then $i < k$. However, if vector a is ccw (counterclockwise) from vector b, and vector b is ccw from vector c, it is not necessarily true that a is ccw from vector c. (Think of a Y shape.)

   Assume that each vector is specified as an x-coordinate and a y-coordinate. Your solution must be comparison-based (no bucketing or radix-sort tricks), and may use orientation tests and comparisons of coordinates (like "`if` $a_x < b_x$ `and` $b_y > 0$"). Orientation tests are performed by calling `orient`(a, b), which returns a positive number if a is ccw of b, a negative number if a is cw of b, and zero if a and b are collinear. Make sure your algorithm works correctly if several vectors point in the same direction (in which case they are treated as being equal) and if there are vectors that point in precisely opposite directions.

7. A *triangulation* of a set of points P is a set of triangles that covers (completely fills) the convex hull of P, where the triangles may not have overlapping interiors. Every triangle vertex must be in P, and every point in P must be a triangle vertex. The rightmost figure below is a triangulation of seven points.



Consider the following algorithm for triangulating P. Assume no two points have the same x-coordinate. Sort the points by x-coordinate, from left to right. Call the first three points p_0, p_1, and p_2, and construct the triangle $\triangle p_0p_1p_2$ (see above left).

The algorithm is incremental, meaning it maintains a triangulation of all the points visited so far, and adds one point at a time, creating a new triangulation that includes the new point as well as the previously visited ones. When we are adding point p_i (for $i = 3, 4, \ldots, n-1$), we find all of the vertices on the convex hull of the previous i points that are visible from p_i. A point p_j is *visible* from p_i if the line segment $\overline{p_ip_j}$ does not intersect the convex hull, except at the point p_j. For each such vertex, we create a triangulation edge $\overline{p_ip_j}$ (which we write to the output). The illustration above demonstrates this algorithm from beginning to end.

At all times, the algorithm maintains a circularly-, doubly-linked list of the vertices of the convex hull (bold edges in the illustration). After each new point is added, the linked list is adjusted to reflect the new convex hull.

Your job is to fill in the details of the algorithm so that it runs in linear time. (Assume we use a linear-time radix sort for the initial sorting step.)

(a) Describe (in clear English) an algorithm that will quickly find the convex hull vertices that are visible from p_i. Be sure to discuss all orientation tests used, and how you use their results. Important: your answer should not check more convex hull vertices than necessary. An algorithm that checks every convex hull vertex for every new point will take quadratic time on some point sets.

(b) Write pseudocode to insert a vertex p_i. The pseudocode should do three things: implement your answer to part (a), write all the newly created triangulation edges to the output, and update the circularly-, doubly-linked list of convex hull vertices. Orientation tests are performed by calling `orient`(a, b, c), which returns a positive number if a, b, and c occur in counterclockwise order, a negative number if they occur in clockwise order, and zero if they are collinear.

To implement the linked list, assume each vertex has a `ccw` and `cw` pointer. For example, p_3`.ccw` references the next vertex counterclockwise of p_3 on the convex hull boundary, and a statement like a`.cw` $\leftarrow b$ sets the next vertex clockwise from a on the convex hull boundary to be b. It's up to you to keep these pointers correct as the convex hull changes. For points not on the convex hull boundary, the values of `ccw` and `cw` are irrelevant.

(c) Show that your algorithm runs in linear time. Hint: any n-vertex triangulation has $O(n)$ triangles and $O(n)$ edges.