Code Jam 2019 - World Finals

Analysis: Juggle Struggle: Part 2

We can represent each pair of jugglers from the input as the endpoints of a segment. The problem then asks us to find two of those segments that do not intersect, or report that there is no such pair.

Test Set 1 (Visible)

Test Set 1 can be solved by checking every possible pair of segments to see if they intersect. Let pq be the segment between points p and q, and rs be the segment between points r and s; then these segments intersect if and only if

- (r - p) × (q - p) has the same sign as (s - p) × (q - p), and
- (p - r) × (s - r) has the same sign as (q - r) × (s - r),

where × stands for the cross product. This works only when no three of the points are collinear, but that is the case in this problem. Since each check takes constant time, this algorithm runs in $O(\mathbf{N}^2)$ time.

Test Set 2 (Hidden)

Let L_i be the line that fully contains segment S_i. Let S_i and S_j be two segments that do not intersect. If L_i and L_j are not parallel, at least one of S_i and S_j does not contain the intersection between L_i and L_j. We can find for every input segment S_i, we can find whether its line L_i intersects some other line at a point outside of S_i (or not at all). Let F be the set of all such segments. Then, every non-intersecting pair contains at least one segment in F and the size of F is at most 25. We can do a linear pass over all other segments to see whether or not they intersect each segment in F to find the segments not in F that also participate in non-intersecting pairs. We focus now on finding all segments to put in F.

We now present three algorithms. The first two are ultimately similar: the first one uses less advanced theory, but requires more problem-specific proofs because of it. They both require somewhat large integers. For C++, `__int128` is enough to represent all values, but since we need to compare fractions that are the ratio of two `__int128`s, we'll need special comparison code. For Java, `BigInteger` will do. The third algorithm is a separate approach that uses a more advanced data structure, and requires only 64-bit integer arithmetic.

A solution using less advanced previous knowledge

First, assume there is a purely vertical segment (that is, its two endpoints have the same x coordinate). If we find more than one of those, we add all of them to F, since they don't overlap. If we find a single one, we can check it against all others like in the previous solution in linear time. In what follows, we assume no vertical segment is present.

Consider the extension of each segment S_i to a full line L_i. We will find the x coordinates of the leftmost and rightmost intersection of L_i with all other L_js and check that they are inside the range of x coordinates where S_i exists. If one of those intersections is not inside that range, then we found one or two segments to put in F. Notice that finding all rightmost intersections is equivalent to finding all leftmost intersections in the input symmetric to the y axis, so if we have an algorithm that finds the leftmost ones, we can just run it twice (and reflecting the input takes

linear time). Moreover, suppose we restrict ourselves to the leftmost intersection of L_i such that L_i is above the other intersecting line to the left of the found intersection. Let us call these "leftmost above intersections". We can use an algorithm that finds those intersections once on the unchanged input and once on the input reflected across the x axis to find the analogous "leftmost below intersections". In summary, we develop an algorithm that finds "leftmost above intersections" and then run it 4 times (using all combinations of reflecting / not reflecting the input across each axis), to find all combinations of "leftmost/rightmost below/above intersections".

To find all "leftmost above intersections", the key observation is that if two lines L_1 and L_2 intersect at x coordinate X, and L_2 is below to the left of the intersection, then L_2 cannot participate in any leftmost below intersection at coordinates X' > X. L_2's own intersections at coordinates X' > X are not leftmost. If L_2 intersects an L_3 that is below L_2 to the left of their intersection at X' > X, then L_3 intersects L_1 to the left of X' because of continuity: L_1 is below L_2 to the right of X.

This leads to the following algorithm: let X_0 be the minimum x coordinate among all endpoints. Sort the lines by y coordinate at X_0. Let L_i be the line with the i-th highest y coordinate. We iterate over the lines in that order, while keeping a list or ranges of x coordinates and which previously seen line is below all others there, since that is the only one that can produce leftmost below intersections in that range. We keep that list as a stack.

At the beginning, we push (X_1, 1) onto the stack, where X_1 is the maximum x coordinate among all input points. This signifies that L_1 is currently below in the entire range of x coordinates. Then, we iterate through L_2, L_3, ... $L_{\mathbf{N}}$. When processing L_i, we find the x coordinate of its intersection with L_j and call it X, where (j, X') is the top of the stack. We check the intersection to see if it is within the x coordinate range of the two corresponding segments. Then, if X < X', we simply push (i, X) onto the stack. Otherwise, we pop (j, X') from the stack and repeat, since j was not the line below all others at X. Notice that this keeps the stack sorted increasingly by line index and decreasingly by intersection coordinate at all times.

Since every line is processed, pushed onto the stack and popped from the stack at most once, and everything else can be done in constant time, this iteration takes linear time. Other than that, the sorting by y coordinate takes time O(**N** log **N**), which is also the overall time complexity of the entire algorithm, since it dominates all other linear steps.

Notice that the way we use the stack in the above algorithm is quite similar to how a stack is used in the most widely known algorithm to compute the convex hull of a set of points. As we show in the next section, that is no coincidence.

Using point-line duality to shortcut to the solution

In this solution we change how we find leftmost intersections. Treating vertical lines and reflecting to find rightmost intersections, and the way to use leftmost/rightmost intersections to find the solution to the problem, are the same as in the solution above.

To find the leftmost intersections, we can apply the point-line duality to the input. With duality between points and lines, a line y=mx+b in the original space can be represented as the point (m, -b) in the dual space. Similarly, a point (a, b) in the original space can be represented as a line of the form y=ax-b in the dual space. Notice that the dual space of the dual space is the original space. Vertical lines have no corresponding point. This duality has the property that when two lines L_1 and L_2 intersect in the original, their intersection point P corresponds to the line dual(P) in the dual space goes through the points dual(L_1) and dual(L_2).

Thus, if we take all lines that are extensions of input segments and consider the points that correspond to them in the dual space, the leftmost intersection for a given line L_1 occurs when intersecting L_2 such that the slope of the segment between dual(L_1) and dual(L_2) is minimal.

We now work on the dual space with an input set of points, and for each point P we want to find another point Q such that the slope of PQ is minimal. For each point in the convex hull of the set, the minimal slope occurs between that point and the next point in the convex hull. For points not in the convex hull, however, the appropriate choice is the temporary "next" point of the convex hull as calculated by the Graham scan. This leads to similar code as for the solution above, but using the duality saves us quite a few hand-made proofs. Just as for the algorithm above, all of the steps of this algorithm take linear time, with the exception of the sorting step needed for the Graham scan, yielding an overall O(**N** log **N**) algorithm.

A solution using incremental convex hull

Another solution requires more code overall, but some of that code might be present in a contestant's comprehensive library. It uses an incremental convex hull, which is a data structure that maintains a convex hull of a set of points and allows us to efficiently (in logarithmic time) add points to the set while updating the convex hull if necessary.

The algorithm checks for a condition that we mentioned in the analysis for Part 1: each segment has the two endpoints of all other segments on different sides. The algorithm uses a rotating sweep line. Assume the endpoints of all input segments are swapped as necessary such that the segments point right (the first endpoint has an x coordinate no greater than the x coordinate of the second endpoint). Then, we sort the segments by slope and consider a rotating line that stops at all those slopes — that is, we iterate through the slopes in order. If we number the segments S_1, S_2, ..., $S_{\mathbf{N}}$ in that order, S_1 must have all left endpoints on one side, and all right endpoints on the other. S_2 is the same, except the left endpoint of S_1 goes with the right endpoints of all others, and vice versa. In general, for S_i we need to check for the left endpoint of all segments S_1, S_2, ..., S_{i-1} to be on one side together with the right endpoints of all segments S_{i+1}, S_{i+2}, ..., $S_{\mathbf{N}}$, and all other endpoints are on the other side. If we find an endpoint of S_j on the wrong side of S_i, then S_i and S_j do not intersect. If we find no such example, the answer is `MAGNIFICENT`.

If we knew the convex hull of all the points that are supposed to be on each side, we could use ternary search on the signed distance between the convex hull and the line to efficiently find the point from that set whose signed distance perpendicular to the current S_i is smallest (for the side where the distances are supposed to be positive) or largest (for the other side). If one of those finds us a point on the wrong side, we are done; otherwise, we know all other points are also on the correct side. Unfortunately, to keep those two convex hulls as we rotate would require us to both add and remove a point from each set. Removing is a lot harder to do, but we can avoid it.

When considering the slope of S_i, instead of using the convex hull of the full set on one side, we can use the convex hull of the left endpoints that are on that side, and separately, the convex hull of the right endpoints on that side. That leaves us one additional candidate to check for that side, but one of those is the optimal candidate. Since we are calculating left and right endpoints separately, the $4 \times \mathbf{N} - 1$ convex hulls we need are the ones of the set of left endpoints of the segments in a prefix of the list of segments, the left endpoints of the segments in a suffix of the list of segments, and similarly, the right endpoints of the segments in a suffix or prefix of the list of segments. We can calculate all of those convex hulls with a data structure that only provides addition of a point by calculating the convex hulls for prefixes in increasing order of segment index, and the ones for suffixes in decreasing order of segment index. Notice that this means we will calculate the convex hulls in an order different from the order in the original form of the algorithm.

We are doing O(**N**) insertions into the convex hull data structure and O(**N**) ternary searches, and each of these operations takes O(log **N**) time, making the time complexity of this algorithm also O(**N** log **N**).

For this particular use, we only need one half of the convex hull: the half that is closer to the line being inspected. In this half convex hull, the points in the hull are sorted by y coordinate, so a tree search can yield us the tentative insertion point, and we can maintain the convex hull by searching and inserting in a sorted tree. This is simple enough that it does not necessarily require prewritten code. Additionally, we can further simplify by using binary search on the angle between the convex hull and the line instead of the ternary search mentioned above.