Problem Set 6 Solutions

Problem 6-1. Same side or not?

Give a data structure that represents \( n \) points in the plane using only \( O(n) \) space and can perform the following operation in \( O(\log n) \) time:

query(\( l \)) - returns “same side” if all the points are on the same side of line \( l \), otherwise “not same side”

You may use a lot of time (say \( O(n^2) \)) to build the data structure initially, but you must end up with a data structure that uses only \( O(n) \) space and can answer each query in only \( O(\log n) \) time.

(Hint: It might help to look at a convex hull.)

Solution:

We begin by following the hint and observing that the points are all on the same side of a line if and only if their convex hull is all on the same side of the line. So if we store the convex hull as the data structure (an array \( P[1..h] \) of the points on the hull in counter-clockwise order), we need only give an \( O(\log h) \)-time procedure to decide if a line intersects a convex polygon.

To achieve \( O(\log h) \) time, we’ll basically do a binary search. The idea is to check a diagonal of the polygon to see if it crosses the line. If so we are done. If not, we know that the half of the polygon on the far side of the the diagonal from the line cannot intersect the line, so we can eliminate that half of the polygon from consideration and split the remainder in half again.

The tricky part about this algorithm is correctly implementing the step of deciding which half of the polygon can be eliminated.

For convenience, we’ll write \( d_i \) for the signed distance from \( P[i] \) to a given query line \( l \). By signed distance we mean the dot product of a unit vector normal to the line and the vector from a point on the line to \( P[i] \). (Actually, since we’ll only compare distances, the normal vector doesn’t have to be unit.) If the line is specified in any reasonable way then we can easily get a point and a unit vector normal to it in \( O(1) \) time. Furthermore, we can choose our unit vector such that \( d_1 \geq 0 \). Then each computation of \( d_i \) can easily be done in \( O(1) \) time, and we can think of the goal of the algorithm as trying to decide whether there is a point with negative \( d_i \).

The key to our binary search is the following:

Claim: If \( i < k \) and \( d_i \leq d_k \) and \( d_{i+1} \geq d_i \), then all of \( d_{i+1}, d_{i+2}, \ldots, d_k \geq d_i \).

Proof: Since we have a convex polygon, if we start at some point and walk around it counterclockwise, there are very few possibilities for the pattern of \( d_j \) we can see. If we start at
the point with minimum \( d_j \), we'll see the \( d_j \) increase and then decrease back to the start. If we start with the maximum \( d_j \) we see the opposite, and otherwise we see either increase, decrease, increase or decrease, increase, decrease. Applying this to the claim, we see that if we start with \( d_i \) that is minimum, the claim is trivially true. Since we have both \( d_k, d_{i+1} \geq d_i \), and a convex hull never has 3 points in a line, the only way \( d_i \) can be maximum is if \( i + 1 = k \), in which case the claim is again trivially true. Otherwise, since the step from \( i \) to \( i + 1 \) is not decreasing, we must be in the increase, decrease, increase case. As there is only one sequence of decreasing, and we must get back to \( d_i \), any decreasing to a point \( j \) with \( d_j < d_i \) can only happen after we pass \( k \).

We can make symmetric arguments for the cases when \( d_i > d_k \) or \( d_{i+1} < d_i \), and this will establish the correctness of our algorithm. We're now ready for the pseudocode:

```
QUERY(l)
1  i ← 1
2  j ← h
3  while (j - i > 1)
4      k ← [(i + j)/2]
5      if (d_k < 0)
6         return “not same side”
7      if (d_i ≤ d_k)
8         if (d_{i+1} ≥ d_i)
9            i ← k
10        else
11            j ← k
12      else
13         if (d_{k+1} ≥ d_k)
14            j ← k
15        else
16            i ← k
17      if d_j < 0 return “not same side”
18  else return “same side”
```

Analysis:
Space is \( O(n) \) and preprocessing time is \( O(n \log n) \). Each iteration of the while loop takes \( O(1) \) time, and the number of iterations is \( O(\log h) = O(\log n) \), because the difference between \( i \) and \( j \) is halved each time.

**Problem 6-2. Make this one simple**

Give an \( O(n \log n) \) time algorithm that takes \( n \) points in the plane and constructs a simple polygon that has each point as a vertex and no other points as vertices. (A simple polygon is a polygon that does not cross itself. For example, the figure on the left is simple, but the figure on the right is not.)
Solution:

The idea for how to solve this problem is related to Graham’s scan. Recall that Graham’s scan starts by finding the lowest point, \( p_0 \), and sorting all other points by the polar angle in counterclockwise order around \( p_0 \) (CLR, p. 902). Our algorithm is simple: use this order as the polygon. That is, \( p_0 \) is connected to \( p_1 \), the first in the order, each \( p_i \) in the order is connected to \( p_{i+1} \), and \( p_{n-1} \) is connected back to \( p_0 \).

Assuming no 3 points are colinear, as suggested in the clarification email, we see that each edge is in its own polar region, so no two edges can intersect, except at a vertex. That is, the polygon does not cross itself, which means our algorithm is correct.

The running time is clearly \( O(n \log n) \), because all the work is in sorting.

Problem 6-3. Approximate Furthest Pair

Give an \( O(n) \) time algorithm that takes \( n \) points in the plane and finds a pair of points who are at least half as far apart as the furthest pair. That is, suppose that \( a \) and \( b \) are the two points in the input that are furthest apart. Find a pair \( c, d \) such that \( \text{dist}(c, d) \geq \text{dist}(a, b)/2 \).

Solution:

The key to this problem is to realize that no point can be close to both members of the furthest pair. More specifically, consider any point \( c \). By the triangle inequality, \( \text{dist}(c, a) + \text{dist}(c, b) \geq \text{dist}(a, b) \). Therefore the larger of \( \text{dist}(c, a) \) and \( \text{dist}(c, b) \) must be at least \( \text{dist}(a, b)/2 \). Based on this we can give a very simple algorithm:

1. Pick any point \( c \)
2. Find \( d \), the point farthest from \( c \), by computing the distance from \( c \) to all other points
3. output \( (c, d) \)

We’ve already established the correctness, and the running time is clearly \( O(n) \).