Complexity and Algorithms: Homework-3 solutions*

February 28, 2017

Answer 1:
The problem can be solved by line sweeping idea. Sort the points by the x coordinates from high to low. Then, apply the sweeping procedure starting from the rightmost point (i.e., starting from the point with the maximum x coordinate). Here, events are the points. We can use a balanced binary tree to maintain the points according to their y coordinate values. When we encounter the first point then we know that the point is present in the maximal layer 1. For the other points we do the following. When we encounter a point $p_i$ then we check if we have any other points above this point. If there is no other point then we know that this point is in the first maximum layer. On the other hand if we have some points that are above the point $p_i$ then we use “range search” to get the maximum layer value so far (say $L_{max}$). Then, assign $L_{max} + 1$ as the layer number for the point $p_i$.

Pseudocode:

allMaximalLayers(P):

1. Sort the points in the set P by x coordinate and store in array A.

2. Initialize a balanced binary tree T.

*Please report if you find any error in the solutions
for every point $p_i$ in $A$ do
  Insert $p_i$ into $T$ using the y-coordinate as key;
  if $p_i$ is root or right-most leaf of $T$ then
    Maximal layer of $p_i = 1$;
  end
  else
    $m =$ Use “Range search” to get the maximum of the maximal layers for
    the scanned points whose y-coordinate values are larger than $p_i$;
    Maximal layer of $p_i = m + 1$;
  end
end

Complexity analysis:
Sorting the $n$ points takes $O(n \log n)$. While sweeping, at each event, we take $O(\log n)$. hence, total time complexity is $O(n \log n)$.

Correctness: While we sweep from right to left, we guarantee that x value for new point encountered must be smaller than existing points swept. So, only the points above it will dominate this point. Hence, the new point should belong to the next maximal layer.

Answer 2:
The problem is a special case of computing the intersections of a set of line segments, and we can apply the same plane sweeping algorithm to solve it. Although, the total number of line segments is $kn$, because we are dealing with convex polygons, we know that the number of line segments intersected by the sweep line at any event is bounded by $4k$, rather than $kn$, and the number of intersections stored in the heap is bounded by $2k$, rather than $(kn)^2$. The result is an algorithm that runs in $O((kn + m) \log k)$.

Complexity analysis: If the vertices of the polygons are given in the clockwise order starting at the leftmost vertex, the initial sort can be done in $O(kn \log k)$ time using a k-way merge procedure. Each iteration of the loop takes $O(\log k)$ time since at any point the sweep line can intersect a maximum of $4k$ line segments. Hence, the total runtime of the algorithm is $O((kn + m) \log k)$, where $m$ is the total number of intersections and $m = O(k^2 n)$.

Correctness: This algorithm is essentially the same as the the algorithm for computing the intersections of line segments, with few minor modifications.

Pseudocode:
CONVEX-POLYGON-INTERSECTIONS($P_1, P_2, \ldots, P_n$)

1. \( R := \) an empty linked list of intersection points
2. \( Z := \) the upper hulls and reversed lower hulls of \( P_1, \ldots, P_n \)
3. \( Q := \text{K-Way-Merge}(Z) \), where points are sorted from left to right, breaking ties by putting lower
   points first, and are stored along with their adjoining line segments
4. \( T := \) an empty balanced BST of line segments, where searches are always performed by evaluating
   each line at a given \( z \)-coordinate
5. \( H := \) an empty max-heap of intersection points, stored along with their adjoining line segments,
   using \( x \)-coordinates as keys
6. \( i := 0 \)
7. while \( H \) is not empty or \( i < |Q| \)
8. \( p := \text{Remove-Min}(H) \)
9. if \( H \) is not empty and the leftmost point in \( H \) is to the left of \( Q[i] \)
10. Append \( p \) to \( T \)
11. In \( T \), swap the two lines \( L_i \) and \( L_{i+1} \) intersecting at \( p \)
12. If \( L_i \) and its nearest upper neighbor intersect, insert the intersection point into \( H \)
13. If \( L_i \) and its nearest lower neighbor intersect, insert the intersection point into \( H \)
14. else
15. \( p := Q[i] \)
16. \( (x, y) := p \)
17. \( L_1 := \) the nearest upper neighbor to \( y \) in \( T \) at \( x \)
18. \( L_2 := \) the nearest lower neighbor to \( y \) in \( T \) at \( x \)
19. if \( p \) is a leftmost vertex,
20. \( L_{UP} := \) the upper line segment adjacent to \( p \)
21. \( L_{LT} := \) the lower line segment adjacent to \( p \)
22. Insert \( L_{UP} \) and \( L_{LT} \) into \( T \) at \( x \)
23. If \( L_i \) exists and \( L_1 \) and \( L_{UP} \) intersect, insert the intersection point into \( H \)
24. If \( L_i \) exists and \( L_2 \) and \( L_{LT} \) intersect, insert the intersection point into \( H \)
25. else if \( p \) is a rightmost vertex,
26. \( L_3 := \) the upper line segment adjacent to \( p \)
27. \( L_4 := \) the lower line segment adjacent to \( p \)
28. Remove \( L_{UP} \) and \( L_{LT} \) from \( T \)
29. If \( L_i \) and \( L_3 \) both exist and intersect, insert the intersection point into \( H \)
30. else,
31. \( L_5 := \) the left line segment adjacent to \( p \)
32. \( L_6 := \) the right line segment adjacent to \( p \)
33. Replace \( L_3 \) in \( T \) with \( L_5 \)
34. If \( L_i \) exists and \( L_4 \) and \( L_6 \) intersect, insert the intersection point into \( H \)
35. If \( L_i \) exists and \( L_2 \) and \( L_6 \) intersect, insert the intersection point into \( H \)
36. \( i := i + 1 \)
37. return \( H \)
Answer 3:

(A): Assume the ghosts and ghostbusters be points in a plane. A good ghostbuster-ghost pair is one for which the number of ghosts on any side of the line equals the number of busters on the same side of the line. We want to prove that there always exists a good ghostbuster-ghost pair and that it can be found in $O(n \log n)$. We can use the angular sweep to find such a ghostbuster-ghost pair. Let $p_0$ be the point with minimum y-coordinate. From the point $p_0$, angular scan (counter clockwise) to sweep all other points. Maintain counters for counting the number of ghosts and ghostbusters present in the right of the angular sweep line. When value in both the counters becomes same then we have detected the desired line.

Pseudocode:

Let $P$ be the set of all the points (ghosts + ghostbusters). Define two counters G and GB to store the number of ghosts and ghostbusters respectively to the right of the angular sweep line.

detectLine($P$):

\[
p_0 = \text{Point with minimum y-coordinate;}
\]

Sort all other points by the angle that the line between the current point and

\[
p_0, \text{ makes with the x-axis and put in the list } A;
\]

fp = the first point in the list A;

if the fp and $p_0$ is an valid pair then

\[
\text{Store } p_0 \text{ and fp;}
\]

\[
\text{return;}
\]

end

else

\[
\text{for every other point } t \text{ in the sorted list } A \text{ do}
\]

\[
\text{if } G == GB \text{ then}
\]

\[
\text{Store } p_0 \text{ and } t;
\]

\[
\text{return;}
\]

end

end

Complexity analysis: Sorting the points before making the angular sweep takes $O(n \log n)$. The angular sweep takes $O(n)$. Hence, total time complexity is $O(n \log n)$.

Correctness: We can prove inductively that given a set of equal number of Ghostbusters and ghosts, there is always an incident when the value in G and GB are equal. At each iteration,
the algorithm maintains counters for each type of point. At some point the values of the counters will need to pass each other on their way to n (number of ghosts or ghostbusters).

(B): We can use the algorithm in the first part to solve this. We use divide and conquer to solve it recursively.

**Pseudocode:**

Let $P$ be the set of ghosts and ghostbusters.

formPairs($P$):

```plaintext
if $|P| \leq 2$
    Connect the ghost and ghostbuster;
    return;
end

L = detectLine($P$);
A = all the points to the one side of the side;
B = all the points to the other side of the line;
formPairs(A);
formPairs(B);
```

**Complexity analysis:** The expected size of the convex hull be given as $O(n^{1/3})$. The recursive relation for the procedure is given by

$$T(n) = T(n - k - 2) + T(k) + O(n \log n)$$

$T(2) = O(1)$

This gives us $T(n) = O(n^2 \log n)$

**Correctness:**

At each recursion we partition the set of ghosts and ghostbusters such that equal number of them are on the same side. Hence, this guarantees that no two ghost and ghostbusters are paired across any other pair.

**Time complexity:**

At each $2n$ endpoints $O(n)$ we need to sort ($O(n \log n)$) other $2n - 1$ endpoints. Hence, the algorithm takes $O(n^2 \log n)$.

**Answer 4(A):** Compute the intersection of the projection of the two lines segments onto the xy plane. If they do not intersect, then return “unrelated”. If they intersect, we evaluate the z axis value for both the lines at (x,y) and report whichever is higher.

**Pseudocode:**

1. Let InterSect($L_1, L_2$) be a function to compute the intersect of the lines.
\( P_1 = \) projection of line 1 onto xy plane. \( P_2 = \) projection of line 2 onto xy plane

\[
p = \text{InterSect}(P_1, P_2) \text{ if } p \text{ exits then}
\]
\[
(x,y) = p;
\]
\[
z_1 = \text{value of 1 at } (x,y);
\]
\[
z_2 = \text{value of 2 at } (x,y);
\]
\[
\text{if } z_1 < z_2 \text{ then}
\]
\[
\quad \text{return BELOW;}
\]
\[
\text{end}
\]
\[
\text{else}
\]
\[
\quad \text{return ABOVE;}
\]
\[
\text{end}
\]
\[
\text{else}
\]
\[
\quad \text{return Unrelated;}
\]
\[
\text{end}
\]
\[
\text{end}
\]

**Answer 4(B): Approach:**

This can be done by modifying the algorithm for computing the intersection of a set of 2D line segments. The difference is that at each sweep line through the 2D plane, we also maintain the relative ordering of the sticks according to the \( z \)-coordinate. If we ever find that the relative ordering of the sticks changes then we know that the sticks cannot be picked up. As long as we remember the relative ordering of the sticks, we can recover a valid order in which to pick them up at the end of the sweep.

The key idea is to find all of the points where two line segments overlap—i.e., other words, the points where they overlap when projected onto the \( x,y \) plane. We know that a line segment that is above another must be picked up before that line segment. We can treat these dependencies as edges in a graph and run a topological sort on the graph to determine a valid pickup order, if it exits.

**Pseudocode:**

Let \( S \) be the set of \( n \) sticks. Let us assume we have access to a subprocedure TopologicalSort(G) which returns a topological sort of the vertices in \( G \) if \( G \) contains no cycles, and returns Not-Possible otherwise. The procedure pickupSticks(S) returns the line segments in \( S \) in a valid order in which they can be picked up, or else Not-Possible if they cannot be picked up.
Complexity analysis:

The algorithm is the simple plane sweep algorithm to detect all the intersection points. Hence, the time complexity for the algorithm is $O((n+k)\log n)$, where $k$ is the number of intersections in the xy projections of the lines.

Correctness:

Since we use the line intersection detection algorithm, we can easily verify its correctness. Let's focus on the modified part of the solution. The algorithm works by finding all of the 2D points where two line segments overlap, and using these intersection points to determine which sticks must be picked up before other sticks. It is impossible to pick up all sticks if and only if we end up with a cyclic pickup order. Therefore, storing the dependencies as edges in a graph, and then running a topological sort on the resulting graph, produces a valid pickup order, or else determines that the sticks cannot be picked up.