Pre-tutorial questions

Do you know the basic concepts of this week's lecture content? These questions are only to test yourself. They will not be explicitly discussed in the tutorial, and no solutions will be given to them.

1. Sweepline approach
   (a) What is the general idea of a Sweepline approach?
   (b) What is an *event point*?
   (c) What is the *status* of a sweepline algorithm?
   (d) How do one generally prove correctness of a Sweepline algorithm?
2. Intersection detection
   (a) Describe the general idea of the sweepline algorithm for detecting intersections.
   (b) What are the event points?
   (c) What is the status structure?
   (d) What is the invariant?

Tutorial

Problem 1
Consider a set S of m line segments (intervals) and a set P of n points on the real line. Design an $O(n + m)\log(n + m))$ time algorithm that reports every point in P that lies on a segment of S.

> **Solution:** Sort the endpoints of the intervals and the points from left to right, these will be the *event points*. Set a counter c to zero. The counter will keep track of the current depth and will be the *status* of the sweepline. Sweep all the event points from left to right. If left endpoint then increment c, and if right endpoint then decrement c. If a point p in P then we have two cases. If $c > 0$ then report p, otherwise ignore the point.
> The running time is dominated by the sorting step which requires $O(n \log n)$ time. The rest of the algorithm runs in $O(n)$ time.

Problem 2
Let R be a set of n red segments in the plane and let B be a set of m blue segments in the plane. Design an algorithm that counts the number of intersections between the segments in R and the segments in B. Prove the running time, space requirement and correctness of your algorithm.

Solution: Run the sweepline intersection reporting algorithm that we went through in class and find all intersections. Recall the algorithm. Sort all the endpoints of the segments from left to right, these will be the initial set of *event points*. The *status* of the sweepline is the segments that intersect it ordered from top to bottom, together with a counter that keeps track of the number of intersections found. These segments are stored in a binary search tree T, thus all relevant operations (insert/delete/search) on T can be done in $O(\log n)$. Sweep a vertical line from left to right, stopping at event points. At an event point three cases can occur.

1. The event point is a left end point of a segment s. In this case we insert s into T and check possible intersections between s and it's two adjacent segments in T. Any intersections are added to the set of event points.

2. The event point is the right end point of a segment s. Delete s from T and check if the two incident segments to s in T intersect. If they do, add the intersection to the set of event points.

3. If the event point is an intersection point between two segments s_1 and s_2 then swap the order of s_1 and s_2 in T. Check if the segments incident to s_1 and s_2 intersect. If they do, add the intersections to the set of event points. Finally, if s_1 and s_2 have different colour then increase the counter by 1.

Since the total number of intersections may be quadratic, the running time is $O(n^2 \log n)$ even if the number of red-blue intersections is small.
This can be improved to $O(n \log n + k)$, but it's not easy...

Problem 3
Given a set R of n pairwise disjoint rectilinear squares (sides are vertical or horizontal) and a set P of m points. Design an $O(n \log n)$ time algorithm that reports all points in S that lie inside a square in R. What if we consider rectangles instead? What if we allow the rectangles to intersect? Does the problem become much harder?

Solution: Sweep from left to right. The set of *event points* are the x-coordinates of the left and right sides of the squares and the set of points. Sorting these requires $O((n+m)\log(n+m))$.
Maintain a balanced binary tree T storing the squares intersecting the sweepline, ordered from top to bottom. This is the status of the sweep line. Note that this is just a set of disjoint intervals in 1D (we only need to store the bottom endpoint of each square, so it's just a binary search tree on a set of points along the vertical line).
During the sweep three events may take place:

1. Left side of a square r: Insert the y-interval of r into the search tree (just the bottom y-coordinate of r).
2. Right side of a square r: Delete the y-interval of r from the search tree.
3. Point p: Check if the point lies in an interval of T. This corresponds to a simple binary search in T for the highest point q in T below p. If p lies inside the rectangle corresponding to q then report p.

Total time: After the sort the sweep requires $O(\log n)$ time per event point. Thus, total time is $O((n+m)\log(n+m))$.

- If the input is a set of disjoint rectangles the exact same approach can be used.
- If the input rectangles can overlap the problem becomes harder. Instead of storing the intervals in a binary search tree one would have to maintain the intervals in a more complex data structure (e.g. interval trees which is outside the scope of this course).

Problem 4
Let S be a set of m disjoint line segments and let P be a set of n points in the plane (no point lie on a segment). Given any query point q in the plane determine all points in P that p can see, that is, every point p in P such that the open segment pq does not intersect any line segment of S. Give an $O((m+n)\log(m+n))$ time algorithm.

Solution: Sort all the endpoints and the points radially around q. These will be the event points. Consider a ray r originating from q. Sweep r counter-clockwise starting from a horizontal position (left-to-right direction). Initialise a binary search tree T (or a heap) such that it contains all the segments of S intersecting r, ordered with respect to the distance between q and the intersection point. The structure T is the status of the sweep "ray".
During the sweep there are three events.

1. start endpoint: insert the segment into T.
2. end endpoint: delete the segment from T.
3. point p: if p lies closer to q then the first segment in T then report p, otherwise ignore it.

For the running time note that there are $2n$ event points, and each event point can be processed in $O(\log n)$ time.

Problem 5
Consider the following algorithm to compute the convex hull of a set S of n points in the plane.
Step 1: Sort the points in S by increasing x-coordinate.
Step 2: Recursively compute the convex hull of the left half of the point set. The resulting convex hull is denoted H_1.

Step 3: Recursively compute the convex hull of the right half of the point set. The resulting convex hull is denoted H_2.

Step 4: From H_1 and H_2 compute the convex hull H of the entire point set.

1. Assume that step 4 can be implemented in time $O(n)$. What is the running time of the algorithm? Prove your time bound.

2. Consider the edge e connecting the highest point in H_1 with the highest point in H_2. Will the edge e be an edge in H? Prove your answer.

3. Consider the points clockwise along H_1 between the highest point of H_1 to the lowest point of H_1. Can any of these points be in the convex hull, H, of the entire set? Prove your answer.

4. Give a correct implementation of step 4, that runs in $O(n)$ time. Prove the correctness and the running time of your algorithm.

Solution:

1. The running time can be described by the recursive formula $T(n) = 2T(n/2) + O(n)$ which solves to $O(n \log n)$. The preprocessing (sorting) requires $O(n \log n)$ time so the total running time is $O(n \log n)$.

2. No, see Fig. 1a.

3. Yes, see Fig. 1b.

4. Merge the two hulls into a common convex hull, H, by computing the upper and lower tangents for H_A and H_B and discarding all the points lying between these two tangents. One thing that simplifies the process of computing the tangents is that the two point sets A and B are separated from each other by a vertical line (assuming no duplicate x coordinates). Let's concentrate on the lower tangent, since the upper tangent is symmetric. The algorithm operates by a simple "walking" procedure. We initialize a to be the rightmost point of H_1 and b is the leftmost point of H_2. (These can be found in linear time.) Lower tangency is a condition that can be tested locally by an orientation test of the two vertices and neighbouring vertices on the hull. Iterate the following two loops, which march a and b down, until they reach the points lower tangency.

   Finding the Lower Tangent

   (1) Let a be the rightmost point of H_1.

   (2) Let b be the leftmost point of H_2.

   (3) While ab is not a lower tangent for H_1 and H_2 do

      (a) While ab is not a lower tangent to H_1 do $a = a - 1$ (move a clockwise along H_1).

      (b) While ab is not a lower tangent to H_2 do $b = b + 1$ (move b counterclockwise along H_2).

   (4) Return ab.

   For the correctness just argue that the resulting segments are tangents (no need to be complete). For the time complexity the important observation is that each vertex on each hull can be visited at most once by the search, and hence its running time is $O(m)$, where $m = |H_1| + |H_2| \leq |A| + |B|$.

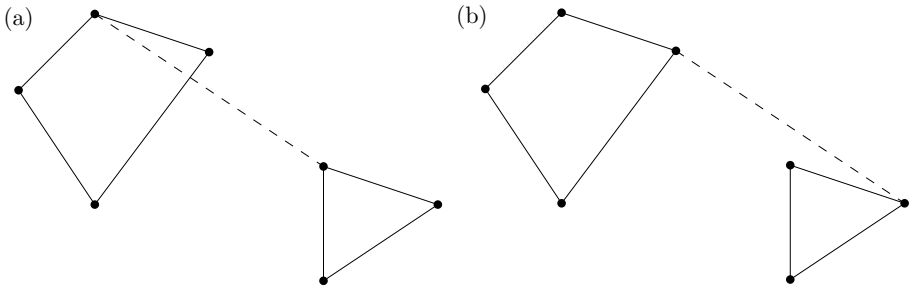

Figure 1: (a) The edge connecting the highest point of H_1 with the highest point of H_2 might not be an edge of the convex hull. (b) The upper convex hull edge between H_1 and H_2 might have one endpoint among the points clockwise along H_1 between the highest point of H_1 to the lowest point of H_1.

Problem 6
Consider the following algorithm for the MST problem:

Algorithm 1 IMPROVING-MST

```
1: function IMPROVING-MST(G, w)
2:     T ← some spanning tree of G
3:     for e ∈ E [in any order] do
4:         T ← T + e
5:         C ← unique cycle in T
6:         f ← heaviest edge in C
7:         T ← T − f
8:     end for
9:     return T
10: end function
```

Prove its correctness and analyze its time complexity. To simplify things, you can assume the weights are different.

Solution: Let $T_0, T_1, T_2, \ldots, T_m$ be the trees kept by the algorithm in each of its m iterations. Consider some edge $(u, v) \in E$ that did not make it to the final tree. It could be that (u, v) was rejected right away by the algorithm, or it was in T for some time and then it was removed. Suppose this rejection took place on the ith iteration. Let p be the u-v in path in T_i. Clearly all edges in p have weight less than $w(u, v)$. It is easy to show using induction that this is true not only in T_i but in all subsequent trees T_j for $j \geq i$.
Let (x, y) be an edge in the final tree T_m. Remove (x, y) from T_m to get two connected components X and Y. In the textbook (Property 4.17) it is shown that if (x, y) is the lightest edge in the cut (X, Y) then every MST contains (x, y). Suppose for the sake of contradiction that there is some edge $(u, v) \in \text{cut}(X, Y)$ such that $w(u, v) < w(x, y)$. This means that (u, v) is not the heaviest edge in the unique u-v path in T_m, which contradicts the conclusion of the previous paragraph. Thus, (x, y) belong to every MST. Since this holds for every edge in T_m, it follows that the T_m itself is an MST.
Regarding the time complexity, we note that finding the cycle in $T + e$ can be done $O(n)$ time using DFS. Similarly finding the heaviest edge and removing it take $O(|C|) = O(n)$ time. There are m iteration, so the overall time complexity is $O(nm)$.

Problem 7
Consider the following algorithm for the MST problem:

Algorithm 2 REVERSE-MST

```
1: function REVERSE-MST(G, w)
2:     sort edges in decreasing weight w
3:     T ← E
4:     for e ∈ E [in this order] do
5:         if T − e is connected then
6:             T ← T − e
7:         end if
8:     end for
9:     return T
10: end function
```

Prove its correctness and analyze its time complexity. To simplify things, you can assume the weights are different.

Solution: Suppose e was one of the edges the algorithm kept. If e belongs to the optimal solution we are done, so assume for the sake of contradiction that this is not the case.
If the algorithm decided to kept e then it must be that $T - e$ was not connected, say $T - e$ had two connected components X and Y. Notice that all edges in the cut (X, Y) have a weight larger than $w(e)$. Therefore, we can take the optimal solution, add e to form a cycle C. This cycle must contain one edge f from $\text{cut}(X, Y)$, so we could remove f and improve the cost of the optimal solution, a contradiction. Therefore, e must be part of the optimal solution. Since this holds for all e in the output the algorithm is optimal.
Regarding the time complexity, we note that checking connectivity can be done $O(n + m)$ time using DFS. There are m iteration, so the overall time complexity is $O(m(n + m))$.

Problem 8
[Advanced] You are given two x-monotone polygonal chains P and Q. Prove that the number of times P and Q can intersect is $O(n)$, where n is the total number of vertices of P and Q.

Solution: First note that two straight lines can intersect at most once. For each of the $2n$ vertices on P and Q draw a vertical line. Since P and Q are x-monotone a vertical line will intersect P and Q at most once. The vertical lines form $2n + 2$ slabs. No vertices of P and Q can lie inside a slab, hence, P and Q can intersect at most once in each slab. This gives the $O(n)$ bound on the number of intersections.