DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

This week we will discuss ways of using augmented search trees to solve 2-D geometry problems. The main idea is to treat the x-axis as time, and the y-axis as the objects maintained in the data structure.

To start with, consider the following problem that may or may not have been inspired by lemmings

Problem 8.1. Given \( n \) horizontal line segments that prevents one from falling, find in \( O(n \log n) \) time the fastest way to get from the topmost to the bottom-most segment by only walking left, right. Assume each distance travelled takes 1 unit of time, and falling is instantaneous.

A slightly ‘souped up’ version of this problem is available at [https://dmoj.ca/problem/tle16c8p6](https://dmoj.ca/problem/tle16c8p6). However, turning that problem into a geometric version requires quite a bit of work, so it will be on the problem set as well.

First, note that once we know where each endpoint of each segment ‘falls’ to, the problem becomes an \( O(n) \) time dynamic program: each segment keeps two states: the fastest time to the bottommost segment if one starts at its left endpoint, or its right endpoint. This is because once one falls onto a segment, the only valid next steps are to walk to either the left, or the right, of the segment.

So it remains to find where the tallest segment below each point is. For this, consider the segments in increasing y order, and maintain the highest height at each x value. The two operations we need to perform are:

1. Set an interval to some value.
2. Query for the value at a point.

This is exactly the range update, point query data structure we discussed last time, so the total runtime is \( O(n \log n) \).

A more general and natural problem is.

Problem 8.2. Given \( n \) axis-parallel rectangles in 2-D (possibly overlapping), compute the area and perimeter of their union.

The perimeter version (with smaller bounds) is available at [https://dmoj.ca/problem/ioi98p4](https://dmoj.ca/problem/ioi98p4).
What’s particularly surprising about this problem is that we cannot explicitly compute the rectilinear polygon corresponding to the union: it may have up to \( \Omega(n^2) \) holes. Consider the case where the rectangles are either very long horizontal slabs, or very long vertical slabs.

We will address this using the idea of plane sweep: because things are axis-parallel, there are only \( O(n) \) \( x \) values where the shape can change. Specifically, all the \( x \) value of all corners/edges of the rectangles. We proceed through these left-to-right, and maintain the structure of the rectangles as we go.

Formally we establish a ‘sweep line’, that is, a vertical line that moves from left to right as we proceed through the algorithm. For simplicity we assume the \( x \) values are integers from 1 to \( n \), and no two corners have shared \( x \) and \( y \) values. Then the key observation is that the profile of rectangles intersecting this line can only change by one. That is, we either add coverage to some range, or decrease coverage to some range. This is exactly where the search tree type data structures we’ve developed so far are useful.

Consider first the area problem. The operations we need to support are:

1. Add \( x \) to a range \([l, r]\),
2. Compute the number of coordinates whose values is more than 1: such points correspond to coordinates covered by at least one rectangle.

This combination of operations is actually tricky to perform in \( O(\log n) \) time when values are allowed to go negative. However, note that because we always encounter the left side of a rectangle before the right side, the coverage values we maintain are never negative. So instead we can consider the following slight extension of range add / range minimum:

1. Add \( x \) to a range \([l, r]\),
2. Compute the minimum, and how many values have that minimum.

That is, the number of uncovered points is just the number of 0s, when 0 is the minimum: otherwise the entire range is covered and we don’t have this issue at all.

The perimeter case is slightly different, and can be solved in two ways:

1. We measure how much the coverage (total non-zero) changed when we add/remove an edge. That difference is how much new perimeter was added. This measures the amount increased horizontally. This has the advantage of using almost the same data structure.

2. We measure the number of zero-nonzero transitions in the current coverage. That is, each node also tracks whether its two extreme endpoints are covered in the current labeling. This extra change is taken into account when merging two nodes.

For both approaches, it’s necessary for us to run things twice: once for the perimeter parallel to each axis.

We now go back to address the simplifications we made. There are two of them:
1. Coordinates may not be unique. This can be addressed by associating multiple updates with each \( x \) value. Note that in the perimeter case, it’s essential we perform the increments before decrements, or the overlaps will get counted twice.

2. Coordinate values may be much larger than \( n \). This can be addressed by assigning values to the coordinates, instead of just maintaining lengths of intervals. Then the area/perimeter can be multiplied by the \( x \) difference between one coordinate and the next.

Robust algorithms for rectangle union are quite powerful. They allow one to compute, for example, the max empty axis-parallel square that don’t intersect with a bunch of rectangular objects (and is in some bounding box).

Now let’s look at a problem that combines a lot of these ideas together.

**Problem 8.3.** Given \( n \) red points, \( n \) blue points, and \( n \) non-intersecting/touching axis-parallel rectangle as obstacles on the 2-D plane. Find how many blue points can one reach from each red point by only walking downward and rightward, without crossing any of the obstacles.

Note that the number of reachable pairs can be as large as \( n^2 \).

We first need to think about when can a red point reach a blue point. Here there are two ‘extreme’ paths to consider: one where one moves right whenever possible, and one where one moves downward whenever possible. The key claim is:

**Lemma 8.4.** \( x \) can reach \( y \) if and only if \( y \) is between the extreme rightward and the extreme downward paths of \( x \).

**Proof.** Proving this critically relies on the non intersection of rectangles. That is, from any point, we can always ‘walk out’ by going leftward and upward.

So for some point \( y \), if we always walk leftward or upward, we will intersect with one of the two curves from \( x \) eventually. At that intersection point, we can reach \( x \) by taking the remainder of that path we intersected with.

So the problem then becomes how to compute the total number of points below such a path. For this, think like an integral: we simply need to sum the number of points below a bunch of horizontal line segments. Furthermore, note that although the total number of segments in these paths may be large, the actual number of line segments is quite small: any path reaches, within 2 steps, some corner of one fo the rectangular obstacles. So we need to find, via several plane sweeps:

1. the first obstacle encountered if we walk leftward/downward from each of the points, as well as all corners of rectangles.

2. along the sides of all rectnagles, intersection points’ line segments to the correp-sonding corners.
This takes $O(n \log n)$ time using a plane sweep (that’s akin to the lemming problem). For the downward drop, we sweep in increasing order of $y$, and ask for the highest obstacle edge present that covers the $x$ coordinate.

Then for each line segment, we need to compute the number of points below it that aren’t separated by some rectangular obstacle. This has two steps:

1. Separate the points/segments by their regions, which can be described by their bounding boxes. We can either:
   
   (a) Do this from scratch using another a plane sweep. If the edge below $x$ is the upper side of some other box, then we need to ‘jump’ to the lower side of that box and look for the outer region encompassing that.
   
   (b) Or just use the last point reached on the extreme paths, which must either be a bottom/right edge of some rectangle, or to infinity.

2. figure out the number of points below each segment using another plane sweep. This is done by sorting the points and segments by $y$ value, and inserting points into a tree that tracks sums along intervals of $x$. 