1 Preliminaries

The *sweep-line* paradigm is a very powerful algorithmic design technique. It's particularly useful for solving geometric problems, but it has other applications as well. We'll illustrate this by presenting algorithms for two problems involving intersecting collections of line segments in 2-D.

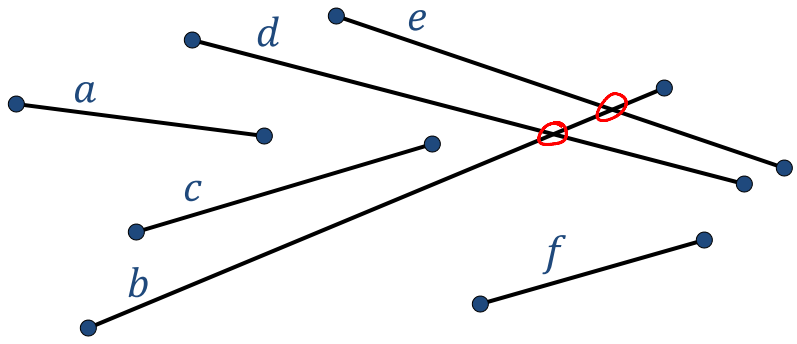
Generally speaking sweepline means that you are processing the data in some order (e.g. left to right order). A data structure is maintained that keeps the information gleened from the part of the data currently to the left of the sweepline. The sweepline moves across absorbing new pieces of the input and incorporating them into its data structure.

Obviously this is very vague. So let's get concrete and solve some problems.

2 Computing all Intersections of a Set of Segments

The input is a set S of line segments in the plane (each defined by a pair of points). The output is a list of all the places where these line segments intersect.

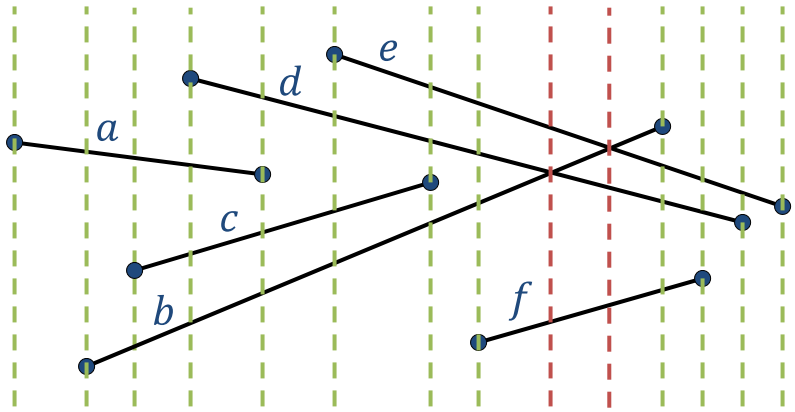
- Input: n line segments in 2D
- Goal: Find the k intersections



As usual, we're going to make our lives easier by making some geometric assumptions. We're going to assume that none of the segments is vertical. We'll also assume that no three segments intersect at the same point. We'll also assume that no segment has an endpoint that is part of another segment. (These assumptions can be avoided by adding some extra cases to the algorithm that do not change the running time bounds.)

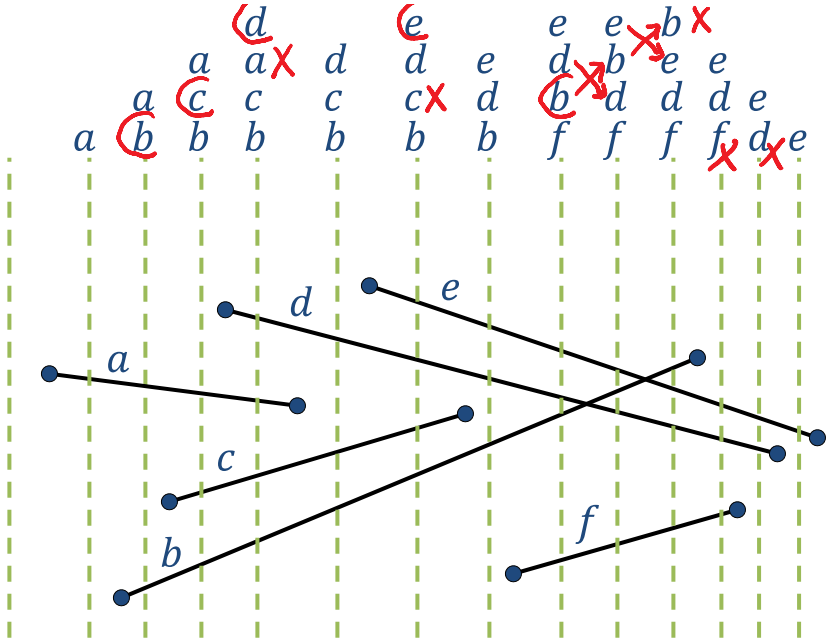
There's a trivial $O(n^2)$ algorithm: Just apply segment intersection to all pairs of segments. The solution we give here will be $O((n+k)\log n)$, where k is the number of segment intersections found by the algorithm.

To get some intuition for what's going to happen, consider the following figure.



Here we have six segments, called a, b, c, d, e, f. There are also two segment intersections. The "interesting" x-coordinates are the ones where something happens: either a segment begins, or a segment ends, or two segments cross.

The following figure shows a vertical dashed line before each interesting x-coordinate. Above the dashed line is a list of the segments in the same order they appear in the diagram, along that dashed line.



Between any two neighboring events, the segments (in that range of x) are in some order by y from bottom to top. This order (and which segments are there) does not change between a pair of neighboring interesting x coordinates.

We're going to maintain a data structure that represents the list shown at the top of the diagram. Let's call this a "segment list". Look at how this list changes as we move across the interesting x values. Only one of three things can happen. (1) A new segment is inserted into the list. (2) A segment is removed from the list. (3) Two neighboring segments in the list cross.

Our goal is to compute the intersections among these segments. The key observation that leads to a good algorithm is that right before two segments cross, they must be neighbors in the segment list. So the key idea of the algorithm is that as our segment list evolves (as we process the interesting x coordinates from left to right), we only need to consider the possible intersections between segments that are neighbors, at some point in time, in the segment list.

So our algorithm is going to maintain two data structures. A segment list (SL) and an event queue (EQ). Each event in the EQ will be labeled with its type ("segment start", "segment end", "segments cross"), as well as the x value where this happens. The EQ is initialized with all the segment starts and segment ends.

The EQ data structure must support `insert`, `findmin`, and `deletemin`. It's just a standard priority queue.

Although a segment is defined by its two endpoints, it's going to be useful to also use a "slope-intercept" representation for the line containing the segment. So segment i will have a left and right endpoint as well as a pair (m_i, b_i) where m_i is the slope of the line and b_i is the y-intercept.

Consider what we need for the SL data structure. The items being stored are a set of segments. The ordering used is the value of $m_i \cdot x + b_i$, where x is the current x value in the ongoing sweep-line algorithm. Here's what it needs to be able to do:

Requirements for the SL data structure:
- Insert a new segment into the data structure.
- Delete a segment from the data structure.
- Find the successor of a segment in the data structure.
- Find the predecessor of a segment in the data structure.

All of these operations can be done in $O(\log n)$ time using any standard search tree data structure (e.g. splay trees, or AVL trees).

Now we can write the complete algorithm. Note that whenever it says "take a pair of segments into consideration" what this means is that this pair of segments are now neighbors in the SL. It's possible that they intersect. So this function tests if they do intersect to the right of the current x coordinate. If they do, this event is added to the EQ.

```
FindSegmentIntersections(S):
    Create an empty EQ.
    For each segment in S, insert its start and end events into EQ.
    Create an empty SL.
    While the EQ is not empty do:
        let E = deletemin(EQ)
        if E is segment start of a segment s then
            Insert s into SL.
            Take the pair (s, the successor of s) into consideration.
            Take the pair (s, the predecessor of s) into consideration.
        else if E is a segment end of a segment s then
            Delete s from SL.
            Take the two former neighbors of s into consideration.
        else if E is a segments cross event involving segments s1 and s2 then
            Output the discovery of the segment intersection between s1 and s2.
            remove s1 and s2 from SL, and reinsert them in the opposite order.
            Take the two new neighbors of s1 and s2 into consderation.
    done.
```

It's easy to see that the running time of the algorithm is $O((n+k)\log n)$ using the data structures described. Because there are $O(n+k)$ events, and each one involves a constant number of operations on the SL and EQ data structures.

3 Counting Intersections of Horizontal and Vertical Segments

Note: this section will be updated for clarity eventually.

Now we switch gears and consider the case when the segments are only vertical or horizontal. Here however, we want to count the total number of intersections (not enumerate them) and we want our algorithm to run in $O(n \log n)$ time. Let V be the set of vertical segments and let H be the set of horizontal segments. We assume that no two vertical segments intersect and no two horizontal segments intersect.

First we form sorted lists of the "interesting" x and y values. A y is interesting of the segment endpoints has that y value. x is analogous. Label these values $y_1 < y_2 < \cdots < y_m$, and $x_1 < x_2 < \cdots < x_n$.

We sweep from left to right with a vertical sweep-line. The events are the interesting x coordinates.

There are three types of events:

We come to a vertical segment.
We come to the beginning of a horizontal segment
We come to the end of a horizontal segment.

At any point in time the vertical sweep-line crosses some number of horizontal segments. We define a variable a_i for each interesting y value y_i. $a_i = 1$ if the current sweep line crosses a horizontal segment with y coordinate equal to y_i. Otherwise it is zero. We maintain these a_i values using a segment tree (which you saw in recitation).

CountSegmentIntersetions(V, H):

Construct the interesting x and y lists.
Create a segment tree ST using the interesting y values.
For each x_i in increasing order do:
If x_i is the right end of a horizontal segment s then
let the y value of s be y_j. then $a_j \leftarrow 0$
else if x_i is the left end of a horizontal segment s then
let the y value of s be y_j. then $a_j \leftarrow 1$
else if x_i is a vertical segment s then
let y_l be the bottom s and y_h be the top of s.
compute `sumRange`(l, r) and add it to a running total.
done

Note that if several events happen at the same x coordinate, then left ends should be processed first, then the vertical segments, then the right ends. This order means that we will not miss an intersection where the left end of the horizontal segment lies on a vertical segment.

The running time is $O(n \log n)$ where n is the total number of segments. This is because all the work can be assigned to the segment trees, which take $O(\log n)$ per opration.