

Spengler: *There's something very important I forgot to tell you.*

Venkman: *What?*

Spengler: *Don't cross the streams.*

Venkman: *Why?*

Spengler: *It would be bad.*

Venkman: *I'm fuzzy on the whole good/bad thing. What do you mean, "bad"?*

Spengler: *Try to imagine all life as you know it stopping instantaneously and every molecule in your body exploding at the speed of light.*

Stantz: *Total protonic reversal.*

Venkman: *Right. That's bad. Okay. All right. Important safety tip. Thanks, Egon.*

— *Ghostbusters* (1984)

O Line Segment Intersection

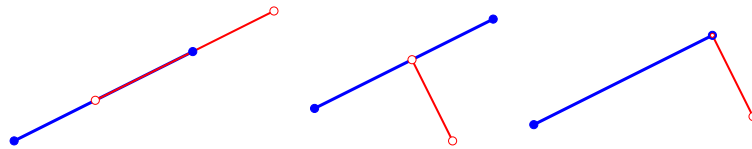
O.1 Introduction

In this lecture, I'll talk about detecting line segment intersections. A line segment is the convex hull of two points, called the *endpoints* (or *vertices*) of the segment. We are given a set of n line segments, each specified by the x - and y -coordinates of its endpoints, for a total of $4n$ real numbers, and we want to know whether any two segments intersect.

To keep things simple, just as in the previous lecture, I'll assume the segments are in *general position*.

- No three endpoints lie on a common line.
- No two endpoints have the same x -coordinate. In particular, no segment is vertical, no segment is just a point, and no two segments share an endpoint.

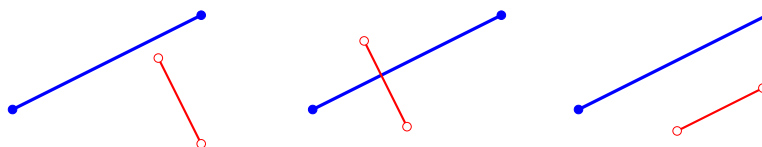
This general position assumption lets us avoid several annoying degenerate cases. Of course, in any real implementation of the algorithm I'm about to describe, you'd have to handle these cases. Real-world data is *full* of degeneracies!



Degenerate cases of intersecting segments that we'll pretend never happen: Overlapping collinear segments, endpoints inside segments, and shared endpoints.

O.2 Two segments

The first case we have to consider is $n = 2$. (The problem is obviously trivial when $n \leq 1$!) How do we tell whether two line segments intersect? One possibility, suggested by a student in class, is to construct the convex hull of the segments. Two segments intersect if and only if the convex hull is a quadrilateral whose vertices alternate between the two segments. In the figure below, the first pair of segments has a triangular convex hull. The last pair's convex hull is a quadrilateral, but its vertices don't alternate.



Some pairs of segments.

Fortunately, we don't need (or want!) to use a full-fledged convex hull algorithm just to test two segments; there's a much simpler test.

Two segments \overline{ab} and \overline{cd} intersect if and only if

- the endpoints a and b are on opposite sides of the line \overleftrightarrow{cd} , and
- the endpoints c and d are on opposite sides of the line \overleftrightarrow{ab} .

To test whether two points are on opposite sides of a line through two other points, we use the same counterclockwise test that we used for building convex hulls. Specifically, a and b are on opposite sides of line \overleftrightarrow{cd} if and only if exactly one of the two triples a, c, d and b, c, d is in counterclockwise order. So we have the following simple algorithm.

```

INTERSECT( $a, b, c, d$ ):
  if  $CCW(a, c, d) = CCW(b, c, d)$ 
    return FALSE
  else if  $CCW(a, b, c) = CCW(a, b, d)$ 
    return FALSE
  else
    return TRUE
```

Or even simpler:

```

INTERSECT( $a, b, c, d$ ):
  return  $[CCW(a, c, d) \neq CCW(b, c, d)] \wedge [CCW(a, b, c) \neq CCW(a, b, d)]$ 
```

O.3 A Sweep Line Algorithm

To detect whether there's an intersection in a set of more than just two segments, we use something called a *sweep line* algorithm. First let's give each segment a unique *label*. I'll use letters, but in a real implementation, you'd probably use pointers/references to records storing the endpoint coordinates.

Imagine sweeping a vertical line across the segments from left to right. At each position of the sweep line, look at the sequence of (labels of) segments that the line hits, sorted from top to bottom. The only times this sorted sequence can change is when the sweep line passes an endpoint or when the sweep line passes an intersection point. In the second case, the order changes because two adjacent labels swap places.¹ Our algorithm will simulate this sweep, looking for potential swaps between adjacent segments.

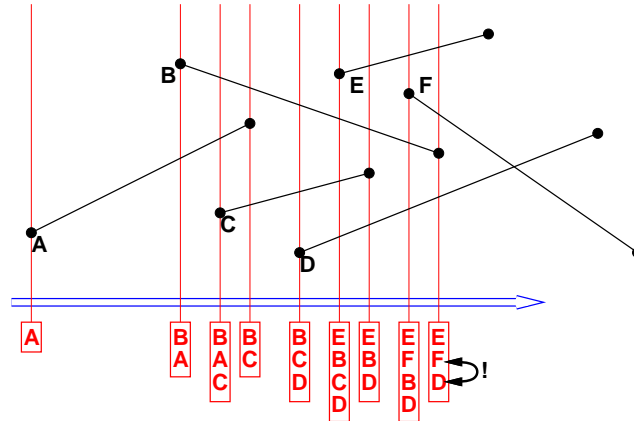
The sweep line algorithm begins by sorting the $2n$ segment endpoints from left to right by comparing their x -coordinates, in $O(n \log n)$ time. The algorithm then moves the sweep line from left to right, stopping at each endpoint.

We store the vertical label sequence in some sort of balanced binary tree that supports the following operations in $O(\log n)$ time. Note that the tree does not store any explicit search keys, only segment labels.

- **Insert** a segment label.
- **Delete** a segment label.
- Find the **neighbors** of a segment label in the sorted sequence.

$O(\log n)$ amortized time is good enough, so we could use a scapegoat tree or a splay tree. If we're willing to settle for an expected time bound, we could use a treap or a skip list instead.

¹Actually, if more than two segments intersect at the same point, there could be a larger reversal, but this won't have any effect on our algorithm.



The sweep line algorithm in action. The boxes show the label sequence stored in the binary tree. The intersection between F and D is detected in the last step.

Whenever the sweep line hits a left endpoint, we insert the corresponding label into the tree in $O(\log n)$ time. In order to do this, we have to answer questions of the form ‘Does the new label X go above or below the old label Y?’ To answer this question, we test whether the new left endpoint of X is above segment Y, or equivalently, if the triple of endpoints $\text{left}(Y), \text{right}(Y), \text{left}(X)$ is in counterclockwise order.

Once the new label is inserted, we test whether the new segment intersects either of its two neighbors in the label sequence. For example, in the figure above, when the sweep line hits the left endpoint of F, we test whether F intersects either B or E. These tests require $O(1)$ time.

Whenever the sweep line hits a right endpoint, we delete the corresponding label from the tree in $O(\log n)$ time, and then check whether its two neighbors intersect in $O(1)$ time. For example, in the figure, when the sweep line hits the right endpoint of C, we test whether B and D intersect.

If at any time we discover a pair of segments that intersects, we stop the algorithm and report the intersection. For example, in the figure, when the sweep line reaches the right endpoint of B, we discover that F and D intersect, and we halt. Note that we may not discover the intersection until long after the two segments are inserted, and the intersection we discover may not be the one that the sweep line would hit first. It’s not hard to show by induction (hint, hint) that the algorithm is correct. Specifically, if the algorithm reaches the n th right endpoint without detecting an intersection, none of the segments intersect.

For each segment endpoint, we spend $O(\log n)$ time updating the binary tree, plus $O(1)$ time performing pairwise intersection tests—at most two at each left endpoint and at most one at each right endpoint. Thus, the entire sweep requires $O(n \log n)$ time in the worst case. Since we also spent $O(n \log n)$ time sorting the endpoints, the overall running time is $O(n \log n)$.

Here’s a slightly more formal description of the algorithm. The input $S[1..n]$ is an array of line segments. The sorting phase in the first line produces two auxiliary arrays:

- $\text{label}[i]$ is the label of the i th leftmost endpoint. I’ll use indices into the input array S as the labels, so the i th vertex is an endpoint of $S[\text{label}[i]]$.
- $\text{isleft}[i]$ is TRUE if the i th leftmost endpoint is a left endpoint and FALSE if it’s a right endpoint.

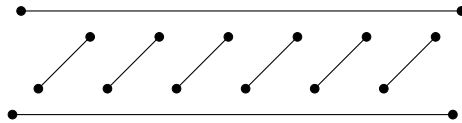
The functions INSERT, DELETE, PREDECESSOR, and SUCCESSOR modify or search through the sorted label sequence. Finally, INTERSECT tests whether two line segments intersect.

```

ANYINTERSECTIONS( $S[1..n]$ ):
  sort the endpoints of  $S$  from left to right
  create an empty label sequence
  for  $i \leftarrow 1$  to  $2n$ 
     $\ell \leftarrow label[i]$ 
    if  $isleft[i]$ 
      INSERT( $\ell$ )
      if INTERSECT( $S[\ell], S[SUCCESSOR(\ell)]$ )
        return TRUE
      if INTERSECT( $S[\ell], S[PREDECESSOR(\ell)]$ )
        return TRUE
    else
      if INTERSECT( $S[SUCCESSOR(\ell)], S[PREDECESSOR(\ell)]$ )
        return TRUE
      DELETE( $label[i]$ )
  return FALSE

```

Note that the algorithm doesn't try to avoid redundant pairwise tests. In the figure below, the top and bottom segments would be checked $n - 1$ times, once at the top left endpoint, and once at the right endpoint of every short segment. But since we've already spent $O(n \log n)$ time just sorting the inputs, $O(n)$ redundant segment intersection tests make no difference in the overall running time.



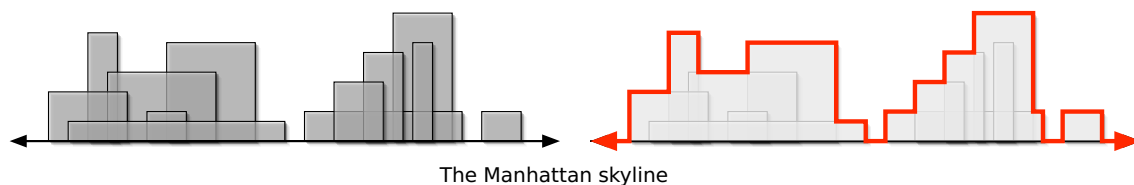
The same pair of segments might be tested $n - 1$ times.

Exercises

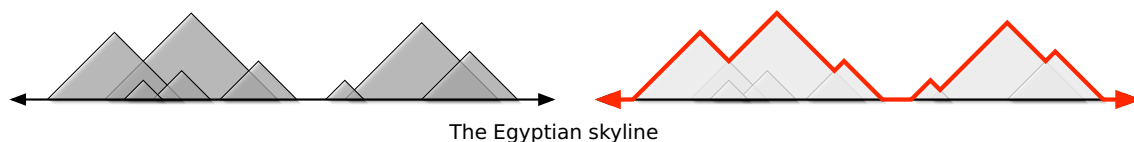
- Let X be a set of n rectangles in the plane, each specified by a left and right x -coordinate and a top and bottom y -coordinate. Thus, the input consists of four arrays $L[1..n]$, $R[1..n]$, $T[1..n]$, and $B[1..n]$, where $L[i] < R[i]$ and $T[i] > B[i]$ for all i .
 - Describe and analyze an algorithm to determine whether any two rectangles in X intersect, in $O(n \log n)$ time.
 - Describe and analyze an algorithm to find a point that lies inside the largest number of rectangles in X , in $O(n \log n)$ time.
 - Describe and analyze an algorithm to compute the area of the union of the rectangles in X , in $O(n \log n)$ time.
- Describe and analyze a sweepline algorithm to determine, given n circles in the plane, whether any two intersect, in $O(n \log n)$ time. Each circle is specified by its center and its radius, so the input consists of three arrays $X[1..n]$, $Y[1..n]$, and $R[1..n]$. Be careful to correctly implement the low-level primitives.
- Describe an algorithm to determine, given n line segments in the plane, a list of all intersecting pairs of segments. Your algorithm should run in $O(n \log n + k \log n)$ time, where n is the number of segments, and k is the number of intersecting pairs.

4. This problem asks you to compute *skylines* of various cities. In each city, all the buildings have a signature geometric shape. Describe an algorithm to compute a description of the union of n such shapes in $O(n \log n)$ time.

- (a) Manhattan: Each building is a rectangle whose bottom edge lies on the x -axis, specified by the left and right x -coordinates and the top y -coordinate.



- (b) Giza: Each building is a right isosceles triangle whose base lies on the x -axis, specified by the (x, y) -coordinates of its apex.



- (c) Nome: Each building is a semi-circular disk whose center lies on the x -axis, specified by its center x -coordinate and radius.

5. [Adapted from CLRS, problem 33-3.] A group of n Ghostbusters are teaming up to fight n ghosts. Each Ghostbuster is armed with a proton pack that shoots a stream along a straight line until it encounters (and neutralizes) a ghost. The Ghostbusters decide that they will fire their proton packs simultaneously, with each Ghostbuster aiming at a different ghost. Crossing the streams would be bad—total protonic reversal, yadda yadda—so it is vital that each Ghostbuster choose his target carefully. Assume that each Ghostbuster and each ghost is a single point in the plane, and that no three of these $2n$ points are collinear.

- (a) Prove that there is a set of n disjoint line segments, each joining one Ghostbuster to one ghost. [Hint: Consider the matching of minimum total length.]
- (b) Prove that the non-collinearity assumption is necessary in part (a).
- (c) A *partitioning line* is a line in the plane such that the number of ghosts on one side of the line is equal to the number of Ghostbusters on the same side of that line. Describe an algorithm to compute a partitioning line in $O(n \log n)$ time.
- (d) Prove that there is a partitioning line with exactly $\lfloor n/2 \rfloor$ ghosts and exactly $\lfloor n/2 \rfloor$ Ghostbusters on either side. This is a special case of the so-called *ham sandwich theorem*. Describe an algorithm to compute such a line as quickly as possible.
- * (e) Describe a randomized algorithm to find an *approximate* ham-sandwich line—that is, a partitioning line with at least $n/4$ ghosts on each side—in $O(n \log n)$ time.
- (f) Describe an algorithm to compute a set of n disjoint line segments, each joining one Ghostbuster to one ghost, as quickly as possible.