LINE SEGMENT INTERSECTION  Ch. 2

input: set $S = \{s_1, s_2, \ldots, s_n\}$ of $n$ segments in plane
output: set $I$ of intersection pts among segments in $S$,
(with segments containing each intersection point)

\[ (\binom{n}{2}) = \Theta(n^2) \text{ (in worst-case)} \]

example: How many intersections possible?

Algorithm: Brute Force Intersect ($S$)

\[
\text{for all pairs } (s_i, s_j) \in S \times S: \text{ } i \neq j
\text{ test if } s_i \text{ intersects } s_j
\]

complexity $\Theta(n^2) \leq \text{worst-case optimal (have to report all intersections)}$

However... there may be many fewer intersections...
Can we spend less time (depending on $|I|$)?

We'd like an algorithm whose running time depends on
$n$ - # input segments
$K$ - # intersections
$K$ - size of the output
we call such algorithms output-sensitive

What would be best time?
$O(n + K)$?
Actually a lower bound of $\Omega(n \log n + k)$ is known.
Goal: An output-sensitive algorithm that avoids testing pairs of segments for intersection if they are "far apart"

Idea y-interval test for $s_i + s_j$
Project segments onto y-axis
0 if $s_i + s_j$ intersect
then so do their y-intervals $(s_i, s_j)$
1 if the y-intervals of $s_i + s_j$ don't intersect
then neither do $s_i + s_j$ $(s_i, s_j)$
*3 if the y-intervals of $s_i + s_j$ do intersect
then maybe $s_i + s_j$ intersect $(s_i, s_j)$

so we may still test extra, but better than before
1. let $l$ be horizontal line above all segments
2. sweep $l$ down over segments
   - as sweep keep track of all segments intersecting it

Plane Sweep Algorithm
• $l$ is sweep line
• status $T$ of sweep line is set of segments intersecting it
• points where status changes are events

in our case... * in general for plane sweep algorithms

At each event point (segment endpoint for now)
• update status of sweep line (add/remove segments from $T$)
• perform some intersection tests
two types of events
(i) lower endpoint of segment $S_i$
   * remove $S_i$ from $T$
(ii) upper endpoint of segment $S_i$
   * add $S_i$ to $T$
   * test $S_i$ for intersection w/ all segments in $T$

\[\begin{align*}
\text{e.g.:} \quad & S_1 & S_3 \\
& S_2 & S_1
\end{align*}\]

- before time $t$: $T = \{S_1, S_3\}$
- at time $t$:
  * event $\Rightarrow$ upper endpoint of $S_3$
    - add $S_3$ to $T \Rightarrow T = \{S_1, S_3, S_3\}$
    - check $S_3$ for intersection w/ $S_1 + S_2$
- at time $t+1$:
  * event $\Rightarrow$ lower endpoint of $S_1$
    - remove $S_1$ from $T \Rightarrow T = \{S_2, S_3\}$

Running Time of Sweepline Algorithm
- first sort segment endpoints by y-coordinate - $O(n \log n)$
  (duplicates - ok or degenerate?)
- 2n event points must be processed.
  the time to process an event point depends on
  * implementation of $T$ (we'll use list...)
  * number of elements in $T$ (when adding new segments)
    (because we check each one for intersection w/ new segs)

note: since # elements in $T$ depends only on # segs intersecting
horizontal sweep line we may still test segs that
are "far apart"

$\Rightarrow$ so alg may take $\Theta(n^2)$ time
even if $k = o(n^2)$

$: This algorithm is not output-sensitive enough
for us $\Rightarrow$ but probably still better than Brute Force.
Ok, so now what?
we want to avoid testing segments that are "far apart"
horizontally... How??

Define segments \( s_i + s_j \) are horizontally adjacent (ha)
on the sweep-line \( l \) if their intersection points on \( l \) are adjacent.

**Lemma** Let \( s_i + s_j \) be two non-horizontal, non-overlapping
segments intersecting at point \( p \). (Assume also no
3rd segment intersects \( p \)).

Then there is an event point above \( p \) where
\( s_i + s_j \) will become adjacent in a horizontal ordering
of the segments intersecting \( l \).

**Proof:**
- Let \( l_p \) be horizontal line thru \( p \)
- Let \( x \) be lowest event point above \( l_p \)

- When sweep line is between \( p + x \), then \( s_i + s_j \)
  are horizontally adjacent on \( l \)
- When algorithm starts \( s_i + s_j \) are not horizontally
  adjacent on \( l \) (since \( l \) starts above \( s_i + s_j \))
- Since the sweep line status changes only at event points
  there must some event point at which \( s_i + s_j \) will
  1st become horizontally adjacent on \( l \).
To use the lemma we'll:

- order segments according to horizontal adjacency on \( L \)
- test two segs for intersection only when first became horiz. adj.

**Status T:** segments intersecting \( L \) AND horizontal adjacency order changes at:

- endpoints of segments (as before)
- at segment intersections (new events)

**Three types of events**

Let \( p \) = event point on \( L \), \( s_L, s_R \) be leftmost + rightmost neighbors of \( p \) in \( T \) (one)

(i) \( p \) is lower endpoint of segment \( s_i \)

- remove \( s_i \) from \( T \)
- check \( s_L + s_R \) for intersection w/ each other if they intersect in pt \( p' \) below \( p \), add \( p' \) to upcoming events

(ii) \( p \) is upper endpoint of \( s_i \)

- add \( s_i \) to \( T \)
- check \( s_L + s_R \) for intersection w/ \( s_i \) if they intersect in ptc(s) \( p' \) below \( p \) add \( p' \) to upcoming events

(iii) \( p \) is intersection point of \( s_i + s_j \)

- swap order of \( s_i + s_j \) in \( T \)
  - (wolog assume \( s_j \) now to left of \( s_i \))
  - check \( s_j + s_L \) and \( s_i + s_R \) for intersection if they intersect in ptc(s) \( p' \) below \( p \) add \( p' \) to upcoming events
General Algorithm

Input: set $S$ of $n$ line segments in the plane
Output: set of intersection points (and segments containing them)
1. $Q := \emptyset$ (event queue)
2. insert all segment endpoints into $Q$
   - Store segment w/ upper endpoint
3. $T := \emptyset$ (status structure)
4. while $Q \neq \emptyset$
   (a) find "next" event point $p$ in $Q$
   (b) process event point $p$
      - remove $p$ from $Q$
      - update $T$
      - report intersections
      - add new (upcoming) event pts to $Q$ (intersection pts)
endwhile

Data Structures

1. $Q$ - Event Queue
   necessary operations:
   - extract next event point $\Rightarrow$ point w/ largest $y$-coord
     + smallest $x$-coord among these
   - insert new event point AND check for duplicates (segments become adj. mult. times)
$\Rightarrow$ balanced binary search tree
   operations cost $O(\log m)$ where $m = \#$elements in $Q$

2. $T$ - Status Structure (maintains sweepline status - order segm. intersect. it)
   necessary operations:
   - insert segments
   - delete segments
   - find left & right neighbors of segs
$\Rightarrow$ balanced binary search tree
for simplicity
- leaves = segments stored here
- internal nodes = store segment that is rightmost leaf in leftmost subtree

E.g.:

```
  s_i
  \_/  \___
  s_j   s_k
   \_   
     s_p
```

- To find left/right neighbors "walk" down tree comparing to segs stored at internal nodes.
  E.g. to find left neighbor of \( p \) → 
    " + right " " →

- Insert/search time \( O(\log n') \), \( n' = \# \text{segs in } T \), \( n' \leq n \)

Correctness of Algorithm

Lemma 2.2 (in text) Alg. finds + reports all intersections
Proof by induction on priority of event pts
  (priority by y-coord + then x-coord)

Running Time

\( n = \# \text{segments} \), \( k = \text{size of output} \)
- initialize \( Q \) - insert \( 2n \) endpoints \( O(n \log n) \)
- while loop - #
  - each iteration - extract event pt from \( Q \) - \( O(\log n) \)
    - process event pt intersect test
    - insert/delete from \( T/\mathcal{Q} \) \( O(\log n) \)
  - # iterations = #event pts = \( 2n + \# \text{intersections} = O(n + k) \)
- Total \( \Rightarrow O((n + k) \log n) \)
Degeneracies

- 3 or more segments intersect in same pt
  - * we can handle these, but need to be careful...
    - insert event pt only once
    - swapping orders in \( T \) when process

- two or more segments share endpoint
  - can deal with this too

- overlapping segment
  - can handle this too, but a bit harder

for all \( I \) need to alter/modify event handling a bit (see text)

Running time analysis

Now we want to show \( O((n+I)\log n) \)

\( I \) intersections (not size of output in terms of segments)

why different?

\( K \) = output size may be larger than \( I \)

- 1 intersection
  - but \( \binom{\log n}{2} \) pairwise intersections...

Not hard, see text.