Planar point location: given a planar subdivision (partition of the plane into disjoint regions by straight line segments), preprocess to quickly locate a query point.

Example: recall the post office problem

- Compute Voronoi region.
- Query becomes: which region contains \( p \)?

- Planar Point Location.
Point location in 1D

use binary search in sorted array, or balanced binary search tree (can handle dynamic case where points are added/deleted)

\[ P = \text{preprocessing time} \]
\[ S = \text{space} \]
\[ Q = \text{query time} \]
\[ (U = \text{update time}) \]

in 1D: \[ P = O(n \log n) \]
\[ S = O(n) \]
\[ Q = O(\log n) \]

Achieve same bounds for planar point location.

1. slab method (not optimal)
2. persistence
3. Kirkpatrick’s triangulation refinement
4. trapezoidal map (expected good behaviour)
1. Slab method: A basic solution to planar point location

- Divide into vertical slabs at vertices
- Each slab is a 1D problem
2. Persistence [Tarjan and Sarnak, 1986]

Observe that the binary search trees for successive slabs do not change much.

We know how to update binary search trees at $O(\log n)$ per insertion/deletion.

New issue: query make take place not in “current” tree but in any previous tree. $T_i$
Persistent data structure

Allow insertions and deletions over time (as in a usual dynamic data structure) BUT allow queries in old versions. The query specifies the time.

Persistent search trees

idea 1: make new tree share as much as possible with old tree

idea 2: give each node one extra pointer to save making new copy
3. Kirkpatrick’s triangulation refinement, 1983

First triangulate the planar subdivision in \(O(n \log n)\) time. Also add a bounding triangle.

Idea: make rougher and rougher versions by deleting vertices, until we have only the bounding triangle. Then search for query point starting backwards.

Same idea as in Delaunay triangulation except that we build the sequence of triangulations from refined (many triangles) to coarse (few triangles).

Details
4. Trapezoidal decomposition (good expected case behaviour)

Recall we saw trapezoidization of a polygon. Same idea for planar subdivision.

Randomized incremental algorithm to build trapezoidal decomposition AND point location data structure
RECALL from Lecture 2
Idea: Incremental algorithm — add edges one by one in random order, maintaining the trapezoidal decomposition.

To add segment $ab$:
- locate $a$ and $b$
  (find the trapezoids they are in)
- cut those 2 trapezoids
RECALL from Lecture 2

- then walk from $a$ to $b$
- "thread the segment"
- cut each trapezoid in two
- then join pairs (vertically)

Timing
- time to locate $a$ and $b$
- time to thread segment $ab$

$= O(\text{# of horizontals cut by } ab)$
RECALL from Lecture 2

Claim: For segments inserted in random order, the expected number of horizontals cut by ab is 4.

Proof: Count the number of incidences of horizontal line with a segment.

\[ \phi \rightarrow \phi \]  
each segment shoots 2 horizontals resulting in \( \leq 2 \) such incidences

\[ \phi \rightarrow \phi \wedge \phi \]  
each

\[ \therefore \text{# incidences is } \leq 4n \]

Average over n segments is 4.
How to do planar point location with trapezoidal decomposition. This will also complete the missing step from Lecture 2 — how to locate endpoints of new segment to be inserted.

Goal: expected cost $O(\log n)$ to locate one point.
Result: $O(n \log n)$ to build the trapezoidal decomposition.

Note: if we find the trapezoid containing a point, we can find the region containing the point. So this solves planar point location.

For each trapezoid, we record its left side segment. For each segment we record the name of the region to the right.
Build a tree-like search structure as we build the trapezoidal decomposition.

Example: search for point $x$.

- At points we decide above/below.
- At segments we decide left/right.
- Trapezoids are leaves.
How to update this search structure, Q, when we insert a new segment.
Summary on planar point location

\[ P = O(n \log n) \]
\[ S = O(n) \]
\[ Q = O(\log n) \]

- persistence
- Kirkpatrick’s triangulation refinement
- trapezoidal map (expected case behaviour)

There are other methods.

Also, the constant inside the \( O(\log n) \) query time can be made 1.

Dynamic Planar Point Location

now the planar subdivision may change over time (insert/delete edges/vertices)

relevant measures:

\[ P = \text{preprocessing time} \]
\[ S = \text{space} \]
\[ Q = \text{query time} \]
\[ U = \text{update time} \]

In 1D we can get \( P = O(n \log n) \), \( S = O(n) \), \( Q = U = O(\log n) \).

In 2D, there are no known solutions that good.
[See recent paper by Chan and Nekrich for current best.]

We will look at a simple solution to a special case — updates on segments, not vertices — with \( Q = U = O(\log^2 n) \).

Data structure: **Segment Tree** [Bentley, 1980], Chapter 10 in de Berg. et al.
Segment Tree, Chapter 10 in [CG]

stores (overlapping) intervals \( s_1, \ldots, s_n \) on the real line
let endpoints of intervals be \( x_1, x_2, \ldots, x_t \) in order \( t \leq 2n \)

\[ \text{intervals} = \text{segments} \]

\[
\begin{array}{c}
S_1 \quad x_4 \\
\hline
x_1 \\
\hline
x_2 \\
S_3 \\
\hline
x_3 \\
x_5 \\
\end{array}
\]

leaves are intervals: \( (x_i, x_{i+1}) \) — also \( (-\infty, x_1) \) and \( (x_t, \infty) \)

Build a balanced binary tree. At node \( v \) store segment \( s \) if \( s \) contains all subintervals descended from \( v \), but this is not true for \( v \)’s parent.
For different applications, store different *secondary structures* at tree nodes.

Claim: Each segment $s_i$ is stored in $\leq \log n$ tree nodes.

Thus a segment tree uses $O(n \log n)$ space.

Proof of claim.
Applications of segment trees:

1. stabbing queries in 1D

   Store segments on real line
   Query: Given point p, which segments contain it?

   Segment tree gives $Q = O(\log n + k)$, $k = \text{output size}$

   (Note: there are better methods with $S$ in $O(n)$.)
Applications of segment trees:

2. Window queries (see Ch. 10 of [CG])

Store non-crossing segments in the plane
Query: Given rectangle R, which segments intersect R?

Useful for trimming a map (geographic information systems)

If segment $s$ intersects $R$ either

1. endpoint of $s$ is in $R$
2. $s$ crosses a side of $R$ (or both)

Find these separately. Type 1 is range searching — next week.

How to find segments crossing right side of $R$ (other sides similar)
Store disjoint segments in the plane
Query: Given a vertical segment r, find segments intersecting r

Make a segment tree on the x-projections of segments.
At node v store secondary structure, $T_v = \text{the sorted list of segments crossing slab}(v)$

Note: $T_v$ does not store ALL segments crossing the slab.

How to answer a query:
Applications of segment trees

3. Dynamic planar point location

For the simple case where vertices are fixed and segments may be added/deleted. Also assume (for now) that each region has a bounded number of edges.

Store $x$-projections of segments in a segment tree.
Secondary structure at node $v$:

\[ T_v = \text{balanced binary search tree of segments sorted by } y \]

With each segment, store the name of the region above.

To answer query for point $p$
How to do updates:
Result: $P, S$ in $O(n \log n)$  $Q$ in $O(\log^2 n)$  $U$ in $O(\log^2 n)$

To handle unbounded number of edges per region we need a data structure to handle:

1. merge two regions (as edge sets)
2. divide a region in two
3. given an edge, return the associated region

(1) + (3) = union-find

(1) + (2) + (3) = special case of union-split-find

More work to handle updates to vertex set.

There are many more sophisticated approaches. Can get $S$ in $O(n)$ with $Q, U$ as above.

For recent results see Chan and Nekrich ’15,

$S$ in $O(n)$  $Q$ in $O(\log n \ (\log \log n)^2)$  $U$ in $O(\log n \ (\log \log n))$