AACPP - Session 8

Assignment 3 Solution, Assignment 4 Intro

Michalis Georgoulakis, Fabian Wenz

Assignment 4 - DMC

Assignment 4 Rewind

Input

5 6
5 3
2
4
8
3
10
1 5
1 2
2 4
4 5
2 3
3 4

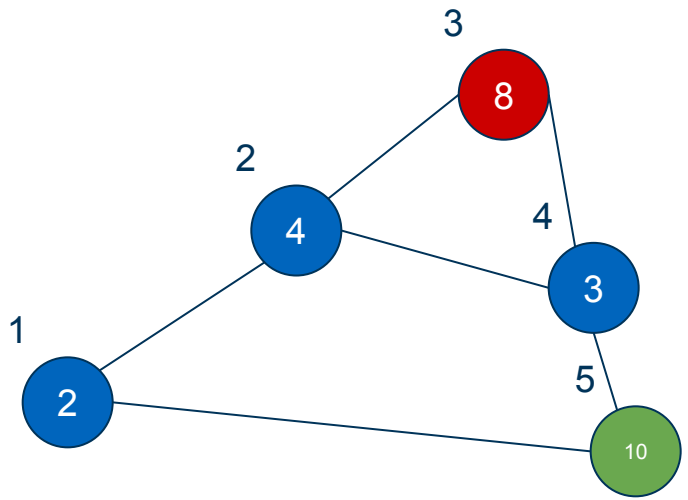
Output

5
1 4

Assignment 4 Rewind

Input

5 6
5 3
2
4
8
3
10
1 5
1 2
2 4
4 5
2 3
3 4



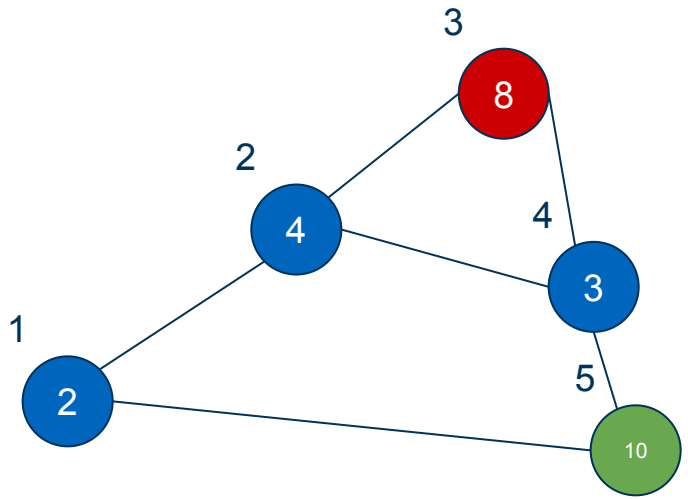
Output

5
1 4

Assignment 4 Rewind

Input

5 6
5 3
2
4
8
3
10
1 5
1 2
2 4
4 5
2 3
3 4



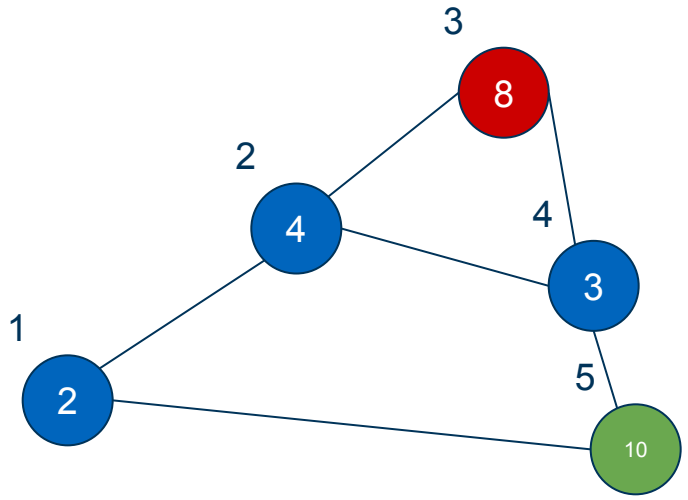
Output

5
1 4

Assignment 4 Rewind

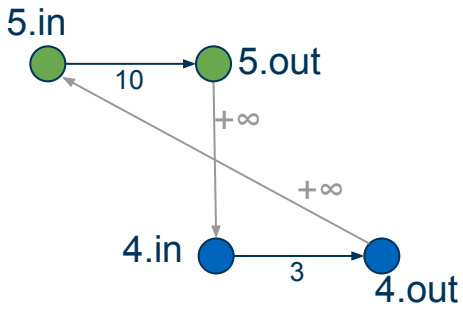
Input

5 6
5 3
2
4
8
3
10
1 5
1 2
2 4
4 5
2 3
3 4



Output

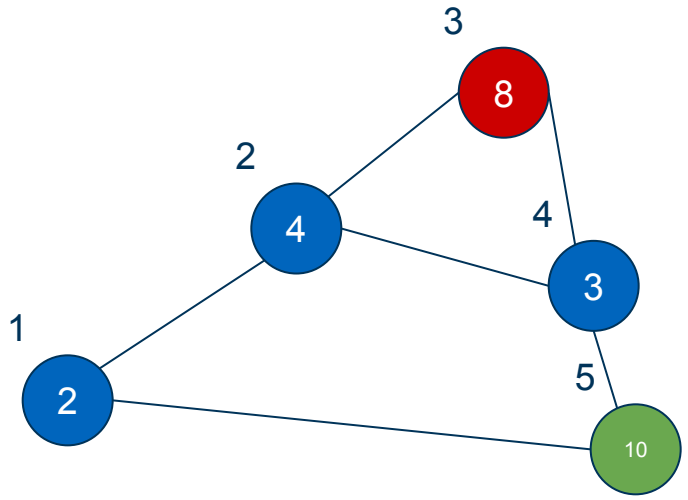
5
1 4



Assignment 4 Rewind

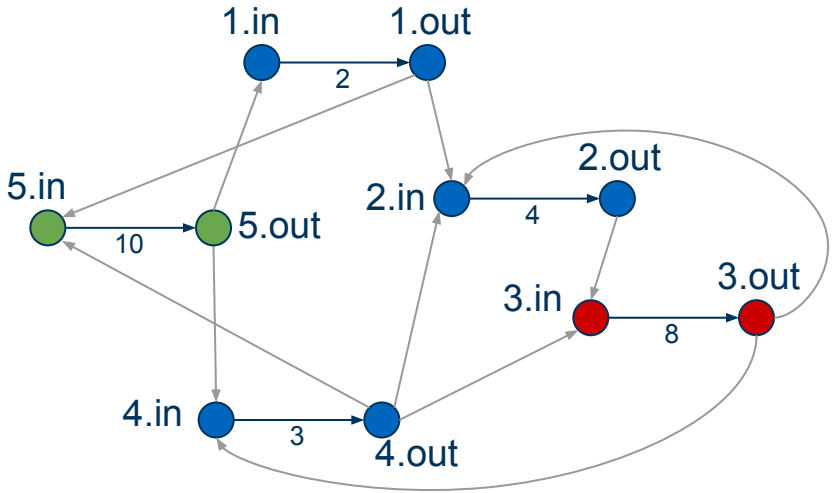
Input

5 6
5 3
2
4
8
3
10
1 5
1 2
2 4
4 5
2 3
3 4



Output

5
1 4



Now find Max-Flow

Graphs – Ford-Fulkerson

The idea is that any path from s to t in the residual network is an *augmenting path*.

Given such a path we can push flow equal to minimum of capacities on the path and increase the overall flow.

The bound on any implementation of this method on $\mathbb{Z}$ capacity values is $\mathcal{O}(m \cdot |f|)$ – finding a path is $\mathcal{O}(m)$ and each path increases the flow.

Graphs – Edmonds-Karp

Always choose the shortest path from s to t with a BFS.

$\mathcal{O}(nm^2)$.

Graphs – Dinitz

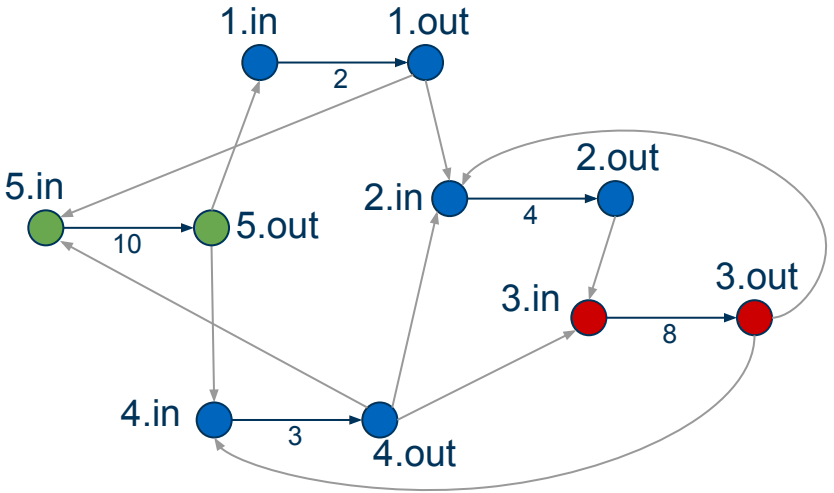
Also known as "Dinic" but Dinitz is correct.

Runs in $\mathcal{O}(n^2m)$ and even faster in special cases.

Idea – find a *blocking flow*, i.e. a *maximal* set of augmenting paths in the residual network.

More precisely:

- construct a DAG such that going forward always brings us closer to the sink;
- saturate the DAG with a series of DFS pushes.

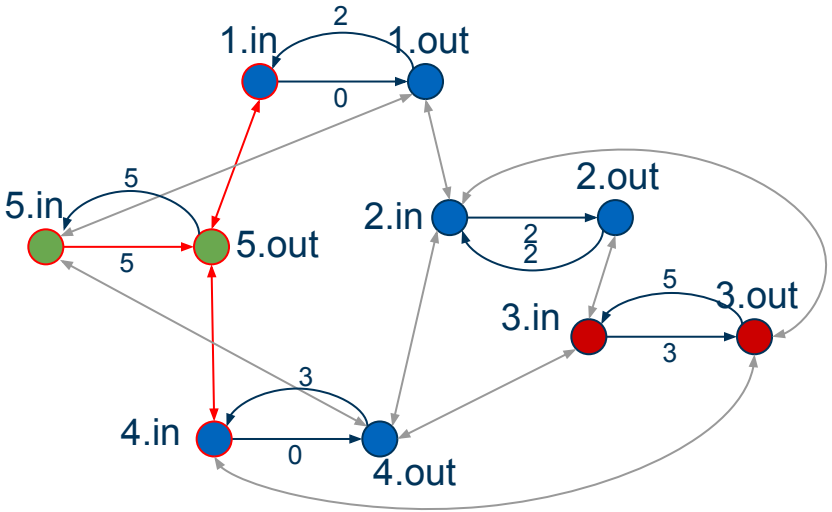


Find the Min-Cut from Max-Flow

1. **Run Dinic to completion** → you have a **max flow** and the final **residual graph**.

2. Do **one BFS/DFS from `s` in the residual graph**, following only edges with **residual capacity > 0**.
   - Let `S` = all vertices reachable from `s`.
   - Let `T` = `V \ S`.
3. Then `(S, T)` is a **minimum cut**.
   - The **cut edges** are all original edges `u -> v` with `u ∈ S` and `v ∈ T` whose **original capacity > 0** (equivalently: they will be **fully saturated** at max flow).

That's it. The extra work is **O(E)** (one graph traversal + one pass over edges).



Geometry

Geometry Primitives

Geometry problems reduce to **basic operations on points** in Euclidean space which maintain the foundations of analytical geometry.

Most algorithms are built from:

- vector arithmetic
- products (dot / cross)
- orientation tests

Points & Vectors

Points and Vectors in 2D

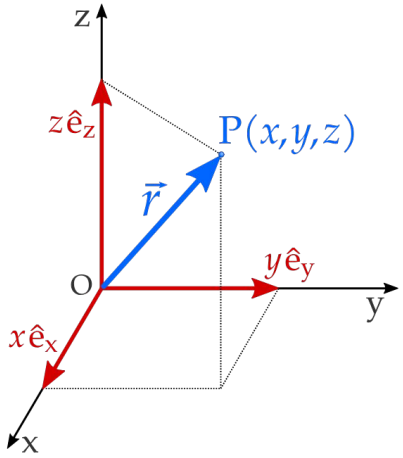
- A point is a pair of coordinates: $P=(x,y,z)$
- A vector is the difference of two points: $\vec{AB} = B - A = (x_B - x_A, y_B - y_A)$

Vector Operations:

- Addition / subtraction $\vec{u}+\vec{v}$, $\vec{u}-\vec{v}$
- Scalar multiplication $k\cdot\vec{u}$

In CP:

- Points and vectors are usually implemented as the same structure
- Operations are just integer arithmetic



Distance Between Points

Euclidean Distance

- For points $A(x_1,y_1)$, $B(x_2,y_2)$

$$d(A,B) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- True geometric distance
- Needed only when the actual length is required

Squared Distance

$$d^2(A,B) = (x_1 - x_2)^2 + (y_1 - y_2)^2$$

- Avoids square roots
- Preserves comparisons and ordering: $d(A,B) < d(C,D) \Leftrightarrow d^2(A,B) < d^2(C,D)$
- Faster and numerically safer than calculating sqrt.

Dot Product

Definition (Computation)

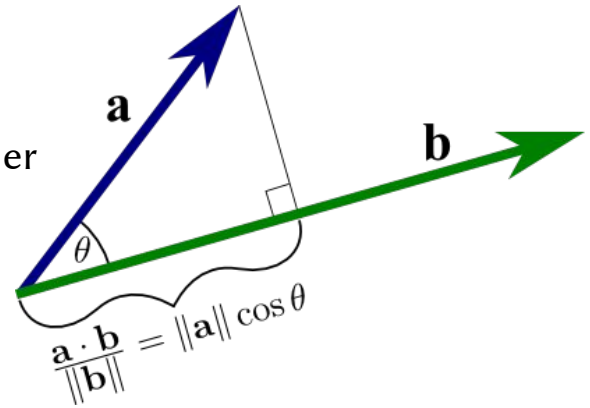
- For vectors $\vec{u}=(x_1,y_1),\vec{v}=(x_2,y_2)$:

$$\vec{u} \cdot \vec{v} = x_1x_2 + y_1y_2$$

Geometric Meaning

- Measures how much one vector points in the direction of another

$$\vec{u} \cdot \vec{v} = \|\vec{u}\|\|\vec{v}\| \cos\theta$$



What It's Used For (CP)

- Angle classification:
  - acute / right / obtuse
- Projection of a point onto a line
- Distance from point to line

Cross Product & Orientations

While the dot product tells us about alignment, the cross product tells us about turning and orientation.

Cross Product (2D)

For vectors $\vec{u}=(x_1,y_1),\vec{v}=(x_2,y_2)$

$$\vec{u}\times\vec{v}=x_1y_2-y_1x_2$$

- Result is a scalar in 2D
- Sign and magnitude both carry meaning

Geometric Meaning

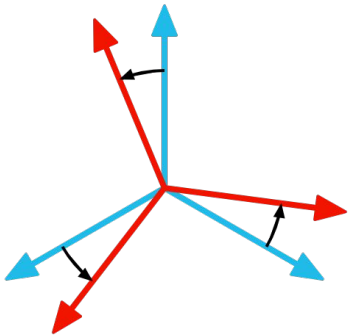
Signed Area: $|\vec{u}\times\vec{v}|=2\times$Area of the triangle formed by $\vec{u},\vec{v}$

- Larger magnitude $\longrightarrow$ larger area
- Zero $\longrightarrow$ vectors are collinear

Direction (Orientation)

The sign of the cross product tells us the turn direction:

- $\vec{u} \times \vec{v} > 0 \longrightarrow$ left turn
- $\vec{u} \times \vec{v} < 0 \longrightarrow$ right turn
- $\vec{u} \times \vec{v} = 0 \longrightarrow$ collinear



Orientation of Three Points

- Given three points A, B, C: Compute $(B - A) \times (C - A)$

Interpretation:

- $0 \longrightarrow A \longrightarrow B \longrightarrow C$ is a left turn
- $< 0 \longrightarrow$ right turn
- $= 0 \longrightarrow$ points are collinear

This is the orientation test.

From Triangles to Polygons

A polygon is a sequence of points $p_0, p_1, \ldots, p_{n-1}$ connected by line segments

- Edges: $(p_i, p_{i+1}), (p_{n-1}, p_0)$

Types

- Simple: no self-intersections
- Convex: all internal angles < 180°
- Non-convex: at least one reflex angle

Key idea:

- Polygons are built from edges and orientations

Oriented Area of a Triangle

Signed Area of Three Points

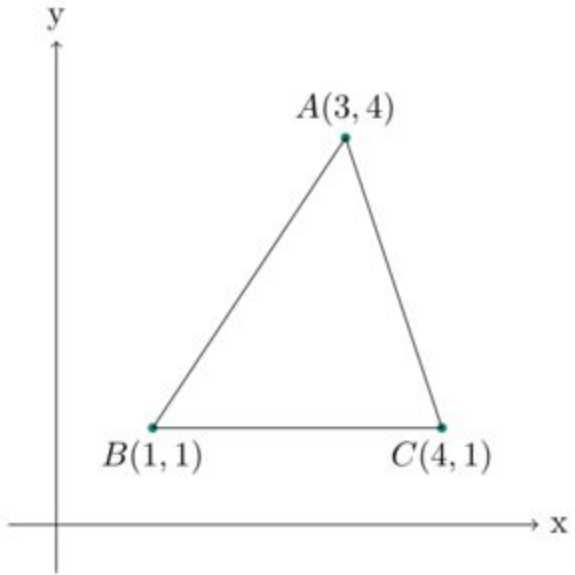
- Given points p_1, p_2, p_3: $2S = (p_2 - p_1) \times (p_3 - p_1))$

Interpretation:

- $S > 0 \longrightarrow$ counter-clockwise
- $S < 0 \longrightarrow$ clockwise
- $S = 0 \longrightarrow$ collinear

This formula gives:

- Orientation
- Triangle area
- This is the core primitive behind polygon algorithms



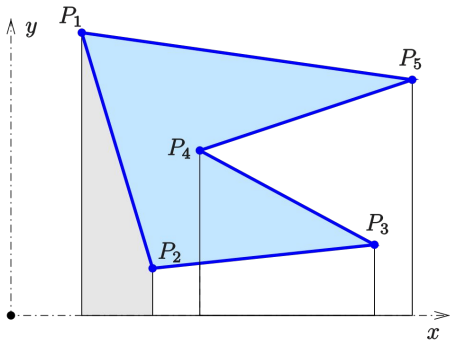
Polygon Area (Shoelace Formula)

Area of a Simple Polygon

- For vertices ordered consistently (CW or CCW):

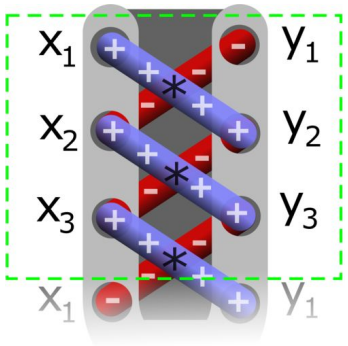
$$2A = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1})$$

(indices modulo n)



This is equivalent to:

- Summing signed triangle areas
- Using cross products of consecutive vertices



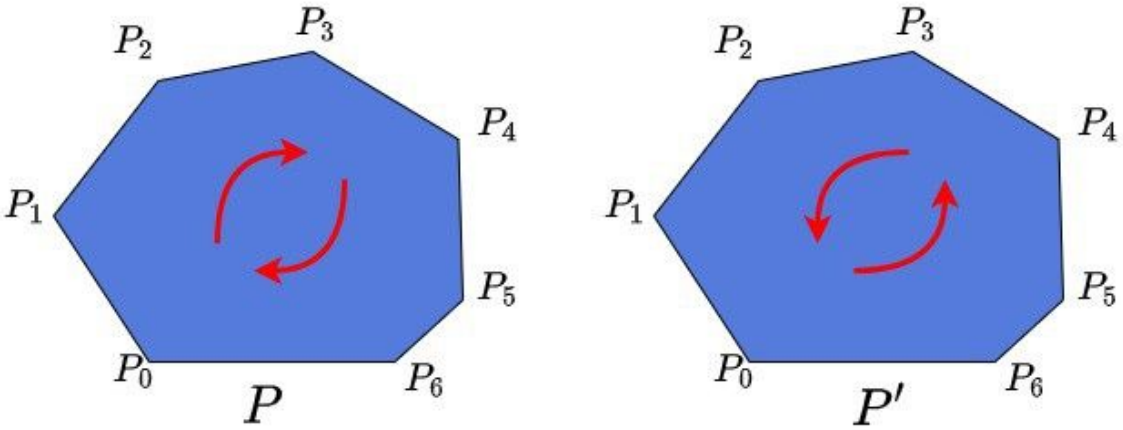
Properties:

- Works for any simple polygon
- Sign gives orientation
- $|A|$ is the actual area

Orientation of a Polygon

Clockwise vs Counter-Clockwise

- A polygon has global orientation
- Determined by the sign of its area $A > 0 \Longrightarrow$ CCW, $A < 0 \Longrightarrow$ CW



Convex vs Non-Convex Polygons

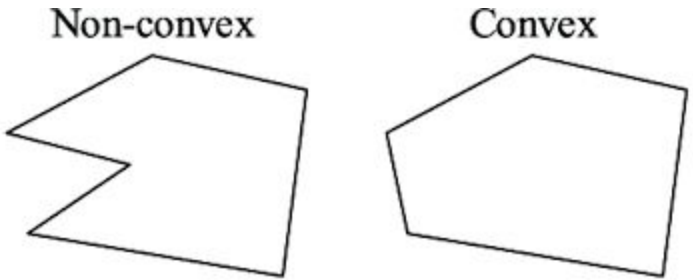
Convex Polygon

- Every turn is in the same direction
- For all triples: $(p_{i+1}-p_i)\times(p_{i+2}-p_{i+1})\geq 0$

- Binary search friendly

Non-Convex Polygon

- At least one “wrong” turn
- Still simple, but harder to process
- This distinction motivates convex hulls



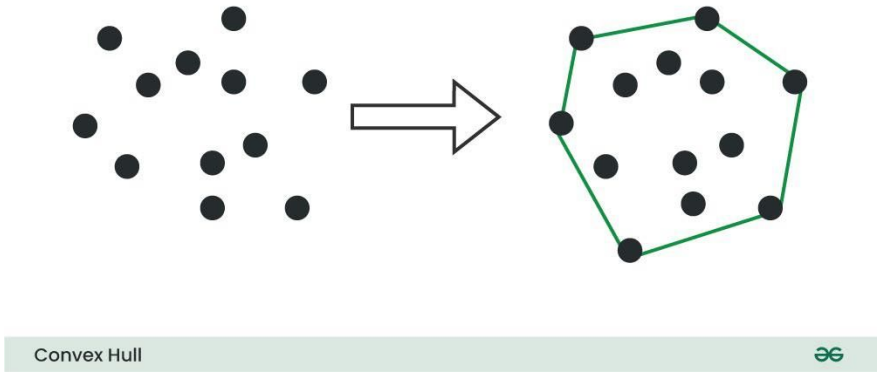
What Is a Convex Hull?

Given a set of points in the plane:

- The convex hull is the smallest convex polygon
- It contains all given points
- All interior points are irrelevant

Intuition:

- Stretching a rubber band around the points
- Points on the hull define the boundary



Convex Hull Construction: High-Level Idea

General Strategy

- Sort points in some order
- Traverse points one by one
- Maintain a candidate hull
- Remove points that break convexity

Key data structure:

- Stack / vector

Time complexity:

- Sorting: $O(n \log n)$
- Processing: $O(n)$

Convex Hull Construction: Graham Scan

Steps:

- Find the starting point $P0$:
  - Lowest y-coordinate
  - If tie: smallest x
- Sort all other points by:
  - Increasing polar angle around $P0$
  - Break ties by increasing distance from $P0$
- Traverse points in sorted order:
  - Maintain a stack of hull points
  - While the last three points do not form a clockwise turn:
    - Remove the middle point
- The stack contains the convex hull in clockwise order

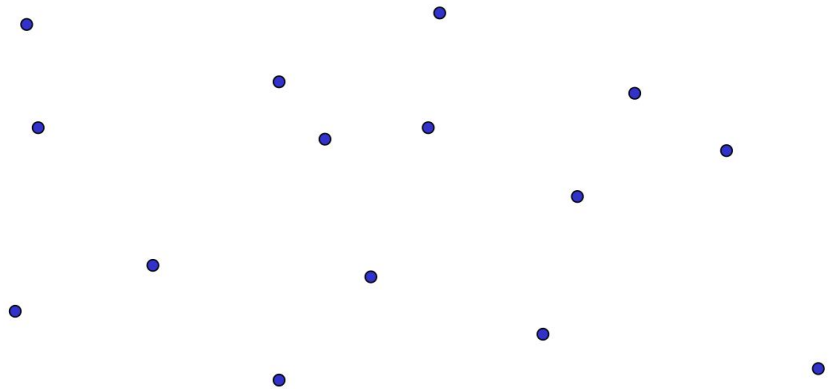
Key tool: **Orientation test** (cross product)

- Time complexity: $O(n \log n)$

Geometric divide-and-conquer

Closest Pair Problem:

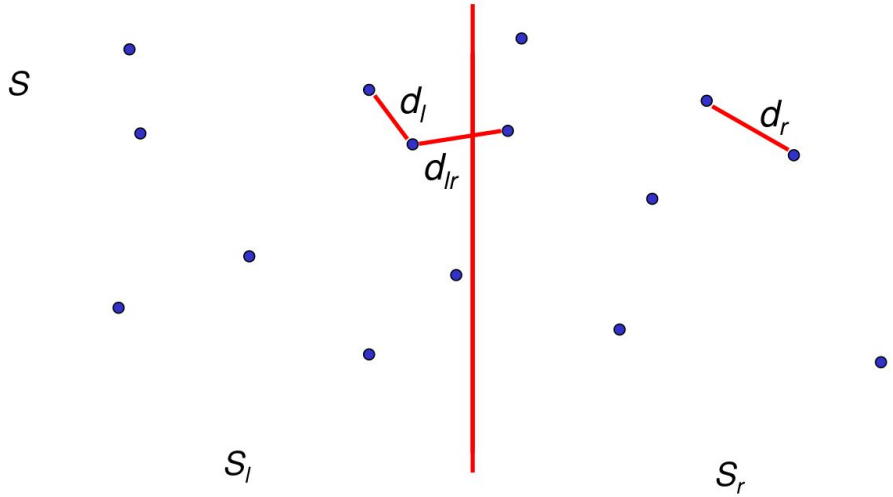
- Given a set S of n points in the plane, find a pair of points with the smallest distance.



Geometric divide-and-conquer

Divide and Conquer Method:

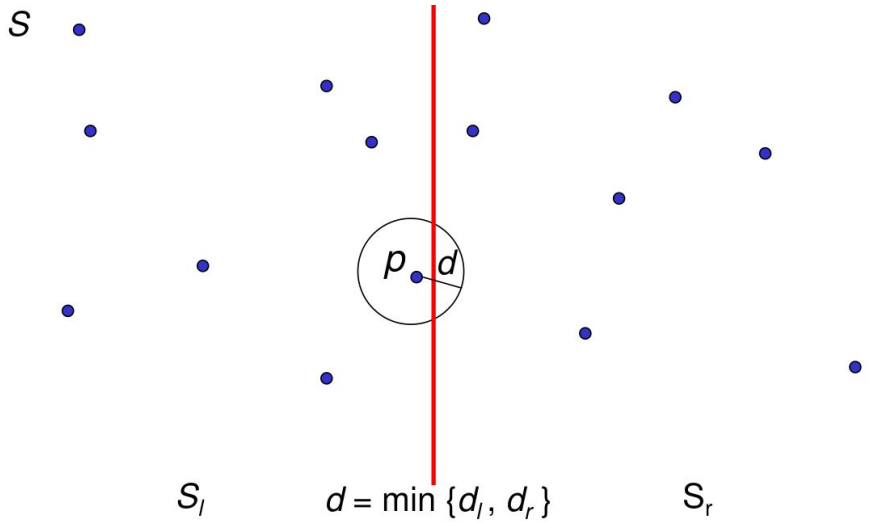
1. **Divide**: Divide S into two equal sized sets S_l und S_r .
2. **Conquer**: d_l = mindist(S_l), d_r = mindist(S_r)
3. **Merge**: d_{lr} = min{ d(p_l ,p_r) | $p_l \in S_l$, $p_r \in S_r$ }
4. return min{d_l , d_r , d_{lr} }



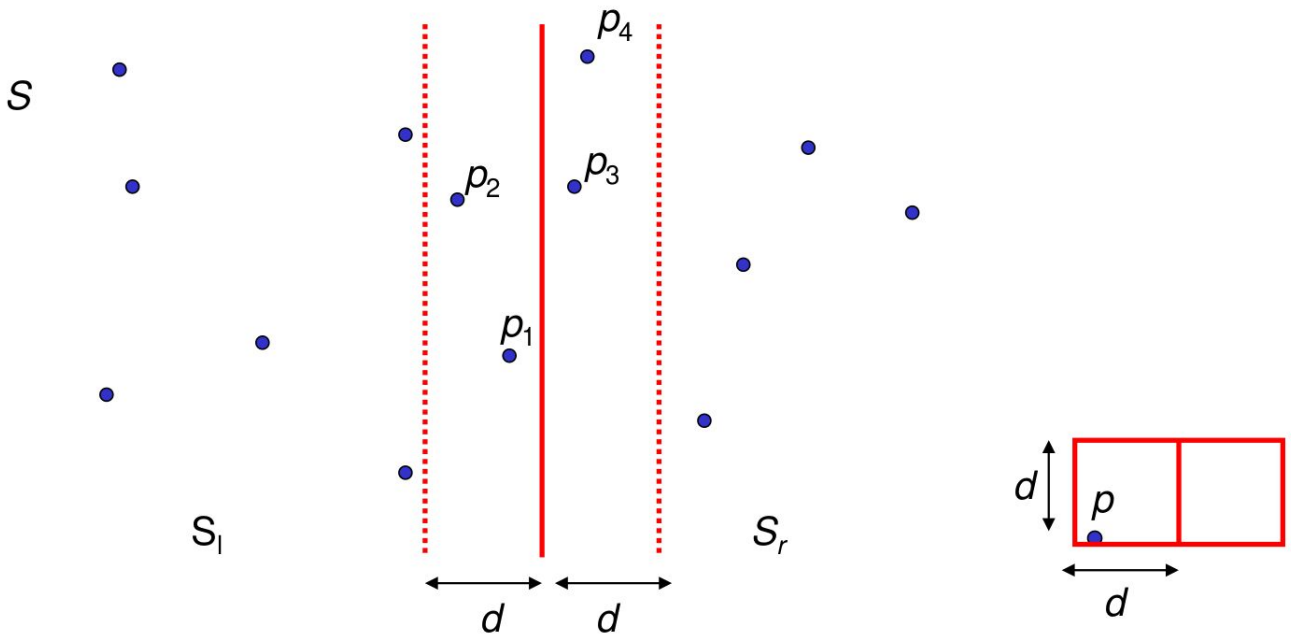
Geometric divide-and-conquer

Divide and Conquer Method:

1. **Divide**: Divide S into two equal sized sets S_l und S_r .
2. **Conquer**: d_l = mindist(S_l), d_r = mindist(S_r)
3. **Merge**: d_{lr} = min{ d(p_l ,p_r) | $p_l \in S_l$, $p_r \in S_r$ }
4. return min{d_l , d_r , d_{lr} }



Merge Step



$d = \min \{ d_l, d_r \}$

Merge Step

- Initially sort the points in S in order of increasing x-coordinates O(n log n).
- Each bisection line can be determined in O(1) time.
- Once the subproblems S_l , S_r are solved, generate a list of the points in S in order of increasing y-coordinates. This can be done by **merging the sorted lists of points** of S_l , S_r (merge sort).

Segment Intersection: The Core Problem

Problem Statement

Given two segments:

- AB
- CD

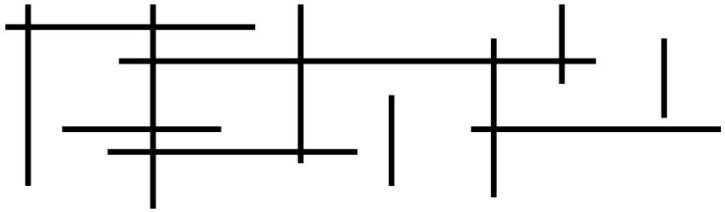
Determine whether they:

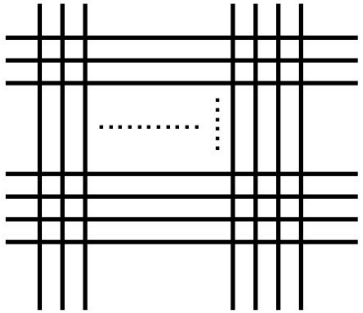
- intersect at a point
- overlap
- do not intersect

This is one of the most common geometry subproblems.

Line Segment Intersection

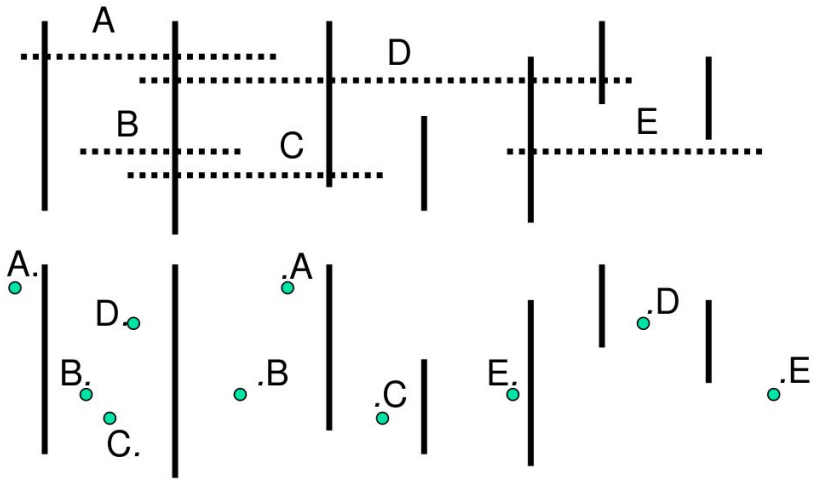
Find all pairs of intersecting line segments





Line Segment Intersection

Find all pairs of intersecting line segments



The representation of the horizontal line segments by their endpoints allows for a vertical partitioning of all objects.

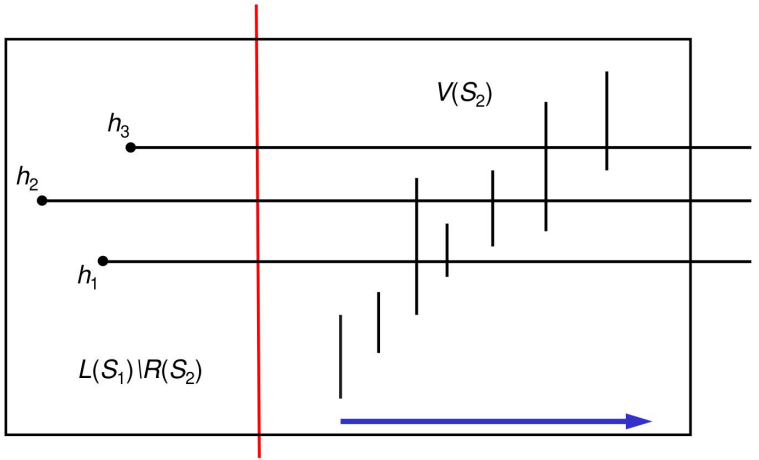
Output of intersections

Can be similarly solved with DnC

- Divide plane by a vertical line
- Solve intersections on left and right
- Detect cross-boundary intersections

Complexity:

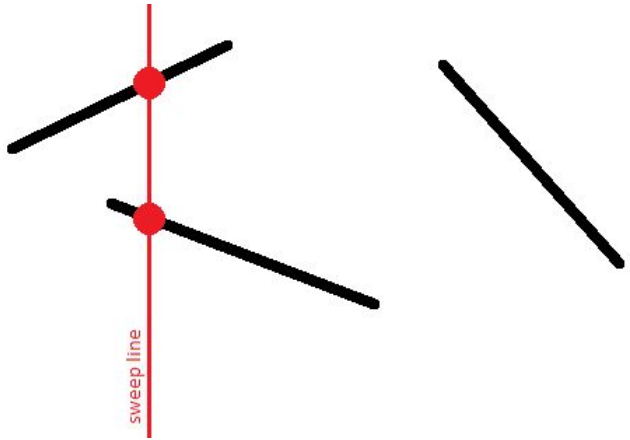
- $O(n\log n+k)$
- k = number of intersections



Another Approach: Sweep Line Algorithm

Idea: Let's draw a vertical line and start moving this line to the right.

- A segment becomes **active** when the sweep line reaches its left endpoint
- The segment remains active while the line intersects it
- It becomes inactive after passing its right endpoint



Assumptions:

- No vertical segments
- Each active segment intersects the sweep line at exactly one point

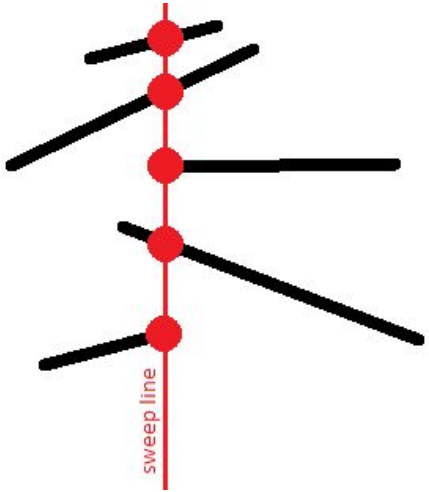
At any time, the sweep line maintains the set of currently active segments.

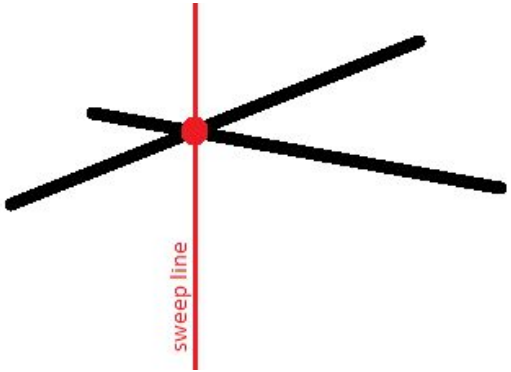
Sweep Line Algorithm

- At any time, consider all segments intersecting the sweep line
- Store these active segments ordered by their y-coordinate at the sweep line
- This vertical order changes only when segments intersect

Two segments intersect if and only if their **relative order** along the sweep line changes.

- Any intersecting pair of segments must become **neighbors** at some moment



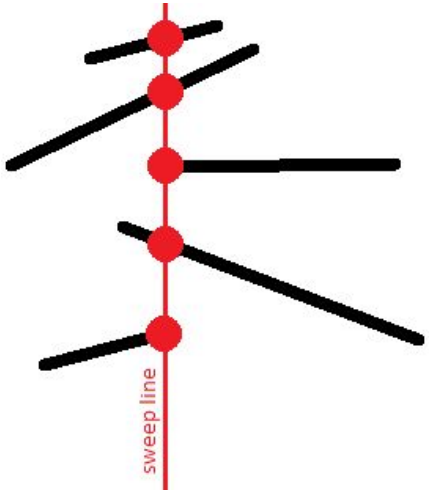


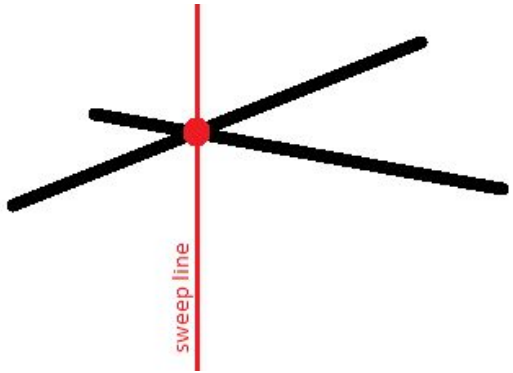
Sweep Line Algorithm: Concept

The set and order of active segments only change when:

- a segment starts
- a segment ends

Thus, it is sufficient to process the sweep line only at the x-coordinates of segment endpoints.





Handling Segment Insertion / Removal

When a segment starts:

- Insert it into the active set at the correct position (by y-order)
  Check for intersections with
  - its immediate neighbor above
  - its immediate neighbor below
- Only neighboring segments can intersect next
- No other segment's relative order is affected

When a segment ends:

- Remove it from the active set
- Its former upper and lower neighbors become adjacent
- Check these two segments for intersection

Checking only neighboring segments at events is sufficient.

Sweep Line for Segment Intersection

- Each segment:
  - is inserted once
  - is removed once

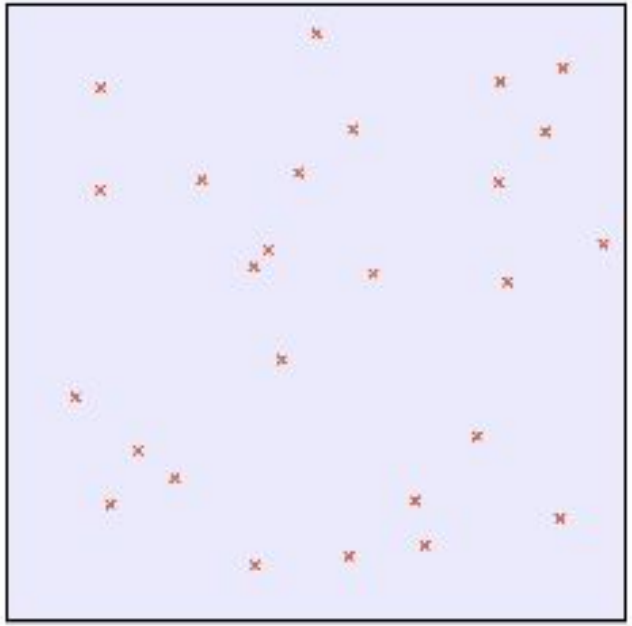
- Each event triggers a constant number of intersection checks
- Operations on the ordered structure take O(logn)

Final complexity: **O(nlogn)**

Sweep Line Applications

Sweep line is used for:

- Segment intersection
- Area of union of rectangles
- Closest pair of points
- Interval overlap in time
- Coverage problems



Assignment 5

Deadline: 14/01/2026

Mat vs. the Backyard Cat Gang

- Scare toys placed as points in the plane
- A point is protected if every direction moves closer to some toy
- Protected region depends on geometry of points

Constraints:

- n, m up to 100,000
- Need fast geometric updates

Happy Christmas !