Computational Geometry in Competitive Programming

Convex Hull – Naïve Algorithm, Graham’s Scan
Plane Sweep – Line segment intersection
Convexity and Convex Hull

A set $S$ is **convex** if for any pair of points $p, q \in S$, the entire line segment $pq \subseteq S$.

• The **convex hull** of a set $S$ is the minimal convex set that contains $S$.

• (Equivalent definition) The intersection of all convex sets that contain $S$. 
Computation of Convex Hull

Input = set of points:
\[ P = \{ p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9 \} \]

Output = representation of the convex hull (clockwise order):
\[ CH(P) = \{ p_4, p_5, p_8, p_2, p_9 \} \]

- \( CH(P) \) is a **convex polygon**

- **Edges** of \( CH(P) \)
  - all points of \( P \setminus \{ p, q \} \) lie to the right of directed line through \( p \) and \( q \) \( \Rightarrow \) \( pq \) is an edge of \( CH(P) \)
Convex Hull — Naïve Algorithm

Algorithm $\text{SLOW}\text{ConvexHull}(P)$

Input. A set $P$ of points in the plane.

Output. A list $\mathcal{L}$ containing the vertices of $\mathcal{C}\mathcal{H}(P)$ in clockwise order.

1. $E \leftarrow \emptyset$.
2. for all ordered pairs $(p, q) \in P \times P$ with $p$ not equal to $q$
3. do valid $\leftarrow$ true
4. for all points $r \in P$ not equal to $p$ or $q$
5. do if $r$ lies to the left of the directed line from $p$ to $q$
6. then valid $\leftarrow$ false.
7. if valid then Add the directed edge $\overrightarrow{pq}$ to $E$.
8. From the set $E$ of edges construct a list $\mathcal{L}$ of vertices of $\mathcal{C}\mathcal{H}(P)$, sorted in clockwise order.
Implementation – Naïve Alg.

• Points

```cpp
struct PT
{
    double x, y;
    // functions
};
```

• Set of points

```cpp
vector<PT> Points;
```

• Directed edges

```cpp
struct EDGE
{
    PT origin, destination;
};
```
Implementation – Naïve Alg.

• If \( r \) lies to the left of the directed line from \( p \) to \( q \)

\[
\overrightarrow{rp} \times \overrightarrow{rq} = \begin{vmatrix} i & j & k \\ x_p - x_r & y_p - y_r & 0 \\ x_q - x_r & y_q - y_r & 0 \end{vmatrix} = (0, 0, \text{cross})
\]

// decide if \( r \) lies on the left, right or on the line \( p \rightarrow q \)
char side(PT& p, PT& q, PT& r)
{
    double cross = (p.x - r.x) * (q.y - r.y) - (p.y - r.y) * (q.x - r.x);

    if (fabs(cross) <= EPS) return 'o'; // o ----------- on the line
    if (cross > 0) return 'l'; // l ----------- left
    return 'r'; // r ----------- right
}
Naïve Alg. — Complexity and Robustness

- **Time Complexity:** $O(n^3)$
  - Number of point pairs: $\binom{n}{2} = \Theta(n^2)$
  - Check all points for each pair: $O(n)$

- **Space Complexity:** $O(n)$

• Not Robust
  • Degenerate case
  • Rounding errors of coordinates
Convex Hull — Graham’s Scan

**Incremental Algorithm**
- Add points from left to right

First upper hull, then lower hull

Update of upper hull
- The rightmost point is on the upper hull
- Check right turn
- Delete middle point
Upper Hull — Graham’s Scan

Algorithm ConvexHull(P)

Input. A set $P$ of points in the plane.

Output. A list containing the vertices of $\mathcal{CH}(P)$ in clockwise order.

1. Sort the points by $x$-coordinate, resulting in a sequence $p_1, \ldots, p_n$.
2. Put the points $p_1$ and $p_2$ in a list $\mathcal{L}_{\text{upper}}$, with $p_1$ as the first point.
3. for $i \leftarrow 3$ to $n$
   4. do Append $p_i$ to $\mathcal{L}_{\text{upper}}$.
5. while $\mathcal{L}_{\text{upper}}$ contains more than two points and the last three points in $\mathcal{L}_{\text{upper}}$ do not make a right turn
   do Delete the middle of the last three points from $\mathcal{L}_{\text{upper}}$. 
Convex Hull — Graham’s Scan

**Algorithm** $\text{CONVEXHULL}(P)$

*Input.* A set $P$ of points in the plane.

*Output.* A list containing the vertices of $\mathcal{CH}(P)$ in clockwise order.

1. Sort the points by $x$-coordinate, resulting in a sequence $p_1, \ldots, p_n$.
2. Put the points $p_1$ and $p_2$ in a list $\mathcal{L}_{\text{upper}}$, with $p_1$ as the first point.
3. for $i \leftarrow 3$ to $n$
   4. do Append $p_i$ to $\mathcal{L}_{\text{upper}}$.
   5. while $\mathcal{L}_{\text{upper}}$ contains more than two points and the last three points in $\mathcal{L}_{\text{upper}}$ do not make a right turn
   6. do Delete the middle of the last three points from $\mathcal{L}_{\text{upper}}$.
5. Put the points $p_n$ and $p_{n-1}$ in a list $\mathcal{L}_{\text{lower}}$, with $p_n$ as the first point.
6. for $i \leftarrow n-2$ downto 1
   7. do Append $p_i$ to $\mathcal{L}_{\text{lower}}$.
   8. while $\mathcal{L}_{\text{lower}}$ contains more than 2 points and the last three points in $\mathcal{L}_{\text{lower}}$ do not make a right turn
   9. do Delete the middle of the last three points from $\mathcal{L}_{\text{lower}}$.
10. Remove the first and the last point from $\mathcal{L}_{\text{lower}}$ to avoid duplication of the points where the upper and lower hull meet.
11. Append $\mathcal{L}_{\text{lower}}$ to $\mathcal{L}_{\text{upper}}$, and call the resulting list $\mathcal{L}$.
12. return $\mathcal{L}$
Time complexity – Graham’s Scan

- Sorting - $O(n \log n)$
- for-loop - $O(n)$
  - while-loop – each point can be deleted only once

\[
\text{for } i \leftarrow 3 \text{ to } n \\
\quad \text{do Append } p_i \text{ to } \mathcal{L}_{\text{upper}}. \\
\quad \text{while } \mathcal{L}_{\text{upper}} \text{ contains more than two points and the last three points in } \mathcal{L}_{\text{upper}} \text{ do not make a right turn} \\
\quad \quad \text{do Delete the middle of the last three points from } \mathcal{L}_{\text{upper}}. 
\]
Implementation – Graham’s Scan

• If $p_3 \rightarrow p_2 \rightarrow p_1$ make a left turn (not right turn)

```cpp
// check p3 (leftmost) \(\rightarrow\) p2 \(\rightarrow\) p1 (rightmost) left turn (not right turn)
bool left(PT& p1, PT& p2, PT& p3)
{
    double cross = (p2.x - p3.x) * (p1.y - p3.y) - (p2.y - p3.y) * (p1.x - p3.x);
    if (fabs(cross) <= EPS)
        return true;  // to delete p2
    if (cross > 0)
        return true;
    return false;
}
```
Implementation – Graham’s Scan

• **Input:** a set of points in the plane (points.txt)
• **Output:** vertices of convex hull in clockwise order

```
5.5 6
2 7
10.5 0.5
3 4
8.5 5.5
12 5
7.5 2.5
6 9
10.5 3.5
8.5 7
```
Line Segment Intersection (LSI)

- **Problem** – Given a set of $n$ closed segments $S = \{s_1, s_2, \ldots, s_n\}$ in the plane
- Report all intersection points
- Count the number of intersection points

- **Assumptions**
  - No line segment is horizontal.
  - No two segments overlap in more than one point.
  - No three segments intersect at a common point.

- **Naïve Algorithm**
  - Check each pair of segments for intersection.
  - Complexity: $\Theta(n^2)$ time, $\Theta(n)$ space.
LSI - Plane Sweep Algorithm

- **Sweep line** – an imaginary line
- A horizontal line sweeping downwards over the plane
- **Status** of sweep line – the set of segments intersecting it
- Update status only at particular points

- **Event points**
  - Upper endpoint
  - Intersection
  - Lower endpoint
LSI — Basic Idea

We are able to identify all intersections by looking **only** at **adjacent** segments in the sweep line status during the sweep.

- **Theorem** – Just before an intersection occurs (infinitesimally-close to it), the two respective segments are adjacent to each other in the sweep-line status.
- **In practice** – Whenever two line segments become adjacent along the sweep line, check for their intersection below the sweep line.
LSI – Alg. Overview

• Imagine moving a horizontal line downwards over the plane
  • Above the sweep line – finished
  • Below the sweep line – to be explored

• Maintain status of the sweep line

• Sweep line halts at event points
  • Endpoints of the segments – known from input
  • Intersection points – computed on the fly

• Take actions at event points
  • Update status
  • Detect intersections
LSI - Plane Sweep Algorithm

- **Data structures** – 2 balanced BST
- **Event queue** $Q$ – store events according to the order of events ($y$ coordinates)
- **Status structure** $T$ – maintain status of the sweep line
  - Used to access neighbors of a given segment
  - Leaves – segments intersecting the sweep line
  - Internal node – rightmost leaf in its left subtree
Algorithm FINDINTERSECTIONS($S$)

Input. A set $S$ of line segments in the plane.

Output. The set of intersection points among the segments in $S$, with for each intersection point the segments that contain it.

1. Initialize an empty event queue $Q$. Next, insert the segment endpoints into $Q$; when an upper endpoint is inserted, the corresponding segment should be stored with it.

2. Initialize an empty status structure $T$.

3. while $Q$ is not empty

4. do Determine the next event point $p$ in $Q$ and delete it.

5. $\text{HANDLEEVENTPOINT}(p)$
Plane Sweep — Handle Events

➢ **Upper Endpoint**

1. **Locate segment** position in the status $T$.
2. **Insert segment** into sweep line status $T$.
3. **Test for intersection** below the sweep line against its 2 neighbors along the sweep line (if exist).
4. **Insert intersection point(s)** (if found) into the event queue.
Plane Sweep — Handle Events

- **Lower Endpoint**
  1. Locate segment position in the status $T$.
  2. Delete segment from sweep line status $T$.
  3. Test for intersection below the sweep line between its 2 neighbors along the sweep line (if exist).
  4. Insert intersection point(s) (if found and distinct) into the event queue.
Plane Sweep — Handle Events

➢ **Intersection Point**

1. **Report/count** the point.
2. **Swap** the two line segments in the sweep line status $T$.
3. **Test for intersection** below the sweep line against each of their new neighbor along the sweep line (if exist).
4. **Insert intersection point(s)** (if found and distinct) into the event queue.
Complexity

➢ Time Complexity - $O(m \log n) = O((n + 1) \log n)$

1. Initialize an empty event queue $Q$. Next, insert the segment endpoints into $Q$; when an upper endpoint is inserted, the corresponding segment should be stored with it. $O(n \log n)$

2. Initialize an empty status structure $\mathcal{T}$. Constant

3. while $Q$ is not empty

4. do Determine the next event point $p$ in $Q$ and delete it.

5. HANDLEEVENTPOINT($p$)

➢ At most 2 new events need to be inserted into $Q$

➢ Deletions, insertions on $Q$ - $O(\log n)$

➢ Number of operations on $T$ linear in $m$ (# all event points)

➢ Deletions, insertions, neighbor finding on $T$ - $O(\log n)$
Implementation – Plane Sweep Alg.

- **Input:** a list of non-overlapping axis-aligned rectangles (rectangles.txt)
- **Output:** the number of neighbors of each rectangle
Implementation – Plane Sweep Alg.

- Sweep line **status**: set (BBST)
- **Events**: 
Implementation – Plane Sweep Alg.

• Sweep line **status**: set (BBST)

• **Events**: 2 lower endpoints of all rectangles

• Handle events
  • Right vertex: remove rectangle from sweep line status
  • Left vertex: find adjacent rectangles from the status, if any neighbor is found, increase count by 1 for both this rectangle and its neighbor(s), insert rectangle into status

• Event queue: static (no insertion needed) ⇒ array / vector

\( (x, y) \quad (x + W, y) \)
How to order events?

How to define tree node of the status?