UCF ICPC Training Camp
Day IV: Geometry

Yongwhan Lim

Friday, March 24, 2022

Yongwhan Lim

Education

Part-time Jobs

Full-time Job

Workshops

Coach/Judge

https://www.yongwhan.io

Yongwhan Lim

- Currently:
    - a **Co-Founder** in a Stealth Mode Startup;
    - ICPC **Internship Director**;
    - Columbia ICPC **Head Coach**;
    - ICPC **Judge** for NAQ and Regionals;
    - **Lecturer** at MIT;
    - **Adjunct** (Associate in CS) at Columbia;

https://www.yongwhan.io

Today's Format

9am ET - 10:20am ET	**Review (UCF ICPC Training Camp Day 3)**
10:30am ET - 12pm ET	**Lecture & Lecture Exercises**
12pm ET - 12:45pm ET	**Lunch**
12:45pm ET - 3:45pm ET	**Practice Contest**
	UCF ICPC Training Camp Day 4
4pm ET - 5:20pm ET	**Review**

Any loose-ends (Mobius inversion code, some problem explanations skipped, etc.) will be covered **tomorrow (Saturday) morning**!

Request 1:1 Meeting, through Calendly

- Use calendly.com/yongwhan/one-on-one to request 1:1 meeting:
  - Mock Interview
  - Resume Critique
  - Career Planning
  - Practice Strategy
  - …
- Always inspired by driven students like yourself!
- Since I'd feel honored/thrilled to talk to you, do not feel shy to sign up!!

Lecture

Today's Overview: Computational Geometry

Main

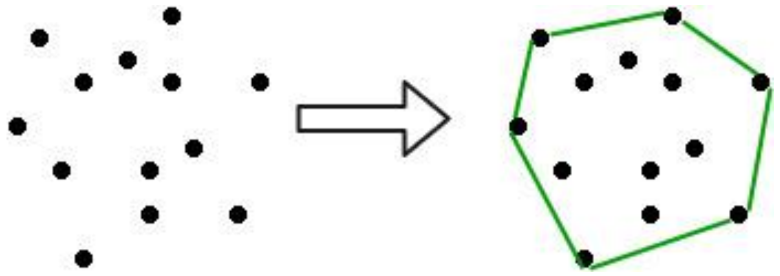
I. Convex Hull Construction
II. Convex Hull Trick and Li Chao Tree
III. Sweep-line

Review

IV. Length of the union of segments
V. Oriented area of a triangle
VI. Area of simple polygon
VII. Pick's Theorem

I. Convex Hull Construction: Motivation

- Consider N points given on a plane.
- The objective is to generate a **convex hull**, i.e. *the smallest convex polygon that contains all the given points*.
- Can be done in **N log N** using:
  - **Graham's scan algorithm** (Graham, 1972)
  - **Monotone chain algorithm** (Andrew, 1979)



I. Graham’s Scan Algorithm: Main Idea

- Find the bottom-most point P_0. If there are multiple points with the same Y coordinate, the one with the smaller X coordinate is considered. This takes O(N).
- All the other points are sorted by polar angle in clockwise order. If the polar angle between two points is the same, the nearest point is chosen instead.
- Iterate through each point one by one, and make sure that the current point and the two before it make a clockwise turn; otherwise, the previous point is discarded, since it would make a non-convex shape.
- Use a stack to store the points, and once we reach P_0, return the stack containing all the points of the convex hull in clockwise order.

I. Graham’s Scan Algorithm: Collinear Case

- **If you need to include the collinear points while doing a Graham scan, you need another step after sorting.**
  - You need to get the points that have the biggest polar distance from P_0 (these should be at the end of the sorted vector) and are collinear.
  - The points in this line should be reversed so that we can output all the collinear points, otherwise the algorithm would get the nearest point in this line and bail.
- **This step shouldn't be included in the non-collinear version of the algorithm, otherwise you wouldn't get the smallest convex hull.**

I. Graham's Scan Algorithm: Implementation

- https://cp-algorithms.com/geometry/convex-hull.html#implementation

I. Monotone Chain Algorithm: Main Idea

- The algorithm first finds the **leftmost** and **rightmost** points A and B.
  - In the event multiple such points exist, the lowest among the left (lowest Y-coordinate) is taken as A, and the highest among the right (highest Y-coordinate) is taken as B.
- Clearly, A and B must both belong to the convex hull as they are the farthest away and they cannot be contained by any line formed by a pair among the given points.

I. Monotone Chain Algorithm: Main Idea

- Draw a line through AB.
- This divides all the other points into two sets, S_1 and S_2, where S_1 contains all the points **above the line** connecting A and B, and S_2 contains all the points **below the line** joining A and B.
  - The points that lie on the line joining A and B may belong to either set.
  - The points A and B belong to both sets.
- Now, we just need to construct the upper set S_1 and the lower set S_2 and then combine them!

I. Monotone Chain Algorithm: Main Idea

- To get the upper set, we sort all points by the **x-coordinate**.
- For each point we check if either - the current point is the last point, **B**, or if the **orientation** between the line between A and the current point and the line between the current point and B is **clockwise**.
- In those cases the current point belongs to the upper set S_1.

I. Monotone Chain Algorithm: Main Idea

- If the given point belongs to the upper set, we **check the angle** made by the line connecting the second last point and the last point in the upper convex hull, with the line connecting the last point in the upper convex hull and the current point.
- If the angle is **not clockwise**, we **remove** the **most recent** point added to the upper convex hull as the current point will be able to contain the previous point once it is added to the convex hull.

I. Monotone Chain Algorithm: Main Idea

- **By symmetry, the same logic applies for the lower set S_2:**
  - If either - the current point is B, or the orientation of the lines, formed by A and the current point and the current point and B, is counterclockwise - then it belongs to S_2.
  - If the given point belongs to the lower set, we act similarly as for a point on the upper set except we check for a counterclockwise orientation instead of a clockwise orientation.

I. Monotone Chain Algorithm: Main Idea

- **By symmetry, the same logic applies for the lower set S_2:**
  - If the angle made by the line connecting the second last point and the last point in the lower convex hull, with the line connecting the last point in the lower convex hull and the current point is not counterclockwise, we remove the most recent point added to the lower convex hull (as the current point will be able to contain the previous point once added to the hull).
- **The final convex hull is obtained from the union of the upper and lower convex hulls.**

I. Monotone Chain Algorithm: Collinear

- Check collinearity in the clockwise/counterclockwise routines.
- However, this allows for a degenerate case where all the input points are collinear in a single line, and the algorithm would output repeated points.
- To solve this, we check whether the upper hull contains all the points, and if it does, we just return the points in reverse, as that is what Graham's implementation would return in this case.

I. Monotone Chain Algorithm: Implementation

- https://cp-algorithms.com/geometry/convex-hull.html#implementation_1

II. Convex Hull Trick and Li Chao Tree: Motivation

- There are n cities.
- You want to travel from city 1 to city n by car.
- To do this, you have to buy some gasoline.
- It is known that a liter of gasoline costs $cost_k$ in the kth city.
- Initially your fuel tank is empty and you spend one liter of gasoline per kilometer.
- Cities are located on the same line in ascending order with kth city having coordinate x_k.
- Also you have to pay $toll_k$ to enter kth city.
- Your task is to make the trip with minimum possible cost.

II. Convex Hull Trick and Li Chao Tree: Motivation

- Need to solve:

$$dp_i = toll_i + \min_{j<i}(cost_j \cdot (x_i - x_j) + dp_j)$$

- Naive approach will give you $O(n^2)$ complexity.

II. Convex Hull Trick and Li Chao Tree: Motivation

- This can be improved to **O(n log n)**.
    - The problem can be reduced to adding linear functions kx + b to the set and finding minimum value of the functions in some particular point x.

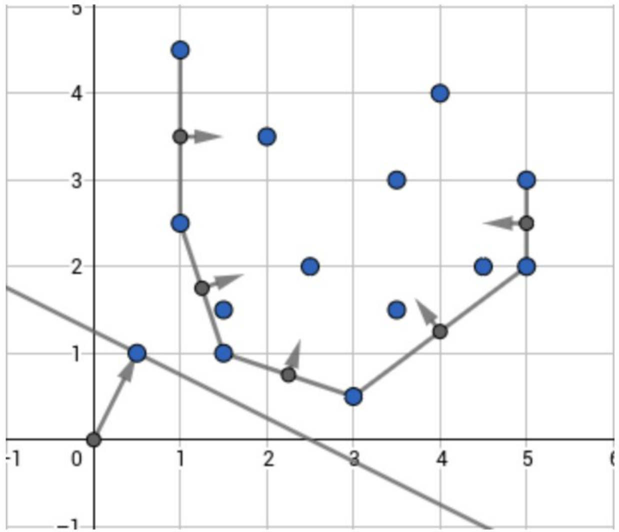
- There are two main approaches one can use here.
    - **Convex Hull Trick**
    - **Li Chao Tree**

II. Convex Hull Trick: Main Idea

- **Maintain a lower convex hull of linear functions.**
- It would be a bit more convenient to consider them not as linear functions, but as points (k;b) on the plane such that we will have to find the point which has **the least dot product** with a given point (x;1); that is, for this point kx+b is minimized which is the same as initial problem.

II. Convex Hull Trick: Main Idea

- Such minimum will necessarily be on lower convex envelope of these points as can be seen below:



II. Convex Hull Trick: Main Idea

- **Keep points on the convex hull and normal vectors of the hull's edges.**
- When you have a (x;1) query, you'll have to find the normal vector closest to it in terms of angles between them, then the optimum linear function will correspond to one of its endpoints.
    - Points having a constant dot product with (x;1) lie on a line which is orthogonal to (x;1); so, the optimum linear function will be the one in which tangent to convex hull which is collinear with normal to (x;1) touches the hull.

II. Convex Hull Trick: Implementation

- https://cp-algorithms.com/geometry/convex_hull_trick.html#convex-hull-trick

II. Li Chao Tree: Main Idea

- Assume you are given a set of functions such that each two can intersect at most once.
- Let's **keep in each vertex of a segment tree** *some function* in such way, that if we go from root to the leaf it will be guaranteed that one of the functions we met on the path will be the one giving the minimum value in that leaf.
- Let’s construct this segment tree!

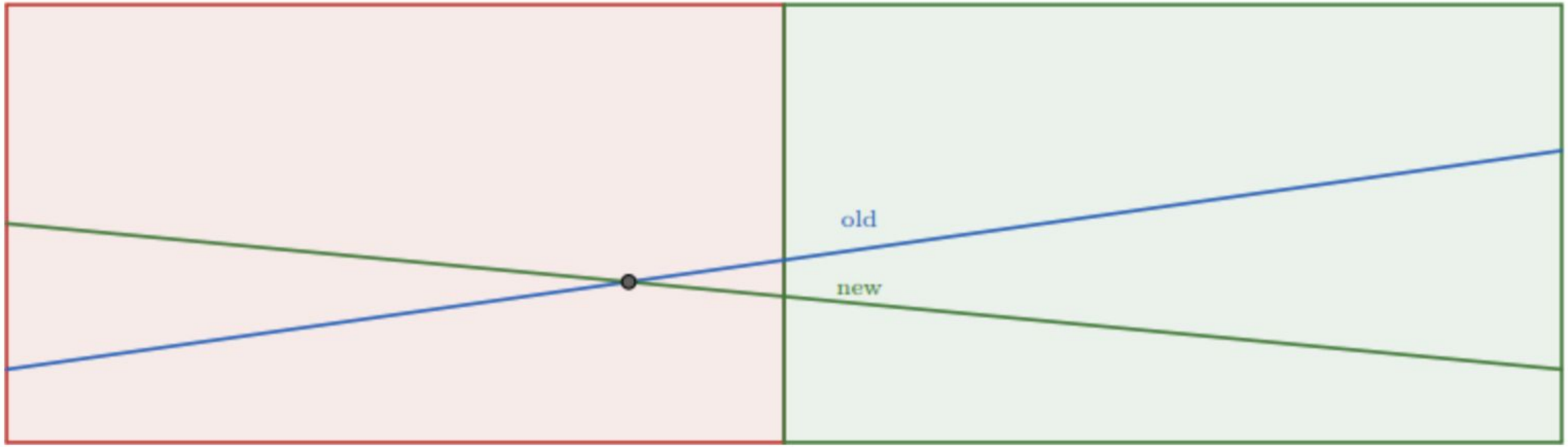
II. Li Chao Tree: Main Idea

- Assume we're in some vertex corresponding to half-segment **[l,r)** and the function $\mathbf{f_{old}}$ is kept there and we add the function $\mathbf{f_{new}}$.
- Then the **intersection point** will be **either in [l;m) or in [m;r)** where m is the midpoint of l and m.
- We can efficiently find that out by comparing the values of the functions in points l and m.

II. Li Chao Tree: Main Idea

- If the **dominating function changes**, then it is in [l;m) otherwise it is in [m;r).
- For the half of the segment with no intersection, we will pick the lower function and write it in the current vertex.
- You can see that it will always be the **one which is lower in point m.**
- After that, we **recursively** go to the other half of the segment with the function which was the upper one.

II. Li Chao Tree: Illustration



II. Li Chao Tree: Implementation

- https://cp-algorithms.com/geometry/convex_hull_trick.html#li-chao-tree

III. Sweep-line: Problem Statement

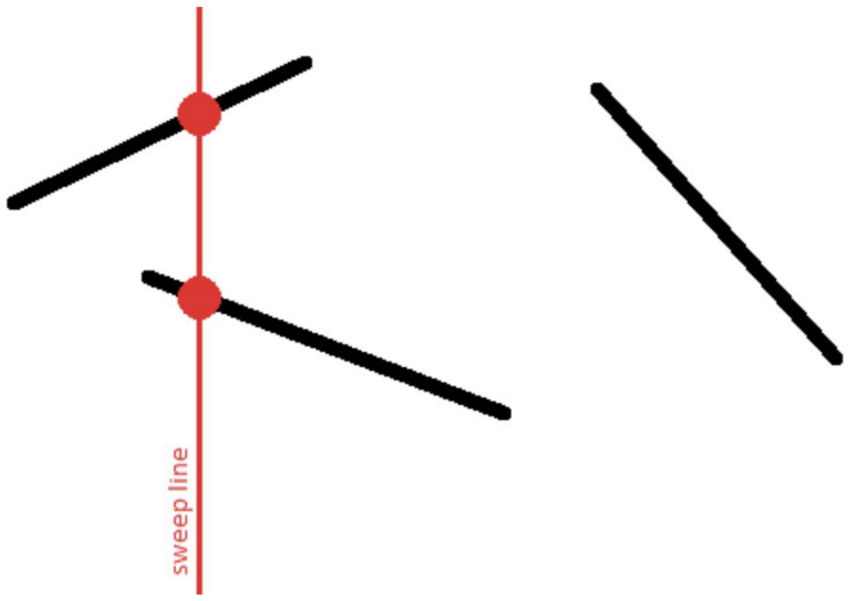
- **Given n line segments on the plane. It is required to check whether at least two of them intersect with each other.**
- If the answer is yes, then **print this pair** of intersecting segments; it is enough to choose any of them among several answers.

- The naive solution algorithm is to iterate over all pairs of segments in $O(n^2)$ and check for each pair whether they intersect or not.
- **Sweep line** algorithm can achieve this in **O(n log n)**!

III. Sweep-line: Main Idea

- Let's draw a vertical line $x = -\infty$ mentally and start moving this line to the right.
- In the course of its movement, this line will meet with segments, and at each time a segment intersect with our line it intersects in exactly one point (we will assume that there are no vertical segments).

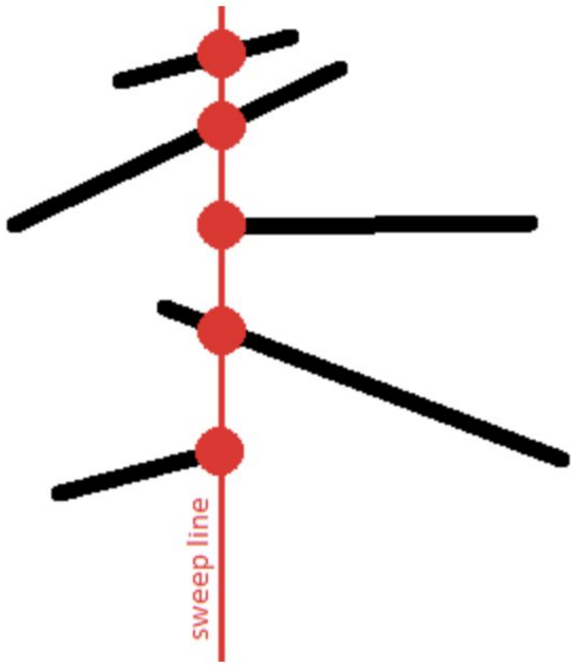
III. Sweep-line: Main Idea



III. Sweep-line: Main Idea

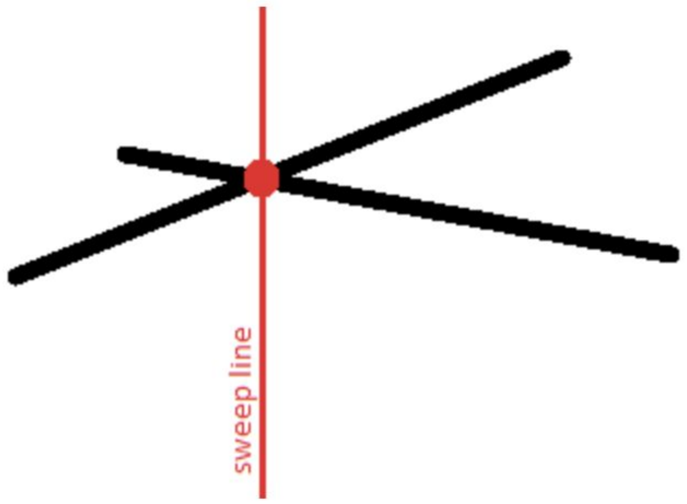
- Thus, for each segment, at some point in time, its point will appear on the sweep line, then with the movement of the line, this point will move, and finally, at some point, the segment will disappear from the line.
- We are interested in the **relative order of the segments** along the vertical. Namely, we will store a list of segments crossing the sweep line at a given time, where the segments will be sorted by their y-coordinate on the sweep line.

III. Sweep-line: Main Idea



III. Sweep-line: Main Idea

- This order is interesting because intersecting segments will have the same y-coordinate at least at one time.



III. Sweep-line: Implementation

- https://cp-algorithms.com/geometry/intersecting_segments.html#implementation

IV. Length of the union of segments: Main Idea

- **Problem**: Given n segments on a line, each described by a pair of coordinates (a_{i1}, a_{i2}). Find the length of their union.

IV. Length of the union of segments: Main Idea

- **Idea** (Klee, 1977) in O(n log n):
  - Store in array x the endpoints of all the segments sorted by their values.
  - Store whether it is a left end or a right end of a segment.
  - Iterate over the array, keeping a counter c of currently opened segments.
  - Whenever the current element is a left end, increase this counter, and otherwise we decrease it.
  - Sum all $x_i - x_{i-1}$ where there is at least one open segment.

IV. Length of the union of segments: Code

```
int length_union(const vector<pair<int, int>> &a) {
    int n = a.size();
    vector<pair<int, bool>> x(n*2);
    for (int i = 0; i < n; i++) {
        x[i*2] = {a[i].first, false};
        x[i*2+1] = {a[i].second, true};
    }
    sort(x.begin(), x.end());
```

IV. Length of the union of segments: Code

```
    int result = 0;
    int c = 0;
    for (int i = 0; i < n * 2; i++) {
        if (i > 0 && x[i].first > x[i-1].first && c > 0)
            result += x[i].first - x[i-1].first;
        if (x[i].second) c--;
        else c++;
    }
    return result;
}
```

V. Oriented area of a triangle: Main Idea

- **Problem**: Given three points p_1, p_2 and p_3, calculate an oriented (signed) area of a triangle formed by them. The sign of the area is determined in the following way: imagine you are standing in the plane at point p_1 and are facing p_2. You go to p_2 and if p_3 is to your right (then we say the three vectors turn "clockwise"), the sign of the area is ***negative***, otherwise it is ***positive***. If the three points are collinear, the area is ***zero***.

V. Oriented area of a triangle: Main Idea

- **Problem**: Given three points p_1, p_2 and p_3, calculate an oriented (signed) area of a triangle formed by them. The sign of the area is determined in the following way: imagine you are standing in the plane at point p_1 and are facing p_2. You go to p_2 and if p_3 is to your right (then we say the three vectors turn "clockwise"), the sign of the area is ***negative***, otherwise it is ***positive***. If the three points are collinear, the area is ***zero***.

$$2S = \begin{vmatrix} x_2 - x_1 & x_3 - x_2 \\ y_2 - y_1 & y_3 - y_2 \end{vmatrix} = (x_2 - x_1)(y_3 - y_2) - (x_3 - x_2)(y_2 - y_1)$$

V. Oriented area of a triangle: Code

```
int signed_area_parallelogram(point2d p1,
                              point2d p2,
                              point2d p3) {
    return cross(p2-p1, p3-p2);
}

double triangle_area(point2d p1,
                     point2d p2,
                     point2d p3) {
    return abs(signed_area_parallelogram(p1,p2,p3))/2.0;
}
```

V. Oriented area of a triangle: Code

```
bool clockwise(point2d p1, point2d p2, point2d p3) {
    return signed_area_parallelogram(p1, p2, p3) < 0;
}

bool counter_clockwise(point2d p1,
                       point2d p2,
                       point2d p3) {
    return signed_area_parallelogram(p1, p2, p3) > 0;
}
```

VI. Area of simple polygon: Main Idea

- **Problem**: Let a simple polygon (i.e. without self intersection, not necessarily convex) be given. Calculate its area given its vertices.

VI. Area of simple polygon: Main Idea

- **Problem**: Let a simple polygon (i.e. without self intersection, not necessarily convex) be given. Calculate its area given its vertices.

$$A = \sum_{(p,q)\in \text{edges}} \frac{(p_x - q_x) \cdot (p_y + q_y)}{2}$$

VI. Area of simple polygon: Code

```
double area(const vector<point>& fig) {
    double res = 0;
    for (unsigned i = 0; i < fig.size(); i++) {
        point p = i ? fig[i - 1] : fig.back();
        point q = fig[i];
        res += (p.x - q.x) * (p.y + q.y);
    }
    return fabs(res) / 2;
}
```

VII. Pick's Theorem: Main Idea

- **Problem**: A polygon without self-intersections is called *lattice* if all its vertices have integer coordinates in some 2D grid. Compute the area of this polygon using the number of vertices that are lying on the boundary and the number of vertices that lie strictly inside the polygon.

VII. Pick's Theorem: Main Idea

- **S**: Area;
  I: the number of points with integer coordinates lying strictly inside the polygon;
  B: the number of points lying on polygon sides;
- Proof is carried out in many stages: from simple polygons to arbitrary ones.

$$S = I + \frac{B}{2} - 1$$

Lecture Exercises

Lecture Exercise #1

- https://www.spoj.com/problems/CIRCINT/

Solution

- https://cp-algorithms.com/geometry/circle-circle-intersection.html

Lecture Exercise #2

- https://open.kattis.com/problems/segmentintersection

Solution

- https://cp-algorithms.com/geometry/check-segments-intersection.html

Lecture Exercise #3

- https://open.kattis.com/problems/polygonarea

Solution

- https://cp-algorithms.com/geometry/area-of-simple-polygon.html

Lecture Exercise #4

- https://open.kattis.com/problems/maxcolinear

Solution

- https://github.com/jowilf/Kattis/blob/master/maxcolinear.py

Lecture Exercise #5

- https://open.kattis.com/problems/birthdaycake

Solution

- #1: https://usaco.guide/problems/kattis-birthday-cake/solution
- #2: https://github.com/mpfeifer1/Kattis/blob/master/birthday.cpp

Further Topics

- Finding the nearest pair of points
- Delaunay triangulation and Voronoi diagram
- Vertical decomposition
- Half-plane intersection: Sort-and-Incremental (S&I) in O(N log N)
- Minkowski sum of convex polygons
- Delaunay Triangulations
- 3D geometry

Further Readings

- CP Algorithms: https://cp-algorithms.com/geometry/basic-geometry.html
- USACO Guide: https://usaco.guide/plat/geo-pri?lang=cpp

- *Computational Geometry*, Mark Overmars, et. al.
- *Computational Geometry in C*, Joseph O'Rourke
- *Discrete and Computational Geometry*, Joseph O'Rourke, et. al.

A Terse Guide on ICPC Contest Strategies

- Please take a look at:
    - A Terse Guide on ICPC Contest Strategies for Columbia team.
    - In addition, we have Google Drive to Terse Guides, of course!

- These documents will be frequently expanded upon later.

Reminder! Discord Servers

- Join the following discord servers, if you have not already!!!

  [ICPC CodeForces Zealots] https://discord.gg/7bvMnMyF6G

Reminder! Practice makes PERFECT!

- Do as **many practice contests** as you can!
    - **Live Contests**
        - CodeForces: Division 1-4
        - AtCoder: Beginner; Regular; Grand;
        - LeetCode: Weekly/Biweekly
    - **ICPC North America Practice Contests** on:
        - **Sundays** from 1pm ET to 6pm ET
    - **Zealot Problem Sets**
        - **Everyday** (24 hours 7 days a week)!

LUNCH BREAK

THANK YOU