Introduction to Algorithms

Science Honors Program (SHP)

Session 10 (Last Session! 🙁😎)

Christian Lim

Saturday, May 4, 2024

3rd Columbia University Local Contest (CULC): Result!

1. 6th (7 / 1299): **sharmaabdd (Abdd Sharma)**
2. 17th (3 / 624): **progrqmmer (Eugene Hwang)**

- **Congratulations!** We will hold 4th CULC in Fall 2024. Be sure to check it out then! 🙂

- I have a **special award** to give out whenever we meet in-person!

Overview

- **Intersection Point of Lines**
- **Check if two segments intersect**
- **Length of the union of segments**
- Break #1
- **Oriented area of triangle**
- **Area of simple polygon**
- **Pick's Theorem**
- Break #2
- **Convex Hull Construction**
- **Convex Hull Trick and Li Chao Tree**

I. Intersection Point of Lines: Main Idea

$$\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases}$$

I. Intersection Point of Lines: Main Idea

$$\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases}$$

Cramer's Rule

$$x = -\frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{c_1b_2 - c_2b_1}{a_1b_2 - a_2b_1}, \qquad y = -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}.$$

I. Intersection Point of Lines: Main Idea

$$\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1 b_2 - a_2 b_1 = 0$$

either: *no* solutions or *infinitely many* solutions.

If the following are both equal 0, the lines overlap:

$$\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}, \begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}$$

I. Intersection Point of Lines: Code

```
struct pt {
    double x, y;
};
struct line {
    double a, b, c;
};
const double EPS = 1e-9;
double det(double a, double b, double c, double d) {
    return a*d - b*c;
}
```

I. Intersection Point of Lines: Code

```
bool intersect(line m, line n, pt & res) {
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS)
        return false;
    res.x = -det(m.c, m.b, n.c, n.b) / zn;
    res.y = -det(m.a, m.c, n.a, n.c) / zn;
    return true;
}
bool parallel(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS;
}
```

I. Intersection Point of Lines: Code

```
bool equivalent(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS
        && abs(det(m.a, m.c, n.a, n.c)) < EPS
        && abs(det(m.b, m.c, n.b, n.c)) < EPS;
}
```

II. Check if two segments intersect: Main Idea

- **Problem**: You are given two segments (a, b) and (c, d). Check if they intersect.

II. Check if two segments intersect: Main Idea

- **Problem**: You are given two segments (a, b) and (c, d). Check if they intersect.

- **Idea**:
  - Firstly, consider the case when the segments are part of the same line. In this case it is sufficient to check if their projections on Ox and Oy intersect.

II. Check if two segments intersect: Main Idea

- **Problem**: You are given two segments (a, b) and (c, d). Check if they intersect.

- **Idea**:
  - Firstly, consider the case when the segments are part of the same line. In this case it is sufficient to check if their projections on Ox and Oy intersect.
  - In the other case a and b must not lie on the same side of line (c, d), and c and d must not lie on the same side of line (a, b). This can be checked with cross products.

II. Check if two segments intersect: Code

```
struct pt {
    long long x, y;

    pt() {}
    pt(long long _x, long long _y) : x(_x), y(_y) {}
    pt operator-(const pt& p) const {
        return pt(x - p.x, y - p.y);
    }
    long long cross(const pt& p) const {
        return x * p.y - y * p.x;
    }
```

II. Check if two segments intersect: Code

```
    long long cross(const pt& a, const pt& b) const {
        return (a - *this).cross(b - *this);
    }
};

int sgn(const long long& x) {
    return x >= 0 ? x ? 1 : 0 : -1;
}
```

II. Check if two segments intersect: Code

```
bool inter1(long long a,
            long long b,
            long long c,
            long long d) {
    if (a > b)
        swap(a, b);
    if (c > d)
        swap(c, d);
    return max(a, c) <= min(b, d);
}
```

II. Check if two segments intersect: Code

```
bool check_inter(const pt& a, const pt& b,
                 const pt& c, const pt& d) {
    if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
        return inter1(a.x, b.x, c.x, d.x) &&
               inter1(a.y, b.y, c.y, d.y);
    return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&
           sgn(c.cross(d, a)) != sgn(c.cross(d, b));
}
```

III. 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.

III. 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.

- **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.

III. 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());
```

III. 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;
}
```


BREAK #1

IV. 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***.

IV. 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***.

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

IV. 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;
}
```

IV. 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;
}
```

V. 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.

V. 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.

- **Method 1**

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

V. 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;
}
```

V. 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.

- **Method 2**: Pick a point O arbitrarily, iterate over all edges adding the oriented area of the triangle formed by the edge and point O. The same code.

VI. 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.

VI. Pick's Theorem: Main Idea

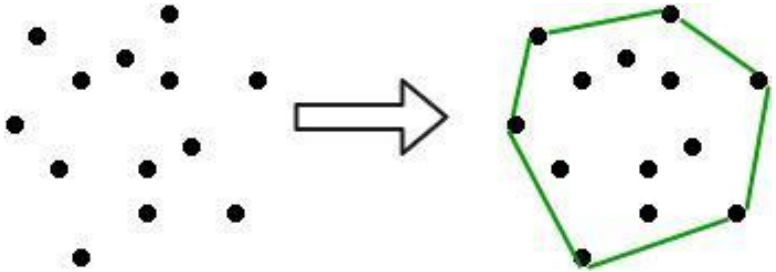
- **S**: Area;
  I: the number of points with integer coordinates lying strictly inside the polygon;
  B: the number of points lying on polygon sides;
- Proof is carried out in many stages: from simple polygons to arbitrary ones.

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


BREAK #2

VII. 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)



VII. 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.

VII. 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.**

VII. Graham's Scan Algorithm: Implementation

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

VII. 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.

VII. 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!

VII. 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.

VII. 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.

VII. 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.

VII. 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.**

VII. 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.

VII. Monotone Chain Algorithm: Implementation

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

VIII. 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 k^{th} 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 k^{th} city having coordinate x_k.
- Also you have to pay $toll_k$ to enter k^{th} city.
- Your task is to make the trip with minimum possible cost.

VIII. 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.

VIII. 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.

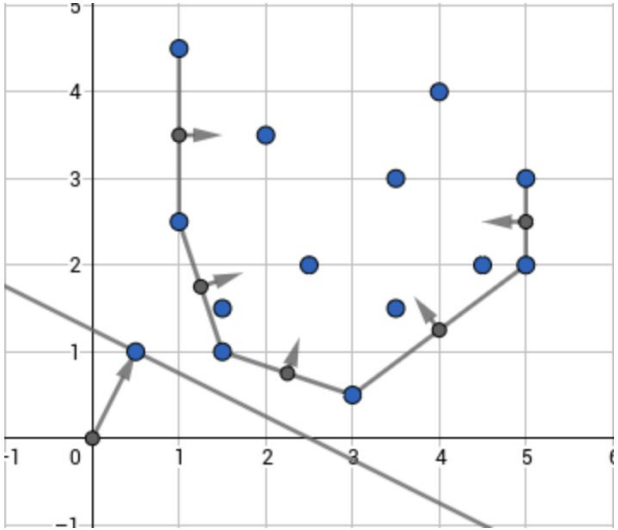
- There are two main approaches one can use here.
  - **Convex Hull Trick**
  - **Li Chao Tree**

VIII. 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.

VIII. Convex Hull Trick: Main Idea

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



VIII. 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.

VIII. Convex Hull Trick: Implementation

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

VIII. 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!

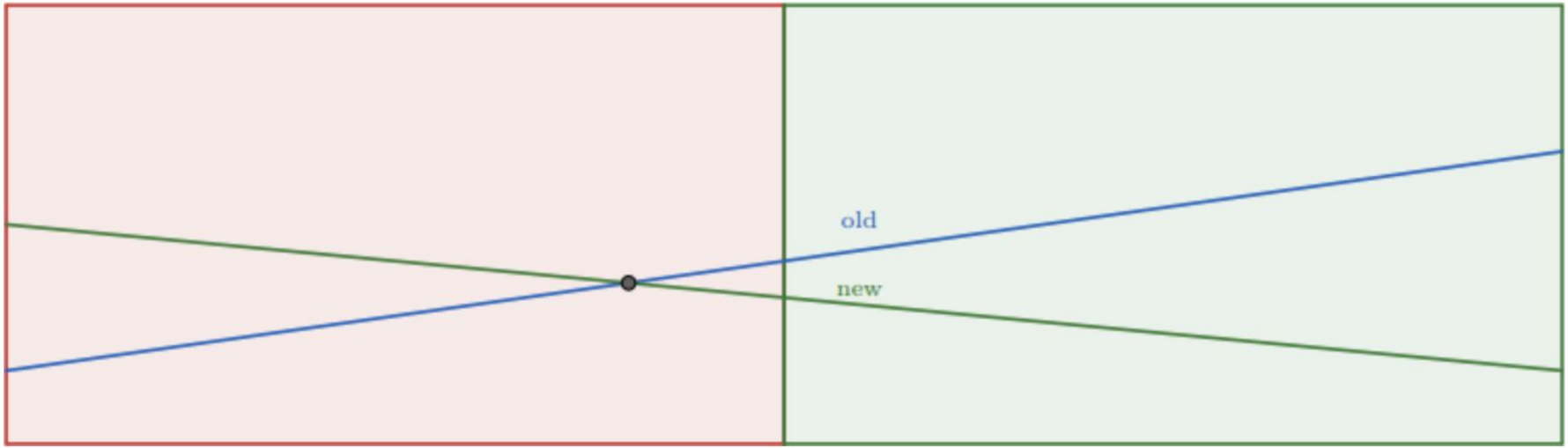
VIII. Li Chao Tree: Main Idea

- 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.

VIII. 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.

VIII. Li Chao Tree: Illustration



VIII. Li Chao Tree: Implementation

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

Programming Zealots @Discord

- Break into **CodeForces** rating of **2200+** as fast as you can!

- Join the discord server!

  bit.ly/programming-zealot

Programming Zealots @CodeForces

- Also, join CodeForces group!

bit.ly/cf-zealots

Review: Success Pathways

- **Programming Zealots @ CodeForces**
- **800 - 2100 (A - N)**
  - **For those who are just starting**
  - To gain some experiences with an explicit goal to enjoy the process of solving new problems;
  - To make it to **USACO Platinum**!

Review: Success Pathways

- **Programming Zealots @ CodeForces**
- **800 - 2100 (A - N)**
  - **For those who are just starting**
  - To gain some experiences with an explicit goal to enjoy the process of solving new problems;
  - To make it to **USACO Platinum**!
- **2200 - 3500 (O - ZB)**
  - **For those who are more serious**
  - To make it to **USACO Programming Camp** and/or **International Olympiad in Informatics**!

Review: Practice Strategy

- If your goal is to get to a rating of **X**, you should practice on problems that are **X + 300** typically, with a spread of 100. So, picking problems within the range of:

  {X + 200, X + 300, X + 400}

  would be sensible!

- So, if you want to target becoming a **red (grandmaster)**, which has a lower-bound of 2400, you should aim to solving {2600, 2700, 2800}.
- **(Eventual) Target**: You should focus on solving it for 30 minutes or less!

Review: Practice Strategy (con't)

- You should focus on solving each problem for **30 minutes or less**; if you cannot, you should consider solving a problem with a lower rating.
- You should aim to solve **~5 problems** each day within this range to expect a rank up within six months.

Review: Practice Strategy (con't)

- You should focus on solving each problem for **30 minutes or less**; if you cannot, you should consider solving a problem with a lower rating.
- You should aim to solve **~5 problems** each day within this range to expect a rank up within six months.
- If you cannot solve a problem, here is a sample recipe you can follow:
  - Look at editorial for **hints**, and try to solve the problem.
  - Look at editorial for **full solutions**, and try to solve the problem.
  - Look at **accepted code**, and try to solve the problem.
  - Make sure you **revisit after two weeks** and see if you can solve it.

Review: Programming Contests

- **CodeForces**
- **AtCoder**

- **Universal Cup**: https://ucup.ac/register

- **Quarterly Contests** from ICPC Curriculum Committee, starting **June 2024**

Review: Training Resources

- **U ICPC**: https://u.icpc.global/training/
- **CP Algorithms**: https://cp-algorithms.com/
- **USACO Guide**: https://usaco.guide/
- **Timus**: https://acm.timus.ru/problemset.aspx
- **Kattis**: https://open.kattis.com/
  - **Methods to Solve**: https://cpbook.net/methodstosolve?oj=kattis&topic=all&quality=all
- **CSES**: https://cses.fi/problemset/
- **solved.ac**: https://solved.ac/en
- **CodeForces**: https://codeforces.com/problemset
- **AtCoder**: https://kenkoooo.com/atcoder

Practice “Strategy”

- The **real keys** to success are:
  - “Upsolving” questions after each session.
  - If solutions are unclear,
    - **STUDY** the algorithms,
    - **IMPLEMENT** them, to make sure you know how to do that,
    - **CHECK** whether you retained them after few weeks,
    - **REPEAT** as many times as needed to learn the algorithms.
  - The **discussions** and **upsolvings** are most probably more important than the simulations! They let you train concepts you DO NOT KNOW!
  - Then, **rinse and repeat** with other problem sets!

2024 Columbia Programming Camp

- In **August 2024**, there will most likely be a programming camp.
- The exact date and contents are **TBD**.

- It will most likely be organized by **Benjamin Rubio**, **Josh Alman**, and **myself**.

- Let me know if you are interested in participating! You may email yongwhan.lim@columbia.edu.

Upcoming Contests

- In **Fall 2024**
  - World Finals (**WF**) @Astana, Kazakhstan;
  - 4th ICPC Columbia University Local Contest (**CULC**);
  - North America Qualifier (**NAQ**);
  - Greater New York Regional (**GNYR**) @Columbia;
- In **Spring 2025** and **Summer 2025**
  - North America Championship (**NAC**);
  - **WF** @?
- In **Fall 2025**
  - ? 1st ICPC High School Contest (**HSC**);

2024 Summer Practices

- **In-Person Practices**
  - **Sundays at 1pm ET**.
  - Alternate between Columbia only and Greater New York.
  - You are welcome to join the practices!

- **Problem Sets**
  - Weekday: **Mondays at 10am ET**;
  - Weekend: **Fridays at 6pm ET**;

Summer 2024 Coaching

- **Christian Lim** can help you getting better in programming contests throughout the summer and beyond via a small team-based coaching.

- So far, **~10** students signed up! Feel free to sign up for **free** sessions!

- https://bit.ly/christian-lim-coaching

Slide Decks

- I will make sure to put all the slide decks in my github: https://github.com/yongwhan/yongwhan.github.io/tree/master/columbia/shp

- Feel free to check it out in the coming week, to get slide decks from session 6 and after! 🥳

My (Personal) Summer "Grinding" Plan

- **AtCoder** (top 500 by solve count)
  - Complete all ABC problems! (~900 problems)
  - Get 50% of all ARC problems! (~250 problems)
- **CodeForces** (top 30 by solve count)
  - Solve all problems up to rating **2100** (~1500 problems)
- **CSES**: Complete it! (185 problems to go!)
- **Kattis** (top 20 by score): Get ~2500 more points (~700 problems)
- **Timus** & **solved.ac** ...
- **Total: Solve ~3500 problems** (~30 problems daily)

THANK YOU SO MUCH FOR SUCH A GREAT SEMESTER!!!

- Thank you for the fun-filled semester over the last **10** sessions!

- Special thanks to **Josh Alman**, **Eric Shao**, and **Innokentiy Kaurov** for covering the sessions while I was away during the semester!

THANK YOU SO MUCH FOR SUCH A GREAT SEMESTER!!!

- Working with you all has been a true **blessing**.

- When I started CP in 2010 and many friends (**Jeffrey**, **Tom**, etc.) helped me out greatly, I pledged to them one day I will pay forward.

- I request kindly that you would do the same one day! 🙂

THANK YOU SO MUCH FOR SUCH A GREAT SEMESTER!!!

- Please stay in touch. Feel free to use:
  - yongwhan.lim@columbia.edu;
  - yongwhan@yongwhan.io;

- Practice makes perfect!
  - I hope to interact with you in the future through continued coaching sessions, over the summer and beyond!
  - Feel free to sign up for the free coaching sessions over the summer!
  - I will send a quick meeting request shortly for those who signed up!


THANK YOU