

# Competitive Programming Week 8

---

Intro. to Computational Geometry



# Alberta Collegiate Programming Contest (ACPC):

This year ACPC will take place on November 25th  
We will be hosting a location here at U of C  
(MS 1st floor)  
There will be \$2550 in prizes across two divisions  
+ an open division  
(no registration required for open division)

Sign Up + More Info:

<https://www.eventbrite.ca/e/alberta-collegiate-programming-contest-2023-tickets-750029397117?aff=oddtcreator>



# Technical Interview Workshop (zoom)

Christian Yongwhan Lim from the ICPC Curriculum Committee will be hosting a workshop on technical interviews on zoom. He will be covering:

- Preparation for interview
- Must know topics
- Ways to impress interviewers

This is a unique opportunity to interact with an entrepreneur who is passionate about teaching, coaching, and mentoring students. Bring your most burning questions!



COMPETITIVE  
PROGRAMMING  
CLUB



*Hosted by Christian Yongwhan Lim  
From ICPC Curriculum Committee*

**Technical Interview  
Workshop** *on Zoom*

*November 10th, 2023 @4:00 PM*

*Meeting ID: 986 4427 0188  
Passcode: 149029*



# Week 7 Topics (cont.)

## Meet in the Middle:

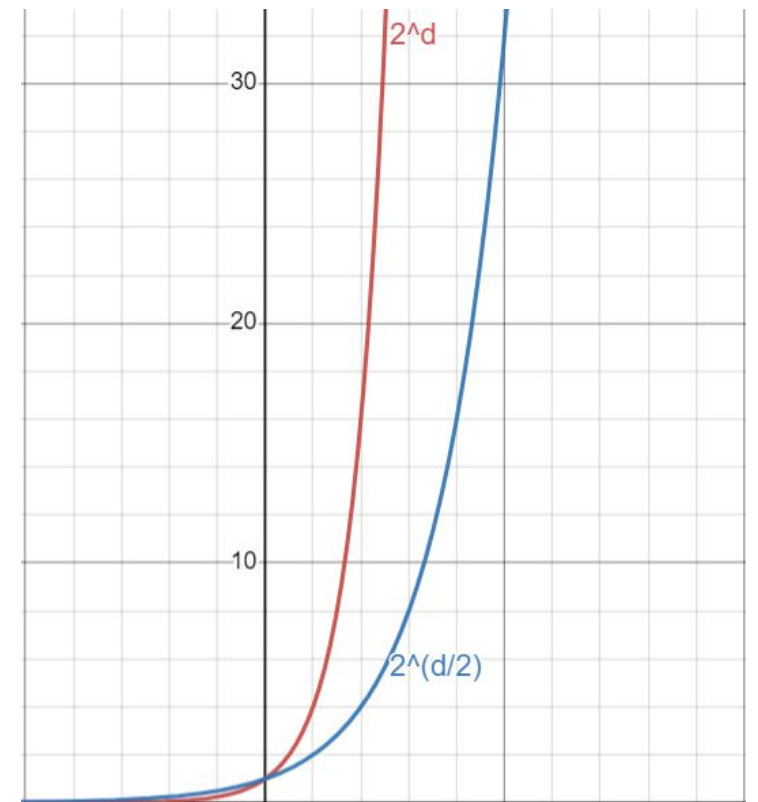
- The example in last week's slides may have been misleading. It did **not** depict an example where MitM was advantageous and was instead meant to demonstrate the end condition (i.e. when the frontwards and backwards searches find a common node)
- MitM offers improvement on graphs where the relationship between the depth from the start and the number of nodes with that depth is exponential. Say we have that the number of nodes  $d$  units from the starting node is  $\sim 2^d$ . This means that if the target node has depth  $d$  then there we might have to visit:

$$1 + 2 + 4 + \dots + 2^d = 2^{d+1} - 1 \approx 2^d$$

With MitM we have to do two searches with depth at most  $d/2$  thus the number of nodes we might visit is around:

$$2(1 + 2 + 4 + \dots + 2^{d/2}) = 2 \cdot (2^{d/2+1} - 1) \approx 2^{d/2+1}$$

which is **much** smaller than  $2^d$  for large values of  $d$



A decorative graphic consisting of a vertical blue line on the left side of the slide and a horizontal red bar extending from the left edge, intersecting the blue line.

# **Week 7 Solutions**

# Walkway

**Problem:** Given a set of trapezoidal stones, the length of a porch, and the length of a gazebo determine the path with the smallest area that connects the porch to the gazebo where each consecutive stone must share a side length.

**Solution:** use dijkstra's on the lengths of the stones connecting the lengths  $l_1$  and  $l_2$  if there is a stone with lengths  $l_1$  and  $l_2$  and weight of that edge being the area of the stone. Note even though the problem allows for duplicate stones it is never optimal to use the same stone twice.

(easy proof left to the reader: show that there is a better solution)

```
typedef struct {
    int a, b;
} Pair;

class Comp {
public:
    bool operator()(Pair a, Pair b) {
        return a.b > b.b;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    cin >> n;
    while (n) {
        vector<vector<Pair>> adj(1001);
        for (int i = 0; i < n; i++) {
            int a, b, h;
            cin >> a >> b >> h;
            int c = (a + b) * h;
            adj[a].push_back({b, c});
            adj[b].push_back({a, c});
        }

        int p, g;
        cin >> p >> g >> n;

        priority_queue<Pair, vector<Pair>, Comp> pq;
        unordered_set<int> seen;

        pq.push({p, 0});
        while (!pq.empty()) {
            Pair cur = pq.top();
            pq.pop();
            if (seen.find(cur.a) != seen.end()) continue;
            if (cur.a == g) {
                printf("%.2f\n", cur.b / 100.0);
                break;
            }
            seen.insert(cur.a);
            for (Pair stone: adj[cur.a])
                pq.push({stone.a, cur.b + stone.b});
        }

        n--;
    }

    return 0;
}
```



# Millionaire Madness

**Problem:** Given a  $n \times m$  grid  $G$  where  $G[i,j]$  is the height of a stack of coins at coordinates  $(i,j)$  determine the smallest ladder required to get from  $(0,0)$  to  $(n-1, m-1)$  moving in the cardinal directions

**Solution:** You can solve this by modifying an mst or sssp algorithm both achieving similar results. I used a modified Dijkstra's where the weight is the max of the current cost and the difference between the last stack and next stack.

```
class Comp {
public:
    bool operator()(Tuple t1, Tuple t2) {
        return t1.k > t2.k;
    }
};

unsigned long long encode(int i, int j) {
    return (static_cast<unsigned long long>(i) << 32) | static_cast<unsigned long long>(j);
}

int dirX[4] = {-1, 1, 0, 0};
int dirY[4] = {0, 0, 1, -1};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int n, m, h;
    cin >> m >> n;
    vector<vector<int>> grid(m);
    for (int i = 0; i < m; i++) {
        grid[i].reserve(n);
        for (int j = 0; j < n; j++) {
            cin >> grid[i][j];
        }
    }

    priority_queue<Tuple, vector<Tuple>, Comp> pq;
    unordered_set<long long> seen;

    pq.push({0, 0, 0});
    while (!pq.empty()) {
        Tuple c = pq.top();
        pq.pop();
        if (seen.find(encode(c.i, c.j)) != seen.end()) continue;
        if (c.i == m - 1 && c.j == n - 1) {
            cout << c.k << endl;
            return 0;
        }

        seen.insert(encode(c.i, c.j));

        for (int l = 0; l < 4; l++) {
            int i = c.i + dirX[l];
            int j = c.j + dirY[l];
            if (i >= 0 && i < m && j >= 0 && j < n) {
                int k = max(c.k, grid[i][j] - grid[c.i][c.j]);
                pq.push({i, j, k});
            }
        }
    }

    return 0;
}
```

# A Feast for Cats

**Problem:** Given a set of cats, the distances between them, and the amount of milk we have determine if it is possible to give at least 1mL of milk to each cat if we spill 1mL/m traveled. (we can also store milk at each cat)

**Solution:** First we will develop an optimal strategy. When we leave a cat  $c$ , bring the minimum amount of milk to feed the other cats we plan to feed before returning to  $c$ . Do this for all cats starting at cat 0. We can then show that this strategy matches a MST where the vertices are the cats and the weights are the distances between them. Pick your favourite MST algorithm\* and check if the cost + the number of cats is less than the amount of milk we have

\*except don't because for some reason Kruskal's isn't fast enough (at least my implementation wasn't 😡) so use Prim's

```
typedef struct {
    int i, j;
} Pair;

class comp {
public:
    bool operator()(Pair p, Pair q) {
        return p.j > q.j;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int n;
    cin >> n;
    while (n--) {
        int m, c;
        cin >> m >> c;
        if (m < c) {
            cout << "no" << endl;
            continue;
        }
        int k = c * (c - 1) >> 1;
        vector<vector<Pair>> adj(c);
        for (int l = 0; l < k; l++) {
            int i, j, w;
            cin >> i >> j >> w;
            adj[i].push_back({j, w});
            adj[j].push_back({i, w});
        }
        int cost = 0;
        unordered_set<int> seen;
        priority_queue<Pair, vector<Pair>, comp> pq;
        pq.push({0, 0});
        while (seen.size() != c && !pq.empty()) {
            int v = pq.top().i;
            int w = pq.top().j;
            pq.pop();
            if (seen.find(v) != seen.end()) continue;
            seen.insert(v);
            cost += w;
            for (Pair p : adj[v])
                pq.push(p);
        }
        cout << ((cost + c <= m) ? "yes" : "no") << endl;
    }
    return 0;
}
```





# Computational Geometry

Lots of info stolen from Dante Bencivenga's 599.4 slides (Thank you Dante)

# Floating Point vs Integer

Numbers are generally represented either in integer form or floating point form.

**Integer:** an  $n$ -bit integer perfectly stores all integers in the range  $[-2^{n-1}, 2^{n-1}]$

**Floating point:** given  $n$ -bits, distribute the bits into two integers  $m$  and  $e$ ; you can then represent fractional numbers as  $m \cdot 2^e$  (IEEE754 defines a standard for the distribution). You can represent much higher/smaller values than with an integer of the same size but you cannot represent more values. In exchange for the ability to represent fractional numbers and very large/small numbers, floating point numbers suffer in precision.

Whenever possible, use integers as integer arithmetic has no error (assuming no overflow) whereas floating point numbers have limited precision. In addition, when comparing floating point numbers, it's better to check if their difference is within some threshold ( $10^{-9}$  for example).

# Vectors and Points

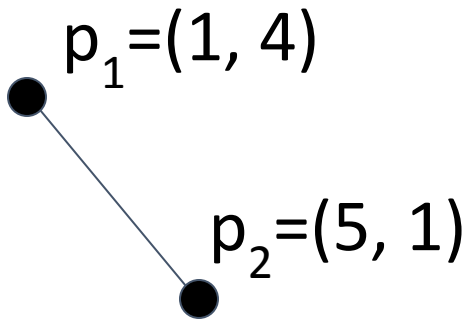
- Point:

Def: A location in some space (usually  $\mathbb{R}^n$ )

Usually represented as  $(x_1, x_2, \dots, x_n)$

Distance between points  $(x_1, x_2, \dots, x_n)$  and  $(y_1, y_2, \dots, y_n)$  is given by the “Euclidean Distance”  $d = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$

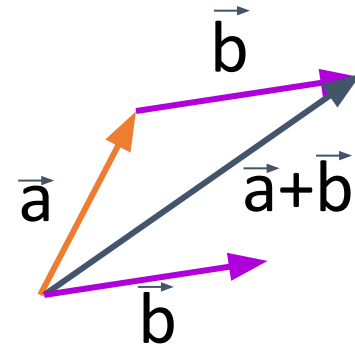
Note: The square root function is generally very slow so when comparing distances you don’t need to include it as the square root is monotone increasing.



$p_1 = (1, 4)$   
 $p_2 = (5, 1)$

$$d = \sqrt{(1-5)^2 + (4-1)^2} = \sqrt{(-4)^2 + (3)^2} = \sqrt{25} = 5$$

# Vectors and Points



- Vector:

Def: A quantity with magnitude and direction

Vectors are usually thought of as arrows rooted at the origin and ending at some point in the ambient space

Vectors in the same ambient space add by taking one vector moving its tail to the tip of the other and taking the point its tip is at.

Vectors scale/multiply by a *scalar* (regular number) by scaling the magnitude while maintaining the direction

All vectors in  $\mathbb{R}^n$  can be written as the linear combination of some *basis vectors*. The most common bases are the axis aligned unit vectors (vectors with magnitude 1)  $\vec{e}_1, \vec{e}_2, \vec{e}_3, \dots, \vec{e}_n$ . We then write that if

$\vec{v} = a_1 \vec{e}_1 + a_2 \vec{e}_2 + a_3 \vec{e}_3 + \dots + a_n \vec{e}_n$  as  $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix}$ . Vectors in this form add and scale componentwise

# Vector Products

Vectors don't have a natural sense of multiplication\* instead we define a couple different *products* for vectors. The main two are the *dot/inner product* and the *cross product* (there are lots of other products).

**Dot product:** Measures how similar two vectors are (projection)

**Cross product:** Measures the area of the parallelogram defined by the origin, the tip of both vectors and the tip of the sum of the vectors.

\*they actually kind of do (see geometric product) but let's not talk about that

# The Dot Product

Written  $a \cdot b = |a| |b| \cos(\theta)$  where  $|v|$  is the magnitude of  $v$  (also the euclidean distance from the origin) and  $\theta$  is the angle between the vectors\*. It is also given by the sum of the product of the components  $\sum a_i b_i$  and this is how we will compute the dot product and is done in  $\theta(n)$  where  $n$  is the dimension of the ambient space. The dot product is **distributive**, **commutative**, and **associative**

The sign of the dot product tells us:

- +: The vectors are somewhat in the same direction
- 0: The vectors are perpendicular
- : One vector points somewhat opposite to the other

The projection of a vector  $v$  onto a vector  $u$  (written:  $\text{proj}_u(v)$ ) is the vector component of  $v$  parallel to  $u$  and is given by  $u[(v \cdot u)/(u \cdot u)]$ . I.e. If you break  $v$  into the sum of 2 vectors: 1 parallel (the projection) and 1 perpendicular (the rejection)

\*most languages come with a function like  $\text{atan2}(x,y)$  that take the  $x$  and  $y$  components of a 2D vector and give the angle of the vector



# The Cross Product

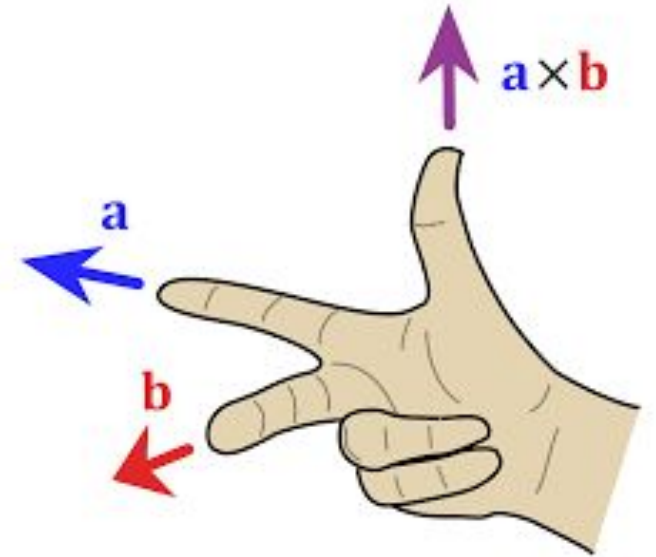
Written  $a \times b$  the cross product in  $\mathbb{R}^2$  is the area of the parallelogram described previously and is given by

$$|a| |b| \sin(\theta) = a_1 b_2 - a_2 b_1$$

In  $\mathbb{R}^3$  the cross product is given by follows the right hand rule

$$\begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} \text{ and}$$

The cross product is **associative** and **distributive** like the dot product but is **anticommutative**. The magnitude of the cross product in both cases is the same but in  $\mathbb{R}^3$  the result is a vector perpendicular to both  $a$  and  $b$  where as the result in  $\mathbb{R}^2$  is a scalar.



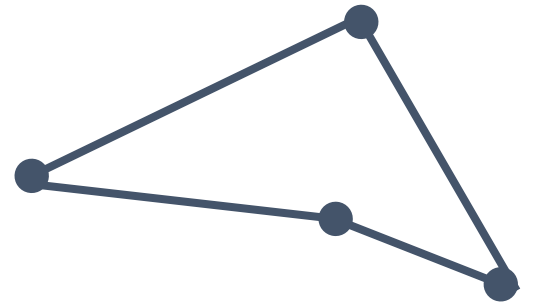
# Polygons

Polygons have many definitions but everyone should at least have a geometric interpretation in mind.

For us a polygon is a set of points usually given as an ordered list wherein there is a line segment between neighbouring points in the list and between the first and last points. Polygons have a well defined inside and outside as well. We will say a point is in the polygon if it is inside it.

Convexity:

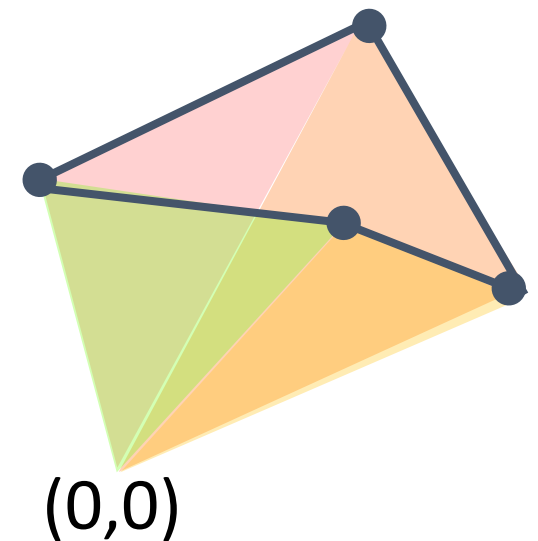
Def: A polygon (set in general) is convex if for every pair of points in it every point on the line segment connecting them is also in it. This also has other geometric interpretations. Say you are traveling along the border of the polygon counterclockwise then every turn you make will be right (if clockwise then left). We can use the signs of cross products of vectors between points to determine if the polygon is convex



# Polygons Properties

- The perimeter of a polygon is the sum of the euclidean distances between successive points
- The area is given by the signed areas of triangles consisting of the origin and successive points (see shoelace formula) which can be done with cross products (must be done in set order)
- Containment:
  - If a point is inside a polygon then if you record the angle you need to turn to reach each vertex of the polygon in order you will get  $360^\circ$
  - If a point is on the boundary you will get  $180^\circ$

Given a point use a series of dot products to figure out the angles you cover.

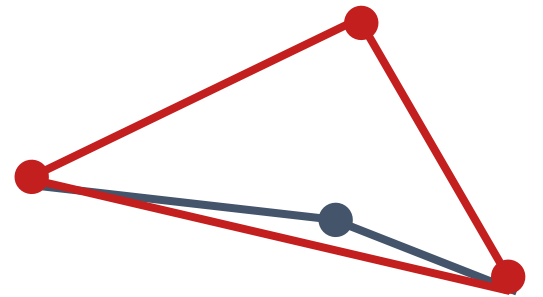


# Convex Hull

Given a polygon the convex hull is the smallest convex polygon that contains all points in the original polygon. This can be thought of as taking a rubber band and stretching it across all vertices of the original polygon and tracing the band. Some problems become much easier or even possible on convex geometries (ex. polygon collision, optimization problems *Math 325*)

Some algorithms for finding the convex hull:

- Graham's Scan
- Andrew's Monotone Chain
- etc.



<https://open.kattis.com/contests/b223wd>

(or look up “CPC Fall 2023 Practice Contest Week 8” in the Kattis contest list)

---

Feel free to ask questions until 7pm, and then throughout the week on Discord!



Register for ACPC!

