Computational Geometry

- Introduction to vector geometry
  - Dot & cross product
- Polar sort
- Convex hull
- Line segment intersection

References

Geometry handbook: https://victorlecomte.com/cp-geo.pdf

CP Algorithms: https://cp-algorithms.com/

Codeforces blog: https://codeforces.com/blog/entry/48122

Precision ⚠️

- Stay in the integers if at all possible
- If you must, defer floating point calculations as late as possible. Avoid compounding calculations.

When comparing floating point numbers, if two numbers are "close enough," we consider them equal. The threshold is typically called the epsilon (1e-6 is generally a good value).

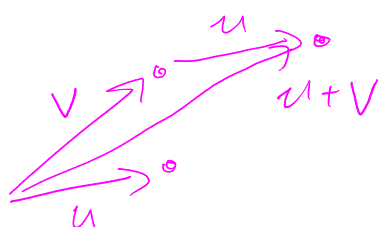| Integers	Floats	
`a < b`	~~`a - EPS < b`~~	$a < b - EPS$
`a <= b`	`a + EPS < b`	$a < b + EPS$
`a == 0`	`abs(a) < EPS`	
`a == b`	`abs(a - b) < EPS`	

Points as vectors

Vectors (not the STL vector) represent moving a fixed distance (magnitude) in a given direction. For comp geom, it's useful to think of a point as a vector. This lets us think about points more intuitively in terms of angle and orientation.
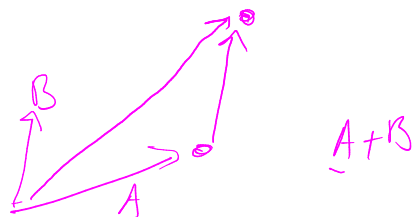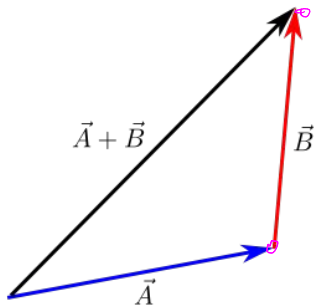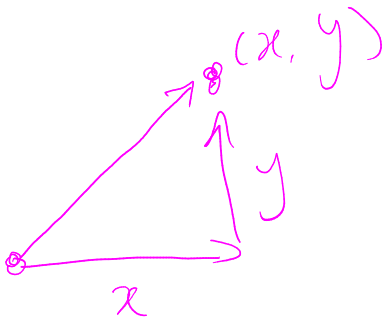
We'll be using the language of vectors to describe some primitives (useful reusable functions) that we'll build more complex algorithms from.

Point class
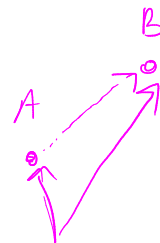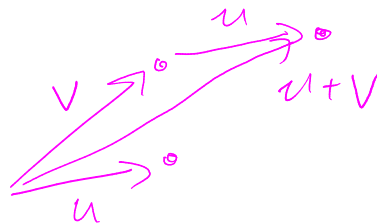
```cpp
struct point {
    long long x, y;
    point(): x(0), y(0) {}
    point(long long _x, long long _y) : x(_x), y(_y) {}

    point operator+(point other) { return {x + other.x, y + other.y}; }
    point operator-(point other) { return {x - other.x, y - other.y}; }
    bool operator==(point other) { return x == other.x && y == other.y; }

    // Use doubles and not long longs if you need these
    point operator*(double scale) { return {x*scale, y*scale}; }
    point operator/(double scale) { return {x/scale, y/scale}; }
};
```
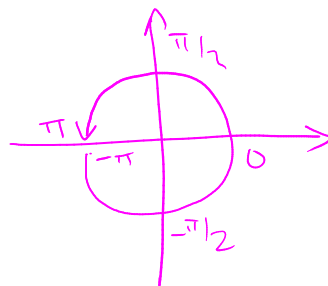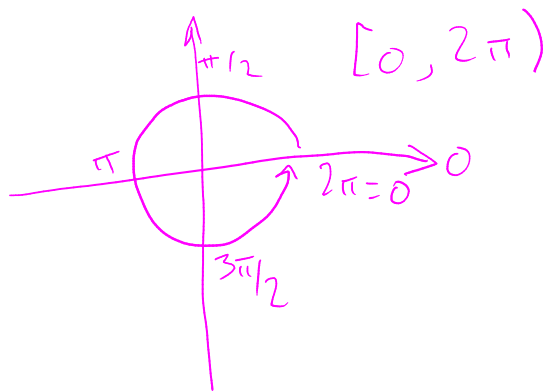
Angles refresher

- One full rotation = 2π
- Usual convention is to keep angles in the range (-π, π]. Counterclockwise is increasing angle.

⚠️ do not compute angles directly. Typically very slow and has precision issues.
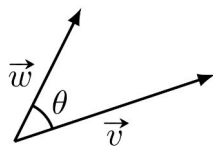
Dot product

The dot product $\vec{v} \cdot \vec{w}$ of two vectors $\vec{v}$ and $\vec{w}$ can be seen as a measure of how similar their directions are. It is defined as

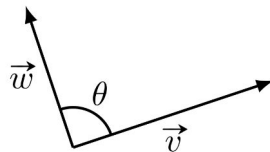$$\vec{v} \cdot \vec{w} = \|\vec{v}\|\|\vec{w}\| \cos\theta$$

where $\|\vec{v}\|$ and $\|\vec{w}\|$ are the lengths of the vectors and θ is amplitude of the angle between $\vec{v}$ and $\vec{w}$.

In general we will take θ in $[0, \pi]$, so that dot product is positive if $\theta < \pi/2$, negative if $\theta > \pi/2$, and zero if $\theta = \pi/2$, that is, if $\vec{v}$ and $\vec{w}$ are perpendicular.

$\theta < \pi/2$
$\vec{v} \cdot \vec{w} = 5$

$\theta = \pi/2$
$\vec{v} \cdot \vec{w} = 0$

$\theta > \pi/2$
$\vec{v} \cdot \vec{w} = -5$

Dot product

Most useful for telling if two vectors are perpendiculars or not.

```
long long dot (point a, point b) {
    return a.x*b.x + a.y*b.y;
}


long long perpendicular(point a, point b) {
    return dot(a, b) == 0;
}
```
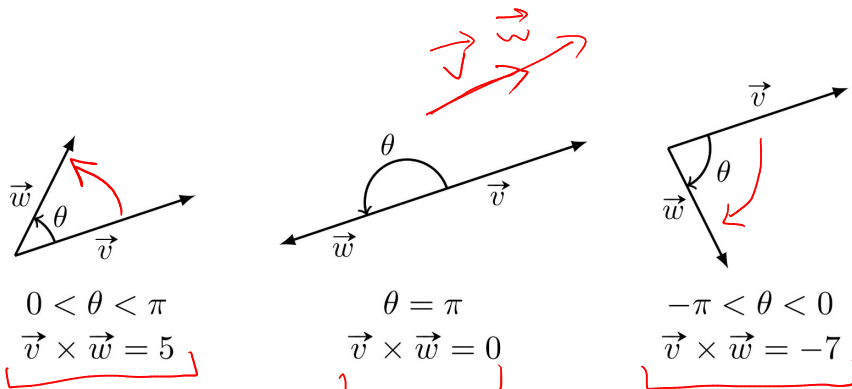
dot(a, b) is the same as dot(b, a)

Cross product

The cross product $\vec{v} \times \vec{w}$ of two vectors $\vec{v}$ and $\vec{w}$ can be seen as a measure of how perpendicular they are. It is defined in 2D as

$$\vec{v} \times \vec{w} = \|\vec{v}\|\|\vec{w}\| \sin \theta$$

where $\|\vec{v}\|$ and $\|\vec{w}\|$ are the lengths of the vectors and θ is amplitude of the oriented angle from $\vec{v}$ to $\vec{w}$.

$0 < \theta < \pi$
$\vec{v} \times \vec{w} = 5$

$\theta = \pi$
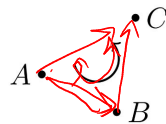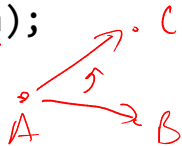$\vec{v} \times \vec{w} = 0$

$-\pi < \theta < 0$
$\vec{v} \times \vec{w} = -7$

Cross product

Most useful for telling which side of a line a point is on. Order is important! cross(a, b) == –cross(b, a)
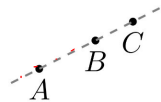
```
long long cross (point a, point b) {
    return a.x*b.x + a.y*b.y;
}
```
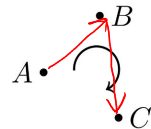
$$a.x \cdot b.y - a.y \cdot b.x;$$

```
long long ccw(point a, point b, point c) {
    return cross(b-a, c-a);
}
```


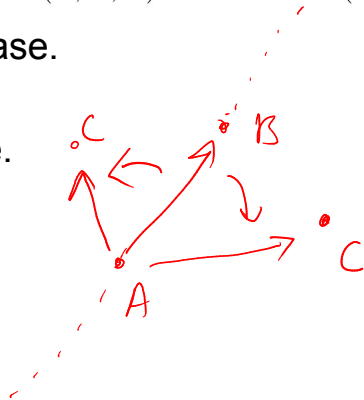left turn
orient$(A, B, C) > 0$

collinear
orient$(A, B, C) = 0$

right turn
orient$(A, B, C) < 0$

Useful for telling which side of a line a point is on. Watch out for the collinear case.

A lot of problems will say "no three points are collinear" to get around this issue.
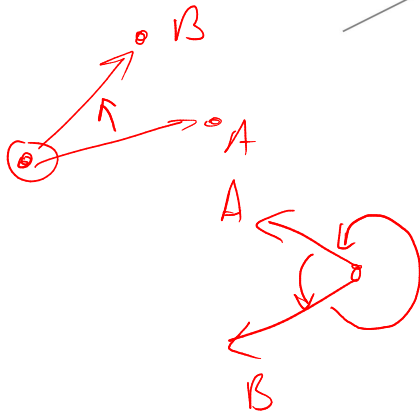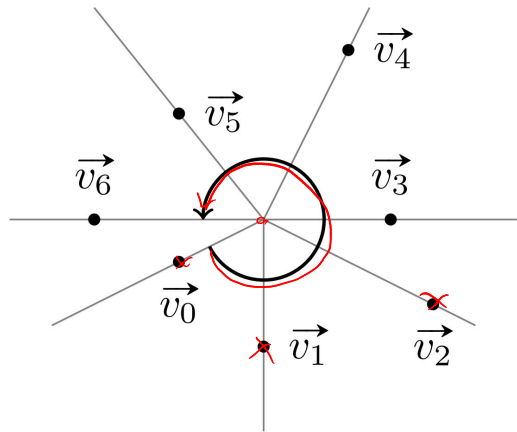**Order is important!** ccw(a, b, c) == –ccw(a, c, b).

Polar sort

Given **N** points, sort them by angle around the origin, from (-π, π].

Would something like this work?

```
void polarSort(vector<pt> &v) {
  sort(v.begin(), v.end(), [](point a, point b) {
    return cross(a, b) > 0;
  });
}
```
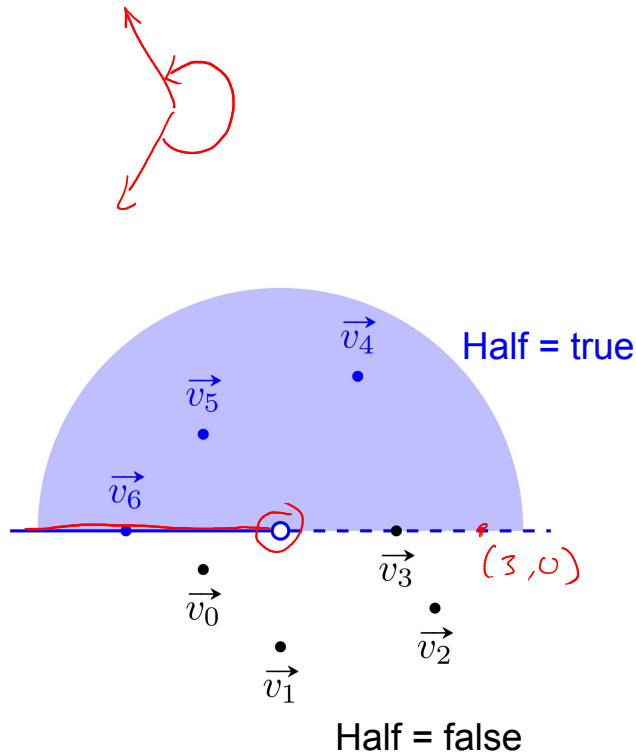
…no. But why?

Polar sort

```
bool half(point p) { // True if in blue half
  assert(p.x != 0 || p.y != 0);
  return p.y > 0 || (p.y == 0 && p.x < 0);
}

void polarSort(vector<point> &v) {
  sort(v.begin(), v.end(), [](point a, point b) {
    return make_pair(half(a), 0ll        ) <
           make_pair(half(b), cross(a, b));
  });
}
```


Half = true

$\vec{v_4}$

$\vec{v_5}$

$\vec{v_6}$

(3,0)

$\vec{v_3}$

$\vec{v_0}$

$\vec{v_2}$

$\vec{v_1}$

Half = false

Polar sort around a different origin point **o**

```
point o = …
void polarSort(vector<point> &v) {
  sort(v.begin(), v.end(), [&o](point _a, point _b) {
    point a = _a - o;
    point b = _b - o;
    return make_pair(half(a), 0ll        ) <
           make_pair(half(b), cross(a, b));
  });
}
```

Polar sort at a different angle

```
point v = …
bool half(point p) { // places v at 0 radians
  return cross(v, p) < 0 || (cross(v, p) == 0 && dot(v, p) < 0);
}
```

Convex Hull (Graham Scan)

Given **N** points where no three are collinear, what's the smallest convex polygon that contains every point?

- Convex means every angle ≤ π (i.e. 180 degrees)
- So pick the point with minimum y-coordinate (tie break by minimum x-coordinate); it definitely is on the convex hull
- Polar sort around that point (since the points on the convex hull appear in this order).
- Then iterate through to find the convex hull.
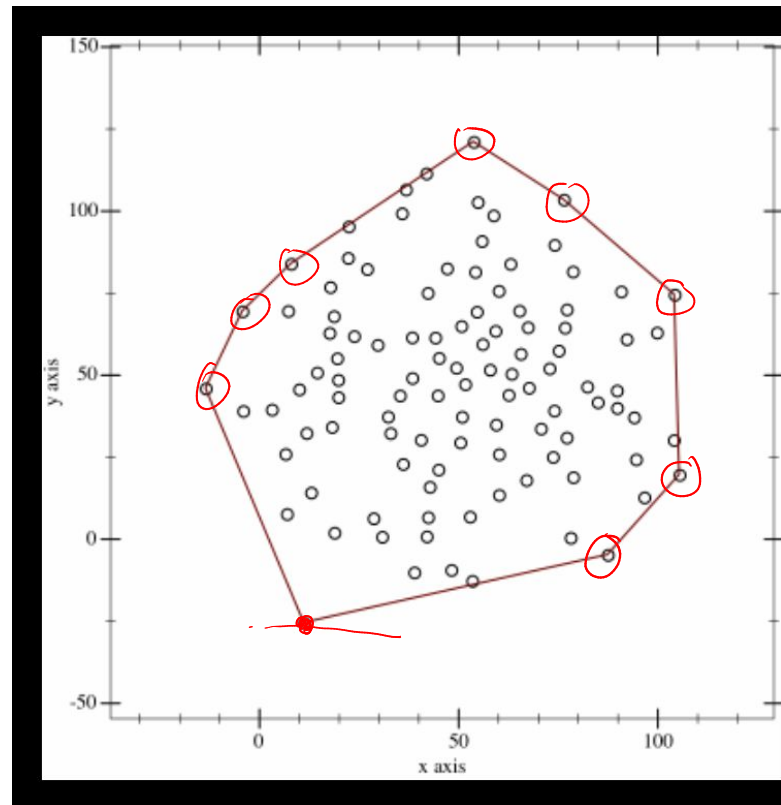
Convex Hull cont.

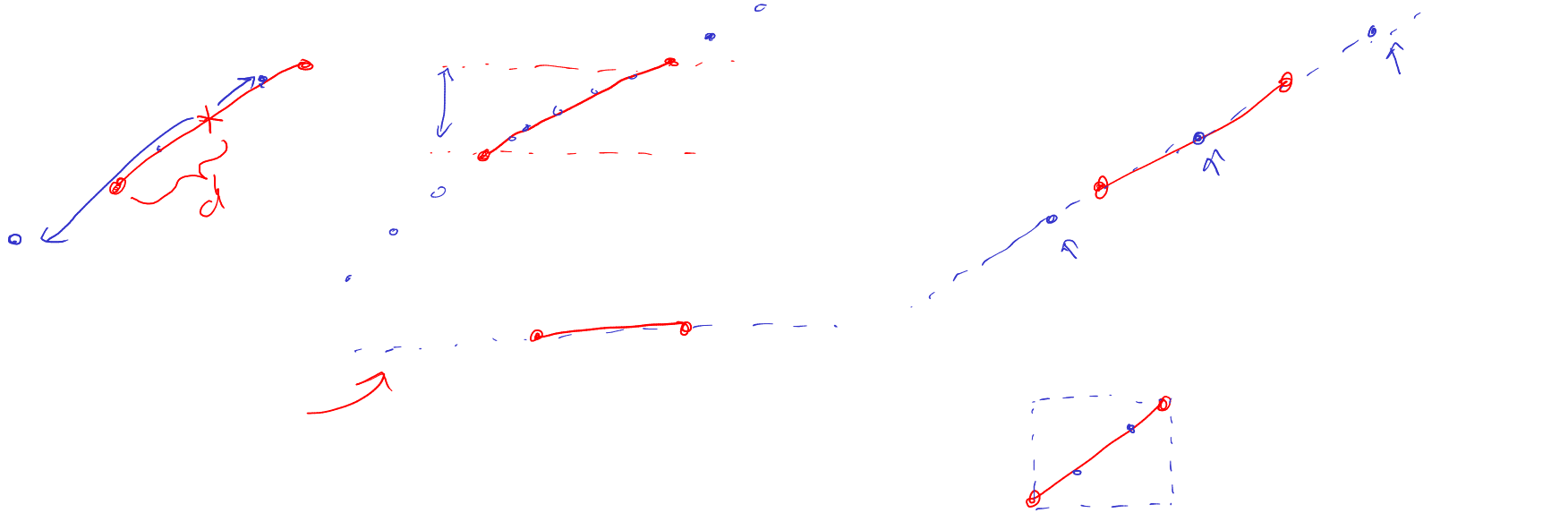Keep a stack that tracks the partial convex hull.

Iteration procedure: Always add the new point p to the stack, but before we do, delete things from the top of the stack if it would make us turn in the wrong direction.
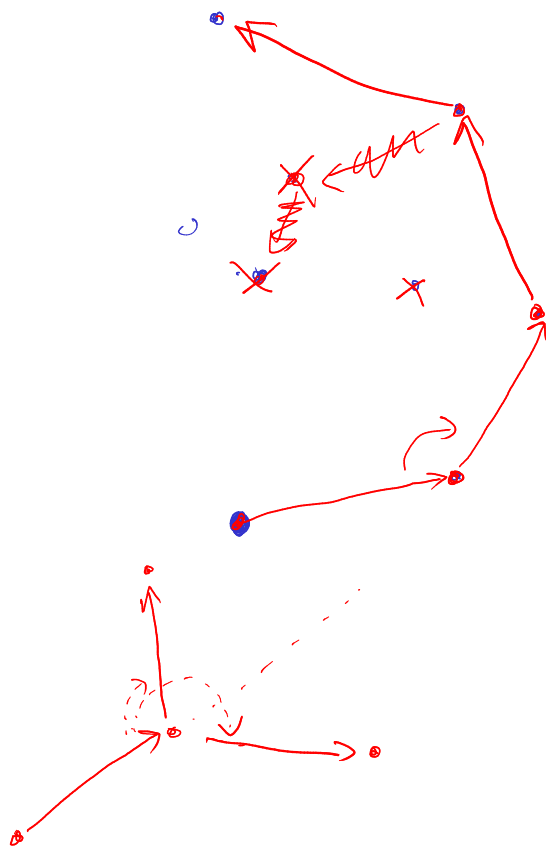
Point on line segment

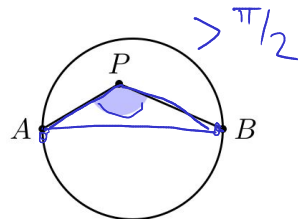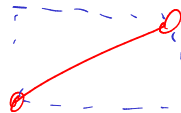How do you check if a point **p** is on the line segment with endpoints **a** and **b**?

Remember the primitives?

Point on line segment
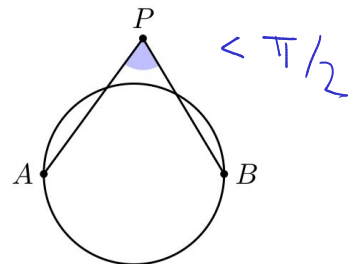
```
bool inDisk(point a, point b, point p) {
  return dot(a-p, b-p) <= 0;
}


bool onSegment(pt a, pt b, pt p) {
  return ccw(a,b,p) == 0 && inDisk(a,b,p);
}
```

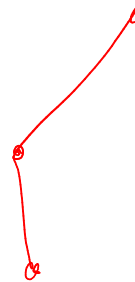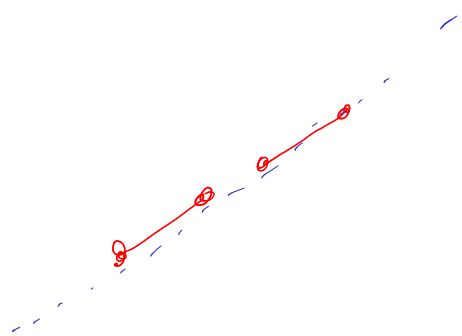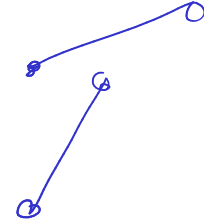$$\overrightarrow{PA} \cdot \overrightarrow{PB} \leq 0$$
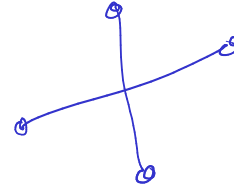
in disk

$$\overrightarrow{PA} \cdot \overrightarrow{PB} > 0$$

out of disk
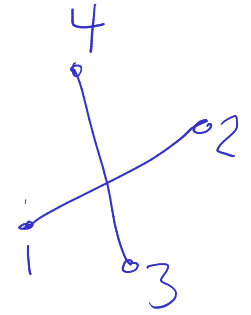
$> \frac{\pi}{2}$

$< \frac{\pi}{2}$

Line segment intersection

How do you check if two line segments **intersect**?
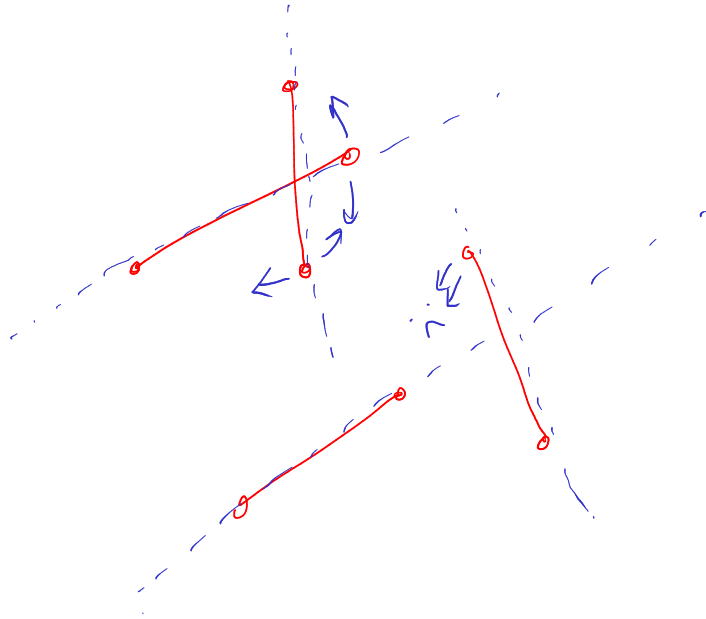
Approach 1: Linear algebra

$$y = mx + c$$

Sorry it's been a long time since I've touched this stuff (8 years?).

A line (not line segment) can be described as a linear equation. Checking intersection is just a matter of solving a system of linear equations. Once you've established the intersection point of the lines, you need to check that it is actually on the line.

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

See https://cp-algorithms.com/geometry/lines-intersection.html
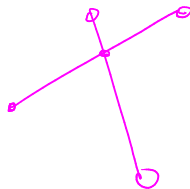
$$x = -\frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{c_1b_2 - c_2b_1}{a_1b_2 - a_2b_1},$$

$$y = -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}.$$
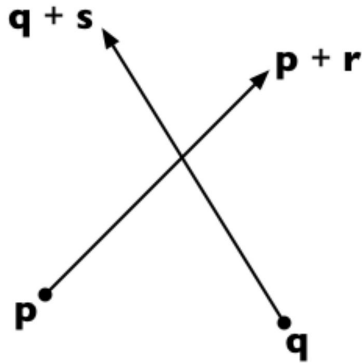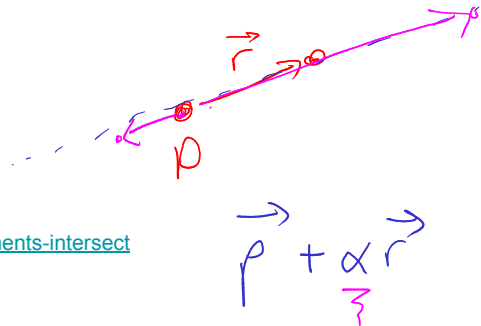
Approach 2: Vector geometry

A line can be described as an initial point and a vector describing the direction the line goes in. Each point on the line is some multiple of this vector. Again, set up and solve the equation.

Different perspective, but equivalent to approach 1.
Personally I follow this better.

See https://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect

See https://cp-algorithms.com/geometry/basic-geometry.html#line-intersection

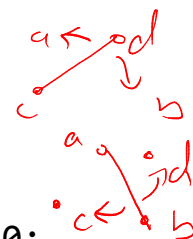And see https://codeforces.com/blog/entry/48122

Approach 3: Cross product check

Two line segments intersect if their endpoints are on opposite sides of the other line.

If you don't need the intersection point, and no three points collinear:

```
bool properInter(point a, point b, point c, point d) {
  double oa = ccw(c,d,a),
         ob = ccw(c,d,b),
         oc = ccw(a,b,c),
         od = ccw(a,b,d);
  return oa*ob < 0 && oc*od < 0;
}
```

⚠️ Watch out for overflow (multiplying up to 4 values). That's why doubles just in case.
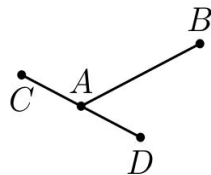
Approach 3: Cross product check cont.

If they can be collinear, then you have to add a check for intersections at the endpoints.

```
bool inters(point a, point b, point c, point d) {
  if (properInter(a,b,c,d)) return true;
  if (onSegment(c,d,a)) return true;
  if (onSegment(c,d,b)) return true;
  if (onSegment(a,b,c)) return true;
  if (onSegment(a,b,d)) return true;
  return false;
}
```
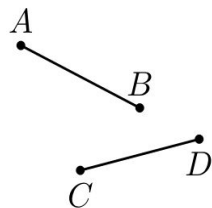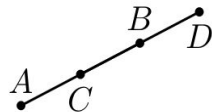
$S = \emptyset$

no intersection

$S = \{A\}$

intersection point

$S = \{B, C\}$

intersection segment

And if you need to calculate the intersection point:

- For a proper intersection: out = (a*ob - b*oa) / (ob-oa)
- For the collinear case, just try all the endpoints