• Homework 10 due Friday at 2pm
• Final Exam:
  • Friday, May 13 from 5:10pm to 7:00pm in WWH 101
  • Must bring a Laptop! (Email me asap if you do not have one!)
• Geometry problems are very common in the ICPC
  • Usually at least one per contest
• Strategy: often not attempted during the early part of the contest due to many pitfalls:
  • Tedious to code
  • Watch out for floating point errors
  • Watch out for edge cases:
    • Vertical lines
    • Parallel lines
    • Collinear points
    • Concave polygons
```java
class point implements Comparable<point> {
    double x, y; // only used if more precision is needed

    point() {
        x = y = 0.0;
    } // default constructor

    point(double _x, double _y) {
        x = _x;
        y = _y;
    } // user-defined

    // use EPS (1e-9) when testing equality of two floating points
    public int compareTo(point other) { // override less than operator
        if (Math.abs(x - other.x) > EPS) // useful for sorting
            return (int) Math.ceil(x - other.x); // first: by x-coordinate
        else if (Math.abs(y - other.y) > EPS)
            return (int) Math.ceil(y - other.y); // second: by y-coordinate
        else
            return 0; // they are equal
    }
};
```
double EuclideanDistance(point p1, point p2) {
    return Math.hypot(p1.x - p2.x, p1.y - p2.y);
}
• Recall equation for line: $y = mx + b$
• Better representation: $ax + by + c = 0$
• Note: this is for lines, not line segments!

```java
class line { double a, b, c; } // a way to represent a line

// the answer is stored in the third parameter
void pointsToLine(point p1, point p2, line l) {
    if (Math.abs(p1.x - p2.x) < EPS) {
        l.a = 1.0;  l.b = 0.0;  l.c = -p1.x;
    } else {
        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
        l.b = 1.0;  // IMPORTANT: we fix the value of b to 1.0
        l.c = -(double)(l.a * p1.x) - p1.y;
    }
}
```
boolean areParallel(line l1, line l2) {
    // check coefficients a & b
    return (Math.abs(l1.a - l2.a) < EPS) && (Math.abs(l1.b - l2.b) < EPS);
}

boolean areSame(line l1, line l2) {
    // also check coefficient c
    return areParallel(l1, l2) && (Math.abs(l1.c - l2.c) < EPS);
}
• If lines intersect and are not the same, they will only do so at one point

```java
boolean areIntersect(line l1, line l2, point p) {
    if (areParallel(l1, l2)) return false; // no intersection
    // solve system of 2 linear algebraic equations with 2 unknowns
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
    // special case: test for vertical line to avoid division by zero
    if (Math.abs(l1.b) > EPS)
        p.y = -(l1.a * p.x + l1.c);
    else
        p.y = -(l2.a * p.x + l2.c);
    return true;
}
```
• Line segments: lines with two endpoints (i.e., finite length)
• Vectors: line segment with a direction starting at (0, 0)

```java
class vec { double x, y; // name: `vec' is different from Java Vector
    vec(double _x, double _y) { x = _x; y = _y; }
}

vec toVec(point a, point b) {
    return new vec(b.x - a.x, b.y - a.y); // convert 2 points to vector
}

vec scale(vec v, double s) {
    return new vec(v.x * s, v.y * s); // nonnegative s = [<1 .. 1 .. >1]
}

point translate(point p, vec v) {
    return new point(p.x + v.x, p.y + v.y); // translate p according to v
}
```
Exercise: A knight is on the road from point A to point B. Suddenly he wants to get to point C. Which way does he turn (left, or right?) (Codeforces 227A)
• Find the cross product of AB and AC
• Magnitude is the area of the parallelogram spanned by AB and AC
  • If zero, then A, B, C are collinear
• Sign of the magnitude indicates which side of line AB the point C is on.

```c
double cross(vec a, vec b) {
    return a.x * b.y - a.y * b.x;
}
```
Recall properties of circles, triangles, quadrilaterals, etc.
Basic trig: line of sines, law of cosines
Heron’s formula:
  • Area of generalized triangle

\[ A = \sqrt{s(s-a)(s-b)(s-c)} \]

where \[ s = \frac{1}{2} (a+b+c) \]
A polygon is **convex** if any line segment draw inside the polygon does not intersect any edges of the polygon.

- Otherwise, it is **concave**.
• To check if a polygon is concave or convex, check if all three consecutive vertices in the polygon form the same turns
  • Recall that polygons are represented as an ordered (CW/CCW) list of points
• If one triple is found that is not consistent with the others, then the polygon is concave
Exercise: the textbook presents the following code for determining if a polygon is concave or convex. Identify two potential errors/ambiguities. The function $ccw$ returns true iff the three points make a counterclockwise turn.

```cpp
bool isConvex(const vector<point> &P) {
    int sz = (int)P.size(); // returns true if all three
    if (sz <= 3) return false; // consecutive vertices of P form the same turns
    bool isLeft = ccw(P[0], P[1], P[2]); // a point/sz=2 or a line/sz=3 is not convex
    for (int i = 1; i < sz-1; i++) // then compare with the others
        if (ccw(P[i], P[i+1], P[(i+2) % sz]) != isLeft)
            return false; // different sign -> this polygon is concave
    return true; // this polygon is convex
}
```
• How do you determine if a point is inside a polygon?
• Given a polygon defined by points $P_i$ and a candidate point $t$
• **Winding number algorithm**
• Consider all angles formed by consecutive polygon vertices and the point: $\{P_i, t, p_{i+1}\}$
• Sum up all angles subtended by each side of the polygon
  • Add up left turns
  • Subtract right turns
• If the final sum is $2\pi$, then the point is inside the polygon
Winding Number Algorithm
• Another algorithm: **ray casting**
  • Draw a line from the point in any fixed direction so that the line intersects edges of the polygon
    • If there are an odd number of intersections, the point is inside the polygon and outside otherwise
  • **Exercise:** what are the pros and cons of ray casting vs. winding number?
Given a convex polygon and a line, you can cut the polygon into two convex polygons.

Algorithm intuition (i.e., return one of the sub-polygons):
- Iterate through polygon vertices, identify if they are to the left or the right of the line.
- If a polygon edge intersects the line, then find the intersection and add it to the resultant polygon.
  - Ignore all subsequent points until we return to the starting side of the line.
Recall the definition of the **convex hull**: for a set of points $P$, the convex hull is the smallest set of points for which each point of $P$ is either on the boundary of the convex hull or is in its interior.
• Pick a **pivot** point: e.g., the bottom-most, right-most point
• Sort all other points based on their angles with respect to the pivot
• A stack is maintained
  • All points from the polygon will be pushed onto the stack once
  • All points that don’t end up on the convex hull will eventually be popped off
• The algorithm maintains that the top three elements of the stack always make a left turn
Graham’s Scan Example
Exercise: Given up to 1M points, find the closest pair.
• Idea: sweep a vertical line across the plane keeping track of:
  • Closest pair out of all points encountered and the distance \(d\) between them
  • All points within \(d\) units to the left of the vertical line in an ordered set \(D\), sorted by \(y\)-coordinate

• Every time the line hits a point \(p\):
  • Remove all points further than \(d\) units away from the line from \(D\)
  • Find the closest point to \(p\) in \(D\), update \(d\) if necessary
• Finding the closest point to $p$ in $D$:
  • Recall that $D$ is sorted by y-coordinates, so you can further restrict the search (the active region)
  • How does this step not make the algorithm $O(n^2)$?

• Note that the horizontal and vertical separation between any two points in $D$ is at least $d$, so the points in the active region has a constant upper bound
• Running time: $O(n\log n)$
Competitive Programming 7.1—7.5