**Notation Clarification**

**Since we deal with double a lot in geometry, we need to set EPS (double error bound) for safety when we convert double to int. Say d = 4.9999999, we want to cast it to 5: int, so we add EPS to d. Usually we set EPS to 1e-8 or 1e-9.**

**Points are denoted like p1, p2, in the input. Each point has field x and y representing x and y coordinates.**

**Lines are denoted like l1, l2. Each line has field a, b, c, meaning this line has formula ax + by = c.**

**toVec(point p1, point p2)** returns the vector from p1 to p2, in the form of [p2.x – p1.x, p2.y – p1.y]

**dot(vec v1, vec v2)** computes dot product of two vectors.

**Norm(vec v)** computes Euclidean norm of vec v.

**cross(vec a, vec b)** {return a[0] \* b[1] – a[1] \* b[0];}

(Equivalent to a.x \* b.y - a.y \* b.x; when a, b are points)

**Vec\_add(vec v1, vec v2)** performs vector element wise addition on v1 and v2.

**Checking if q, p, r makes a counter-clockwise (CCW) or clockwise turn (CW):** bool ccw(p, q, r) {return cross(toVec(p, q), toVec(p, r)) > EPS;}

**Checking if p, q, r are collinear:** bool collinear(p, q, r) {return abs(cross(toVec(p, q), toVec(p, r))) < EPS}

**List of Commonly Used Functions**

**Rotate coordinate (x, y) by angle alpha (in radian)**

Rotate(x, y, alpha) = [cos(alpha) \* x - sin(alpha) \* y, sin(alpha) \* x + cos(theta) \* y]

**line that passes through 2 points (returns l = {a, b, c} for line l in the form of ax + by = c):**

pointsToLine(point p1, point p2){

if(abs(p1.x - p2.x) < EPS (like 1e-9)){

l = {1.0, 0.0, -p1.x};

}

else{

l = {-(double)(p1.y - p2.y) / (p1.x - p2.x), 1.0, -(double)(p1.y - p2.y) / (p1.x - p2.x) \* p1.x - p1.y)};

}

return l;

}

**line that passes through 1 point with slope m:**

l = {-m, 1.0, -((-m \* p.x) + p.y)};

**intersectOfLines**(line l1, line l2){

if(l1 and l2 parallel)

return null;

x = (l2.b \* l1.c - l1.b \* l2.c) / (l2.a \* l1.b - l1.a \* l2.b);

if(abs(l1.b) > EPS)

y = -(l1.a \* x + l1.c);

else

y = -(l2.a \* x + l2.c);

return point(x, y);

}

**compute angle aob** (point a, point o, point b){

vec oa = toVec(o, a), ob = toVec(o, b);

return acos(dot(oa, ob) / sqrt(norm(oa) \* norm(ob)));

}

**double distToLine**(point p, point a, point b) (a, b defines a line, want to compute distance from p to ab){

vec ap = toVec(a, p), ab = toVec(a, b);

double u = dot(ap, ab) / norm(ab);

c = vec\_add(a, u \* ab);

return dist(p, c);

}

**double distToLineSegment**(point p, point a, point b)(a, b defines a line segment, want to compute distance from p to line segment ab){

vec ap = toVec(a, p), ab = toVec(a, b);

double u = dot(ap, ab) / norm(ab);

if(u < 0.0){

return dist(p, a);

}

else if(u > 1.0){

return dist(p, b);

}

else{

return distToLine(p, a, b);

}

}

**length of chord on a circle with central angle alpha and radius r:** sqrt(2 \* r \* r \* (1 - cos(alpha)))

**Given 2 points on a circle (p1 and p2) and radius, give two possible centers c1 and c2 of the circle**

{

double d2 = (p1.x - p2.x)^2 + (p1.y - p2.y)^2;

double det = r \* r / d2 - 0.25;

if(det < EPS)

return false;

double h = sqrt(det);

c1 = {(p1.x + p2.x) / 2 + (p1.y - p2.y) \* h, (p1.y + p2.y) / 2 + (p2.x - p1.x) \* h}.

c2 = (c1 but reverse p1 and p2).

}

**inscribed circle radius = Area / semi-perimeter**

**Area of triangle with 3 sides a, b, c:**

s = (a + b + c) / 2

A = sqrt(s \* (s - a) \* (s - b) \* (s - c))

**center of inscribed circle = meeting point of triangle's angle bisectors**

**Point inscribedCircleCenter**(p1, p2, p3) (coordinates of 3 vertices of triangle given as input){

r = radius of inCircle(p1, p2, p3); // computed from Area / semi-perimeter, see what’s immediately above

if(abs(r) < EPS) return false;

line l1, l2; //2 angle bisectors

double ratio = dist(p1, p2) / dist(p1, p3);

point p = vec\_add(p2, toVec(p2, p3) \* (ratio / (1 + ratio))));

l1 = pointsToLine(p1, p);

ratio = dist(p2, p1) / dist(p2, p3);

p = p1 + toVec(p1, p3) \* (ratio / (1 + ratio));

l2 = pointsToLine(p2, p);

return intersectOfLines(l1, l2);

}

**Circumscribed circle radius R = a \* b \* c / (4 \* Area of Triangle);**

**Center of circumscribed circle: meeting point of perpendicular bisectors (easy to compute, for perpendicular bisector of ab, the slope is 1/slope(ab), then compute the equation of the perpendicular bisector by the equation of a line with slope m (1 / slope(ab)) that passes through midpoint of ab ([(a.x + b.x) / 2, (a.y + b.y) / 2])**

**law of cosines, let gamma be angle C.**

We have c^2 = a^2 + b^2 - 2ab\*cos(gamma), gamma = acos((a^2 + b^2 - c^2) / (2ab));

**law of sines: alpha, beta, gamma = angle A, angle B, angle C, respectively:**

We have a / sin(alpha) = b / sin(beta) = c / sin(gamma) = 2R (R = radius of circumscribed circle)

**Area of Polygon:**

double area(list of point P){

double res = 0.0;

for(int i = 0; i < P.size() - 1; ++i){

res += (P[i].x \* P[i + 1].y - P[i + 1].x \* P[i].y);

}

return res / 2.0;

}

**Checking polygon is convex? Just do CCW on each angle to see if they are all CCW or all CW.**

**Checking if a point pt is in the polygon: sum up the angles made by {P[i], pt, P[i + 1]}. If the sum equals = 2\*pi, then yes, else, no.**

**int insidePolygon**(point pt, list of point P){

int n = P.size();

if(n <= 3)

return -1;

//check if pt on polygon

for(int i = 0; i < n - 1; ++i){

if(abs(dist(P[i], pt) + dist(pt, P[i + 1]) - dist(P[i], P[i + 1])) < EPS){

return 0;

}

}

double sum = 0.0;

for(int i = 0; i < n - 1; ++i){

if(ccw(pt, P[i], P[i + 1])){

sum += angle(P[i], pt, P[i + 1]);

}

else{

sum -= angle(P[i], pt, P[i + 1]);

}

}

return abs(sum) > 2 \* Math.PI ? 1 : -1;

}

**All intersections with a polygon Q and a line denoted by (point A, point B)**

**point lineIntersectSeg**(point p, point q, point A, point B){

double a = B.y - A.y, b = A.x - B.x, c = B.x \* A.y - A.x \* B.y;

double u = abs(a \* p.x + b \* p.y + c);

double v = abs(a \* q.x + b \* q.y + c);

return point((p.x \* v + q.x \* u) / (u + v), (p.y \* v + q.y \* u) / (u + v));

}

**list<point> cutPolygon**(point A, point B, list of point Q){

list<point> l = new ArrayList<>();

for(int i = 0; i < Q.size(); ++i){

double left1 = cross(toVec(A, B), toVec(A, Q[i])), left2 = 0.0;

if(i != Q.size() - 1)

left2 = cross(toVec(A, B), toVec(A, Q[i + 1]));

//Q[i] on the left

if(left1 > -EPS)

l.add(Q[i]);

//crossesline AB

if(left1 \* left2 < -EPS)

l.add(lineIntersectSeg(Q[i], Q[i + 1], A, B));

}

if((!l.isEmpty()) && !(l.back() == l.front())){

l.add(l.front());

}

return l;

}

**convex hull: credit Leetcode**

public class Solution {

public int orientation(int[] p, int[] q, int[] r) {

return (q[1] - p[1]) \* (r[0] - q[0]) - (q[0] - p[0]) \* (r[1] - q[1]);

}

public int[][] outerTrees(int[][] points) {

Arrays.sort(points, new Comparator<int[]> () {

public int compare(int[] p, int[] q) {

return q[0] - p[0] == 0 ? q[1] - p[1] : q[0] - p[0];

}

});

Stack<int[]> hull = new Stack<>();

for (int i = 0; i < points.length; i++) {

while (hull.size() >= 2 && orientation(hull.get(hull.size() - 2), hull.get(hull.size() - 1), points[i]) > 0)

hull.pop();

hull.push(points[i]);

}

hull.pop();

for (int i = points.length - 1; i >= 0; i--) {

while (hull.size() >= 2 && orientation(hull.get(hull.size() - 2), hull.get(hull.size() - 1), points[i]) > 0)

hull.pop();

hull.push(points[i]);

}

// remove redundant elements from the stack

HashSet<int[]> ret = new HashSet<>(hull);

return ret.toArray(new int[ret.size()][]);

}

}