#include <algorithm>

#include <cstdio>

#include <cmath>

#include <vector>

using namespace std;

#define INF 1e9

#define EPS 1e-9

#define PI acos(-1.0) // important constant; alternative #define PI (2.0 \* acos(0.0))

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

// struct point\_i { int x, y; }; // basic raw form, minimalist mode

struct point\_i { int x, y; // whenever possible, work with point\_i

point\_i() { x = y = 0; } // default constructor

point\_i(int \_x, int \_y) : x(\_x), y(\_y) {} }; // user-defined

struct 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

bool operator < (point other) const { // override less than operator

if (fabs(x - other.x) > EPS) // useful for sorting

return x < other.x; // first criteria , by x-coordinate

return y < other.y; } // second criteria, by y-coordinate

// use EPS (1e-9) when testing equality of two floating points

bool operator == (point other) const {

return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) { // Euclidean distance

// hypot(dx, dy) returns sqrt(dx \* dx + dy \* dy)

return hypot(p1.x - p2.x, p1.y - p2.y); } // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)

point rotate(point p, double theta) {

double rad = DEG\_to\_RAD(theta); // multiply theta with PI / 180.0

return point(p.x \* cos(rad) - p.y \* sin(rad),

p.x \* sin(rad) + p.y \* cos(rad)); }

struct line { double a, b, c; }; // a way to represent a line

// the answer is stored in the third parameter (pass by reference)

void pointsToLine(point p1, point p2, line &l) {

if (fabs(p1.x - p2.x) < EPS) { // vertical line is fine

l.a = 1.0; l.b = 0.0; l.c = -p1.x; // default values

} 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;

} }

// not needed since we will use the more robust form: ax + by + c = 0 (see above)

struct line2 { double m, c; }; // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {

if (abs(p1.x - p2.x) < EPS) { // special case: vertical line

l.m = INF; // l contains m = INF and c = x\_value

l.c = p1.x; // to denote vertical line x = x\_value

return 0; // we need this return variable to differentiate result

}

else {

l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);

l.c = p1.y - l.m \* p1.x;

return 1; // l contains m and c of the line equation y = mx + c

} }

bool areParallel(line l1, line l2) { // check coefficients a & b

return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) { // also check coefficient c

return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect

bool 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 (fabs(l1.b) > EPS) p.y = -(l1.a \* p.x + l1.c);

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

return true; }

struct vec { double x, y; // name: `vec' is different from STL vector

vec(double \_x, double \_y) : x(\_x), y(\_y) {} };

vec toVec(point a, point b) { // convert 2 points to vector a->b

return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) { // nonnegative s = [<1 .. 1 .. >1]

return vec(v.x \* s, v.y \* s); } // shorter.same.longer

point translate(point p, vec v) { // translate p according to v

return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line

void pointSlopeToLine(point p, double m, line &l) {

l.a = -m; // always -m

l.b = 1; // always 1

l.c = -((l.a \* p.x) + (l.b \* p.y)); } // compute this

void closestPoint(line l, point p, point &ans) {

line perpendicular; // perpendicular to l and pass through p

if (fabs(l.b) < EPS) { // special case 1: vertical line

ans.x = -(l.c); ans.y = p.y; return; }

if (fabs(l.a) < EPS) { // special case 2: horizontal line

ans.x = p.x; ans.y = -(l.c); return; }

pointSlopeToLine(p, 1 / l.a, perpendicular); // normal line

// intersect line l with this perpendicular line

// the intersection point is the closest point

areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line

void reflectionPoint(line l, point p, point &ans) {

point b;

closestPoint(l, p, b); // similar to distToLine

vec v = toVec(p, b); // create a vector

ans = translate(translate(p, v), v); } // translate p twice

double dot(vec a, vec b) { return (a.x \* b.x + a.y \* b.y); }

double norm\_sq(vec v) { return v.x \* v.x + v.y \* v.y; }

// returns the distance from p to the line defined by

// two points a and b (a and b must be different)

// the closest point is stored in the 4th parameter (byref)

double distToLine(point p, point a, point b, point &c) {

// formula: c = a + u \* ab

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

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

c = translate(a, scale(ab, u)); // translate a to c

return dist(p, c); } // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by

// two points a and b (still OK if a == b)

// the closest point is stored in the 4th parameter (byref)

double distToLineSegment(point p, point a, point b, point &c) {

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

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

if (u < 0.0) { c = point(a.x, a.y); // closer to a

return dist(p, a); } // Euclidean distance between p and a

if (u > 1.0) { c = point(b.x, b.y); // closer to b

return dist(p, b); } // Euclidean distance between p and b

return distToLine(p, a, b, c); } // run distToLine as above

double angle(point a, point o, point b) { // returns angle aob in rad

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

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

double cross(vec a, vec b) { return a.x \* b.y - a.y \* b.x; }

//// another variant

//int area2(point p, point q, point r) { // returns 'twice' the area of this triangle A-B-c

// return p.x \* q.y - p.y \* q.x +

// q.x \* r.y - q.y \* r.x +

// r.x \* p.y - r.y \* p.x;

//}

// note: to accept collinear points, we have to change the `> 0'

// returns true if point r is on the left side of line pq

bool ccw(point p, point q, point r) {

return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq

bool collinear(point p, point q, point r) {

return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

int main() {

point P1, P2, P3(0, 1); // note that both P1 and P2 are (0.00, 0.00)

printf("%d\n", P1 == P2); // true

printf("%d\n", P1 == P3); // false

vector<point> P;

P.push\_back(point(2, 2));

P.push\_back(point(4, 3));

P.push\_back(point(2, 4));

P.push\_back(point(6, 6));

P.push\_back(point(2, 6));

P.push\_back(point(6, 5));

// sorting points demo

sort(P.begin(), P.end());

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

printf("(%.2lf, %.2lf)\n", P[i].x, P[i].y);

// rearrange the points as shown in the diagram below

P.clear();

P.push\_back(point(2, 2));

P.push\_back(point(4, 3));

P.push\_back(point(2, 4));

P.push\_back(point(6, 6));

P.push\_back(point(2, 6));

P.push\_back(point(6, 5));

P.push\_back(point(8, 6));

/\*

// the positions of these 7 points (0-based indexing)

6 P4 P3 P6

5 P5

4 P2

3 P1

2 P0

1

0 1 2 3 4 5 6 7 8

\*/

double d = dist(P[0], P[5]);

printf("Euclidean distance between P[0] and P[5] = %.2lf\n", d); // should be 5.000

// line equations

line l1, l2, l3, l4;

pointsToLine(P[0], P[1], l1);

printf("%.2lf \* x + %.2lf \* y + %.2lf = 0.00\n", l1.a, l1.b, l1.c); // should be -0.50 \* x + 1.00 \* y - 1.00 = 0.00

pointsToLine(P[0], P[2], l2); // a vertical line, not a problem in "ax + by + c = 0" representation

printf("%.2lf \* x + %.2lf \* y + %.2lf = 0.00\n", l2.a, l2.b, l2.c); // should be 1.00 \* x + 0.00 \* y - 2.00 = 0.00

// parallel, same, and line intersection tests

pointsToLine(P[2], P[3], l3);

printf("l1 & l2 are parallel? %d\n", areParallel(l1, l2)); // no

printf("l1 & l3 are parallel? %d\n", areParallel(l1, l3)); // yes, l1 (P[0]-P[1]) and l3 (P[2]-P[3]) are parallel

pointsToLine(P[2], P[4], l4);

printf("l1 & l2 are the same? %d\n", areSame(l1, l2)); // no

printf("l2 & l4 are the same? %d\n", areSame(l2, l4)); // yes, l2 (P[0]-P[2]) and l4 (P[2]-P[4]) are the same line (note, they are two different line segments, but same line)

point p12;

bool res = areIntersect(l1, l2, p12); // yes, l1 (P[0]-P[1]) and l2 (P[0]-P[2]) are intersect at (2.0, 2.0)

printf("l1 & l2 are intersect? %d, at (%.2lf, %.2lf)\n", res, p12.x, p12.y);

// other distances

point ans;

d = distToLine(P[0], P[2], P[3], ans);

printf("Closest point from P[0] to line (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d);

closestPoint(l3, P[0], ans);

printf("Closest point from P[0] to line V2 (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, dist(P[0], ans));

d = distToLineSegment(P[0], P[2], P[3], ans);

printf("Closest point from P[0] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to A (or P[2]) = (2.00, 4.00)

d = distToLineSegment(P[1], P[2], P[3], ans);

printf("Closest point from P[1] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to midway between AB = (3.20, 4.60)

d = distToLineSegment(P[6], P[2], P[3], ans);

printf("Closest point from P[6] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to B (or P[3]) = (6.00, 6.00)

reflectionPoint(l4, P[1], ans);

printf("Reflection point from P[1] to line (P[2]-P[4]): (%.2lf, %.2lf)\n", ans.x, ans.y); // should be (0.00, 3.00)

printf("Angle P[0]-P[4]-P[3] = %.2lf\n", RAD\_to\_DEG(angle(P[0], P[4], P[3]))); // 90 degrees

printf("Angle P[0]-P[2]-P[1] = %.2lf\n", RAD\_to\_DEG(angle(P[0], P[2], P[1]))); // 63.43 degrees

printf("Angle P[4]-P[3]-P[6] = %.2lf\n", RAD\_to\_DEG(angle(P[4], P[3], P[6]))); // 180 degrees

printf("P[0], P[2], P[3] form A left turn? %d\n", ccw(P[0], P[2], P[3])); // no

printf("P[0], P[3], P[2] form A left turn? %d\n", ccw(P[0], P[3], P[2])); // yes

printf("P[0], P[2], P[3] are collinear? %d\n", collinear(P[0], P[2], P[3])); // no

printf("P[0], P[2], P[4] are collinear? %d\n", collinear(P[0], P[2], P[4])); // yes

point p(3, 7), q(11, 13), r(35, 30); // collinear if r(35, 31)

printf("r is on the %s of line p-r\n", ccw(p, q, r) ? "left" : "right"); // right

/\*

// the positions of these 6 points

E<-- 4

3 B D<--

2 A C

1

-4-3-2-1 0 1 2 3 4 5 6

-1

-2

F<-- -3

\*/

// translation

point A(2.0, 2.0);

point B(4.0, 3.0);

vec v = toVec(A, B); // imagine there is an arrow from A to B (see the diagram above)

point C(3.0, 2.0);

point D = translate(C, v); // D will be located in coordinate (3.0 + 2.0, 2.0 + 1.0) = (5.0, 3.0)

printf("D = (%.2lf, %.2lf)\n", D.x, D.y);

point E = translate(C, scale(v, 0.5)); // E will be located in coordinate (3.0 + 1/2 \* 2.0, 2.0 + 1/2 \* 1.0) = (4.0, 2.5)

printf("E = (%.2lf, %.2lf)\n", E.x, E.y);

// rotation

printf("B = (%.2lf, %.2lf)\n", B.x, B.y); // B = (4.0, 3.0)

point F = rotate(B, 90); // rotate B by 90 degrees COUNTER clockwise, F = (-3.0, 4.0)

printf("F = (%.2lf, %.2lf)\n", F.x, F.y);

point G = rotate(B, 180); // rotate B by 180 degrees COUNTER clockwise, G = (-4.0, -3.0)

printf("G = (%.2lf, %.2lf)\n", G.x, G.y);

return 0;

}