#include <cstdio>

#include <cmath>

using namespace std;

#define EPS 1e-9

#define PI acos(-1.0)

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; // whenever possible, work with point\_i

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

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

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) {} }; // constructor

double dist(point p1, point p2) {

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

double perimeter(double ab, double bc, double ca) {

return ab + bc + ca; }

double perimeter(point a, point b, point c) {

return dist(a, b) + dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {

// Heron's formula, split sqrt(a \* b) into sqrt(a) \* sqrt(b); in implementation

double s = 0.5 \* perimeter(ab, bc, ca);

return sqrt(s) \* sqrt(s - ab) \* sqrt(s - bc) \* sqrt(s - ca); }

double area(point a, point b, point c) {

return area(dist(a, b), dist(b, c), dist(c, a)); }

//====================================================================

// from ch7\_01\_points\_lines

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;

} }

bool areParallel(line l1, line l2) { // check coefficient a + b

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

//====================================================================

double rInCircle(double ab, double bc, double ca) {

return area(ab, bc, ca) / (0.5 \* perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {

return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written

// returns 1 if there is an inCircle center, returns 0 otherwise

// if this function returns 1, ctr will be the inCircle center

// and r is the same as rInCircle

int inCircle(point p1, point p2, point p3, point &ctr, double &r) {

r = rInCircle(p1, p2, p3);

if (fabs(r) < EPS) return 0; // no inCircle center

line l1, l2; // compute these two angle bisectors

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

point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));

pointsToLine(p1, p, l1);

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

p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));

pointsToLine(p2, p, l2);

areIntersect(l1, l2, ctr); // get their intersection point

return 1; }

double rCircumCircle(double ab, double bc, double ca) {

return ab \* bc \* ca / (4.0 \* area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {

return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written

// returns 1 if there is a circumCenter center, returns 0 otherwise

// if this function returns 1, ctr will be the circumCircle center

// and r is the same as rCircumCircle

int circumCircle(point p1, point p2, point p3, point &ctr, double &r){

double a = p2.x - p1.x, b = p2.y - p1.y;

double c = p3.x - p1.x, d = p3.y - p1.y;

double e = a \* (p1.x + p2.x) + b \* (p1.y + p2.y);

double f = c \* (p1.x + p3.x) + d \* (p1.y + p3.y);

double g = 2.0 \* (a \* (p3.y - p2.y) - b \* (p3.x - p2.x));

if (fabs(g) < EPS) return 0;

ctr.x = (d\*e - b\*f) / g;

ctr.y = (a\*f - c\*e) / g;

r = dist(p1, ctr); // r = distance from center to 1 of the 3 points

return 1; }

// returns true if point d is inside the circumCircle defined by a,b,c

int inCircumCircle(point a, point b, point c, point d) {

return (a.x - d.x) \* (b.y - d.y) \* ((c.x - d.x) \* (c.x - d.x) + (c.y - d.y) \* (c.y - d.y)) +

(a.y - d.y) \* ((b.x - d.x) \* (b.x - d.x) + (b.y - d.y) \* (b.y - d.y)) \* (c.x - d.x) +

((a.x - d.x) \* (a.x - d.x) + (a.y - d.y) \* (a.y - d.y)) \* (b.x - d.x) \* (c.y - d.y) -

((a.x - d.x) \* (a.x - d.x) + (a.y - d.y) \* (a.y - d.y)) \* (b.y - d.y) \* (c.x - d.x) -

(a.y - d.y) \* (b.x - d.x) \* ((c.x - d.x) \* (c.x - d.x) + (c.y - d.y) \* (c.y - d.y)) -

(a.x - d.x) \* ((b.x - d.x) \* (b.x - d.x) + (b.y - d.y) \* (b.y - d.y)) \* (c.y - d.y) > 0 ? 1 : 0;

}

bool canFormTriangle(double a, double b, double c) {

return (a + b > c) && (a + c > b) && (b + c > a); }

int main() {

double base = 4.0, h = 3.0;

double A = 0.5 \* base \* h;

printf("Area = %.2lf\n", A);

point a; // a right triangle

point b(4.0, 0.0);

point c(4.0, 3.0);

double p = perimeter(a, b, c);

double s = 0.5 \* p;

A = area(a, b, c);

printf("Area = %.2lf\n", A); // must be the same as above

double r = rInCircle(a, b, c);

printf("R1 (radius of incircle) = %.2lf\n", r); // 1.00

point ctr;

int res = inCircle(a, b, c, ctr, r);

printf("R1 (radius of incircle) = %.2lf\n", r); // same, 1.00

printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y); // (3.00, 1.00)

printf("R2 (radius of circumcircle) = %.2lf\n", rCircumCircle(a, b, c)); // 2.50

res = circumCircle(a, b, c, ctr, r);

printf("R2 (radius of circumcircle) = %.2lf\n", r); // same, 2.50

printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y); // (2.00, 1.50)

point d(2.0, 1.0); // inside triangle and circumCircle

printf("d inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, d));

point e(2.0, 3.9); // outside the triangle but inside circumCircle

printf("e inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, e));

point f(2.0, -1.1); // slightly outside

printf("f inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, f));

// Law of Cosines

double ab = dist(a, b);

double bc = dist(b, c);

double ca = dist(c, a);

double alpha = RAD\_to\_DEG(acos((ca \* ca + ab \* ab - bc \* bc) / (2.0 \* ca \* ab)));

printf("alpha = %.2lf\n", alpha);

double beta = RAD\_to\_DEG(acos((ab \* ab + bc \* bc - ca \* ca) / (2.0 \* ab \* bc)));

printf("beta = %.2lf\n", beta);

double gamma = RAD\_to\_DEG(acos((bc \* bc + ca \* ca - ab \* ab) / (2.0 \* bc \* ca)));

printf("gamma = %.2lf\n", gamma);

// Law of Sines

printf("%.2lf == %.2lf == %.2lf\n", bc / sin(DEG\_to\_RAD(alpha)), ca / sin(DEG\_to\_RAD(beta)), ab / sin(DEG\_to\_RAD(gamma)));

// Phytagorean Theorem

printf("%.2lf^2 == %.2lf^2 + %.2lf^2\n", ca, ab, bc);

// Triangle Inequality

printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 5, canFormTriangle(3, 4, 5)); // yes

printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 7, canFormTriangle(3, 4, 7)); // no, actually straight line

printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 8, canFormTriangle(3, 4, 8)); // no

return 0;

}