#include <algorithm>

#include <cstdio>

#include <cmath>

#include <stack>

#include <vector>

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 { 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 {

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

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); }

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

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

// returns the perimeter, which is the sum of Euclidian distances

// of consecutive line segments (polygon edges)

double perimeter(const vector<point> &P) {

double result = 0.0;

for (int i = 0; i < (int)P.size()-1; i++) // remember that P[0] = P[n-1]

result += dist(P[i], P[i+1]);

return result; }

// returns the area, which is half the determinant

double area(const vector<point> &P) {

double result = 0.0, x1, y1, x2, y2;

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

x1 = P[i].x; x2 = P[i+1].x;

y1 = P[i].y; y2 = P[i+1].y;

result += (x1 \* y2 - x2 \* y1);

}

return fabs(result) / 2.0; }

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; }

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; }

// 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; }

// returns true if we always make the same turn while examining

// all the edges of the polygon one by one

bool isConvex(const vector<point> &P) {

int sz = (int)P.size();

if (sz <= 3) return false; // a point/sz=2 or a line/sz=3 is not convex

bool isLeft = ccw(P[0], P[1], P[2]); // remember one result

for (int i = 1; i < sz-1; i++) // then compare with the others

if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)

return false; // different sign -> this polygon is concave

return true; } // this polygon is convex

// returns true if point p is in either convex/concave polygon P

bool inPolygon(point pt, const vector<point> &P) {

if ((int)P.size() == 0) return false;

double sum = 0; // assume the first vertex is equal to the last vertex

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

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

sum += angle(P[i], pt, P[i+1]); // left turn/ccw

else sum -= angle(P[i], pt, P[i+1]); } // right turn/cw

return fabs(fabs(sum) - 2\*PI) < EPS; }

// line segment p-q intersect with line A-B.

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

double a = B.y - A.y;

double b = A.x - B.x;

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

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

double v = fabs(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)); }

// cuts polygon Q along the line formed by point a -> point b

// (note: the last point must be the same as the first point)

vector<point> cutPolygon(point a, point b, const vector<point> &Q) {

vector<point> P;

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

double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;

if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));

if (left1 > -EPS) P.push\_back(Q[i]); // Q[i] is on the left of ab

if (left1 \* left2 < -EPS) // edge (Q[i], Q[i+1]) crosses line ab

P.push\_back(lineIntersectSeg(Q[i], Q[i+1], a, b));

}

if (!P.empty() && !(P.back() == P.front()))

P.push\_back(P.front()); // make P's first point = P's last point

return P; }

point pivot;

bool angleCmp(point a, point b) { // angle-sorting function

if (collinear(pivot, a, b)) // special case

return dist(pivot, a) < dist(pivot, b); // check which one is closer

double d1x = a.x - pivot.x, d1y = a.y - pivot.y;

double d2x = b.x - pivot.x, d2y = b.y - pivot.y;

return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; } // compare two angles

vector<point> CH(vector<point> P) { // the content of P may be reshuffled

int i, j, n = (int)P.size();

if (n <= 3) {

if (!(P[0] == P[n-1])) P.push\_back(P[0]); // safeguard from corner case

return P; // special case, the CH is P itself

}

// first, find P0 = point with lowest Y and if tie: rightmost X

int P0 = 0;

for (i = 1; i < n; i++)

if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))

P0 = i;

point temp = P[0]; P[0] = P[P0]; P[P0] = temp; // swap P[P0] with P[0]

// second, sort points by angle w.r.t. pivot P0

pivot = P[0]; // use this global variable as reference

sort(++P.begin(), P.end(), angleCmp); // we do not sort P[0]

// third, the ccw tests

vector<point> S;

S.push\_back(P[n-1]); S.push\_back(P[0]); S.push\_back(P[1]); // initial S

i = 2; // then, we check the rest

while (i < n) { // note: N must be >= 3 for this method to work

j = (int)S.size()-1;

if (ccw(S[j-1], S[j], P[i])) S.push\_back(P[i++]); // left turn, accept

else S.pop\_back(); } // or pop the top of S until we have a left turn

return S; } // return the result

int main() {

// 6 points, entered in counter clockwise order, 0-based indexing

vector<point> P;

P.push\_back(point(1, 1));

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

P.push\_back(point(9, 1));

P.push\_back(point(12, 4));

P.push\_back(point(9, 7));

P.push\_back(point(1, 7));

P.push\_back(P[0]); // loop back

printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // 31.64

printf("Area of polygon = %.2lf\n", area(P)); // 49.00

printf("Is convex = %d\n", isConvex(P)); // false (P1 is the culprit)

//// the positions of P6 and P7 w.r.t the polygon

//7 P5--------------P4

//6 | \

//5 | \

//4 | P7 P3

//3 | P1\_\_\_ /

//2 | / P6 \ \_\_\_ /

//1 P0 P2

//0 1 2 3 4 5 6 7 8 9 101112

point P6(3, 2); // outside this (concave) polygon

printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); // false

point P7(3, 4); // inside this (concave) polygon

printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); // true

// cutting the original polygon based on line P[2] -> P[4] (get the left side)

//7 P5--------------P4

//6 | | \

//5 | | \

//4 | | P3

//3 | P1\_\_\_ | /

//2 | / \ \_\_\_ | /

//1 P0 P2

//0 1 2 3 4 5 6 7 8 9 101112

// new polygon (notice the index are different now):

//7 P4--------------P3

//6 | |

//5 | |

//4 | |

//3 | P1\_\_\_ |

//2 | / \ \_\_\_ |

//1 P0 P2

//0 1 2 3 4 5 6 7 8 9

P = cutPolygon(P[2], P[4], P);

printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // smaller now 29.15

printf("Area of polygon = %.2lf\n", area(P)); // 40.00

// running convex hull of the resulting polygon (index changes again)

//7 P3--------------P2

//6 | |

//5 | |

//4 | P7 |

//3 | |

//2 | |

//1 P0--------------P1

//0 1 2 3 4 5 6 7 8 9

P = CH(P); // now this is a rectangle

printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // precisely 28.00

printf("Area of polygon = %.2lf\n", area(P)); // precisely 48.00

printf("Is convex = %d\n", isConvex(P)); // true

printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); // true

printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); // true

return 0;

}