ACM/ICPC: Competitive Programming Notebook

Lucas Mattioli, Marlon Mendes, Simião Carvalho

Contents

Points

Comparing floating point values

Returns true if double values a and b are equal

```
const double EPS { 1e-9 };
bool equals(double a, double b)
{
    return fabs(a - b) < EPS;
}
```

Listing 1: equals

Lines

General equation of a line

Non-normalized form: ax + by + c = 0

```
class Line {
public:
    double a;
    double b;
    double c;

    Line(double av, double bv, double cv) : a(av), b(bv), c(cv) {}

    Line(const Point& p, const Point& q)
    {
        a = p.y - q.y;
        b = q.x - p.x;
        c = p.x * q.y - p.y * q.x;
    }
};
```

Listing 2: General equation of a line

General equation of a line normalized

```
class Line {
public:
    double a;
    double b;
    double c;

    Line(double av, double bv, double cv) : a(av), b(bv), c(cv) {}

    Line(const Point& p, const Point& q)
    {
        a = p.y - q.y;
        b = q.x - p.x;
        c = p.x * q.y - p.y * q.x;

        auto k = a ? a : b;

        a /= k;
        b /= k;
```

```
        c /= k;
    }
};
```

Listing 3: General equation of a line

Point on a line

Is the given point located on the given Line?

```
template<typename T>
struct Line {
    bool contains(const Point<T>& P) const
    {
        return equals(a*P.x + b*P.y + c, 0);
    }
};
```

Listing 4: Point on line

Equal and parallel lines

```
template<typename T>
struct Line {
    bool operator==(const Line& r) const
    {
        auto k = a ? a : b;
        auto s = r.a ? r.a : r.b;

        return equals(a*s, r.a*k) && equals(b*s, r.b*k)
            && equals(c*s, r.c*k);
    }

    bool parallel(const Line& r) const
    {
        auto det = a*r.b - b*r.a;
        return det == 0 and !(*this == r);
    }
};
```

Orthogonal

```
template<typename T>
struct Line
{
    bool orthogonal(const Line& r) const
    {
        return equals(a * r.a + b * r.b, 0);
    }
};
```

Intersection

```
const int INF { -1 };
template<typename T>
std::pair<int, Point<T>> intersections(const Line<T>& r, const Line<T>& s)
```

```
{
    auto det = r.a * s.b - r.b * s.a;

    if (equals(det, 0))      // Coincidentes ou paralelas
    {
        int qtd = (r == s) ? INF : 0;
        return std::pair<int, Point<T>>(qtd, Point());
    } else                   // Concorrentes
    {
        auto x = (-r.c * s.b + s.c * r.b) / det;
        auto y = (-s.c * r.a + r.c * s.a) / det;

        return std::pair<int, Point<T>>(1, Point<T>(x, y));
    }
}
```

Angle between lines

```
template<typename T>
double angle(const Point<T>& P, const Point<T>& Q,
             const Point<T>& R, const Point<T>& S)
{
    auto ux = P.x - Q.x;
    auto uy = P.y - Q.y;

    auto vx = R.x - S.x;
    auto vy = R.y - S.y;

    auto num = ux * vx + uy * vy;
    auto den = hypot(ux, uy) * hypot(vx, vy);
    return acos(num / den);
}
```

Distance to point

```
#include <cmath>
#include <iostream>

template<typename T>
struct Point {
    T x, y;
};

template<typename T>
struct Line {
    T a, b, c;

    double distance(const Point<T>& p) const
    {
        return fabs(a*p.x + b*p.y + c)/hypot(a, b);
    }

    Point<T> closest(const Point<T>& p) const
    {
        auto den = (a*a + b*b);

        auto x = (b*(b*p.x - a*p.y) - a*c)/den;
        auto y = (a*(-b*p.x + a*p.y) - b*c)/den;
```

```cpp
        return Point<T> { x, y };
    }
};

int main()
{
    Point<double> P { 1.0, 4.0 };
    Line<double> r { 1.0, -1.0, 0 };

    std::cout << "Distance: " << r.distance(P) << '\n';

    auto Q = r.closest(P);

    std::cout << "Closest: Q = (" << Q.x << ", " << Q.y << ")\n";

    return 0;
}
```

Bissector / Mediatriz

```cpp
typename<template T>
Line<T> perpendicular_bisector(const Point<T>& P, const  Point<T>& Q)
{
    auto a = 2*(Q.x - P.x);
    auto b = 2*(Q.y - P.y);
    auto c = (P.x * P.x + P.y * P.y) - (Q.x * Q.x + Q.y * Q.y);

    return Line<T>(a, b, c);
}
```

Orientation between point and line

```cpp
typedef pair<long long, long long> ii;

// D = 0: R lies on line PQ
// D > 0: R is to the left of line PQ
// D < 0: R is to the right of line PQ
long long D(const ii &a, const ii &b, const ii &c) {
    return (a.first * b.second + a.second * c.first + b.first * c.second)
       - (c.first * b.second + c.second * a.first + b.first * a.second);
}
```

Line segments

Contains point

```cpp
template<typename T>
bool contains(const Point<T>& A, const Point<T>& B, const Point<T>& P)
{
    if (P == A || P == B)
        return true;

    auto xmin = min(A.x, B.x);
    auto xmax = max(A.x, B.x);
    auto ymin = min(A.y, B.y);
    auto ymax = max(A.y, B.y);
```

```

    if (P.x < xmin || P.x > xmax || P.y < ymin || P.y > ymax)
        return false;

    return equals((P.y - A.y)*(B.x - A.x), (P.x - A.x)*(B.y - A.y));
}
```

Closest point

```
template<typename T>
struct Segment {
    Point<T> A, B;

    bool contains(const Point<T>& P) const
    {
        if (equals(A.x, B.x))
            return min(A.y, B.y) <= P.y and P.y <= max(A.y, B.y);
        else
            return min(A.x, B.x) <= P.x and P.x <= max(A.x, B.x);
    }

    Point<T> closest(const Point<T>& P)
    {
        Line<T> r(A, B);
        auto Q = r.closest(P);

        if (this->contains(Q))
            return Q;

        auto distA = P.distanceTo(A);
        auto distB = P.distanceTo(B);

        if (distA <= distB)
            return A;
        else
            return B;
    }
}
```

Intersectin with segment

```
template<typename T>
class Segment {
public:
    Point<T> A, B;

    bool intersect(const Segment& s) const
    {
        auto d1 = D(A, B, s.A);
        auto d2 = D(A, B, s.B);

        if ((equals(d1, 0) && contains(s.A)) ||
           (equals(d2, 0) && contains(s.B)))
            return true;

        auto d3 = D(s.A, s.B, A);
        auto d4 = D(s.A, s.B, B);

        if ((equals(d3, 0) && s.contains(A)) ||
```

```
            (equals(d4, 0) && s.contains(B)))
             return true;

        return (d1 * d2 < 0) && (d3 * d4 < 0);
    }
}
```

Vectors

Angle between vector and X-axis

Returns an angle in radians in the interval $[-\pi, +\pi]$. A positive angle means in the COUNTER-clockwise direction. A negative angle is measured in the clockwise direction. Note that the atan2 swaped the parameters.

```
inline double angle(double x, double y) {
    return atan2(y, x);
}
```

Listing 5: angle between X-axis and vectorx, y

Translation

```
Point translate(const Point& P, double dx, double dy)
{
    return Point { P.x + dx, P.y + dy };
}
```

Listing 6: Translate point

Rotation around origin

```
Point rotate(const Point& P, double angle)
{
    auto x = cos(angle) * P.x - sin(angle) * P.y;
    auto y = sin(angle) * P.x + cos(angle) * P.y;

    return Point { x, y };
}
```

Rotation around another point

```
Point rotate(const Point& P, double angle, const Point& C)
{
    auto Q = translate(P, -C.x, -C.y);
    Q = rotate(Q, angle);
    Q = translate(Q, C.x, C.y);

    return Q;
}
```

Rotation around origin 3D

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \quad R_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scale

```
Point scale(double sx, double sy)
{
    return Point(sx * P.x, sy * P.y);
}
```

Listing 7: Scale vector by a factor of sx and sy

Normalization

```
Vector normalize(const Vector& v)
{
    auto len = v.length();
    auto u = Vector(v.x / len, v.y / len);

    return u;
}
```

Listing 8: Returns a unit vector with the same direction as the given vector

Dot product

$$< \vec{u}, \vec{v} >= \vec{u} \cdot \vec{v} = u_x v_x + u_y v_y = |\vec{u}||\vec{v}| \cos\theta$$

```
double dot_product(const Vector& u, const Vector& v)
{
    return u.x * v.x + u.y * v.y;
}
```

Angle between vectors

```
double angle(const Vector& u, const Vector& v)
{
    auto lu = u.length();
    auto lv = v.length();
    auto prod = dot_product(u, v);

    return acos(prod/(lu * lv));
}
```

Cross product

$$u \times v = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$

- $|\vec{u} \times \vec{v}| = |\vec{u}||\vec{v}| \sin \theta$
- where $\vec{i}, \vec{j}, \vec{k}$ are unity vectors on the same direction and orientation as x, y, z, respectively
- the result vector $\vec{w}$ is orthogonal to both $\vec{u}$ and $\vec{v}$
- it is the area of the parallelogram formed by $\vec{u}$ and $\vec{v}$

```
Vector cross_product(const Vector& u, const Vector& v)
{
    auto x = u.y*v.z - v.y*u.z;
    auto y = u.z*v.x - u.x*v.z;
    auto z = u.x*v.y - u.y*v.x;

    return Vector(x, y, z);
}
```

Circles

Definition

```
template<typename T>
struct Circle {
    Point<T> C;
    T r;
};
```

Perimeter, Area

```
template<typename T>
struct Circle
{
    double perimeter() const
    {
        return 2.0 * PI * r;
    }

    double area() const
    {
```

```
        return PI * r * r;
    }

    double arc(double theta) const
    {
        return theta * r;
    }

    double chord(double theta) const
    {
        return 2 * r * sin(theta/2);
    }

  double sector(double theta) const
    {
        return (theta * r * r)/2;
    }

    double segment(double a) const
    {
        auto c = chord(a);
        auto s = (r + r + c)/2.0;
        auto T = sqrt(s*(s - r)*(s - r)*(s - c));

        return sector(a) - T;
    }

};
```

From 2 points

```
#include <optional>

template<typename T>
struct Circle {
    static std::optional<Circle>
    from_2_points_and_r(const Point<T>& P, const Point<T>& Q, T r)
    {
        double d2 = (P.x - Q.x) * (P.x - Q.x) + (P.y - Q.y) * (P.y - Q.y);
        double det = r * r / d2 - 0.25;

        if (det < 0.0)
            return { };

        double h = sqrt(det);

        auto x = (P.x + Q.x) * 0.5 + (P.y - Q.y) * h;
        auto y = (P.y + Q.y) * 0.5 + (Q.x - P.x) * h;

        return Circle { Point(x, y), r };
    }
}
```

From 3 points

```
#include <optional>

template<typename T>
struct Circle {
```

```cpp
    static std::optional<Circle>
    from_3_points(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
    {
        auto a = 2*(Q.x - P.x);
        auto b = 2*(Q.y - P.y);
        auto c = 2*(R.x - P.x);
        auto d = 2*(R.y - P.y);

        auto det = a*d - b*c;
        if (equals(det, 0))
            return { };

        auto k1 = (Q.x*Q.x + Q.y*Q.y) - (P.x*P.x + P.y*P.y);
        auto k2 = (R.x*R.x + R.y*R.y) - (P.x*P.x + P.y*P.y);

        auto cx = (k1*d - k2*b)/det;
        auto cy = (a*k2 - c*k1)/det;

        Point<T> C { cx, cy };
        auto r = distance(P, C);

        return Circle<T>(C, r);
    }
};
```

Intersection between 2 circles

```cpp
#include <variant>
#include <vector>

const int oo { 2000000000 };

template<typename T> std::variant<int, std::vector<Point<T>>>
intersection(const Circle<T>& c1, const Circle<T>& c2)
{
    double d = distance(c1.C, c2.C);

    if (d > c1.r + c2.r or d < fabs(c1.r - c2.r))
        return 0;

    if (equals(d, 0.0) and equals(c1.r, c2.r))
        return oo;

    auto a = (c1.r * c1.r - c2.r * c2.r + d * d)/(2 * d);
    auto h = sqrt(c1.r * c1.r - a * a);

    auto x = c1.C.x + (a/d)*(c2.C.x - c1.C.x);
    auto y = c1.C.y + (a/d)*(c2.C.y - c1.C.y);

    auto P = Point<T> { x, y };

    x = P.x + (h/d)*(c2.C.y - c1.C.y);
    y = P.y - (h/d)*(c2.C.x - c1.C.x);

    auto P1 = Point<T> { x, y };

    x = P.x - (h/d)*(c2.C.y - c1.C.y);
    y = P.y + (h/d)*(c2.C.x - c1.C.x);

    auto P2 = Point<T> { x, y };
```

```
    return std::vector<Point<T>> { P1, P2 };
}
```

Intersection between circle and line

```
template<typename T> std::vector<Point<T>>
intersection(const Circle<T>& c, const Point<T>& P, const Point<T>& Q)
{
    auto a = pow(Q.x - P.x, 2.0) + pow(Q.y - P.y, 2.0);
    auto b = 2*((Q.x - P.x) * (P.x - c.C.x) + (Q.y - P.y) * (P.y - c.C.y));
    auto d = pow(c.C.x, 2.0) + pow(c.C.y, 2.0) + pow(P.x, 2.0)
        + pow(P.y, 2.0) + 2*(c.C.x * P.x + c.C.y * P.y);
    auto D = b * b - 4 * a * d;

    if (D < 0)
        return { };
    else if (equals(D, 0))
    {
        auto u = -b/(2*a);
        auto x = P.x + u*(Q.x - P.x);
        auto y = P.y + u*(Q.y - P.y);
        return { Point { x, y } };
    }

    auto u = (-b + sqrt(D))/(2*a);

    auto x = P.x + u*(Q.x - P.x);
    auto y = P.y + u*(Q.y - P.y);

    auto P1 = Point { x, y };

    u = (-b - sqrt(D))/(2*a);

    x = P.x + u*(Q.x - P.x);
    y = P.y + u*(Q.y - P.y);

    auto P2 = Point { x, y };

    return { P1, P2 };
}
```

Intersection between circle and point

```
template<typename T>
struct Circle {
    Point<T> C;
    T r;

    enum { IN, ON, OUT } PointPosition;

    PointPosition position(const Point& P) const
    {
        auto d = dist(P, C);

        return equals(d, r) ? ON : (d < r ? IN : OUT);
    }
};
```

Triangles

Perimeter

```
template<typename T>
struct Triangle {
    Point<T> A, B, C;

    double perimeter() const
    {
        auto a = dist(A, B);
        auto b = dist(B, C);
        auto c = dist(C, A);

        return a + b + c;
    }
};
```

Area

```
// Defini  o das estruturas Point e Line

template<typename T>
struct Triangle {
    Point<T> A, B, C;

    double area() const
    {
        Line<T> r(A, B);

        auto b = dist(A, B);
        auto h = r.distance(C);

        return (b * h)/2;
    }
};
```

```
// Defini  o da estrutura Point

template<typename T>
struct Triangle {
    Point<T> A, B, C;

    double area() const
    {
        auto a = dist(A, B);
        auto b = dist(B, C);
        auto c = dist(C, A);

        auto s = (a + b + c)/2

        return sqrt(s)*sqrt(s - a)*sqrt(s - b)*sqrt(s - c);
    }
};
```

```
// Defini  o da estrutura Point

template<typename T>
```

```
struct Triangle {
    Point<T> A, B, C;

    double area() const
    {
        double det = (A.x*B.y + A.y*C.x + B.x*C.y)
                   - (C.x*B.y + C.y*A.x + B.x*A.y);

        return 0.5 * fabs(det);
    }
};
```

Side classification

By sides

```
template<typename T>
struct Triangle {
    Point<T> A, B, C;

    enum Sides { EQUILATERAL, ISOSCELES, SCALENE };

    Sides classification_by_sides() const
    {
        auto a = dist(A, B);
        auto b = dist(B, C);
        auto c = dist(C, A);

        if (equals(a, b) and equals(b, c))
            return EQUILATERAL;

        if (equals(a, b) or equals(a, c) or equals(b, c))
            return ISOSCELES;

        return SCALENE;
    }
};
```

By angles

```
// Defini  o da classe Point, da fun  o de compara  o equals() e
// da fun  o de dist ncia entre pontos dist()

template<typename T>
struct Triangle {
    Point<T> A, B, C;

    enum Angles { RIGHT, ACUTE, OBTUSE };

    Angles classification_by_angles() const
    {
        auto a = dist(A, B);
        auto b = dist(B, C);
        auto c = dist(C, A);

        auto alpha = acos((a*a - b*b - c*c)/(-2*b*c));
        auto beta = acos((b*b - a*a - c*c)/(-2*a*c));
        auto gamma = acos((c*c - a*a - b*b)/(-2*a*b));

```

```
        auto right = PI / 2.0;

        if (equals(alpha, right) || equals(beta, right)
            || equals(gamma, right))
            return RIGHT;

        if (alpha > right || beta > right || gamma > right)
            return OBTUSE;

        return ACUTE;
    }
};
```

Important points

Barycenter

```
// Defini  o da estrutura Point

template<typename T>
struct Triangle {
    Point<T> A, B, C;

    Point<T> barycenter() const
    {
        auto x = (A.x + B.x + C.x) / 3.0;
        auto y = (A.y + B.y + C.y) / 3.0;

        return Point<T> { x, y };
    }
};
```

Incenter

```
template<typename T>
struct Triangle {
    Point<T> A, B, C;

    // Defini  o dos m todos area() e perimeter()

    double inradius() const
    {
        return (2 * area()) / perimeter();
    }

    Point<T> incenter() const
    {
        auto P = perimeter();
        auto x = (a*A.x + b*B.x + c*C.x)/P;
        auto y = (a*A.y + b*B.y + c*C.y)/P;

        return { x, y };
    }
};
```

Orthocenter

```cpp
#include <iostream>

using namespace std;

template<typename T>
struct Point {
    T x, y;
};

template<typename T>
struct Line {
    T a, b, c;

    Line(T av, T bv, T cv) : a(av), b(bv), c(cv) {}

    Line(const Point<T>& P, const Point<T>& Q)
        : a(P.y - Q.y), b (Q.x - P.x), c(P.x * Q.y - Q.x * P.y)
    {
    }
};

template<typename T>
struct Triangle {
    Point<T> A, B, C;

    Point<T> orthocenter() const
    {
        Line<T> r(A, B), s(A, C);

        Line<T> u { r.b, -r.a, -(C.x*r.b - C.y*r.a) };
        Line<T> v { s.b, -s.a, -(B.x*s.b - B.y*s.a) };

        auto det = u.a * v.b - u.b * v.a;
        auto x = (-u.c * v.b + v.c * u.b) / det;
        auto y = (-v.c * u.a + u.c * v.a) / det;

        return { x, y };
    }
};

int main()
{
    Point<double> A { 0, 0 }, B { 3, 6 },  C { 9, 1 };
    Triangle<double> T { A, B, C };

    auto O = T.orthocenter();

    cout << "(" << O.x << ", " << O.y << ")\n";

    return 0;
}
```

Circumcircle

```cpp
// Defini  o da estrutura Point e da fun  o de dist ncia
// entre pontos dist()

template<typename T>
struct Triangle {
    Point<T> A, B, C;
```

```

    // Defini  o do m todo area()

    double circumradius() const
    {
        auto a = dist(B, C);
        auto b = dist(A, C);
        auto c = dist(A, B);

        return (a * b * c)/(4 * area());
    }

    Point<T> circumcenter() const
    {
        auto D = 2*(A.x*(B.y - C.y) + B.x*(C.y - A.y) + C.x*(A.y - B.y));

        auto A2 = A.x*A.x + A.y*A.y;
        auto B2 = B.x*B.x + B.y*B.y;
        auto C2 = C.x*C.x + C.y*C.y;

        auto x = (A2*(B.y - C.y) + B2*(C.y - A.y) + C2*(A.y - B.y))/D;
        auto y = (A2*(C.x - B.x) + B2*(A.x - C.x) + C2*(B.x - A.x))/D;

        return { x, y };
    }
};
```

Quadrilaterals

Area

Trapezium

```
template<typename T>
struct Trapezium {
    T b, B, h;

    T area() const
    {
        return (b + B) * h / 2;
    }
};
```

Quadrilateral

```
// Defini  o das estruturas Point e Line

template<typename T>
struct Triangle {
    Point<T> A, B, C;

    double area() const
    {
        Line<T> r(A, B);

        auto b = dist(A, B);
        auto h = r.distance(C);

```

```
        return (b * h)/2;
    }
};
```

Rectangles

From 2 points

```
// Defini  o da estrutura Point

template<typename T>
class Rectangle {
public:
    Point<T> P, Q;
    T b, h;

    Rectangle(const Point<T>& p, const Point<T>& q) : P(p), Q(q)
    {
        b = max(P.x, Q.x) - min(P.x, Q.x);
        h = max(P.y, Q.y) - min(P.y, Q.y);
    }

    Rectangle(const T& base, const T& height)
        : P(0, 0), Q(base, height), b(base), h(height) {}
};
```

Intersection between rectangles

```
// Defini  o da classe Point
template<typename T>
struct Rectangle {
    // Membros e construtores

    Rectangle intersection(const Rectangle& r) const
    {
        using interval = pair<T, T>;

        auto I = interval(min(P.x, Q.x), max(P.x, Q.x));
        auto U = interval(min(r.P.x, r.Q.x), max(r.P.x, r.Q.x));

        auto a = max(I.first, U.first);
        auto b = min(I.second, U.second);

        if (b < a)
            return { {-1, -1}, {-1, -1} };

        I = interval(min(P.y, Q.y), may(P.y, Q.y));
        U = interval(min(r.P.y, r.Q.y), may(r.P.y, r.Q.y));

        auto c = max(I.first, U.first);
        auto d = min(I.second, U.second);

        if (d < c)
            return { {-1, -1}, {-1, -1} };

        inter = Rectangle(Point(a, c), Point(b, d));

        return { {a, c}, {b, d} };
    }
```

```
};
```

Polygons

Definition

```cpp
#include <bits/stdc++.h>

using namespace std;

template<typename T>
struct Point { T x, y; };

template<typename T>
class Polygon {
private:
    vector<Point<T>> vs;
    int n;

public:
    // O par metro deve conter os n v rtices do pol gono
    Polygon(const vector<Point<T>>& ps) : vs(ps), n(vs.size())
    {
        vs.push_back(vs.front());
    }

private:
    T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R) const
    {
        return (P.x * Q.y + P.y * R.x + Q.x * R.y) -
            (R.x * Q.y + R.y * P.x + Q.x * P.y);
    }

public:
    bool convex() const {
        // Um pol gono deve ter, no minimo, 3 v rtices
        if (n < 3) return false;

        int P = 0, N = 0, Z = 0;

        for (int i = 0; i < n; ++i) {
            auto d = D(vs[i], vs[(i + 1) % n], vs[(i + 2) % n]);
            d ? (d > 0 ? ++P : ++N) : ++Z;
        }

        return not ((P and N) or (P == 0 and N == 0));
    }

private:
    double distance(const Point<T>&P, const Point<T>& Q)
    {
        return hypot(P.x - Q.x, P.y - Q.y);
    }
public:
    double perimeter() const
    {
        auto p = 0.0;

        for (int i = 0; i < n; ++i)
```

```
            p += distance(vs[i], vs[i + 1]);

        return p;
    }

    double area() const
    {
        auto a = 0.0;

        for (int i = 0; i < n; ++i)
        {
            a += vs[i].x * vs[i + 1].y;
            a -= vs[i + 1].x * vs[i].y;
        }

        return 0.5 * fabs(a);
    }

private:
    //  ngulo APB, em radianos
    double angle(const Point<T>& P, const Point<T>& A, const Point<T>& B)
    {
        auto ux = P.x - A.x;
        auto uy = P.y - A.y;

        auto vx = P.x - B.x;
        auto vy = P.y - B.y;

        auto num = ux * vx + uy * vy;
        auto den = hypot(ux, uy) * hypot(vx, vy);

        // Caso especial: se den == 0, algum dos vetores   degenerado: os
        // dois pontos s o iguais. Neste caso, o  ngulo n o est  definido

        return acos(num / den);
    }

    bool equals(double x, double y)
    {
        static const double EPS { 1e-9 };

        return fabs(x - y) < EPS;
    }

public:
    bool contains(const Point<T>& P) const
    {
        if (n < 3)
            return false;

        auto sum = 0.0;

        for (int i = 0; i < n; ++i)
        {
            auto d = D(P, vs[i], vs[i + 1]);

            // Pontos sobre as arestas ou v rtices s o considerados
            // interiores
            if (equals(d, 0) and AB_contains(P))
                return true;

            auto a = angle(P, vs[i], vs[i + 1]);
```

```
            sum += d > 0 ? a : -a;
        }

        static const double PI = acos(-1.0);

        return equals(fabs(sum), 2*PI);
    }

private:
    // Interse  o entre a reta AB e o segmento de reta PQ
    Point<T> intersection(const Point<T>& P, const Point<T>& Q,
                          const Point<T>& A, const Point<T>& B)
    {
        auto a = B.y - A.y;
        auto b = A.x - B.x;
        auto c = B.x * A.y - A.x * B.y;
        auto u = fabs(a * P.x + b * P.y + c);
        auto v = fabs(a * Q.x + b * Q.y + c);

        // M dia ponderada pelas dist ncias de P e Q at  a reta AB
        return {(P.x * v + Q.x * u)/(u + v), (P.y * v + Q.y * u)/(u + v)};
    }

public:
    // Corta o pol gono com a reta r que passa por A e B
    Polygon cut_polygon(const Point<T>& A, const Point<T>& B) const
    {
        vector<Point<T>> points;
        const double EPS { 1e-9 };

        for (int i = 0; i < n; ++i)
        {
            auto d1 = D(A, B, vs[i]);
            auto d2 = D(A, B, vs[i + 1]);

            // V rtice   esquerda da reta
            if (d1 > -EPS)
                points.push_back(vs[i]);

            // A aresta cruza a reta
            if (d1 * d2 < -EPS)
                points.push_back(intersection(vs[i], vs[i + 1], A, B));
        }

        return Polygon(points);
    }

    double circumradius() const
    {
        auto s = distance(vs[0], vs[1]);
        const double PI { acos(-1.0) };

        return (s/2.0)*(1.0/sin(PI/n));
    }

    double apothem() const
    {
        auto s = distance(vs[0], vs[1]);
        const double PI { acos(-1.0) };

        return (s/2.0)*(1.0/tan(PI/n));
```

```
    }
};

int main()
{

    vector<Point<int>> xs { { 0, 0 }, { 2, 1 }, { 3, 4 }, { 5, 2 }, { 4, 0 } };
    vector<Point<int>> ys { { 6, 1 }, { 9, 3 }, { 9, 1 }, { 6, 3 } };
    vector<Point<double>> zs { { 0, 0 }, { 1, 0 }, { 0.5, 0.5 }, { 1, 1 }, { 0, 1 }, {
    0.5, 0.5 } };

    Polygon<int> A(xs), B(ys);
    Polygon<double> C(zs);

    cout << "A is convex? " << A.convex() << '\n';
    cout << "B is convex? " << B.convex() << '\n';
    cout << "C is convex? " << C.convex() << '\n';

    cout << C.area() << '\n';

    return 0;
}
```

UVA 11265

```
#include <bits/stdc++.h>

using namespace std;

struct Point {
    double x, y;

    Point(double xv = 0, double yv = 0) : x(xv), y(yv) {}

    double distance(const Point& P) const
    {
        return hypot(x - P.x, y - P.y);
    }

    bool operator==(const Point& P) const
    {
        const double EPS { 1e-6 };
        return fabs(x - P.x) < EPS and fabs(y - P.y) < EPS;
    }
};

struct Polygon {
    vector<Point> vs;
    int n;

    Polygon(const vector<Point>& vs) : vs(vs), n(vs.size())
    {
        vs.push_back(vs[0]);
    }

    double area() const {
        double a = 0;

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

```
            a += vs[i].x * vs[i+1].y;
            a -= vs[i+1].x * vs[i].y;
        }

        return 0.5 * fabs(a);
    }
};

Point intersection(const Point& P, const Point& Q,
    const Point& A, const Point& B)
{
    auto a = B.y - A.y;
    auto b = A.x - B.x;
    auto c = B.x * A.y - A.x * B.y;
    auto u = fabs(a * P.x + b * P.y + c);
    auto 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));
}

double D(const Point& P, const Point& Q, const Point& R)
{
    return (P.x * Q.y + P.y * R.x + Q.x * R.y)
        - (R.x * Q.y + R.y * P.x + Q.x * P.y);
}

Polygon cut_polygon(const Polygon& P, const Point& A, const Point& B)
{
    vector<Point> points;

    for (int i = 0; i < P.n; ++i)
    {
        auto d1 = D(A, B, P.vs[i]);
        auto d2 = D(A, B, P.vs[i + 1]);

        if (d1 > -EPS)
            points.push_back(P.vs[i]);

        if (d1 * d2 < -EPS)
            points.push_back(intersection(P.vs[i], P.vs[i+1], A, B));
    }

    return Polygon(points);
}

int main() {
    int N, W, H, x, y, test = 0;

    while (cin >> N >> W >> x >> y) {
        Polygon p({ Point(0, 0), Point(W, 0), Point(W, H), Point(0, H) });
        Point F(x, y);

        while (N--) {
            Point Q, R;
            cin >> Q.x >> Q.y >> R.x >> R.y;

            if (D(Q, R, F) > 0)
                p = cut_polygon(p, Q, R);
            else
                p = cut_polygon(p, R, Q);
        }
```

```
        printf("Case #%d: %.3f\n", ++test, p.area());
    }
    return 0;
}
```

Codeforces 1C

```
#include <bits/stdc++.h>

using namespace std;

const double PI { acos(-1.0) };
const int MAX { 110 };

double angles[MAX];

struct Point {
    double x, y;

    double distance(const Point& P) const
    {
        return hypot(x - P.x, y - P.y);
    }

    Point translate(const Point& P) const
    {
        return Point { x + P.x, y + P.y };
    }

    Point rotate(double angle) const
    {
        auto xv = x*cos(angle) - y*sin(angle);
        auto yv = x*sin(angle) + y*cos(angle);

        return Point { xv, yv };
    }

    bool operator==(const Point& P) const
    {
        const double EPS { 1e-5 };

        return fabs(x - P.x) < EPS and fabs(y - P.y) < EPS;
    }
};

struct Triangle {
    Point A, B, C;

    double area() const
    {
        auto a = A.distance(B);
        auto b = B.distance(C);
        auto c = C.distance(A);
        auto s = (a + b + c) / 2;

        return sqrt(s*(s - a)*(s - b)*(s - c));
    }

    double circumradius() const
    {
```

```
        auto a = A.distance(B);
        auto b = B.distance(C);
        auto c = C.distance(A);

        return (a * b * c)/(4 * area());
    }

    Point circumcenter() const
    {
        auto d = 2*(A.x*(B.y - C.y) + B.x*(C.y - A.y) + C.x*(A.y - B.y));

        auto A2 = A.x*A.x + A.y*A.y;
        auto B2 = B.x*B.x + B.y*B.y;
        auto C2 = C.x*C.x + C.y*C.y;

        auto x = (A2*(B.y - C.y) + B2*(C.y - A.y) + C2*(A.y - B.y))/d;
        auto y = (A2*(C.x - B.x) + B2*(A.x - C.x) + C2*(B.x - A.x))/d;

        return Point { x, y };
    }
};

void precomp()
{
    for (int i = 1; i < MAX; ++i)
        angles[i] = (2.0*PI)/i;
}

int sides(const Point& P, const Point& Q, const Point& R)
{
    for (int i = 3; i < 100; ++i)
    {
        auto angle = angles[i];
        int match = 0;
        Point S { P };

        for (int j = 0; j < i; ++j)
        {
            if (Q == S)
                ++match;

            if (R == S)
                ++match;

            S = S.rotate(angle);
        }

        if (match == 2)
            return i;
    }

    return 100;
}

int main()
{
    precomp();

    Point P, Q, R;

    cin >> P.x >> P.y >> Q.x >> Q.y >> R.x >> R.y;

```

```
    Triangle t { P, Q, R };

    auto r = t.circumradius();
    auto C = t.circumcenter();

    P = P.translate(Point { -C.x, -C.y } );
    Q = Q.translate(Point { -C.x, -C.y } );
    R = R.translate(Point { -C.x, -C.y } );

    int min_sides = sides(P, Q, R);

    auto area = (r * r * min_sides*sin(angles[min_sides]))/2.0;

    cout.precision(6);
    cout << fixed << area << '\n';

    return 0;
}
```

URI 2202

```
#include <bits/stdc++.h>

using namespace std;
using ll = long long;

struct Point {
    ll x, y;

    bool operator<(const Point& P) const
    {
        return x == P.x ? y < P.y : x < P.x;
    }
};

struct Line {
    ll a, b, c;

    Line(const Point& p, const Point& q)
    {
        a = p.y - q.y;
        b = q.x - p.x;
        c = p.x * q.y - p.y * q.x;
    }

    double distanceTo(const Point& P) const
    {
        auto num = a*P.x + b*P.y + c;
        auto den = sqrt(a*a + b*b);

        return fabs(num/den);
    }
};

ll D(const Point& P, const Point& Q, const Point& R)
{
    return (P.x * Q.y + P.y * R.x + Q.x * R.y) -
           (R.x * Q.y + R.y * P.x + Q.x * P.y);
}

```

```
// Andrew monotonic chain
vector<Point> convex_hull(vector<Point>& P)
{
    sort(P.begin(), P.end());

    vector<Point> L, U;

    for (auto p : P)
    {
        while (L.size() >= 2 and D(L[L.size() - 2], L[L.size() -1], p) < 0)
            L.pop_back();

        L.push_back(p);
    }

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

    for (auto p : P)
    {
        while (U.size() >= 2 and D(U[U.size() - 2], U[U.size() -1], p) < 0)
            U.pop_back();

        U.push_back(p);
    }

    L.pop_back();
    U.pop_back();

    L.reserve(L.size() + U.size());
    L.insert(L.end(), U.begin(), U.end());

    return L;
}

int main()
{
    ios::sync_with_stdio(false);

    int n, test = 0;

    while (cin >> n, n)
    {
        vector<Point> ps(n);

        for (int i = 0; i < n; ++i)
            cin >> ps[i].x >> ps[i].y;

        auto ch = convex_hull(ps);
        ch.push_back(ch.front());
cout << "convex hull (size = " << ch.size() << ", n = " << n << ") = ";
for (const auto& v : ch)
    cout << "(" << v.x << ", " << v.y << ") ";
cout << '\n';
        auto min_dist = 1000000000.0;

        for (int i = 0; i < n; ++i)
        {
            auto A = ch[i];
            auto B = ch[i + 1];
            auto max_dist = 0.0;

            Line r(A, B);
```

```
cout << "A = (" << A.x << ", " << A.y << ")\n";
cout << "B = (" << B.x << ", " << B.y << ")\n";
            for (int j = 0; j < n; ++j)
            {
                auto d = r.distanceTo(ch[j]);
cout << "dist = " << d << "\n";
cout << "P = (" << ch[j].x << ", " << ch[j].y << ")\n";
                max_dist = max(max_dist, d);
            }
cout << "max dist = " << max_dist << '\n';
            min_dist = min(min_dist, max_dist);
cout << "\n";
        }

        cout << "Case " << ++test << ": ";
        cout.precision(2);
        cout << fixed << min_dist << '\n';
    }

    return 0;
}
```

Convex Hull

Graham

```
#include <bits/stdc++.h>

using namespace std;

const double EPS { 1e-6 };

template<typename T>
bool equals(T a, T b)
{
    if (std::is_floating_point<T>::value)
        return fabs(a - b) < EPS;
    else
        return a == b;
}

template<typename T>
struct Point
{
    T x, y;

    double distance(const Point& P) const
    {
        return hypot(x - P.x, y - P.y);
    }
};

template<typename T>
T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
{
    return (P.x * Q.y + P.y * R.x + Q.x * R.y) -
           (R.x * Q.y + R.y * P.x + Q.x * P.y);
}
```

```
template<typename T>
class GrahamScan
{
private:
    static Point<T> pivot(vector<Point<T>>& P)
    {
        size_t idx = 0;

        for (size_t i = 1; i < P.size(); ++i)
            if (P[i].y < P[idx].y or
                (equals(P[i].y, P[idx].y) and P[i].x > P[idx].x))
                    idx = i;

        swap(P[0], P[idx]);

        return P[0];
    }

    static void sort_by_angle(vector<Point<T>>& P)
    {
        auto P0 = pivot(P);

        sort(P.begin() + 1, P.end(),
            [&](const Point<T>& A, const Point<T>& B) {
                // pontos colineares: escolhe-se o mais pr ximo do piv
                if (equals(D(P0, A, B), 0))
                    return A.distance(P0) < B.distance(P0);

                auto alfa = atan2(A.y - P0.y, A.x - P0.x);
                auto beta = atan2(B.y - P0.y, B.x - P0.x);

                return alfa < beta;
            }
        );
    }

public:
    static vector<Point<T>> convex_hull(const vector<Point<T>>& points)
    {
        vector<Point<T>> P(points);
        auto N = P.size();

        // Corner case: com 3 v rtices ou menos, P   o pr prio convex hull
        if (N <= 3)
            return P;

        sort_by_angle(P);

        vector<Point<T>> ch;
        ch.push_back(P[N - 1]);
        ch.push_back(P[0]);
        ch.push_back(P[1]);

        size_t i = 2;

        while (i < N)
        {
            auto j = ch.size() - 1;

            if (D(ch[j - 1], ch[j], P[i]) > 0)
                ch.push_back(P[i++]);
            else
```

```
                ch.pop_back();
        }

        // O envolt rio   um caminho fechado: o primeiro ponto   igual
        // ao  ltimo
        return ch;
    }
};

int main()
{
    vector<Point<int>> P { { 0, 0 }, { 5, 3 }, { 8, -2 }, { 4, 4 }, { 2, 1 }, { 2, 5 }, { 3, -1 },
        { 7, 2 }, { 5, 0 }, { 0, 4 }, { 1, -1 }, { 7, -2 }, { 6, 4 }, { 6, 0 }, { 1, 3 } };

    auto ch = GrahamScan<int>::convex_hull(P);

    for (size_t i = 0; i < ch.size(); ++i)
        cout << i + 1 << ": (" << ch[i].x << ", " << ch[i].y << ")\n";

    return 0;
}
```