二维几何

库实数类实现（双精度）

```
using Real = int;
using Point = complex<Real>;

Real cross(const Point &a, const Point &b) {
    return (conj(a) * b).imag();
}
Real dot(const Point &a, const Point &b) {
    return (conj(a) * b).real();
}
```

平面几何必要初始化

字符串读入浮点数

```
const int Knum = 4;
int read(int k = Knum) {
    string s;
    cin >> s;

    int num = 0;
    int it = s.find('.');
    if (it != -1) { // 存在小数点
        num = s.size() - it - 1; // 计算小数位数
        s.erase(s.begin() + it); // 删除小数点
    }
    for (int i = 1; i <= k - num; i++) { // 补全小数位数
        s += '0';
    }
    return stoi(s);
}
```

预置函数

```
using ld = long double;
const ld PI = acos(-1);
const ld EPS = 1e-7;
const ld INF = numeric_limits<ld>::max();
#define cc(x) cout << fixed << setprecision(x);

ld fgcd(ld x, ld y) { // 实数域gcd
    return abs(y) < EPS ? abs(x) : fgcd(y, fmod(x, y));
}
template<class T, class S> bool equal(T x, S y) {
    return -EPS < x - y && x - y < EPS;
}
template<class T> int sign(T x) {
```

```
    if (-EPS < x && x < EPS) return 0;
    return x < 0 ? -1 : 1;
}
```

点线封装

```
template<class T> struct Point { // 在C++17下使用 emplace_back 绑定可能会导致CE!
    T x, y;
    Point(T x_ = 0, T y_ = 0) : x(x_), y(y_) {} // 初始化
    template<class U> operator Point<U>() { // 自动类型匹配
        return Point<U>(U(x), U(y));
    }
    Point &operator+=(Point p) & { return x += p.x, y += p.y, *this; }
    Point &operator+=(T t) & { return x += t, y += t, *this; }
    Point &operator-=(Point p) & { return x -= p.x, y -= p.y, *this; }
    Point &operator-=(T t) & { return x -= t, y -= t, *this; }
    Point &operator*=(T t) & { return x *= t, y *= t, *this; }
    Point &operator/=(T t) & { return x /= t, y /= t, *this; }
    Point operator-() const { return Point(-x, -y); }
    friend Point operator+(Point a, Point b) { return a += b; }
    friend Point operator+(Point a, T b) { return a += b; }
    friend Point operator-(Point a, Point b) { return a -= b; }
    friend Point operator-(Point a, T b) { return a -= b; }
    friend Point operator*(Point a, T b) { return a *= b; }
    friend Point operator*(T a, Point b) { return b *= a; }
    friend Point operator/(Point a, T b) { return a /= b; }
    friend bool operator<(Point a, Point b) {
        return equal(a.x, b.x) ? a.y < b.y - EPS : a.x < b.x - EPS;
    }
    friend bool operator>(Point a, Point b) { return b < a; }
    friend bool operator==(Point a, Point b) { return !(a < b) && !(b < a); }
    friend bool operator!=(Point a, Point b) { return a < b || b < a; }
    friend auto &operator>>(istream &is, Point &p) {
        return is >> p.x >> p.y;
    }
    friend auto &operator<<(ostream &os, Point p) {
        return os << "(" << p.x << ", " << p.y << ")";
    }
};
template<class T> struct Line {
    Point<T> a, b;
    Line(Point<T> a_ = Point<T>(), Point<T> b_ = Point<T>()) : a(a_), b(b_) {}
    template<class U> operator Line<U>() { // 自动类型匹配
        return Line<U>(Point<U>(a), Point<U>(b));
    }
    friend auto &operator<<(ostream &os, Line l) {
        return os << "<" << l.a << ", " << l.b << ">";
    }
};
```

叉乘

定义公式 $a \times b = |a||b| \sin \theta$。

```
template<class T> T cross(Point<T> a, Point<T> b) { // 叉乘
    return a.x * b.y - a.y * b.x;
}
template<class T> T cross(Point<T> p1, Point<T> p2, Point<T> p0) { // 叉乘 (p1 - p0) x (p2 - p0);
    return cross(p1 - p0, p2 - p0);
}
```

点乘

定义公式 $a \times b = |a||b| \cos \theta$。

```
template<class T> T dot(Point<T> a, Point<T> b) { // 点乘
    return a.x * b.x + a.y * b.y;
}
template<class T> T dot(Point<T> p1, Point<T> p2, Point<T> p0) { // 点乘 (p1 - p0) * (p2 - p0);
    return dot(p1 - p0, p2 - p0);
}
```

欧几里得距离公式

最常用的距离公式。**需要注意**，开根号会丢失精度，如无强制要求，先不要开根号，留到最后一步一起开。

```
template <class T> ld dis(T x1, T y1, T x2, T y2) {
    ld val = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
    return sqrt(val);
}
template <class T> ld dis(Point<T> a, Point<T> b) {
    return dis(a.x, a.y, b.x, b.y);
}
```

曼哈顿距离公式

```
template <class T> T dis1(Point<T> p1, Point<T> p2) { // 曼哈顿距离公式
    return abs(p1.x - p2.x) + abs(p1.y - p2.y);
}
```

将向量转换为单位向量

```
Point<ld> standardize(Point<ld> vec) { // 转换为单位向量
    return vec / sqrt(vec.x * vec.x + vec.y * vec.y);
}
```

向量旋转

将当前向量移动至原点后顺时针旋转 90° ，即获取垂直于当前向量的、起点为原点的向量。在计算垂线时非常有用。例如，要想获取点 a 绕点 o 顺时针旋转 90° 后的点，可以这样书写代码：`auto ans = o + rotate(o, a);` ；如果是逆时针旋转，那么只需更改符号即可：`auto ans = o - rotate(o, a);` 。

```cpp
template<class T> Point<T> rotate(Point<T> p1, Point<T> p2) { // 旋转
    Point<T> vec = p1 - p2;
    return {-vec.y, vec.x};
}
```

平面角度与弧度

弧度角度相互转换

```cpp
ld toDeg(ld x) { // 弧度转角度
    return x * 180 / PI;
}
ld toArc(ld x) { // 角度转弧度
    return PI / 180 * x;
}
```

正弦定理

$\dfrac{a}{\sin A}=\dfrac{b}{\sin B}=\dfrac{c}{\sin C}=2R$ ，其中 R 为三角形外接圆半径；

余弦定理（已知三角形三边，求角）

$\cos C=\dfrac{a^2+b^2-c^2}{2ab},\cos B=\dfrac{a^2+c^2-b^2}{2ac},\cos A=\dfrac{b^2+c^2-a^2}{2bc}$。可以借此推导出三角形面积公式 $S_{\triangle ABC}=\dfrac{ab\cdot\sin C}{2}=\dfrac{bc\cdot\sin A}{2}=\dfrac{ac\cdot\sin B}{2}$。

注意，计算格式是：由 b,c,a 三边求 $\angle A$；由 a,c,b 三边求 $\angle B$；由 a,b,c 三边求 $\angle C$。

```cpp
ld angle(ld a, ld b, ld c) { // 余弦定理
    ld val = acos((a * a + b * b - c * c) / (2.0 * a * b)); // 计算弧度
    return val;
}
```

求两向量的夹角

能够计算 $[0^\circ,180^\circ]$ 区间的角度。

```cpp
ld angle(Point<ld> a, Point<ld> b) {
    ld val = abs(cross(a, b));
    return abs(atan2(val, a.x * b.x + a.y * b.y));
}
```

向量旋转任意角度

逆时针旋转，转换公式：$\begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$

```
Point<ld> rotate(Point<ld> p, ld rad) {
    return {p.x * cos(rad) - p.y * sin(rad), p.x * sin(rad) + p.y * cos(rad)};
}
```

点绕点旋转任意角度

逆时针旋转，转换公式：$\begin{cases} x' = (x_0 - x_1)\cos\theta + (y_0 - y_1)\sin\theta + x_1 \\ y' = (x_1 - x_0)\sin\theta + (y_0 - y_1)\cos\theta + y_1 \end{cases}$

```
Point<ld> rotate(Point<ld> a, Point<ld> b, ld rad) {
    ld x = (a.x - b.x) * cos(rad) + (a.y - b.y) * sin(rad) + b.x;
    ld y = (b.x - a.x) * sin(rad) + (a.y - b.y) * cos(rad) + b.y;
    return {x, y};
}
```

平面点线相关

点是否在直线上（三点是否共线）

```
template<class T> bool onLine(Point<T> a, Point<T> b, Point<T> c) {
    return sign(cross(b, a, c)) == 0;
}
template<class T> bool onLine(Point<T> p, Line<T> l) {
    return onLine(p, l.a, l.b);
}
```

点是否在向量（直线）左侧

需要注意，向量的方向会影响答案；点在向量上时不视为在左侧。

```
template<class T> bool pointOnLineLeft(Pt p, Lt l) {
    return cross(l.b, p, l.a) > 0;
}
```

两点是否在直线同侧/异侧

```
template<class T> bool pointOnLineSide(Pt p1, Pt p2, Lt vec) {
    T val = cross(p1, vec.a, vec.b) * cross(p2, vec.a, vec.b);
    return sign(val) == 1;
}
template<class T> bool pointNotOnLineSide(Pt p1, Pt p2, Lt vec) {
    T val = cross(p1, vec.a, vec.b) * cross(p2, vec.a, vec.b);
    return sign(val) == -1;
}
```

两直线相交交点

在使用前需要先判断直线是否平行。

```
Pd lineIntersection(Ld l1, Ld l2) {
    ld val = cross(l2.b - l2.a, l1.a - l2.a) / cross(l2.b - l2.a, l1.a - l1.b);
    return l1.a + (l1.b - l1.a) * val;
}
```

两直线是否平行/垂直/相同

```
template<class T> bool lineParallel(Lt p1, Lt p2) {
    return sign(cross(p1.a - p1.b, p2.a - p2.b)) == 0;
}
template<class T> bool lineVertical(Lt p1, Lt p2) {
    return sign(dot(p1.a - p1.b, p2.a - p2.b)) == 0;
}
template<class T> bool same(Line<T> l1, Line<T> l2) {
    return lineParallel(Line{l1.a, l2.b}, {l1.b, l2.a}) &&
           lineParallel(Line{l1.a, l2.a}, {l1.b, l2.b}) && lineParallel(l1, l2);
}
```

点到直线的最近距离与最近点

```
pair<Pd, ld> pointToLine(Pd p, Ld l) {
    Pd ans = lineIntersection({p, p + rotate(l.a, l.b)}, l);
    return {ans, dis(p, ans)};
}
```

如果只需要计算最近距离，下方的写法可以减少书写的代码量，效果一致。

```
template<class T> ld disPointToLine(Pt p, Lt l) {
    ld ans = cross(p, l.a, l.b);
    return abs(ans) / dis(l.a, l.b); // 面积除以底边长
}
```

点是否在线段上

```
template<class T> bool pointOnSegment(Pt p, Lt l) { // 端点也算作在直线上
    return sign(cross(p, l.a, l.b)) == 0 && min(l.a.x, l.b.x) <= p.x && p.x <= max(l.a.x, l.b.x) &&
           min(l.a.y, l.b.y) <= p.y && p.y <= max(l.a.y, l.b.y);
}
template<class T> bool pointOnSegment(Pt p, Lt l) { // 端点不算
    return pointOnSegment(p, l) && min(l.a.x, l.b.x) < p.x && p.x < max(l.a.x, l.b.x) &&
           min(l.a.y, l.b.y) < p.y && p.y < max(l.a.y, l.b.y);
}
```

点到线段的最近距离与最近点

```
pair<Pd, ld> pointToSegment(Pd p, Ld l) {
    if (sign(dot(p, l.b, l.a)) == -1) { // 特判到两端点的距离
        return {l.a, dis(p, l.a)};
    } else if (sign(dot(p, l.a, l.b)) == -1) {
        return {l.b, dis(p, l.b)};
    }
    return pointToLine(p, l);
}
```

点在直线上的投影点（垂足）

```
Pd project(Pd p, Ld l) { // 投影
    Pd vec = l.b - l.a;
    ld r = dot(vec, p - l.a) / (vec.x * vec.x + vec.y * vec.y);
    return l.a + vec * r;
}
```

线段的中垂线

```
template<class T> Lt midSegment(Lt l) {
    Pt mid = (l.a + l.b) / 2; // 线段中点
    return {mid, mid + rotate(l.a, l.b)};
}
```

两线段是否相交及交点

该扩展版可以同时返回相交状态和交点，分为四种情况：0 代表不相交；1 代表普通相交；2 代表重叠（交于两个点）；3 代表相交于端点。**需要注意**，部分运算可能会使用到直线求交点，此时务必保证变量类型为浮点数！

```
template<class T> tuple<int, Pt, Pt> segmentIntersection(Lt l1, Lt l2) {
    auto [s1, e1] = l1;
    auto [s2, e2] = l2;
    auto A = max(s1.x, e1.x), AA = min(s1.x, e1.x);
    auto B = max(s1.y, e1.y), BB = min(s1.y, e1.y);
    auto C = max(s2.x, e2.x), CC = min(s2.x, e2.x);
    auto D = max(s2.y, e2.y), DD = min(s2.y, e2.y);
    if (A < CC || C < AA || B < DD || D < BB) {
        return {0, {}, {}};
    }
    if (sign(cross(e1 - s1, e2 - s2)) == 0) {
        if (sign(cross(s2, e1, s1)) != 0) {
            return {0, {}, {}};
        }
        Pt p1(max(AA, CC), max(BB, DD));
        Pt p2(min(A, C), min(B, D));
        if (!pointOnSegment(p1, l1)) {
            swap(p1.y, p2.y);
        }
        if (p1 == p2) {
```

```
                return {3, p1, p2};
            } else {
                return {2, p1, p2};
            }
        }
        auto cp1 = cross(s2 - s1, e2 - s1);
        auto cp2 = cross(s2 - e1, e2 - e1);
        auto cp3 = cross(s1 - s2, e1 - s2);
        auto cp4 = cross(s1 - e2, e1 - e2);
        if (sign(cp1 * cp2) == 1 || sign(cp3 * cp4) == 1) {
            return {0, {}, {}};
        }
        // 使用下方函数时请使用浮点数
        Pd p = lineIntersection(l1, l2);
        if (sign(cp1) != 0 && sign(cp2) != 0 && sign(cp3) != 0 && sign(cp4) != 0) {
            return {1, p, p};
        } else {
            return {3, p, p};
        }
}
```

如果不需要求交点，那么使用快速排斥+跨立实验即可，其中重叠、相交于端点均视为相交。

```
template<class T> bool segmentIntersection(Lt l1, Lt l2) {
    auto [s1, e1] = l1;
    auto [s2, e2] = l2;
    auto A = max(s1.x, e1.x), AA = min(s1.x, e1.x);
    auto B = max(s1.y, e1.y), BB = min(s1.y, e1.y);
    auto C = max(s2.x, e2.x), CC = min(s2.x, e2.x);
    auto D = max(s2.y, e2.y), DD = min(s2.y, e2.y);
    return A >= CC && B >= DD && C >= AA && D >= BB &&
           sign(cross(s1, s2, e1) * cross(s1, e1, e2)) == 1 &&
           sign(cross(s2, s1, e2) * cross(s2, e2, e1)) == 1;
}
```

平面圆相关（浮点数处理）

点到圆的最近点

同时返回最近点与最近距离。**需要注意**，当点为圆心时，这样的点有无数个，此时我们视作输入错误，直接返回圆心。

```
pair<Pd, ld> pointToCircle(Pd p, Pd o, ld r) {
    Pd U = o, V = o;
    ld d = dis(p, o);
    if (sign(d) == 0) { // p 为圆心时返回圆心本身
        return {o, 0};
    }
    ld val1 = r * abs(o.x - p.x) / d;
    ld val2 = r * abs(o.y - p.y) / d * ((o.x - p.x) * (o.y - p.y) < 0 ? -1 : 1);
    U.x += val1, U.y += val2;
    V.x -= val1, V.y -= val2;
```

```
    if (dis(U, p) < dis(V, p)) {
        return {U, dis(U, p)};
    } else {
        return {V, dis(V, p)};
    }
}
```

根据圆心角获取圆上某点

将圆上最右侧的点以圆心为旋转中心，逆时针旋转 `rad` 度。

```
Point<ld> getPoint(Point<ld> p, ld r, ld rad) {
    return {p.x + cos(rad) * r, p.y + sin(rad) * r};
}
```

直线是否与圆相交及交点

0 代表不相交；1 代表相切；2 代表相交。

```
tuple<int, Pd, Pd> lineCircleCross(Ld l, Pd o, ld r) {
    Pd P = project(o, l);
    ld d = dis(P, o), tmp = r * r - d * d;
    if (sign(tmp) == -1) {
        return {0, {}, {}};
    } else if (sign(tmp) == 0) {
        return {1, P, {}};
    }
    Pd vec = standardize(l.b - l.a) * sqrt(tmp);
    return {2, P + vec, P - vec};
}
```

线段是否与圆相交及交点

0 代表不相交；1 代表相切；2 代表相交于一个点；3 代表相交于两个点。

```
tuple<int, Pd, Pd> segmentCircleCross(Ld l, Pd o, ld r) {
    auto [type, U, V] = lineCircleCross(l, o, r);
    bool f1 = pointOnSegment(U, l), f2 = pointOnSegment(V, l);
    if (type == 1 && f1) {
        return {1, U, {}};
    } else if (type == 2 && f1 && f2) {
        return {3, U, V};
    } else if (type == 2 && f1) {
        return {2, U, {}};
    } else if (type == 2 && f2) {
        return {2, V, {}};
    } else {
        return {0, {}, {}};
    }
}
```

两圆是否相交及交点

0 代表内含；1 代表相离；2 代表相切；3 代表相交。

```
tuple<int, Pd, Pd> circleIntersection(Pd p1, ld r1, Pd p2, ld r2) {
    ld x1 = p1.x, x2 = p2.x, y1 = p1.y, y2 = p2.y, d = dis(p1, p2);
    if (sign(abs(r1 - r2) - d) == 1) {
        return {0, {}, {}};
    } else if (sign(r1 + r2 - d) == -1) {
        return {1, {}, {}};
    }
    ld a = r1 * (x1 - x2) * 2, b = r1 * (y1 - y2) * 2, c = r2 * r2 - r1 * r1 - d * d;
    ld p = a * a + b * b, q = -a * c * 2, r = c * c - b * b;
    ld cosa, sina, cosb, sinb;
    if (sign(d - (r1 + r2)) == 0 || sign(d - abs(r1 - r2)) == 0) {
        cosa = -q / p / 2;
        sina = sqrt(1 - cosa * cosa);
        Point<ld> p0 = {x1 + r1 * cosa, y1 + r1 * sina};
        if (sign(dis(p0, p2) - r2)) {
            p0.y = y1 - r1 * sina;
        }
        return {2, p0, p0};
    } else {
        ld delta = sqrt(q * q - p * r * 4);
        cosa = (delta - q) / p / 2;
        cosb = (-delta - q) / p / 2;
        sina = sqrt(1 - cosa * cosa);
        sinb = sqrt(1 - cosb * cosb);
        Pd ans1 = {x1 + r1 * cosa, y1 + r1 * sina};
        Pd ans2 = {x1 + r1 * cosb, y1 + r1 * sinb};
        if (sign(dis(ans1, p1) - r2)) ans1.y = y1 - r1 * sina;
        if (sign(dis(ans2, p2) - r2)) ans2.y = y1 - r1 * sinb;
        if (ans1 == ans2) ans1.y = y1 - r1 * sina;
        return {3, ans1, ans2};
    }
}
```

两圆相交面积

上述所言四种相交情况均可计算，之所以不使用三角形面积计算公式是因为在计算过程中会出现“负数”面积（扇形面积与三角形面积的符号关系会随圆的位置关系发生变化），故公式全部重新推导，这里采用的是扇形面积减去扇形内部的那个三角形的面积。

```
ld circleIntersectionArea(Pd p1, ld r1, Pd p2, ld r2) {
    ld x1 = p1.x, x2 = p2.x, y1 = p1.y, y2 = p2.y, d = dis(p1, p2);
    if (sign(abs(r1 - r2) - d) >= 0) {
        return PI * min(r1 * r1, r2 * r2);
    } else if (sign(r1 + r2 - d) == -1) {
        return 0;
    }
    ld theta1 = angle(r1, dis(p1, p2), r2);
    ld area1 = r1 * r1 * (theta1 - sin(theta1 * 2) / 2);
    ld theta2 = angle(r2, dis(p1, p2), r1);
    ld area2 = r2 * r2 * (theta2 - sin(theta2 * 2) / 2);
    return area1 + area2;
}
```

三点确定一圆

```
tuple<int, Pd, ld> getCircle(Pd A, Pd B, Pd C) {
    if (onLine(A, B, C)) { // 特判三点共线
        return {0, {}, 0};
    }
    Ld l1 = midSegment(Line{A, B});
    Ld l2 = midSegment(Line{A, C});
    Pd O = lineIntersection(l1, l2);
    return {1, O, dis(A, O)};
}
```

求解点到圆的切线数量与切点

```
pair<int, vector<Point<ld>>> tangent(Point<ld> p, Point<ld> A, ld r) {
    vector<Point<ld>> ans; // 储存切点
    Point<ld> u = A - p;
    ld d = sqrt(dot(u, u));
    if (d < r) {
        return {0, {}};
    } else if (sign(d - r) == 0) { // 点在圆上
        ans.push_back(u);
        return {1, ans};
    } else {
        ld ang = asin(r / d);
        ans.push_back(getPoint(A, r, -ang));
        ans.push_back(getPoint(A, r, ang));
        return {2, ans};
    }
}
```

求解两圆的内公、外公切线数量与切点

同时返回公切线数量以及每个圆的切点。

```
tuple<int, vector<Point<ld>>, vector<Point<ld>>> tangent(Point<ld> A, ld Ar, Point<ld> B, ld Br) {
```

```
    vector<Point<ld>> a, b; // 储存切点
    if (Ar < Br) {
        swap(Ar, Br);
        swap(A, B);
        swap(a, b);
    }
    int d = disEx(A, B), dif = Ar - Br, sum = Ar + Br;
    if (d < dif * dif) { // 内含，无
        return {0, {}, {}};
    }
    ld base = atan2(B.y - A.y, B.x - A.x);
    if (d == 0 && Ar == Br) { // 完全重合，无数条外公切线
        return {-1, {}, {}};
    }
    if (d == dif * dif) { // 内切，1条外公切线
        a.push_back(getPoint(A, Ar, base));
        b.push_back(getPoint(B, Br, base));
        return {1, a, b};
    }
    ld ang = acos(dif / sqrt(d));
    a.push_back(getPoint(A, Ar, base + ang)); // 保底2条外公切线
    a.push_back(getPoint(A, Ar, base - ang));
    b.push_back(getPoint(B, Br, base + ang));
    b.push_back(getPoint(B, Br, base - ang));
    if (d == sum * sum) { // 外切，多1条内公切线
        a.push_back(getPoint(A, Ar, base));
        b.push_back(getPoint(B, Br, base + PI));
    } else if (d > sum * sum) { // 相离，多2条内公切线
        ang = acos(sum / sqrt(d));
        a.push_back(getPoint(A, Ar, base + ang));
        a.push_back(getPoint(A, Ar, base - ang));
        b.push_back(getPoint(B, Br, base + ang + PI));
        b.push_back(getPoint(B, Br, base - ang + PI));
    }
    return {a.size(), a, b};
}
```

平面三角形相关（浮点数处理）

三角形面积

```
ld area(Point<ld> a, Point<ld> b, Point<ld> c) {
    return abs(cross(b, c, a)) / 2;
}
```

三角形外心

三角形外接圆的圆心，即三角形三边垂直平分线的交点。

```
template<class T> Pt center1(Pt p1, Pt p2, Pt p3) { // 外心
    return lineIntersection(midSegment({p1, p2}), midSegment({p2, p3}));
}
```

三角形内心

三角形内切圆的圆心，也是三角形三个内角的角平分线的交点。其到三角形三边的距离相等。

```
Pd center2(Pd p1, Pd p2, Pd p3) { // 内心
    #define atan2(p) atan2(p.y, p.x) // 注意先后顺序
    Line<ld> U = {p1, {}}, V = {p2, {}};
    ld m, n, alpha;
    m = atan2((p2 - p1));
    n = atan2((p3 - p1));
    alpha = (m + n) / 2;
    U.b = {p1.x + cos(alpha), p1.y + sin(alpha)};
    m = atan2((p1 - p2));
    n = atan2((p3 - p2));
    alpha = (m + n) / 2;
    V.b = {p2.x + cos(alpha), p2.y + sin(alpha)};
    return lineIntersection(U, V);
}
```

三角形垂心

三角形的三条高线所在直线的交点。锐角三角形的垂心在三角形内；直角三角形的垂心在直角顶点上；钝角三角形的垂心在三角形外。

```
Pd center3(Pd p1, Pd p2, Pd p3) { // 垂心
    Ld U = {p1, p1 + rotate(p2, p3)}; // 垂线
    Ld V = {p2, p2 + rotate(p1, p3)};
    return lineIntersection(U, V);
}
```

平面直线方程转换

浮点数计算直线的斜率

一般很少使用到这个函数，因为斜率的取值不可控（例如接近平行于 x, y 轴时）。**需要注意**，当直线平行于 y 轴时斜率为 `inf` 。

```
template <class T> ld slope(Pt p1, Pt p2) { // 斜率，注意 inf 的情况
    return (p1.y - p2.y) / (p1.x - p2.x);
}
template <class T> ld slope(Lt l) {
    return slope(l.a, l.b);
}
```

分数精确计算直线的斜率

调用分数四则运算精确计算斜率，返回最简分数，只适用于整数计算。

```
template<class T> Frac<T> slopeEx(Pt p1, Pt p2) {
    Frac<T> U = p1.y - p2.y;
    Frac<T> V = p1.x - p2.x;
    return U / V; // 调用分数精确计算
}
```

两点式转一般式

返回由三个整数构成的方程，在输入较大时可能找不到较小的满足题意的一组整数解。可以处理平行于 x, y 轴、两点共点的情况。

```
template<class T> tuple<int, int, int> getfun(Lt p) {
    T A = p.a.y - p.b.y, B = p.b.x - p.a.x, C = p.a.x * A + p.a.y * B;
    if (A < 0) { // 符号调整
        A = -A, B = -B, C = -C;
    } else if (A == 0) {
        if (B < 0) {
            B = -B, C = -C;
        } else if (B == 0 && C < 0) {
            C = -C;
        }
    }
    if (A == 0) { // 数值计算
        if (B == 0) {
            C = 0; // 共点特判
        } else {
            T g = fgcd(abs(B), abs(C));
            B /= g, C /= g;
        }
    } else if (B == 0) {
        T g = fgcd(abs(A), abs(C));
        A /= g, C /= g;
    } else {
        T g = fgcd(fgcd(abs(A), abs(B)), abs(C));
        A /= g, B /= g, C /= g;
    }
    return tuple{A, B, C}; // Ax + By = C
}
```

一般式转两点式

由于整数点可能很大或者不存在，故直接采用浮点数；如果与 x, y 轴有交点则取交点。可以处理平行于 x, y 轴的情况。

```
Line<ld> getfun(int A, int B, int C) { // Ax + By = C
    ld x1 = 0, y1 = 0, x2 = 0, y2 = 0;
    if (A && B) { // 正常
        if (C) {
```

```
            x1 = 0, y1 = 1. * C / B;
            y2 = 0, x2 = 1. * C / A;
        } else { // 过原点
            x1 = 1, y1 = 1. * -A / B;
            x2 = 0, y2 = 0;
        }
    } else if (A && !B) { // 垂直
        if (C) {
            y1 = 0, x1 = 1. * C / A;
            y2 = 1, x2 = x1;
        } else {
            x1 = 0, y1 = 1;
            x2 = 0, y2 = 0;
        }
    } else if (!A && B) { // 水平
        if (C) {
            x1 = 0, y1 = 1. * C / B;
            x2 = 1, y2 = y1;
        } else {
            x1 = 1, y1 = 0;
            x2 = 0, y2 = 0;
        }
    } else { // 不合法，请特判
        assert(false);
    }
    return {{x1, y1}, {x2, y2}};
}
```

抛物线与 x 轴是否相交及交点

0 代表没有交点；1 代表相切；2 代表有两个交点。

```
tuple<int, ld, ld> getAns(ld a, ld b, ld c) {
    ld delta = b * b - a * c * 4;
    if (delta < 0.) {
        return {0, 0, 0};
    }
    delta = sqrt(delta);
    ld ans1 = -(delta + b) / 2 / a;
    ld ans2 = (delta - b) / 2 / a;
    if (ans1 > ans2) {
        swap(ans1, ans2);
    }
    if (sign(delta) == 0) {
        return {1, ans2, 0};
    }
    return {2, ans1, ans2};
}
```

SMU_inch

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

#define endl '\n'
#define append push_back
#define pop pop_back
#define list vector
//#include <bits/extc++.h>
using namespace std;
//using namespace __gnu_pbds;
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
const int N = 2e5 + 5, inf = 0x3f3f3f3f, MOD = 998244353, mod = 1e9 + 7;
const ll llinf = 0x3f3f3f3f3f3f3f3f;
//const double PI=acos(-1);
typedef double db;
const db EPS = 1e-9;

// long double的区分精度大约为2^-64,1e-15~1e-18
// double的区分精度大约为2^-53,1e-12~1e-15
//精度问题,求两个1e9内的点的斜率,误差为1e-18

inline int sign(db a) { return a < -EPS ? -1 : a > EPS; }

inline int cmp(db a, db b) { return sign(a - b); }

struct P {
    db x, y;

    P() {}

    P(db _x, db _y) : x(_x), y(_y) {}

    P operator+(P p) { return {x + p.x, y + p.y}; }

    P operator-(P p) { return {x - p.x, y - p.y}; }

    P operator*(db d) { return {x * d, y * d}; }

    P operator/(db d) { return {x / d, y / d}; }

    bool operator<(P p) const {
        int c = cmp(x, p.x);
        if (c)return c == -1;
        return cmp(y, p.y) == -1;
    }

    bool operator==(P o) const {
        //没有传递性
```

```
        return cmp(x, o.x) == 0 && cmp(y, o.y) == 0;
    }


    db dot(P p) { return x * p.x + y * p.y; }//点积, |a|*|b|*cos(an) 结果 大于0,两个向量夹角小于90度;等于0,两个向量夹角等于90度;小于0,两个向量夹角大于90度
    db det(P p) {
        return x * p.y - y * p.x;
    }//叉积, |a|*|b|*sin(an) an为有向角, an为a逆时针旋转多少度到b, a x b = - (b x a). 结果大于0,b在a的逆时针方向;等于0,共线;小于0,b在a的顺时针方向

    db disTo(P p) { return (*this - p).abs(); }//两点距离
    db disTo2(P p) { return (*this - p).abs2(); }//两点距离的平方
    db alpha() { return atan2(y, x); }//求极角
    void readint() {
        int x_, y_;
        cin >> x_ >> y_;
        x = x_, y = y_;
    }//输入整数
    void readdb() { cin >> x >> y; }

    void write() { cout << "(" << x << ", " << y << ")" << endl; }//输出
    db abs() { return sqrt(abs2()); }//原点距离
    db abs2() { return x * x + y * y; }//原点距离的平方
    P rot90() { return P(-y, x); }//原点旋转90
    int quad() const { return sign(y) == 1 || (sign(y) == 0 && sign(x) >= 0); }//判断点在上半边还是下半边
    P unit() { return *this / abs(); }//单位向量

    P rot(db an) {
        return {x * cos(an) - y * sin(an), x * sin(an) + y * cos(an)};
    }// 绕原点旋转an度表示: (x+yi)(cos(an)+sin(an)i)

};

#define cross(p1, p2, p3)((p2.x-p1.x)*(p3.y-p1.y)-(p3.x-p1.x)*(p2.y-p1.y))
#define crossOp(p1, p2, p3) sign(cross(p1,p2,p3))

//如果crossop大于0,表示p1,p2,p3为逆时针关系,小于0表示为顺时针关系,等于0为共线
//也可以解释为p3在p1,p2的上方还是下方,还是p3在直线p1,p2上
int cmp2(P A, P B) { return A.det(B) > 0 || (A.det(B) == 0 && A.abs2() < B.abs2()); }

bool chkLL(P p1, P p2, P q1, P q2) {
    ////两个线段是否平行
    db a1 = cross(q1, q2, p1);
    db a2 = -cross(q1, q2, p2);
    return sign(a1 + a2) != 0;
}

P isLL(P p1, P p2, P q1, P q2) {
    ////求出交点
    db a1 = cross(q1, q2, p1);
    db a2 = -cross(q1, q2, p2);
```

```cpp
    return (p1 * a2 + p2 * a1) / (a1 + a2);
}

bool intersect(db l1, db r1, db l2, db r2) {
    ////判断[l1,r1],[l2,r2]是否相交
    if (l1 > r1) swap(l1, r1);
    if (l2 > r2) swap(l2, r2);
    return !(cmp(r1, l2) == -1 || cmp(r2, l1) == -1);
}

bool isSS(P p1, P p2, P q1, P q2) {
    ////线段是否相交
    return intersect(p1.x, p2.x, q1.x, q2.x) && intersect(p1.y, p2.y, q1.y, q2.y) &&
           crossOp(p1, p2, q1) * crossOp(p1, p2, q2) <= 0 && crossOp(q1, q2, p1) * crossOp(q1, q2, p2) <= 0;
}

bool isSS_strict(P p1, P p2, P q1, P q2) {
    ////线段是否严格相交
    ////严格相交指:只有一个公共点,且不能端点相交,就是一个x的形状
    return crossOp(p1, p2, q1) * crossOp(p1, p2, q2) < 0 && crossOp(q1, q2, p1) * crossOp(q1, q2, p2) < 0;
}

bool isMiddle(db a, db b, db m) {
    ////点m在不在区间[a,b]上
    if (a > b)swap(a, b);
    return cmp(a, m) <= 0 && cmp(m, b) <= 0;
}

bool isMiddle(P a, P b, P m) {
    ////判断直线q1q2和直线p1p2的交点在不在线段p1,p2上,可以调用isMiddle,精度比onSeg更优
    return isMiddle(a.x, b.x, m.x) && isMiddle(a.y, b.y, m.y);
}

bool onSeg(P p1, P p2, P q) {
    ////p在不在线段p1,p2上
    //可能精度有点问题
    return crossOp(p1, p2, q) == 0 && isMiddle(p1, p2, q);
}

bool onSeg_strict(P p1, P p2, P q) {
    ////p是不是严格在线段p1,p2上
    return crossOp(p1, p2, q) == 0 && sign((q - p1).dot(p1 - p2)) * sign((q - p2).dot(p1 - p2)) < 0;
}

P proj(P p1, P p2, P q) {
    ////求q到p1p2的垂足,且p1!=p2
    if (p1 == p2)return p1;
    P dir = p2 - p1;
    return p1 + dir * (dir.dot(q - p1) / dir.abs2());
}
```

```

P reflect(P p1, P p2, P q) {
    ////求q关于p1p2的反射
    return proj(p1, p2, q) * 2 - q;
}

db nearest(P p1, P p2, P q) {
    ////求q到线段p1p2的最小距离
    if (p1 == p2)return p1.disTo(q);
    P h = proj(p1, p2, q);
    if (isMiddle(p1, p2, h))return q.disTo(h);
    return min(p1.disTo(q), p2.disTo(q));
}

db disSS(P p1, P p2, P q1, P q2) {
    ////求线段p1p2到q1q2的距离
    if (isSS(p1, p2, q1, q2))return 0;
    return min(min(nearest(p1, p2, q1), nearest(p1, p2, q1)), min(nearest(q1, q2, p1),
nearest(q1, q2, p2)));
}
//极角排序
/*
sort(p,p+n,[&](const P &a,const P &b){
    int qa = a.quad(),qb=b.quad();
    if(qa!=qb) return qa<qb;
    return sign(a.det(b)) > 0;
});
*/
bool cmp1(P a, const P &b) {
    int qa = a.quad(), qb = b.quad();
    if (qa != qb) return qa < qb;
    return sign(a.det(b)) > 0;
}

int type(P o1, db r1, P o2, db r2) {
    ///求两个圆的关系
    /// 4 : 相离
    /// 3 : 外切
    /// 2 : 相交
    /// 1 : 内切
    /// 0 : 内含
    db d = o1.disTo(o2);
    if (cmp(d, r1 + r2) == 1) return 4;
    if (cmp(d, r1 + r2) == 0) return 3;
    if (cmp(d, abs(r1 - r2)) == 1) return 2;
    if (cmp(d, abs(r1 - r2)) == 0) return 1;
    return 0;
}

vector<P> isCL(P o, db r, P p1, P p2) {
    ///求圆和直线的交点,返回的两个点属于p1->p2方向
    if (cmp(abs((o - p1).det(p2 - p1) / p1.disTo(p2)), r) > 0) return {};
```

```
    db x = (p1 - o).dot(p2 - p1), y = (p2 - p1).abs2(), d = x * x - y * ((p1 - o).abs2() - r * r);
    d = max(d, (db) 0.0);
    P m = p1 - (p2 - p1) * (x / y), dr = (p2 - p1) * (sqrt(d) / y);
    return {m - dr, m + dr};
}

vector<P> isCC(P o1, db r1, P o2, db r2) {
    ///两个圆的交点,需要判断两个圆是否全等
    ///返回的交点沿着第一个圆的逆时针方向
    db d = o1.disTo(o2);
    if (cmp(d, r1 + r2) == 1)return {};
    if (cmp(d, abs(r1 - r2)) == -1)return {};
    d = min(d, r1 + r2);
    db y = (r1 * r1 + d * d - r2 * r2) / (2 * d), x = sqrt(r1 * r1 - y * y);
    P dr = (o2 - o1).unit();
    P q1 = o1 + dr * y, q2 = dr.rot90() * x;
    return {q1 - q2, q1 + q2};
}

vector<pair<P, P>> tanCC(P o1, db r1, P o2, db r2) {
    ///两个圆的外切线,如果需要内切线,把r2传入负值即可,如果需要点到圆的切线,把r2传为0即可
    P d = o2 - o1;
    db dr = r1 - r2, d2 = d.abs2(), h2 = d2 - dr * dr;
    if (sign(d2) == 0 || sign(h2) < 0)return {};
    h2 = max((db) 0.0, h2);
    vector<pair<P, P>> ret;
    for (db sign: {-1, 1}) {
        P v = (d * dr + d.rot90() * sqrt(h2) * sign) / d2;
        ret.push_back({o1 + v * r1, o2 + v * r2});
    }
    if (sign(h2) == 0)ret.pop_back();
    return ret;
}

db rad(P p1, P p2) {
    ///求两个向量的夹角弧度
    return atan2l(p1.det(p2), p1.dot(p2));
}

db areaCT(P o, db r, P p1, P p2) {
    ///圆和其中一个顶点是圆心的三角形的面积交,返回有向面积
    p1 = p1 - o;
    p2 = p2 - o;
    vector<P> is = isCL(P(0, 0), r, p1, p2);
    if (is.empty()) return r * r * rad(p1, p2) / 2;
    bool b1 = cmp(p1.abs2(), r * r) == 1, b2 = cmp(p2.abs2(), r * r) == 1;
    if (b1 && b2) {
        P md = (is[0] + is[1]) / 2;
        if (sign((p1 - md).dot(p2 - md)) <= 0)
            return r * r * (rad(p1, is[0]) + rad(is[1], p2)) / 2 + is[0].det(is[1]) / 2;
        else return r * r * rad(p1, p2) / 2;
```

```
    }
    if (b1) return (r * r * rad(p1, is[0]) + is[0].det(p2)) / 2;
    if (b2) return (p1.det(is[1]) + r * r * rad(is[1], p2)) / 2;
    return p1.det(p2) / 2;
}


P inCenter(P A, P B, P C) {
    ///三角形内心
    double a = (B - C).abs(), b = (C - A).abs(), c = (A - B).abs();
    return (A * a + B * b + C * c) / (a + b + c);
}

P circumCenter(P a, P b, P c) {
    ///三角形外心
    P bb = b - a, cc = c - a;
    double db = bb.abs2(), dc = cc.abs2(), d = 2 * bb.det(cc);
    return a - P(bb.y * dc - cc.y * db, cc.x * db - bb.x * dc) / d;
}

P othroCenter(P a, P b, P c) {
    ///三角形垂心
    P ba = b - a, ca = c - a, bc = b - c;
    double Y = ba.y * ca.y * bc.y,
           A = ca.x * ba.y - ba.x * ca.y,
           x0 = (Y + ca.x * ba.y * b.x - ba.x * ca.y * c.x) / A,
           y0 = -ba.x * (x0 - c.x) / ba.y + ca.y;
    return {x0, y0};
}

pair<P, db> min_circle(vector<P> ps) {
    ///最小圆覆盖,给定若干个点,求最小的一个圆能够覆盖这些点,复杂度为O(n)
    random_shuffle(ps.begin(), ps.end());
    int n = ps.size();
    P o = ps[0];
    db r = 0;
    for (int i = 1; i < n; ++i) {
        if (o.disTo(ps[i]) > r + EPS)
            o = ps[i], r = 0;
        for (int j = 0; j < i; ++j)
            if (o.disTo(ps[j]) > r + EPS) {
                o = (ps[i] + ps[j]) / 2;
                r = o.disTo(ps[i]);
                for (int k = 0; k < j; ++k)
                    if (o.disTo(ps[k]) > r + EPS) {
                        o = circumCenter(ps[i], ps[j], ps[k]);
                        r = o.disTo(ps[i]);
                    }
            }
    }
    return {o, r};
}

```

```cpp

db area(vector<P> ps) {
    ////计算多边形面积
    db ret = 0;
    int n = ps.size();
    for (int i = 0; i < ps.size(); ++i) {
        ret += ps[i].det(ps[(i + 1) % n]);
    }
    return ret / 2;
}


int containP(const vector<P> &ps, P p) {
    ////判断点是否在多边形内部
    ////如果返回 0:不在内部;1:在边界上;2:在内部
    int n = ps.size(), ret = 0;
    for (int i = 0; i < n; ++i) {
        P u = ps[i], v = ps[(i + 1) % n];
        if (onSeg(u, v, p)) return 1;
        if (cmp(u.y, v.y) <= 0) swap(u, v);
        if (cmp(p.y, u.y) > 0 || cmp(p.y, v.y) <= 0)continue;
        ret ^= crossOp(p, u, v) > 0;
    }
    return ret * 2;
}


vector<P> convexHull(vector<P> ps) {
    ////求严格凸包
    int n = ps.size();
    if (n <= 1)return ps;
    sort(ps.begin(), ps.end());
    vector<P> qs(n * 2);
    int k = 0;
    for (int i = 0; i < n; qs[k++] = ps[i++]) {//求下凸壳
        while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)--k;
    }
    for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--]) {//求上凸壳
        while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)--k;
    }
    qs.resize(k - 1);
    return qs;
}

vector<P> convexHullnonstrict(vector<P> ps) {
    ////求不严格凸包,需要先去重
    int n = ps.size();
    if (n <= 1)return ps;
    sort(ps.begin(), ps.end());
    vector<P> qs(n * 2);
    int k = 0;
    for (int i = 0; i < n; qs[k++] = ps[i++]) {//求下凸壳
        while (k > 1 && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)--k;
```

```
    }
    for (int i = n - 2, t = k; i >= 0; qs[k++] = ps[i--]) {//求上凸壳
        while (k > t && crossOp(qs[k - 2], qs[k - 1], ps[i]) <= 0)--k;
    }
    qs.resize(k - 1);
    return qs;
}

db convexDiamter(vector<P> ps) {
    ////求凸包最大直径
    int n = ps.size();
    if (n <= 1)return 0;
    int is = 0;
    int js = 0;
    for (int k = 1; k < n; ++k) {
        is = ps[k] < ps[is] ? k : is, js = ps[js] < ps[k] ? k : js;
    }
    int i = is, j = js;
    db ret = ps[i].disTo(ps[j]);
    do {
        if ((ps[(i + 1) % n] - ps[i]).det(ps[(j + 1) % n] - ps[j]) >= 0)
            (++j) %= n;
        else
            (++i) %= n;
        ret = max(ret, ps[i].disTo(ps[j]));
    } while (i != is || j != js);
    return ret;
}


vector<P> convexCut(const vector<P> &ps, P q1, P q2) {
    ////用直线切割ps,返回切线左边的点以及交点
    vector<P> qs;
    int n = ps.size();
    for (int i = 0; i < n; ++i) {
        P p1 = ps[i], p2 = ps[(i + 1) % n];
        int d1 = crossOp(q1, q2, p1), d2 = crossOp(q1, q2, p2);
        if (d1 >= 0) qs.push_back(p1);
        if (d1 * d2 < 0) qs.push_back(isLL(p1, p2, q1, q2));
    }
    return qs;
}

vector<P> isLD(const vector<P> &ps, P q1, P q2) {
    ////返回直线和多边形的所有交点
    int n = ps.size();
    vector<P> qs;
    for (int i = 0; i < n; ++i) {
        if (crossOp(q1, q2, ps[i]) == 0)qs.push_back(ps[i]);
        if (crossOp(q1, q2, ps[i]) * crossOp(q1, q2, ps[(i + 1) % n]) < 0)
            qs.push_back(isLL(q1, q2, ps[i], ps[(i + 1) % n]));
    }
    sort(qs.begin(), qs.end());
```

```
    qs.erase(unique(qs.begin(), qs.end()), qs.end());
    return qs;
}

vector<P> isSD(const vector<P> &ps, P q1, P q2) {
    ////返回直线和多边形的所有交点
    int n = ps.size();
    vector<P> qs;
    qs.push_back(q1);
    qs.push_back(q2);
    for (int i = 0; i < n; ++i) {
        if (crossOp(q1, q2, ps[i]) == 0)qs.push_back(ps[i]);
        if (crossOp(q1, q2, ps[i]) * crossOp(q1, q2, ps[(i + 1) % n]) < 0)
            qs.push_back(isLL(q1, q2, ps[i], ps[(i + 1) % n]));
    }
    sort(qs.begin(), qs.end());
    qs.erase(unique(qs.begin(), qs.end()), qs.end());
    int s = -1, t = -1;
    for (int i = 0; i < n; ++i) {
        if (q1 == qs[i])s = i;
        if (q2 == qs[i])t = i;
    }
    if (s > t)swap(s, t);
    vector<P> ks;
    for (int i = s; i < t; ++i) {
        ks.push_back(qs[i]);
    }
    return ks;
}


bool containSeg(vector<P> ps, P p1, P p2) {
    ////判断线段是否在内部
    vector<P> qs = isSD(ps, p1, p2);
    int n = qs.size();
    for (int i = 0; i < n - 1; ++i) {
        P m = (qs[i] + qs[i + 1]) / 2;
        if (containP(qs, m) == 0)return false;
    }
    return true;
}

vector<P> Minkowski(vector<P> A, vector<P> B) {
    vector<P> C(A.size() + B.size() + 1), v1(A.size()), v2(B.size());
    for (int i = 0; i < (int) A.size(); i++)v1[i] = A[(i + 1) % A.size()] - A[i];
    for (int i = 0; i < (int) B.size(); i++)v2[i] = B[(i + 1) % B.size()] - B[i];
    int cnt = 0;
    C[cnt] = (A[0] + B[0]);
    int p1 = 0, p2 = 0;
    while (p1 < (int) A.size() && p2 < (int) B.size()) {
        ++cnt;
        if (sign(v1[p1].det(v2[p2])) >= 0)
            C[cnt] = C[cnt - 1] + v1[p1++];
```

```
        else
            C[cnt] = C[cnt - 1] + v2[p2++];
    }
    while (p1 < (int) A.size()) {
        ++cnt;
        C[cnt] = C[cnt - 1] + v1[p1++];
    }
    while (p2 < (int) B.size()) {
        ++cnt;
        C[cnt] = C[cnt - 1] + v2[p2++];
    }
    return C;
}

bool containPs(const vector<P> &ts, P q) {
    ///判断点集是否在线段内,要保证ps[0]={0,0};
    int ps = upper_bound(ts.begin(), ts.end(), q, cmp2) - ts.begin() - 1;
    return (crossOp(ts[ps], ts[(ps + 1) % ts.size()], q) >= 0);
}


void solve() {


}


int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
//    freopen(".\\Template\\CHECK\\data.in", "r", stdin);
//    freopen(".\\Template\\CHECK\\std.out", "w", stdout);
    int cases;
    cin >> cases;
    while (cases--)
        solve();
}
```