基础

点(Point)

```
struct Point {
    db x, y;
};

inline db dis(const Point &a, const Point &b) {
    db dx = a.x - b.x;
    db dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
```

$$x\prime = x \cdot cos\theta - y \cdot sin\theta$$

$$y\prime = x \cdot sin\theta + y \cdot cos\theta$$

向量(Vector)

```
using Vector = Point;

inline db dot(const Vector &a, const Vector &b) {
    return a.x * b.x + a.y * b.y;
}

inline db cross(const Vector &a, const Vector &b) {
    return a.x * b.y - a.y * b.x;
}

inline Vector operator+(const Point &a, const Point &b) {
    return Vector{a.x + b.x, a.y + b.y};
}

inline Vector operator-(const Point &a, const Point &b) {
    return Vector{a.x - b.x, a.y - b.y};
}

inline Vector operator*(const Vector &a, const db &b) {
    return Vector{a.x * b, a.y * b};
}
```

基础用法

- 将一个向量$\vec{a}$逆时针旋转θ度

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} a_x \\ b_x \end{bmatrix} = \begin{bmatrix} \cos\theta\, a_x & - & \sin\theta\, a_y \\ \sin\theta\, a_x & + & \cos\theta\, a_y \end{bmatrix}$$

点积(Dot)

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y$$

叉积(Cross)

$$\vec{a} \times \vec{b} = a_x b_y - a_y b_x$$

- 平行四边形面积: $\|\vec{a}\|\|\vec{b}\| \, |\sin\theta| = \|\vec{a} \times \vec{b}\|$
- 向量平行: $\vec{a} \times \vec{b} = 0$
- **TO_LEFT测试**
    - 判断点P在向量AB的左侧还是右侧
    - $\vec{a} \times \vec{b} > 0$ P在向量AB左侧
    - $\vec{a} \times \vec{b} < 0$ P在向量AB右侧
    - $\vec{a} \times \vec{b} = 0$ P在向量AB上

线段(Segment)

```cpp
struct Segment {
    Point a, b;
};
```

基础用法

- 判断点P是否在线段AB上(含端点)
    - $\vec{a} \times \vec{b} = \vec{0}$
    - $\vec{a} \cdot \vec{b} \leq 0$
- 判断线段AB，CD是否相交
    - 特判三点共线和四点共线
    - 通过叉积判断
    - 点C和点D在线段AB的不同侧
    - 点A和点B在线段CD的不同侧

直线

点向式 `struct Line { Point p; Vector v; };`

基础用法

- 输入直线$(P,\vec{v})$与点A，求A到直线距离

$$\|\vec{AB}\| = \|\vec{PA}\| \, |\sin\theta| = \frac{\|\vec{v} \times \vec{PA}\|}{\|\vec{v}\|}$$

- 输入直线$(P,\vec{v})$与点A，求A在直线上的投影点B 点乘算投影
- 两直线求交点 直线 $\{P_1,\vec{v}_1\}\{P_2,\vec{v}_2\}$

$$\begin{cases} \dfrac{\|P_1Q\|}{\sin\alpha} = \dfrac{\|P_1P_2\|}{\sin\beta} \\ \|\vec{v}_2 \times \vec{P_2P_1}\| = \|\vec{v}_2\|\|\vec{P_2P_1}\|\sin\alpha \end{cases}$$

$$\|\vec{v}_1 \times \vec{v}_2\| = \|\vec{v}_1\|\|\vec{v}_2\|\sin\beta$$

$$\|P_1Q\| = \frac{\|\vec{v}_2 \times \vec{P_2P_1}\|\|\vec{v}_1\|}{\|\vec{v}_1 \times \vec{v}_2\|}$$

$$\vec{OQ} = \vec{OP_1} + \vec{P_1Q} = \vec{OP_1} + \frac{\|P_1Q\|}{\|\vec{v}_1\|}\vec{v}_1 = \vec{OP_1} + \frac{\|\vec{v}_2 \times \vec{P_2P_1}\|}{\|\vec{v}_1 \times \vec{v}_2\|}\vec{v}_1$$

```
inline Point l_to_l(const Line &l1, const Line &l2) {
    Vector w = l2.p - l1.p; // P2 - P1
    db denom = cross(l1.v, l2.v);
    if (fabs(denom) < 1e-9) return {1e18, 1e18};
    db t = cross(w, l2.v) / denom;
    return {l1.p.x + t * l1.v.x, l1.p.y + t * l1.v.y};
}
```

- 判断射线与线段是否相交

```
// 射线 r 与线段 s 是否相交
inline bool r_to_s(const Line &r, const Seg &s) {
    Vector d = r.v; // 射线方向
    Vector v = s.b - s.a; // 线段方向
    Vector w = r.p - s.a; // 射线起点到线段起点向量

    db denom = cross(v, d);
    if (fabs(denom) < eps) return false;

    db t = cross(w, v) / denom;
    db u = cross(w, d) / denom;

    return t >= -eps && u >= -eps && u <= 1 + eps;
}
```

多边形

`struct Polygon { vector<Point> p; };` 一般默认按照逆时针排序

基础用法

- 计算多边形面积(三角剖分) $S = \frac{1}{2}\|\sum_{i=0}^{n-1}\vec{OP_I} \times \vec{OP_{(i+1)\,mod\,n}}\|$
- 判断点是否在多边形内部
    1. 从该点引出一条射线，如果与多边形有奇数个交点，则在内部，否则在多边形外部 (不能交到顶点)
    2. 遍历多边形的点，如果转动圈数为0，点在多边形外部，否则在内部 (计算角度有精度误差)
    3. 水平引出一条射线，逆时针依次遍历边，如果边从上向下穿过射线，val--，否则val++，如果val=0则点在多边形外 (优秀)

```

```

模版

点(Point)

```
constexpr db eps = 1e-9;
const db pi = acos(-1);

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

    bool operator==(const point &a) const { return (abs(x - a.x) <= eps && abs(y - a.y) <= eps); }
    point operator+(const point &a) const { return {x + a.x, y + a.y}; }
    point operator-(const point &a) const { return {x - a.x, y - a.y}; }
    point operator-() const { return {-x, -y}; }
    point operator*(const T k) const { return {k * x, k * y}; }
    point operator/(const T k) const { return {x / k, y / k}; }
    T operator*(const point &a) const { return x * a.x + y * a.y; } // Dot
    T operator^(const point &a) const { return x * a.y - y * a.x; } // Cross
    bool operator<(const point &a) const {
        if (abs(x - a.x) <= eps) return y < a.y - eps;
        return x < a.x - eps;
    }

    bool is_par(const point &a) const { return abs((*this) ^ a) <= eps; } // 平行
    bool is_ver(const point &a) const { return abs((*this) * a) <= eps; } // 垂直

    int toleft(const point &a) const {
        auto t = (*this) ^ a;
        return (t > eps) - (t < -eps);
    }


    T len2() const { return (*this) * (*this); }
    T dis2(const point &a) const { return (a - (*this)).len2(); }
    double len() const { return sqrt(len2()); }
    double dis(const point &a) const { return (a - (*this)).len(); }
    double ang(const point &a) const { return acos(((*this) * a) / (this->len() * a.len())); } // 普通夹角（只返回大小 [0, π]）
    double signed_ang(const point &a) const { return atan2((*this) ^ a, (*this) * a); } // 带方向夹角：逆时针为正，顺时针为负，范围 (-π, π]
    point rot(const double rad) const { return {x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)}; }


    point rot(const long double &sinr) const {
        const long double cosr = sqrt(1 - sinr * sinr);
        return Point(x * cosr - y * sinr, x * sinr + y * cosr);
    }
};
```

```cpp

using Point = point<double>;

bool is_on(const Point p, const Point a, const Point b) {
    return abs((p - a) ^ (p - b)) <= eps && (p - a) * (p - b) <= eps;
}

bool argcmp(const Point &a, const Point &b) {
    auto quad = [](const Point &a) {
        if (a.y < -eps) return 1;
        if (a.y > eps) return 4;
        if (a.x > eps) return 3;
        if (a.x < -eps) return 5;
        return 2;
    };
    int x = quad(a), y = quad(b);
    if (x != y) return x < y;
    auto res = a ^ b;
    // if (abs(res) < eps) return a * a < b * b - eps; // 相同位置按距离原点长度排序
    return res > eps;
}

```

线(Line)

```cpp
template<typename T>
struct line {
    point<T> p, v; //p+kv

    bool operator==(const line &a) const { return v.is_par(a.v) && v.is_par(p - a.p); }
    bool is_par(const line &a) const { return v.is_par(a.v) && !v.is_par(p - a.p); } //排除共线
    bool is_ver(const line &a) const { return v.is_ver(a.v); }
    bool is_on(const point<T> &a) const { return v.is_par(a - p); }
    int toleft(const point<T> &a) const { return v.toleft(a - p); }
    point<T> inter(const line &a) const { return p + v * ((a.v ^ (p - a.p)) / (v ^ a.v)); }
    double dis(const point<T> &a) const { return abs(v ^ (a - p)) / v.len(); }
    point<T> proj(const point<T> &a) const { return p + v * ((v * (a - p)) / (v * v)); }

    bool operator<(const line &a) const {
        if (abs(v ^ a.v) <= eps && v * a.v >= -eps) return toleft(a.p) == -1;
        return argcmp(v, a.v);
    }
};

using Line = line<double>;
```

多边形(Polygon)

```cpp
template<typename T>
struct Polygon {
    vector<Point<T> > p;

    size_t nxt(const size_t i) const { return i == p.size() - 1 ? 0 : i + 1; }
    size_t pre(const size_t i) const { return i == 0 ? p.size() - 1 : i - 1; }

    // 计算绕多边形一圈转了几圈
    pair<bool, int> winding(const Point<T> &a) const {
        int cnt = 0;
        for (size_t i = 0; i < p.size(); i++) {
            Point<T> u = p[i], v = p[nxt(i)];
            if (is_on(a, u, v)) return {true, 0};
            if (abs(u.y - v.y) <= eps) continue;
            Line<T> uv = {u, v - u};
            if (u.y < v.y - eps && uv.toleft(a) <= 0) continue;
            if (u.y > v.y + eps && uv.toleft(a) >= 0) continue;
            if (u.y < a.y - eps && v.y >= a.y - eps) cnt++;
            if (u.y >= a.y - eps && v.y < a.y - eps) cnt--;
        }
        return {false, cnt};
    }
};
```

计算一条直线在一个多边形内的最长直线

```cpp
int n;
Polygon<db> poly;
set<pair<Point<db>, Point<db> > > edges;

template<typename T>
db calc(const Line<T> &l) {
    vector<tuple<Point<db>, Point<db>, Point<db> > > vec;
    for (int i = 0; i < n; i++) {
        auto u = poly.p[i], v = poly.p[poly.nxt(i)];
        int c1 = l.toleft(u), c2 = l.toleft(v);
        if (c1 * c2 <= 0) {
            if (c1 == 0 && c2 == 0) {
                vec.emplace_back(u, u, v);
                vec.emplace_back(v, u, v);
            } else {
                auto s = l.inter({u, u - v});
                vec.emplace_back(s, u, v);
            }
        }
    }
    sort(vec.begin(), vec.end());
    int cnt = 0;
    Point pre = {1e12, 1e12};
```

```
    db len = 0, maxlen = 0;
    while (!vec.empty()) {
        auto [now,u,v] = vec.back();
        if (cnt || edges.count({now, pre})) {
            len += now.dis(pre);
        } else {
            maxlen = max(maxlen, len);
            len = 0;
        }
        while (!vec.empty() && get<0>(vec.back()) == now) {
            auto [p,u,v] = vec.back();
            vec.pop_back();
            if (l.toleft(u) == -1) cnt++;
            else if (l.toleft(v) == -1) cnt--;
        }
        pre = now;
    }
    return max(maxlen, len);
}
```

V图

```
std::vector<line> cut(const std::vector<line> & o, line l) {
    std::vector<line> res;
    int n = size(o);
    for(int i = 0;i < n;++i) {
        line a = o[i], b = o[(i + 1) % n], c = o[(i + 2) % n];
        int va = check(a, b, l), vb = check(b, c, l);
        if(va > 0 || vb > 0 || (va == 0 && vb == 0)) {
            res.push_back(b);
        }
        if(va >= 0 && vb < 0) {
            res.push_back(l);
        }
    }
    return res;
}
std::vector<std::vector<line>> voronoi(std::vector<vec2> p) {
    int n = p.size();
    auto b = p; shuffle(b.begin(), b.end(), gen);
    const db V = 3e4;
    std::vector<std::vector<line>> a(n, {
        {V, 0, V * V}, {0, V, V * V},
            {-V, 0, V * V}, {0, -V, V * V},
    });
    for(int i = 0;i < n;++i) {
        for(vec2 x : b) if((x - p[i]).abs() > eps) {
            a[i] = cut(a[i], bisector(p[i], x));
        }
    }
    return a;
}
```

求角度

```
ldb get_angle(const Point &a, const Point &b) {
    return atan2(a.x * b.y - a.y * b.x, a.x * b.x + a.y * b.y);
}
```

jjgg

```cpp
#include <bits/stdc++.h>
using std::numeric_limits;
using std::abs, std::max, std::min, std::swap;
using std::pair, std::make_pair;
using std::tuple, std::make_tuple;
using std::vector, std::deque;
using std::set, std::multiset;

using T = long double; //全局数据类型



// 点与向量
struct Point {
    T x, y;

    bool operator==(const Point &a) const { return (abs(x - a.x) <= eps && abs(y - a.y) <= eps); }

    bool operator<(const Point &a) const {
        if (abs(x - a.x) <= eps) return y < a.y - eps;
        return x < a.x - eps;
    }

    bool operator>(const Point &a) const { return !(*this < a || *this == a); }
    Point operator+(const Point &a) const { return {x + a.x, y + a.y}; }
    Point operator-(const Point &a) const { return {x - a.x, y - a.y}; }
    Point operator-() const { return {-x, -y}; }
    Point operator*(const T k) const { return {k * x, k * y}; }
    Point operator/(const T k) const { return {x / k, y / k}; }
    T operator*(const Point &a) const { return x * a.x + y * a.y; } // 点积
    T operator^(const Point &a) const { return x * a.y - y * a.x; } // 叉积，注意优先级
    int toleft(const Point &a) const {
        const auto t = (*this) ^ a;
        return (t > eps) - (t < -eps);
    } // to-left 测试
    T len2() const { return (*this) * (*this); } // 向量长度的平方
    T dis2(const Point &a) const { return (a - (*this)).len2(); } // 两点距离的平方
    int quad() const // 象限判断 0:原点 1:x轴正 2:第一象限 3:y轴正 4:第二象限 5:x轴负 6:第三象限 7:y轴负 8:第四象限
    {
        if (abs(x) <= eps && abs(y) <= eps) return 0;
        if (abs(y) <= eps) return x > eps ? 1 : 5;
        if (abs(x) <= eps) return y > eps ? 3 : 7;
        return y > eps ? (x > eps ? 2 : 4) : (x > eps ? 8 : 6);
    }

    // 必须用浮点数
    T len() const { return sqrtl(len2()); } // 向量长度
```

```cpp
    T dis(const Point &a) const { return sqrtl(dis2(a)); } // 两点距离
    T ang(const Point &a) const { return acosl(max(-1.0l, min(1.0l, ((*this) * a) / (len() * a.len())))); } // 向量夹角
    Point rot(const T rad) const { return {x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)}; } // 逆时针旋转（给定角度）
    Point rot(const T cosr, const T sinr) const { return {x * cosr - y * sinr, x * sinr + y * cosr}; }
    // 逆时针旋转（给定角度的正弦与余弦）
};

// 极角排序
struct Argcmp {
    bool operator()(const Point &a, const Point &b) const {
        const int qa = a.quad(), qb = b.quad();
        if (qa != qb) return qa < qb;
        const auto t = a ^ b;
        // if (abs(t)<=eps) return a*a<b*b-eps;  // 不同长度的向量需要分开
        return t > eps;
    }
};

// 直线
struct Line {
    Point p, v; // p 为直线上一点，v 为方向向量

    bool operator==(const Line &a) const { return v.toleft(a.v) == 0 && v.toleft(p - a.p) == 0; }
    int toleft(const Point &a) const { return v.toleft(a - p); } // to-left 测试
    bool operator<(const Line &a) const // 半平面交算法定义的排序
    {
        if (abs(v ^ a.v) <= eps && v * a.v >= -eps) return toleft(a.p) == -1;
        return Argcmp()(v, a.v);
    }

    // 必须用浮点数
    Point inter(const Line &a) const { return p + v * ((a.v ^ (p - a.p)) / (v ^ a.v)); } // 直线交点
    T dis(const Point &a) const { return abs(v ^ (a - p)) / v.len(); } // 点到直线距离
    Point proj(const Point &a) const { return p + v * ((v * (a - p)) / (v * v)); } // 点在直线上的投影
};

//线段
struct Segment {
    Point a, b;

    bool operator<(const Segment &s) const { return make_pair(a, b) < make_pair(s.a, s.b); }

    // 判定性函数建议在整数域使用

    // 判断点是否在线段上
    // -1 点在线段端点 | 0 点不在线段上 | 1 点严格在线段上
```

```cpp
    int is_on(const Point &p) const {
        if (p == a || p == b) return -1;
        return (p - a).toleft(p - b) == 0 && (p - a) * (p - b) < -eps;
    }

    // 判断线段直线是否相交
    // -1 直线经过线段端点 | 0 线段和直线不相交 | 1 线段和直线严格相交
    int is_inter(const Line &l) const {
        if (l.toleft(a) == 0 || l.toleft(b) == 0) return -1;
        return l.toleft(a) != l.toleft(b);
    }

    // 判断两线段是否相交
    // -1 在某一线段端点处相交 | 0 两线段不相交 | 1 两线段严格相交
    int is_inter(const Segment &s) const {
        if (is_on(s.a) || is_on(s.b) || s.is_on(a) || s.is_on(b)) return -1;
        const Line l{a, b - a}, ls{s.a, s.b - s.a};
        return l.toleft(s.a) * l.toleft(s.b) == -1 && ls.toleft(a) * ls.toleft(b) == -1;
    }

    // 点到线段距离（必须用浮点数）
    T dis(const Point &p) const {
        if ((p - a) * (b - a) < -eps || (p - b) * (a - b) < -eps) return min(p.dis(a), p.dis(b));
        const Line l{a, b - a};
        return l.dis(p);
    }

    // 两线段间距离（必须用浮点数）
    T dis(const Segment &s) const {
        if (is_inter(s)) return 0;
        return min({dis(s.a), dis(s.b), s.dis(a), s.dis(b)});
    }
};

// 多边形
struct Polygon {
    vector<Point> p; // 以逆时针顺序存储

    size_t nxt(const size_t i) const { return i == p.size() - 1 ? 0 : i + 1; }
    size_t pre(const size_t i) const { return i == 0 ? p.size() - 1 : i - 1; }

    // 回转数
    // 返回值第一项表示点是否在多边形边上
    // 对于狭义多边形，回转数为 0 表示点在多边形外，否则点在多边形内
    pair<bool, int> winding(const Point &a) const {
        int cnt = 0;
        for (size_t i = 0; i < p.size(); i++) {
            const Point u = p[i], v = p[nxt(i)];
            if (abs((a - u) ^ (a - v)) <= eps && (a - u) * (a - v) <= eps) return {true, 0};
            if (abs(u.y - v.y) <= eps) continue;
```

```cpp
            const Line uv = {u, v - u};
            if (u.y < v.y - eps && uv.toleft(a) <= 0) continue;
            if (u.y > v.y + eps && uv.toleft(a) >= 0) continue;
            if (u.y < a.y - eps && v.y >= a.y - eps) cnt++;
            if (u.y >= a.y - eps && v.y < a.y - eps) cnt--;
        }
        return {false, cnt};
    }

    // 多边形面积的两倍
    // 可用于判断点的存储顺序是顺时针或逆时针
    T area() const {
        T sum = 0;
        for (size_t i = 0; i < p.size(); i++) sum += p[i] ^ p[nxt(i)];
        return sum;
    }

    // 多边形的周长
    T circ() const {
        T sum = 0;
        for (size_t i = 0; i < p.size(); i++) sum += p[i].dis(p[nxt(i)]);
        return sum;
    }
};

//凸多边形
struct Convex : Polygon {
    // 闵可夫斯基和
    Convex operator+(const Convex &c) const {
        const auto &p = this->p;
        vector&lt;Segment&gt; e1(p.size()), e2(c.p.size()), edge(p.size() + c.p.size());
        vector<Point> res;
        res.reserve(p.size() + c.p.size());
        const auto cmp = [](const Segment &u, const Segment &v) { return Argcmp()(u.b - u.a, v.b - v.a); };
        for (size_t i = 0; i < p.size(); i++) e1[i] = {p[i], p[this->nxt(i)]};
        for (size_t i = 0; i < c.p.size(); i++) e2[i] = {c.p[i], c.p[c.nxt(i)]};
        rotate(e1.begin(), min_element(e1.begin(), e1.end(), cmp), e1.end());
        rotate(e2.begin(), min_element(e2.begin(), e2.end(), cmp), e2.end());
        merge(e1.begin(), e1.end(), e2.begin(), e2.end(), edge.begin(), cmp);
        const auto check = [](const vector<Point> &res, const Point &u) {
            const auto back1 = res.back(), back2 = *prev(res.end(), 2);
            return (back1 - back2).toleft(u - back1) == 0 && (back1 - back2) * (u - back1) >= -eps;
        };
        auto u = e1[0].a + e2[0].a;
        for (const auto &v: edge) {
            while (res.size() > 1 && check(res, u)) res.pop_back();
            res.push_back(u);
            u = u + v.b - v.a;
        }
        if (res.size() > 1 && check(res, res[0])) res.pop_back();
        return {res};
```

```cpp
    }

    // 旋转卡壳
    // 例：凸多边形的直径的平方
    T rotcaliper() const {
        const auto &p = this->p;
        if (p.size() == 1) return 0;
        if (p.size() == 2) return p[0].dis2(p[1]);
        const auto area = [](const Point &u, const Point &v, const Point &w) { return (w - u) ^ (w - v); };
        T ans = 0;
        for (size_t i = 0, j = 1; i < p.size(); i++) {
            const auto nxti = this->nxt(i);
            ans = max({ans, p[j].dis2(p[i]), p[j].dis2(p[nxti])});
            while (area(p[this->nxt(j)], p[i], p[nxti]) >= area(p[j], p[i], p[nxti])) {
                j = this->nxt(j);
                ans = max({ans, p[j].dis2(p[i]), p[j].dis2(p[nxti])});
            }
        }
        return ans;
    }

    // 判断点是否在凸多边形内
    // 复杂度 O(logn)
    // -1 点在多边形边上 | 0 点在多边形外 | 1 点在多边形内
    int is_in(const Point &a) const {
        const auto &p = this->p;
        if (p.size() == 1) return a == p[0] ? -1 : 0;
        if (p.size() == 2) return Segment{p[0], p[1]}.is_on(a) ? -1 : 0;
        if (a == p[0]) return -1;
        if ((p[1] - p[0]).toleft(a - p[0]) == -1 || (p.back() - p[0]).toleft(a - p[0]) == 1) return 0;
        const auto cmp = [&](const Point &u, const Point &v) { return (u - p[0]).toleft(v - p[0]) == 1; };
        const size_t i = lower_bound(p.begin() + 1, p.end(), a, cmp) - p.begin();
        if (i == 1) return Segment{p[0], p[i]}.is_on(a) ? -1 : 0;
        if (i == p.size() - 1 && Segment{p[0], p[i]}.is_on(a)) return -1;
        if (Segment{p[i - 1], p[i]}.is_on(a)) return -1;
        return (p[i] - p[i - 1]).toleft(a - p[i - 1]) > 0;
    }

    // 凸多边形关于某一方向的极点
    // 复杂度 O(logn)
    // 参考资料：https://codeforces.com/blog/entry/48868
    template<typename F>
    size_t extreme(const F &dir) const {
        const auto &p = this->p;
        const auto check = [&](const size_t i) { return dir(p[i]).toleft(p[this->nxt(i)] - p[i]) >= 0; };
        const auto dir0 = dir(p[0]);
        const auto check0 = check(0);
        if (!check0 && check(p.size() - 1)) return 0;
```

```cpp
        const auto cmp = [&](const Point &v) {
            const size_t vi = &v - p.data();
            if (vi == 0) return 1;
            const auto checkv = check(vi);
            const auto t = dir0.toleft(v - p[0]);
            if (vi == 1 && checkv == check0 && t == 0) return 1;
            return checkv ^ (checkv == check0 && t <= 0);
        };
        return partition_point(p.begin(), p.end(), cmp) - p.begin();
    }

    // 过凸多边形外一点求凸多边形的切线，返回切点下标
    // 复杂度 O(logn)
    // 必须保证点在多边形外
    pair<size_t, size_t> tangent(const Point &a) const {
        const size_t i = extreme([&](const Point &u) { return u - a; });
        const size_t j = extreme([&](const Point &u) { return a - u; });
        return {i, j};
    }

    // 求平行于给定直线的凸多边形的切线，返回切点下标
    // 复杂度 O(logn)
    pair<size_t, size_t> tangent(const Line &a) const {
        const size_t i = extreme([&](...) { return a.v; });
        const size_t j = extreme([&](...) { return -a.v; });
        return {i, j};
    }
};

// 圆
struct Circle {
    Point c;
    T r; // 一般来说必须用浮点数

    bool operator==(const Circle &a) const { return c == a.c && abs(r - a.r) <= eps; }
    T circ() const { return 2 * PI * r; } // 周长
    T area() const { return PI * r * r; } // 面积

    // 点与圆的关系
    // -1 圆上 | 0 圆外 | 1 圆内
    int is_in(const Point &p) const {
        const T d = p.dis(c);
        return abs(d - r) <= eps ? -1 : d < r - eps;
    }

    // 直线与圆关系
    // 0 相离 | 1 相切 | 2 相交
    int relation(const Line &l) const {
        const T d = l.dis(c);
        if (d > r + eps) return 0;
        if (abs(d - r) <= eps) return 1;
        return 2;
    }
```

```cpp

    // 圆与圆关系
    // -1 相同 | 0 相离 | 1 外切 | 2 相交 | 3 内切 | 4 内含
    int relation(const Circle &a) const {
        if (*this == a) return -1;
        const T d = c.dis(a.c);
        if (d > r + a.r + eps) return 0;
        if (abs(d - r - a.r) <= eps) return 1;
        if (abs(d - abs(r - a.r)) <= eps) return 3;
        if (d < abs(r - a.r) - eps) return 4;
        return 2;
    }

    // 直线与圆的交点
    vector<Point> inter(const Line &l) const {
        const T d = l.dis(c);
        const Point p = l.proj(c);
        const int t = relation(l);
        if (t == 0) return vector<Point>();
        if (t == 1) return vector<Point>{p};
        const T k = sqrt(r * r - d * d);
        return vector<Point>{p - (l.v / l.v.len()) * k, p + (l.v / l.v.len()) * k};
    }

    // 圆与圆交点
    vector<Point> inter(const Circle &a) const {
        const T d = c.dis(a.c);
        const int t = relation(a);
        if (t == -1 || t == 0 || t == 4) return vector<Point>();
        Point e = a.c - c;
        e = e / e.len() * r;
        if (t == 1 || t == 3) {
            if (r * r + d * d - a.r * a.r >= -eps) return vector<Point>{c + e};
            return vector<Point>{c - e};
        }
        const T costh = (r * r + d * d - a.r * a.r) / (2 * r * d), sinth = sqrt(1 - costh * costh);
        return vector<Point>{c + e.rot(costh, -sinth), c + e.rot(costh, sinth)};
    }

    // 圆与圆交面积
    T inter_area(const Circle &a) const {
        const T d = c.dis(a.c);
        const int t = relation(a);
        if (t == -1) return area();
        if (t < 2) return 0;
        if (t > 2) return min(area(), a.area());
        const T costh1 = (r * r + d * d - a.r * a.r) / (2 * r * d), costh2 =
                (a.r * a.r + d * d - r * r) / (2 * a.r * d);
        const T sinth1 = sqrt(1 - costh1 * costh1), sinth2 = sqrt(1 - costh2 * costh2);
        const T th1 = acos(costh1), th2 = acos(costh2);
        return r * r * (th1 - costh1 * sinth1) + a.r * a.r * (th2 - costh2 * sinth2);
```

```
    }

    // 过圆外一点圆的切线
    vector<Line> tangent(const Point &a) const {
        const int t = is_in(a);
        if (t == 1) return vector<Line>();
        if (t == -1) {
            const Point v = {-(a - c).y, (a - c).x};
            return vector<Line>{{a, v}};
        }
        Point e = a - c;
        e = e / e.len() * r;
        const T costh = r / c.dis(a), sinth = sqrt(1 - costh * costh);
        const Point t1 = c + e.rot(costh, -sinth), t2 = c + e.rot(costh, sinth);
        return vector<Line>{{a, t1 - a}, {a, t2 - a}};
    }

    // 两圆的公切线
    vector<Line> tangent(const Circle &a) const {
        const int t = relation(a);
        vector<Line> lines;
        if (t == -1 || t == 4) return lines;
        if (t == 1 || t == 3) {
            const Point p = inter(a)[0], v = {-(a.c - c).y, (a.c - c).x};
            lines.push_back({p, v});
        }
        const T d = c.dis(a.c);
        const Point e = (a.c - c) / (a.c - c).len();
        if (t <= 2) {
            const T costh = (r - a.r) / d, sinth = sqrt(1 - costh * costh);
            const Point d1 = e.rot(costh, -sinth), d2 = e.rot(costh, sinth);
            const Point u1 = c + d1 * r, u2 = c + d2 * r, v1 = a.c + d1 * a.r, v2 = a.c + d2 * a.r;
            lines.push_back({u1, v1 - u1});
            lines.push_back({u2, v2 - u2});
        }
        if (t == 0) {
            const T costh = (r + a.r) / d, sinth = sqrt(1 - costh * costh);
            const Point d1 = e.rot(costh, -sinth), d2 = e.rot(costh, sinth);
            const Point u1 = c + d1 * r, u2 = c + d2 * r, v1 = a.c - d1 * a.r, v2 = a.c - d2 * a.r;
            lines.push_back({u1, v1 - u1});
            lines.push_back({u2, v2 - u2});
        }
        return lines;
    }

    // 圆的反演
    // auto result = circle.inverse(line);
    // if (std::holds_alternative<Circle>(result))
    // Circle c = std::get<Circle>(result);
    std::variant<Circle, Line> inverse(const Line &l) const {
        if (l.toleft(c) == 0) return l;
```

```cpp
        const Point v = l.toleft(c) == 1 ? Point{l.v.y, -l.v.x} : Point{-l.v.y,
l.v.x};
        const T d = r * r / l.dis(c);
        const Point p = c + v / v.len() * d;
        return Circle{(c + p) / 2, d / 2};
    }

    std::variant<Circle, Line> inverse(const Circle &a) const {
        const Point v = a.c - c;
        if (a.is_in(c) == -1) {
            const T d = r * r / (a.r + a.r);
            const Point p = c + v / v.len() * d;
            return Line{p, {-v.y, v.x}};
        }
        if (c == a.c) return Circle{c, r * r / a.r};
        const T d1 = r * r / (c.dis(a.c) - a.r), d2 = r * r / (c.dis(a.c) + a.r);
        const Point p = c + v / v.len() * d1, q = c + v / v.len() * d2;
        return Circle{(p + q) / 2, p.dis(q) / 2};
    }
};

// 圆与多边形面积交
T area_inter(const Circle &circ, const Polygon &poly) {
    const auto cal = [](const Circle &circ, const Point &a, const Point &b) {
        if ((a - circ.c).toleft(b - circ.c) == 0) return 0.0l;
        const auto ina = circ.is_in(a), inb = circ.is_in(b);
        const Line ab = {a, b - a};
        if (ina && inb) return ((a - circ.c) ^ (b - circ.c)) / 2;
        if (ina && !inb) {
            const auto t = circ.inter(ab);
            const Point p = t.size() == 1 ? t[0] : t[1];
            const T ans = ((a - circ.c) ^ (p - circ.c)) / 2;
            const T th = (p - circ.c).ang(b - circ.c);
            const T d = circ.r * circ.r * th / 2;
            if ((a - circ.c).toleft(b - circ.c) == 1) return ans + d;
            return ans - d;
        }
        if (!ina && inb) {
            const Point p = circ.inter(ab)[0];
            const T ans = ((p - circ.c) ^ (b - circ.c)) / 2;
            const T th = (a - circ.c).ang(p - circ.c);
            const T d = circ.r * circ.r * th / 2;
            if ((a - circ.c).toleft(b - circ.c) == 1) return ans + d;
            return ans - d;
        }
        const auto p = circ.inter(ab);
        if (p.size() == 2 && Segment{a, b}.dis(circ.c) <= circ.r + eps) {
            const T ans = ((p[0] - circ.c) ^ (p[1] - circ.c)) / 2;
            const T th1 = (a - circ.c).ang(p[0] - circ.c), th2 = (b - circ.c).ang(p[1]
- circ.c);
            const T d1 = circ.r * circ.r * th1 / 2, d2 = circ.r * circ.r * th2 / 2;
            if ((a - circ.c).toleft(b - circ.c) == 1) return ans + d1 + d2;
            return ans - d1 - d2;
```

```cpp
        }
        const T th = (a - circ.c).ang(b - circ.c);
        if ((a - circ.c).toleft(b - circ.c) == 1) return circ.r * circ.r * th / 2;
        return -circ.r * circ.r * th / 2;
    };

    T ans = 0;
    for (size_t i = 0; i < poly.p.size(); i++) {
        const Point a = poly.p[i], b = poly.p[poly.nxt(i)];
        ans += cal(circ, a, b);
    }
    return ans;
}

// 点集的凸包
// Andrew 算法，复杂度 O(nlogn)
Convex convexhull(vector<Point> p) {
    vector<Point> st;
    if (p.empty()) return Convex{st};
    sort(p.begin(), p.end());
    const auto check = [](const vector<Point> &st, const Point &u) {
        const auto back1 = st.back(), back2 = *prev(st.end(), 2);
        return (back1 - back2).toleft(u - back1) <= 0;
    };
    for (const Point &u: p) {
        while (st.size() > 1 && check(st, u)) st.pop_back();
        st.push_back(u);
    }
    size_t k = st.size();
    p.pop_back();
    reverse(p.begin(), p.end());
    for (const Point &u: p) {
        while (st.size() > k && check(st, u)) st.pop_back();
        st.push_back(u);
    }
    st.pop_back();
    return Convex{st};
}

// 半平面交
// 排序增量法，复杂度 O(nlogn)
// 输入与返回值都是用直线表示的半平面集合
vector<Line> halfinter(vector<Line> l, const T lim = 1e9) {
    const auto check = [](const Line &a, const Line &b, const Line &c) { return a.toleft(b.inter(c)) < 0; };
    // 无精度误差的方法，但注意取值范围会扩大到三次方
    /*const auto check=[](const Line &a,const Line &b,const Line &c)
    {
        const Point p=a.v*(b.v^c.v),q=b.p*(b.v^c.v)+b.v*(c.v^(b.p-c.p))-a.p*(b.v^c.v);
        return p.toleft(q)<0;
    };*/
    l.push_back({{-lim, 0}, {0, -1}});
    l.push_back({{0, -lim}, {1, 0}});
```

```
    l.push_back({{lim, 0}, {0, 1}});
    l.push_back({{0, lim}, {-1, 0}});
    sort(l.begin(), l.end());
    deque<Line> q;
    for (size_t i = 0; i < l.size(); i++) {
        if (i > 0 && l[i - 1].v.toleft(l[i].v) == 0 && l[i - 1].v * l[i].v > eps)
continue;
        while (q.size() > 1 && check(l[i], q.back(), q[q.size() - 2])) q.pop_back();
        while (q.size() > 1 && check(l[i], q[0], q[1])) q.pop_front();
        if (!q.empty() && q.back().v.toleft(l[i].v) <= 0) return vector<Line>();
        q.push_back(l[i]);
    }
    while (q.size() > 1 && check(q[0], q.back(), q[q.size() - 2])) q.pop_back();
    while (q.size() > 1 && check(q.back(), q[0], q[1])) q.pop_front();
    return vector<Line>(q.begin(), q.end());
}

// 点集形成的最小最大三角形
// 极角序扫描线，复杂度 O(n^2logn)
// 最大三角形问题可以使用凸包与旋转卡壳做到 O(n^2)
pair<T, T> minmax_triangle(const vector<Point> &vec) {
    if (vec.size() <= 2) return {0, 0};
    vector<pair<int, int> > evt;
    evt.reserve(vec.size() * vec.size());
    T maxans = 0, minans = INF;
    for (size_t i = 0; i < vec.size(); i++) {
        for (size_t j = 0; j < vec.size(); j++) {
            if (i == j) continue;
            if (vec[i] == vec[j]) minans = 0;
            else evt.push_back({i, j});
        }
    }
    sort(evt.begin(), evt.end(), [&](const pair<int, int> &u, const pair<int, int> &v)
{
        const Point du = vec[u.second] - vec[u.first], dv = vec[v.second] -
vec[v.first];
        return Argcmp()({du.y, -du.x}, {dv.y, -dv.x});
    });
    vector<size_t> vx(vec.size()), pos(vec.size());
    for (size_t i = 0; i < vec.size(); i++) vx[i] = i;
    sort(vx.begin(), vx.end(), [&](int x, int y) { return vec[x] < vec[y]; });
    for (size_t i = 0; i < vx.size(); i++) pos[vx[i]] = i;
    for (auto [u,v]: evt) {
        const size_t i = pos[u], j = pos[v];
        const size_t l = min(i, j), r = max(i, j);
        const Point vecu = vec[u], vecv = vec[v];
        if (l > 0) minans = min(minans, abs((vec[vx[l - 1]] - vecu) ^ (vec[vx[l - 1]]
- vecv)));
        if (r < vx.size() - 1) minans = min(minans, abs((vec[vx[r + 1]] - vecu) ^
(vec[vx[r + 1]] - vecv)));
        maxans = max({
            maxans, abs((vec[vx[0]] - vecu) ^ (vec[vx[0]] - vecv)),
            abs((vec[vx.back()] - vecu) ^ (vec[vx.back()] - vecv))
```

```
        });
        if (i < j) swap(vx[i], vx[j]), pos[u] = j, pos[v] = i;
    }
    return {minans, maxans};
}

// 平面最近点对
// 扫描线，复杂度 O(nlogn)
T closest_pair(vector<Point> points) {
    sort(points.begin(), points.end());
    const auto cmpy = [](const Point &a, const Point &b) {
        if (abs(a.y - b.y) <= eps) return a.x < b.x - eps;
        return a.y < b.y - eps;
    };
    multiset<Point, decltype(cmpy)> s{cmpy};
    T ans = INF;
    for (size_t i = 0, l = 0; i < points.size(); i++) {
        const T sqans = sqrtl(ans) + 1;
        while (l < i && points[i].x - points[l].x >= sqans) s.erase(s.find(points[l++]));
        for (auto it = s.lower_bound(Point{-INF, points[i].y - sqans}); it != s.end() && it->y - points[i].y <= sqans;
             it++) {
            ans = min(ans, points[i].dis2(*it));
        }
        s.insert(points[i]);
    }
    return ans;
}

// 判断多条线段是否有交点
// 扫描线，复杂度 O(nlogn)
bool segs_inter(const vector&lt;Segment&gt; &segs) {
    if (segs.empty()) return false;
    using seq_t = tuple<T, int, Segment>; // x坐标 出入点 线段
    const auto seqcmp = [](const seq_t &u, const seq_t &v) {
        const auto [u0,u1,u2] = u;
        const auto [v0,v1,v2] = v;
        if (abs(u0 - v0) <= eps) return make_pair(u1, u2) < make_pair(v1, v2);
        return u0 < v0 - eps;
    };
    vector<seq_t> seq;
    for (auto seg: segs) {
        if (seg.a.x > seg.b.x + eps) swap(seg.a, seg.b);
        seq.push_back({seg.a.x, 0, seg});
        seq.push_back({seg.b.x, 1, seg});
    }
    sort(seq.begin(), seq.end(), seqcmp);
    T x_now;
    auto cmp = [&](const Segment &u, const Segment &v) {
        if (abs(u.a.x - u.b.x) <= eps || abs(v.a.x - v.b.x) <= eps) return u.a.y < v.a.y - eps;
```

```
        return ((x_now - u.a.x) * (u.b.y - u.a.y) + u.a.y * (u.b.x - u.a.x)) * (v.b.x - v.a.x) < (
                   (x_now - v.a.x) * (v.b.y - v.a.y) + v.a.y * (v.b.x - v.a.x)) * (u.b.x - u.a.x) - eps;
    };
    multiset&lt;Segment, decltype(cmp)> s{cmp};
    for (const auto [x,o,seg]: seq) {
        x_now = x;
        const auto it = s.lower_bound(seg);
        if (o == 0) {
            if (it != s.end() && seg.is_inter(*it)) return true;
            if (it != s.begin() && seg.is_inter(*prev(it))) return true;
            s.insert(seg);
        } else {
            if (next(it) != s.end() && it != s.begin() && (*prev(it)).is_inter(*next(it))) return true;
            s.erase(it);
        }
    }
    return false;
}

// 多边形面积并
// 轮廓积分，复杂度 O(n^2logn)，n为边数
// ans[i] 表示被至少覆盖了 i+1 次的区域的面积
vector<T> area_union(const vector<Polygon> &polys) {
    const size_t siz = polys.size();
    vector<vector<pair<Point, Point> > > segs(siz);
    const auto check = [](const Point &u, const Segment &e) { return !((u < e.a && u < e.b) || (u > e.a && u > e.b)); };

    auto cut_edge = [&](const Segment &e, const size_t i) {
        const Line le{e.a, e.b - e.a};
        vector<pair<Point, int> > evt;
        evt.push_back({e.a, 0});
        evt.push_back({e.b, 0});
        for (size_t j = 0; j < polys.size(); j++) {
            if (i == j) continue;
            const auto &pj = polys[j];
            for (size_t k = 0; k < pj.p.size(); k++) {
                const Segment s = {pj.p[k], pj.p[pj.nxt(k)]};
                if (le.toleft(s.a) == 0 && le.toleft(s.b) == 0) {
                    evt.push_back({s.a, 0});
                    evt.push_back({s.b, 0});
                } else if (s.is_inter(le)) {
                    const Line ls{s.a, s.b - s.a};
                    const Point u = le.inter(ls);
                    if (le.toleft(s.a) < 0 && le.toleft(s.b) >= 0) evt.push_back({u, -1});
                    else if (le.toleft(s.a) >= 0 && le.toleft(s.b) < 0) evt.push_back({u, 1});
                }
            }
```

```cpp
        }
        sort(evt.begin(), evt.end());
        if (e.a > e.b) reverse(evt.begin(), evt.end());
        int sum = 0;
        for (size_t i = 0; i < evt.size(); i++) {
            sum += evt[i].second;
            const Point u = evt[i].first, v = evt[i + 1].first;
            if (!(u == v) && check(u, e) && check(v, e)) segs[sum].push_back({u, v});
            if (v == e.b) break;
        }
    };

    for (size_t i = 0; i < polys.size(); i++) {
        const auto &pi = polys[i];
        for (size_t k = 0; k < pi.p.size(); k++) {
            const Segment ei = {pi.p[k], pi.p[pi.nxt(k)]};
            cut_edge(ei, i);
        }
    }
    vector<T> ans(siz);
    for (size_t i = 0; i < siz; i++) {
        T sum = 0;
        sort(segs[i].begin(), segs[i].end());
        int cnt = 0;
        for (size_t j = 0; j < segs[i].size(); j++) {
            if (j > 0 && segs[i][j] == segs[i][j - 1]) segs[i +
(++cnt)].push_back(segs[i][j]);
            else cnt = 0, sum += segs[i][j].first ^ segs[i][j].second;
        }
        ans[i] = sum / 2;
    }
    return ans;
}

// 圆面积并
// 轮廓积分，复杂度 O(n^2logn)
// ans[i] 表示被至少覆盖了 i+1 次的区域的面积
vector<T> area_union(const vector<Circle> &circs) {
    const size_t siz = circs.size();
    using arc_t = tuple<Point, T, T, T>;
    vector<vector<arc_t> > arcs(siz);
    const auto eq = [](const arc_t &u, const arc_t &v) {
        const auto [u1,u2,u3,u4] = u;
        const auto [v1,v2,v3,v4] = v;
        return u1 == v1 && abs(u2 - v2) <= eps && abs(u3 - v3) <= eps && abs(u4 - v4)
<= eps;
    };

    auto cut_circ = [&](const Circle &ci, const size_t i) {
        vector<pair<T, int> > evt;
        evt.push_back({-PI, 0});
        evt.push_back({PI, 0});
        int init = 0;
```

```
        for (size_t j = 0; j < circs.size(); j++) {
            if (i == j) continue;
            const Circle &cj = circs[j];
            if (ci.r < cj.r - eps && ci.relation(cj) >= 3) init++;
            const auto inters = ci.inter(cj);
            if (inters.size() == 1) evt.push_back({atan2l((inters[0] - ci.c).y, (inters[0] - ci.c).x), 0});
            if (inters.size() == 2) {
                const Point dl = inters[0] - ci.c, dr = inters[1] - ci.c;
                T argl = atan2l(dl.y, dl.x), argr = atan2l(dr.y, dr.x);
                if (abs(argl + PI) <= eps) argl = PI;
                if (abs(argr + PI) <= eps) argr = PI;
                if (argl > argr + eps) {
                    evt.push_back({argl, 1});
                    evt.push_back({PI, -1});
                    evt.push_back({-PI, 1});
                    evt.push_back({argr, -1});
                } else {
                    evt.push_back({argl, 1});
                    evt.push_back({argr, -1});
                }
            }
        }
        sort(evt.begin(), evt.end());
        int sum = init;
        for (size_t i = 0; i < evt.size(); i++) {
            sum += evt[i].second;
            if (abs(evt[i].first - evt[i + 1].first) > eps)
                arcs[sum].push_back({
                    ci.c, ci.r, evt[i].first, evt[i + 1].first
                });
            if (abs(evt[i + 1].first - PI) <= eps) break;
        }
    };

    const auto oint = [](const arc_t &arc) {
        const auto [cc,cr,l,r] = arc;
        if (abs(r - l - PI - PI) <= eps) return 2.0l * PI * cr * cr;
        return cr * cr * (r - l) + cc.x * cr * (sin(r) - sin(l)) - cc.y * cr * (cos(r) - cos(l));
    };

    for (size_t i = 0; i < circs.size(); i++) {
        const auto &ci = circs[i];
        cut_circ(ci, i);
    }
    vector<T> ans(siz);
    for (size_t i = 0; i < siz; i++) {
        T sum = 0;
        sort(arcs[i].begin(), arcs[i].end());
        int cnt = 0;
        for (size_t j = 0; j < arcs[i].size(); j++) {
```

```
            if (j > 0 && eq(arcs[i][j], arcs[i][j - 1])) arcs[i + (++cnt)].push_back(arcs[i][j]);
            else cnt = 0, sum += oint(arcs[i][j]);
        }
        ans[i] = sum / 2;
    }
    return ans;
}
```