```cpp
#include <bits/stdc++.h>
#define maxn 800005
#define int long long
using namespace std;


// +--------------------------------+
// |                                |
// |  Geometry Template BASIC(Sgn)  |
// |                                |
// +--------------------------------+


//const long long eps = 0;
const long double eps = 1e-8;

//long double情况下使用的sgn函数
int sgn(long double x){
    if(fabs(x) <= eps)    return 0;
    return x > 0 ? 1 : -1;
}
/*
//long long情况下使用的sgn函数
int sgnLL(long long  x){
    if(abs(x) == eps)    return 0;
    return x > 0 ? 1 : -1;
}
*/

// +----------------------------+
// |                            |
// |  Geometry Template Struct  |
// |                            |
// +----------------------------+

/* *** 点（基础模板） *** */
template<typename T> struct TP{
    T x, y;
    TP(){}
    TP(T _x, T _y){ x = _x; y = _y; }
    TP operator -() const {
        return {-x, -y};
    }
    friend TP operator +(const TP &a, const TP &b){
        return {a.x + b.x, a.y + b.y};
    }
    friend TP operator -(const TP &a, const TP &b){
        return {a.x - b.x, a.y - b.y};
    }
    friend T operator *(const TP &a, const TP &b){
        return a.x * b.x + a.y * b.y;
    }
    friend T operator ^(const TP &a, const TP &b){
        return a.x * b.y - a.y * b.x;
    }
    friend bool operator ==(const TP &a, const TP &b){
        return sgn(a.x - b.x) == 0 && sgn(a.y - b.y) == 0;
    }
    friend bool operator <(const TP &a, const TP &b){
        if(sgn(a.x - b.x) == 0)    return sgn(a.y - b.y) < 0;
        return sgn(a.x - b.x) < 0;
    }
    TP operator *(const long double &k) const{
        return {x * k, y * k};
    }
    TP operator /(const long double &k) const{
        return {x / k, y / k};
    }
    //a在逆时针方向：1，顺时针方向：-1，其他：0
    int toleft(const TP &a) const {
        auto t = (*this) ^ a;
        return (t > eps) - (t < -eps);
    }
    //返回极角，(-PI, PI]
    long double angle(){
        return atan2(y, x);
    }
    //返回长度
    long double len() const {
        return sqrt(len2());
    }
    //返回两点间距离
    long double dis(const TP &a) const {
        return sqrt(dis2(a));
    }
    //返回长度的平方
    T len2() const {
        return (*this) * (*this);
    }
    //返回两点间距离的平方
    T dis2(const TP &a) const {
        return TP(x - a.x, y - a.y).len2();
    }
    //返回两向量的夹角
    long double ang(const TP &a) const {
        return acos(max(-1.0, min(1.0, ((*this) * a) / (len() * a.len()))));
    }
    //返回逆时针旋转rad后的结果
    TP rot(const long double rad) const {
        return {x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)};
    }
    //旋转（指定cos和sin）
    TP rot(const long double cosr,const long double sinr) const {
        return {x * cosr - y * sinr, x * sinr + y * cosr};
    }
    //返回长度为R的向量
    TP trunc(long double r){
        long double l = len();
        if(!sgn(l))    return (*this);
        r /= l;
        return {x * r, y * r};
    }
    void input(){
        cin >> x >> y;
    }
    void print(){
        cout << "[Point]\n";
        cout << x << " " << y << '\n';
    }
}; using Point = TP<long double>;

/* *** 线（基础模板） *** */
template<typename T> struct TL{
    TP<T> s, e;
    TL(){}
    TL(TP<T> _s, TP<T> _e){ s = _s; e = _e; }
    friend T operator *(const TL &la, const TL &lb){
        return (la.e - la.s) * (lb.e - lb.s);
    }
    friend T operator ^(const TL &la, const TL &lb){
        return (la.e - la.s) ^ (lb.e - lb.s);
    }
    friend bool operator ==(const TL &la, const TL &lb){
        return la.parallel(lb) && la.isOnSeg(lb.s);
    }
    //点到直线的距离
    long double length(){
        return (e - s).len();
    }
    //点到直线的距离
    long double disLine(const TP<T> &p) const {
        return fabs((p - s) ^ (e - s)) / length();
    }
    //点到线段的距离
    long double disSeg(const TP<T> &p) const{
        if(sgn((p - s) * (e - s)) < 0 || sgn((p - e) * (s - e)) < 0){
            return min(p.dis(s), p.dis(e));
        }
        return disLine(p);
    }
    //点到直线的投影
    TP<T> proj(const TP<T> &p) const {
        return s + (((e - s) * ((e - s) * (p - s))) / ((e - s).len2()));
    }
    //关于直线的对称点
    TP<T> symmetryPoint(TP<T> p){
        Point q = proj(p);
        return Point(2 * q.x - p.x, 2 * q.y - p.y);
    }
    //两直线平行
    bool parallel(TL v){
        return sgn((e - s) ^ (v.e - v.s)) == 0;
    }
    //两直线交点
    TP<T> crosspoint(TL v){
        auto a1 = (v.e - v.s) ^ (s - v.s);
        auto a2 = (v.e - v.s) ^ (e - v.s);
        return {(s.x * a2 - e.x * a1) / (a2 - a1), (s.y * a2 - e.y * a1) / (a2 - a1)};
    }
    //点在线段上，端点：-1，线段内：1，其他：0
    int isOnSeg(const TP<T> &p){
        if(p == s || p == e)    return -1;
        return sgn((p - s) ^ (e - s)) == 0 && sgn((p - s) * (p - e)) <= 0;
    }
    //2 -> 规范相交，1 -> 非规范相交，0 -> 不相交
    int segCrossSeg(TL v){
        int d1 = sgn((e - s) ^ (v.s - s));
        int d2 = sgn((e - s) ^ (v.e - s));
        int d3 = sgn((v.e - v.s) ^ (s - v.s));
        int d4 = sgn((v.e - v.s) ^ (e - v.s));
        if((d1 ^ d2) == -2 && (d3 ^ d4) == -2)    return 2;
        return (d1 == 0 && sgn((v.s - s) * (v.s - e)) <= 0) ||
               (d2 == 0 && sgn((v.e - s) * (v.e - e)) <= 0) ||
               (d3 == 0 && sgn((s - v.s) * (s - v.e)) <= 0) ||
               (d4 == 0 && sgn((e - v.s) * (e - v.e)) <= 0);
    }
    // this -> Line, v -> Seg, 2，规范相交，1，非规范相交，0，不相交
    int lineCrossSeg(TL v){
        int d1 = sgn((e - s) ^ (v.s - s));
        int d2 = sgn((e - s) ^ (v.e - s));
        if((d1 ^ d2) == -2)    return 2;
        return (d1 == 0 || d2 == 0);
    }
    //两直线关系，0，平行，1，重合，2，相交
    int lineRelation(TL v){
        if((*this).parallel(v)){
            return v.toleft(s) == 0;
        }
        return 2;
    }
    //a在直线的，逆时针方向：1，顺时针方向：-1，其他：0
    int toleft(const TP<T> &p) const {
        int c = sgn((p - s) ^ (e - s));
        if(c < 0)       return 1;
        else if(c > 0)  return -1;
        else            return 0;
    }
    void print(){
        cout << "[Line]\n";
        cout << s.x << " " << s.y << " " << e.x << " " << e.y << '\n';
    }
}; using Line = TL<long double>;

/* *** 圆（基础模板） *** */
const long double PI = acos(-1.0);
template<typename T> struct TC{
    TP<T> c;
    long double r;
    TC(){}
    TC(TP<T> _c, long double _r){ c = _c; r = _r; }
    //根据极角返回圆上一点
    TP<T> point(long double a) {
        return TP<T>(c.x + cos(a) * r, c.y + sin(a) * r);
    }
    long double area(){
        return PI * r * r;
    }
    //5 -> 相离，4 -> 外切， 3 -> 相交，2 -> 内切，1 -> 内含
    int relationCircle(TC v){
        long double d = c.dis(v.c);
        if(sgn((d - r - v.r)) > 0)    return 5;
        if(sgn((d - r - v.r)) == 0)   return 4;
        long double l = fabs(r - v.r);
        if(sgn((d - r - v.r)) < 0 && sgn(d - l) > 0)    return 3;
        if(sgn(d - l) == 0)    return 2;
        if(sgn(d - l) < 0)     return 1;
    }
    //【已测试：ZOJ1597】
    //两圆相交得到的面积
    long double areaCircle(TC v){
        int rel = relationCircle(v);
        if(rel >= 4)    return 0.0;
        if(rel <= 2)    return min(area(), v.area());
        long double d = c.dis(v.c);
        long double hf = (r + v.r + d) / 2.0;
        long double ss = 2 * sqrt(hf * (hf - r) * (hf - v.r) * (hf - d));
        long double a1 = acos((r * r + d * d - v.r * v.r) / (2.0 * r * d));
```

```cpp
        a1 = a1 * r * r;
        long double a2 = acos((v.r * v.r + d * d - r * r) / (2.0 * v.r * d));
        a2 = a2 * v.r * v.r;
        return a1 + a2 - ss;
    }
}; using Circle = TC<long double>;

/* *** 多边形（基础模板） *** */
template<typename T> struct TG{
    vector<TP<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;}
    //求面积的二倍（逆时针存点则为正）
    T getArea2(){
        int siz = p.size();
        T sum = 0;
        for(int i = 0; i < siz; i++){
            sum += (p[i] ^ p[(i + 1) % siz]);
        }
        return sum;
    }
    //Winding，判断点与多边形关系，{true, 0} -> 点在边上，{false, cnt} -> 回转数为0 -> 外部，其他 -> 内部
    pair<bool, int> winding(const Point &a) {
        int cnt = 0;
        for(int i = 0; i < p.size(); i++){
            Point u = p[i], v = p[nxt(i)];
            if(sgn((a - u) ^ (a - v)) == 0 && sgn((a - u)*(a - v)) <= 0)    return {true, 0};
            if(sgn(u.y - v.y) == 0)    continue;
            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};
    }
    void print(){
        cout << "[Polygon]\n";
        for(int i = 0; i < p.size(); i++){
            cout << i << " " << p[i].x << " " << p[i].y << '\n';
        }
    }
}; using Polygon = TG<long double>;


// +------------------------------+
// |                              |
// |  Geometry Template Function  |
// |                              |
// +------------------------------+


//凸包点集调整 -> 起点变为下凸壳最左侧的点；
void adjustConvexHull(vector<Point> &P, vector<Point> &tmp){
    int n = P.size(); tmp.resize(n);
    int pos = -1;
    long double minX = 1e60, maxY = -1e60;
    for(int i = 0; i < n; i++){
        if(P[i].x < minX || (fabs(P[i].x - minX) <= eps && P[i].y > maxY)){
            pos = i;
            minX = P[i].x; maxY = P[i].y;
        }
    }
    int cnt = 0;
    for(int i = pos; i < n; i++)    tmp[cnt++] = P[i];
    for(int i = 0; i < pos; i++)    tmp[cnt++] = P[i];
    for(int i = 0; i < n; i++)      P[i] = tmp[i];
}
//求解点集p的凸包（Andrew算法），逆时针存于点集ans中。
#define bk1(x) (x.back())
#define bk2(x) (*(x.rbegin() + 1))
void findConvexHull(vector<Point> p, vector<Point> &ans){
    vector<Point> st;
    sort(p.begin(), p.end(), [&](const Point &A, const Point &B){ return sgn(A.x - B.x) ? A.x < B.x : A.y < B.y; });
    for(Point u : p){
        while(st.size() > 1 && ((bk1(st) - bk2(st)).toleft(u - bk2(st))) <= 0){
            st.pop_back();
        }
        st.push_back(u);
    }
    int k = st.size();
    p.pop_back(); reverse(p.begin(), p.end());
    for(Point u : p){
        while(st.size() > k && ((bk1(st) - bk2(st)).toleft(u - bk2(st))) <= 0){
            st.pop_back();
        }
        st.push_back(u);
    }
    st.pop_back();
    ans.clear();
    for(auto x : st)    ans.push_back(x);
}

//叉积排序函数
bool argcmpC(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 5;
        if(a.x > +eps)    return 3;
        return 2;
    };
    int qa = Quad(a), qb = Quad(b);
    if(qa != qb)    return qa < qb;
    auto cross = (a ^ b);
    return cross > eps;
}

//极角序，自动去重，返回<方向，个数>的集合，需依赖上方的argcmpC函数
vector<pair<Point, int>> polarUniqueTrans(vector<Point> &p){
    map<Point, int, decltype(&argcmpC)> uni{&argcmpC};
    vector<pair<Point, int>> res;
    for(auto x : p)     uni[x]++;
    for(auto x : uni)   res.push_back(x);
    return res;
}
//【已测试：P4525 「模板」自适应辛普森法 1】
//自适应simp积分，近似求int[a, b]
long double functionVal(long double x){
    //returns f(x)
    return 0;
}
long double simp(long double l, long double r){
```

```cpp
    long double mid = (l + r) / 2.0;
    return (r - l) * (functionVal(l) + 4 * functionVal(mid) + functionVal(r)) / 6.0;
}
//需注意eps精度问题
long double asr(long double l, long double r, long double ans){
    long double mid = (l + r) / 2.0;
    long double vL = simp(l, mid), vR = simp(mid, r), tmp = vL + vR - ans;
    if(fabs(tmp) <= eps)    return ans;
    else                    return asr(l, mid, vL) + asr(mid, r, vR);
}
//求解两圆公切线：返回切线的条数，-1表示无穷多条切线，a -> A上的切点，b -> B上的切点
int getCircleTangents(Circle A, Circle B, vector<Point> &a, vector<Point> &b){
    int cnt = 0;
    if (A.r < B.r) {
        swap(A, B);
        swap(a, b);
    }
    double d2 = (A.c.x - B.c.x) * (A.c.x - B.c.x) + (A.c.y - B.c.y) * (A.c.y - B.c.y);
    double rdiff = A.r - B.r;
    double rsum = A.r + B.r;
    if (sgn(d2 - rdiff * rdiff) < 0) return 0;  // 内含
    double base = atan2(B.c.y - A.c.y, B.c.x - A.c.x);
    //无限多条切线
    if (sgn(d2) == 0 && sgn(A.r - B.r) == 0) return -1;
    //内切，一条切线
    if (sgn(d2 - rdiff * rdiff) == 0) {
        a.push_back(A.point(base));
        b.push_back(B.point(base));
        cnt++;
        return cnt;
    }
    //有外公切线
    double ang = acos(rdiff / sqrt(d2));
    a.push_back(A.point(base + ang));
    b.push_back(B.point(base + ang));
    a.push_back(A.point(base - ang));
    b.push_back(B.point(base - ang));
    cnt += 2;
    if(sgn(d2 - rsum * rsum) == 0){  // 一条内公切线
        a.push_back(A.point(base));
        b.push_back(B.point(PI + base));
        cnt++;
    } else if(sgn(d2 - rsum * rsum) > 0){  // 两条内公切线
        double ang = acos(rsum / sqrt(d2));
        a.push_back(A.point(base + ang));
        b.push_back(B.point(PI + base + ang));
        a.push_back(A.point(base - ang));
        b.push_back(B.point(PI + base - ang));
        cnt += 2;
    }
    return cnt;
}

/* 圆的反演：C2C,C2L,L2C */
/*
 * 反演变换
 * 适用于题目中存在多个圆/直线之间的相切关系的情况。
 * 1. 圆O外的点的反演点在圆O内，反之亦然，圆O上的点的反演点为其自身。
 * 2. 不过点O的圆，其反演图形也是不过点O的圆。
 * 3. 过点O的圆，其反演图形是不过点O的直线。
 * 4. 两个图形相切，则他们的反演图形也相切。（*）
 * 5. 两个不经过反演点的外切的圆，反演之后的图形为相交的两条直线。
 *     如果其中一个圆经过反演点，那么反演之后的图形为一个圆和它的一条切线并且反演点和反演后的圆的圆心在切线的【同一侧】。
 *     内切的话反演中心和反演圆的圆心在【异侧】。
 */
//【已测试：HDU4773: Problem of Apollonius】
//点O在圆A外，求圆A的反演圆B，R是反演半径
Circle inversionC2C(Point O, long double R, Circle A){
    long double OA = (A.c - O).len();
    long double RB = 0.5 * ((1 / (OA - A.r)) - (1 / (OA + A.r))) * R * R;
    long double OB = OA * RB / A.r;
    long double Bx = O.x + (A.c.x - O.x) * OB / OA;
    long double By = O.y + (A.c.y - O.y) * OB / OA;
    return Circle(Point(Bx, By), RB);
}
//【已测试：HDU4773: Problem of Apollonius】
//直线反演为过O点的圆B，R是反演半径
Circle inversionL2C(Point O, long double R, Point A, Point B){
    Point P = Line(A, B).proj(O);
    long double d = (O - P).len();
    long double RB = R * R / (2 * d);
    Point VB = (P - O) / d * RB;
    return Circle(O + VB, RB);
}
//【已测试：HDU4773: Problem of Apollonius】
//圆A经过反演中心O，反演得到直线L
Line inversionC2L(Point O, long double R, Circle A){
    long double angle = (O - A.c).angle();
    if(sgn(angle) < 0)    angle += 2 * PI;
    long double angleL = angle + PI / 2;
    long double angleR = angle - PI / 2;
    if(angleL < 0)    angleL += 2 * PI;
    if(angleR < 0)    angleR += 2 * PI;
    Point PL = A.point(angleL), PR = A.point(angleR), dirL = PL - O, dirR = PR - O;
    long double disL = O.dis(PL), disrL = R * R / disL, disR = O.dis(PR), disrR = R * R / disR;
    return Line(O + dirL.trunc(disrL), O + dirR.trunc(disrR));
}
/*
 * 根据两个圆的位置关系来确定情况：
 * (1) 两个圆内含，没有公共点，没有公切线
 * (2) 两圆内切，有一个条公切线
 * (3) 两圆完全重合，有无数条公切线
 * (4) 两圆相交。有2条公切线
 * (5) 两圆外切，有3条公切线
 * (6) 两圆相离，有4条公切线
 */


// +------------------------------+
// |                              |
// |  Geometry Template ExStruct  |
// |                              |
// +------------------------------+


/* 拓展自Polygon：求解点是否在凸包内，以及凸包外一点对该凸包的切线 */
struct Convex : Polygon{
    //闵可夫斯基和，对应凸包
    //【已测试：BZOJ2564. 集合的面积】
    Convex operator +(const Convex &c){
        const auto &p = this->p;
        vector<Line> e1(p.size()), e2(c.p.size()), edge(p.size() + c.p.size());
        Convex res;
```

```cpp
        res.p.reserve(p.size() + c.p.size());
        for(int i = 0; i < p.size(); i++){
            e1[i] = {p[i], p[this -> nxt(i)]};
        }
        for(int i = 0; i < c.p.size(); i++){
            e2[i] = {c.p[i], c.p[c.nxt(i)]};
        }
        const auto cmp = [](const Line &u,const Line &v) { return argcmpC(u.e - u.s, v.e - v.s); };
        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> &p, const Point &u){
            const auto back1 = p.back(), back2 = *prev(p.end(), 2);
            return (back1 - back2).toleft(u - back1) == 0 && (back1 - back2) * (u - back1) >= -eps;
        };
        auto u = e1[0].s + e2[0].s;
        for(const auto &v : edge){
            while(res.p.size() > 1 && check(res.p, u)){
                res.p.pop_back();
            }
            res.p.push_back(u);
            u = u + v.e - v.s;
        }
        if(res.p.size() > 1 && check(res.p, res.p[0]))    res.p.pop_back();
        return res;
    }
    //【已测试：Enclosure】
    //O(logN)判断点是否在凸包内，1：在凸包内，0：在凸包外，-1：在凸包上
    int inConvex(const Point &a){
        auto &p = this->p;
        int l = 1, r = (int)(p.size()) - 2;
        while(l <= r){
            auto mid = (l + r) / 2;
            auto t1 = (p[mid] - p[0]).toleft(a - p[0]);
            auto t2 = (p[mid + 1] - p[0]).toleft(a - p[0]);
            if(t1 >= 0 && t2 <= 0){
                if(mid == 1 && Line(p[0], p[mid]).isOnSeg(a))    return -1;
                if(mid + 1 == (int)(p.size()) - 1 && Line(p[0], p[mid + 1]).isOnSeg(a))    return -1;
                if(Line(p[mid], p[mid + 1]).isOnSeg(a))    return -1;
                return (p[mid + 1] - p[mid]).toleft(a - p[mid]) > 0;
            }
            if(t1 < 0)    r = mid - 1;
            else          l = mid + 1;
        }
        return false;
    }
    //【已测试：USACO03FALL - Beauty Contest G】
    //旋转卡壳，求解内容取决于传入的函数F
    template<typename F> void rotcaliper(const F &func) {
        const auto &p = this->p;
        const auto area = [](const Point &u, const Point &v, const Point &w){ return fabs((w - u) ^ (w - v)); };
        for(int i = 0, j = 1; i < p.size(); i++){
            const auto nxti = this -> nxt(i);
            func(p[i], p[nxti], p[j]);
            while(area(p[this -> nxt(j)], p[i], p[nxti]) >= area(p[j], p[i], p[nxti])){
                j = this -> nxt(j);
                func(p[i], p[nxti], p[j]);
            }
        }
    }
    //【已测试：USACO03FALL - Beauty Contest G】
    //旋转卡壳，求凸包直径（平方），需根据选定类型确定返回值（long long double/long long）
    long double diameter2(){
        const auto &p = this -> p;
        if(p.size() == 1) return 0;
        if(p.size() == 2) return p[0].dis2(p[1]);
        long double ans = 0;
        auto func = [&](const Point &u, const Point &v, const Point &w){
            ans = max(ans, max(w.dis2(u), w.dis2(v)));
        };
        rotcaliper(func);
        return ans;
    }
    //【已测试：Enclosure】
    //O(logN)求解凸包外一点切线（返回其中一个切点），配合tangent使用
    template<typename F> int extreme(const F &dir){
        auto &p = this -> p;
        auto check = [&](const int i){
            return dir(p[i]).toleft(p[this->nxt(i)] - p[i]) >= 0;
        };
        auto dir0 = dir(p[0]);
        auto check0 = check(0);
        if(check0 == 0 && check((int)(p.size()) - 1))    return 0;
        int l = 0, r = p.size() - 1;
        while(l < r){
            auto mid = (l + r) / 2;
            auto checkm = check(mid);
            if(checkm == check0){
                auto t = dir0.toleft(p[mid] - p[0]);
                if((check0 == 0 && t <= 0) || (check0 && t < 0))    checkm ^= 1;
            }
            if(checkm)    l = mid + 1;
            else          r = mid;
        }
        return r;
    }
    //【已测试：Enclosure】
    //凸包外一点切点（返回两个切点下标）
    pair<int, int> tangent(const Point &a){
        int i = extreme([&](const Point &u){ return u - a; });
        int j = extreme([&](const Point &u){ return a - u; });
        return {i, j};
    }
    //【已测试：A highway and the seven dwarfs】
    //求直线与凸包上点的关系
    pair<int, int> tangent(const Line &l, const Point &dir){
        int i = extreme([&](...){ return dir; });
        int j = extreme([&](...){ return -dir; });
        return {i, j};
    }
    //O(logN)求直线是否穿过凸包
    bool isLineCrossConvex(const Line &l, const Point &dir){
        if(p.size() <= 1)    return true;
        if(p.size() == 2)    return l.toleft(p[0]) == l.toleft(p[1]);
        auto t = tangent(l, dir);
        return l.toleft(p[t.first]) == l.toleft(p[t.second]);
    }
};

//【已测试：Enclosure】
/* 拓展自Convex：利用前缀和求解凸包中若干【连续】点构成的小凸包的面积 */
struct sumConvex : Convex{
    vector<long double> sum;
    void init(){
        getSum();
    }
    void getSum(){
        auto &p = this->p;
        vector<long double> a(p.size());
        for(int i = 0; i < p.size(); i++){
            a[i] = p[this->pre(i)] ^ p[i];
        }
        sum.resize(p.size());
        partial_sum(a.begin(), a.end(), sum.begin());
    }
    long double queryTangentSum(const Point &a){
        auto &p = this->p;
        pair<int, int> result = this->tangent(a);
        int l = result.second, r = result.first;
        return querySum(l, r);
    }
    long double querySum(){
        return sum.back();
    }
    long double querySum(int l, int r){
        if(l <= r)    return sum[r] - sum[l] + (p[r] ^ p[l]);
        return sum[p.size() - 1] - sum[l] + sum[r] + (p[r] ^ p[l]);
    }
};
//【已测试：[HNOI2012]射箭】
/* 半平面（单个） */
struct Halfplane : Line{
    Halfplane(){}
    Halfplane(Point _s, Point _e){s = _s; e = _e;}
    //叉积排序，减少精度损失
    bool operator <(const Halfplane &b){
        Point A = e - s, B = b.e - b.s;
        return argcmpC(A, B);
    }
};

/* 半平面交（集合） */
struct Halfplanes{
    int n, st, ed, que[maxn];
    Point p[maxn]; Halfplane hp[maxn];
    //去重 & 便于判断非法情况
    void unique(){
        int m = 1;
        for(int i = 1; i < n; i++){
            if(!(sgn(hp[i] ^ hp[i - 1]) == 0 && sgn(hp[i] * hp[i - 1]) >= 0)){
                hp[m++] = hp[i];
            }
            else if(sgn((hp[m - 1].e - hp[m - 1].s) ^ (hp[i].s - hp[m - 1].s)) > 0){
                hp[m - 1] = hp[i];
            }
        }
        n = m;
    }
    //True -> 存在半平面交
    bool Halfplaneinsert(){
        sort(hp, hp + n); unique();
        que[st = 0] = 0; que[ed = 1] = 1;
        p[1] = hp[0].crosspoint(hp[1]);
        for(int i = 2; i < n; i++){
            while(st < ed && sgn((hp[i].e - hp[i].s) ^ (p[ed] - hp[i].s)) < 0)    ed--;
            while(st < ed && sgn((hp[i].e - hp[i].s) ^ (p[st + 1] - hp[i].s)) < 0)    st++;
            que[++ed] = i;
            if(hp[i].parallel(hp[que[ed - 1]]))    return false;
            p[ed] = hp[i].crosspoint(hp[que[ed - 1]]);
        }
        while(st < ed && sgn((hp[que[st]].e - hp[que[st]].s) ^ (p[ed] - hp[que[st]].s)) < 0)    ed--;
        while(st < ed && sgn((hp[que[ed]].e - hp[que[ed]].s) ^ (p[st + 1] - hp[que[ed]].s)) < 0)    st++;
        if(st + 1 >= ed)    return false;
        return true;
    }
    //Halfplaneinsert True -> 得到半平面交对应的凸包
    void getConvex(Polygon &con){
        p[st] = hp[que[st]].crosspoint(hp[que[ed]]);
        for(int j = st, i = 0; j <= ed; i++, j++){
            con.p.push_back(p[j]);
        }
    }
    //压入新的半平面
    void push(Halfplane tmp){
        hp[n++] = tmp;
    }
};

//【已测试：70D - Professor's task】
/* 动态凸包（set维护）：需选用前三个点确定BASIC点，而后进行维护，不支持删除 */
Point DBASIC;
bool argcmpB(const Point &A, const Point &B){
    Point p1 = A - DBASIC, p2 = B - DBASIC;
    //此处可以换用叉积版本维护
    long double len1 = p1.len2(), len2 = p2.len2();
    long double ang1 = p1.angle(), ang2 = p2.angle();
    if(sgn(ang1 - ang2) == 0)    return len1 < len2;
    return ang1 < ang2;
}
struct DConvexHull{
    set<Point, decltype(&argcmpB)> Set{&argcmpB};
    void init(const Point &A, const Point &B, const Point &C){
        //A,B,C为任意三点，处理完毕后直接调用Insert即可
        DBASIC = {(A.x + B.x + C.x) / 3,
                  (A.y + B.y + C.y) / 3};
        Set.insert(A); Set.insert(B); Set.insert(C);
    }
    set<Point>::iterator Pre(set<Point>::iterator it){
        if(it == Set.begin())    it = Set.end();
        return --it;
    }
    set<Point>::iterator Nxt(set<Point>::iterator it){
        ++it;
        return it == Set.end() ? Set.begin() : it;
    }
    //询问点是否在凸包内
    bool Query(Point v){
        auto it = Set.lower_bound(v);
        if(it == Set.end())    it = Set.begin();
        Point v1 = v - (*Pre(it));
        Point v2 = (*it) - (*Pre(it));
        return sgn(v1 ^ v2) <= 0;
    }
    //往凸包中插入一个新的点
    void Insert(Point v){
        if(Query(v))    return;
        Set.insert(v);
        auto it = Nxt(Set.find(v));
        while(Set.size() > 3 && sgn((v - (*Nxt(it))) ^ ((*it) - (*Nxt(it)))) <= 0){
            Set.erase(it);
            it = Nxt(Set.find(v));
        }
        it = Pre(Set.find(v));
        while(Set.size() > 3 && sgn((v - (*it)) ^ ((*it) - (*Pre(it)))) >= 0){
            Set.erase(it);
            it = Pre(Set.find(v));
        }
    }
};

signed main(void){
    cout << "Helloworld!\n";
}
```