动态开点线段树

```cpp
#include <bits/stdc++.h>
using ld = long double;
using i64 = long long;
using namespace std;

const int maxn = 1e5 + 5;
int mod = 1e9 + 7;
int rt = 1;

// 使用前先 init()
struct DST {
    int tot = 1;
    struct Node {
        int ls, rs, v, laz;
    } t[maxn * 66]; // Q(查询次数) * 2logV(值域大小)
    // 极端情况下单次修改可能创建 接近 2logV 个节点

#define me t[p]
#define lc t[t[p].ls]
#define rc t[t[p].rs]

    void init() {
        for (int i = 0; i <= tot; i++) {
            t[i] = {0, 0, 0, 0};
        }
        tot = 1;
    }
    void pushup(int p) {
        me.v = (lc.v + rc.v) % mod; // pushup
    }
    void pushdown(int p, int L, int R) {
        if (!me.laz) return;
        int M = L + R >> 1;
        if (!me.ls) me.ls = ++tot;
        if (!me.rs) me.rs = ++tot;
        lc.v = (lc.v + 1LL * me.laz * (M - L + 1)) % mod;
        rc.v = (rc.v + 1LL * me.laz * (R - M)) % mod;
        lc.laz = (lc.laz + me.laz) % mod;
        rc.laz = (rc.laz + me.laz) % mod;
        me.laz = 0;
    }
    // int rt = 1;
    // add(rt, 1, n, l, r, d);
    void add(int &p, int L, int R, int l, int r, int d) {
        if (!p) p = ++tot; // 开点
        if (r < L || R < l) return;
        if (l <= L && R <= r) {
```

```
            me.v = (1LL * me.v + 1LL * (R - L + 1) * d) % mod;
            me.laz = (me.laz + d) % mod;
            return;
        }
        pushdown(p, L, R);
        int M = L + R >> 1;
        add(me.ls, L, M, l, r, d);
        add(me.rs, M + 1, R, l, r, d);
        pushup(p);
    }
    int query(int p, int L, int R, int l, int r) {
        if (!p) return 0;
        if (l <= L && R <= r) return t[p].v;
        pushdown(p, L, R);
        int res = 0;
        int M = L + R >> 1;
        if (l <= M) res = (res + query(t[p].ls, L, M, l, r)) % mod;
        if (r > M) res = (res + query(t[p].rs, M + 1, R, l, r)) % mod;
        return res;
    }

#undef me
#undef lc
#undef rc

} T;

int main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);

    T.init();
    int m, n = 1000000000;
    cin >> m >> mod;
    while (m--) {
        int op;
        cin >> op;
        if (op == 1) {
            int x, y, z;
            cin >> x >> y >> z;
            T.add(rt, 1, n, x, y, z);
        } else {
            int x, y;
            cin >> x >> y;
            cout << T.query(rt, 1, n, x, y) << '\n';
        }
    }

    return 0;
}

```

二维全集

```cpp
#include <bits/stdc++.h>
using ld = long double;
using i64 = long long;
using namespace std;

// 有浮点数但是大部分都是 i64 时，可以选择用 pair<ld, ld> 替换必需内容
// 然后删去强制浮点数的函数，改用手写 pair<ld, ld> 替代。
using T = ld; // 全局数据类型

const T eps = 1e-12; // 注意一旦使用 T = i64，eps 就是 0LL ！！！
const T INF = numeric_limits<T>::max();
const T PI = acosl(-1); // 注意一旦使用 T = i64，PI 就是 3LL ！！！
#define setp(x) cout << fixed << setprecision(x)
int sgn(T x) { return x < -eps ? -1 : x > eps; }

// 点与向量
struct Pt {
    T x, y;
    bool operator==(const Pt &a) const { return (abs(x - a.x) <= eps && abs(y - a.y) <= eps); }
    bool operator!=(const Pt &a) const { return !(*this == a); }
    bool operator<(const Pt &a) const {
        if (abs(x - a.x) <= eps) return y < a.y - eps;
        return x < a.x - eps;
    }
    bool operator>(const Pt &a) const { return !(*this < a || *this == a); }
    Pt operator+(const Pt &a) const { return {x + a.x, y + a.y}; }
    Pt operator-(const Pt &a) const { return {x - a.x, y - a.y}; }
    Pt operator-() const { return {-x, -y}; }
    Pt operator*(const T k) const { return {k * x, k * y}; }
    Pt operator/(const T k) const { return {x / k, y / k}; }
    T operator*(const Pt &a) const { return x * a.x + y * a.y; } // 点积
    T operator^(const Pt &a) const { return x * a.y - y * a.x; } // 叉积，注意优先级
    int toleft(const Pt &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 Pt &a) const { return (a - (*this)).len2(); } // 两点距离的平方
    // 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
    int quad() const {
        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);
    }
    // i64 + pair<ld, ld>
```

```
49      // ld dis(const pair<ld, ld> &a) { return sqrtl((a.first - x) * (a.first - x) + (a.second - y)
    * (a.second - y)); }
50      // 必须用浮点数
51      T len() const { return sqrtl(len2()); }
    // 向量长度
52      T dis(const Pt &a) const { return sqrtl(dis2(a)); }
    // 两点距离
53      T ang(const Pt &a) const { return acosl(max(-1.0L, min(1.0L, ((*this) * a) / (len() *
    a.len())))); } // 向量夹角
54      Pt rot(const T rad) const { return {x * cosl(rad) - y * sinl(rad), x * sinl(rad) + y *
    cosl(rad)}; } // 逆时针旋转（给定角度）
55      Pt 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）
56  };
57
58  // pair<ld, ld> 叉积
59  ld operator^(const pair<ld, ld> &a, const pair<ld, ld> &b) {
60      return a.first * b.second - a.second * b.first;
61  }
62
63  // 极角排序
64  struct Argcmp {
65      bool operator()(const Pt &a, const Pt &b) const {
66          const int qa = a.quad(), qb = b.quad();
67          if (qa != qb) return qa < qb;
68          const auto t = a ^ b;
69          // 不同长度的向量需要分开
70          // if (abs(t) <= eps) return a * a < b * b - eps;
71          return t > eps;
72      }
73  };
74
75  // 直线
76  struct Lt {
77      Pt p, v; // p 为直线上一点，v 为方向向量
78      bool operator==(const Lt &a) const { return v.toleft(a.v) == 0 && v.toleft(p - a.p) == 0; }
79      int toleft(const Pt &a) const { return v.toleft(a - p); } // to-left 测试
80      // 半平面交算法定义的排序
81      bool operator<(const Lt &a) const {
82          if (abs(v ^ a.v) <= eps && v * a.v >= -eps) return toleft(a.p) == -1;
83          return Argcmp()(v, a.v);
84      }
85      // i64 + pair<ld, ld>
86      // pair<ld, ld> inter(const Lt &a) const {
87      //      ld k = 1.0L * (a.v ^ (p - a.p)) / (v ^ a.v);
88      //      return {p.x + v.x * k, p.y + v.y * k};
89      // }
90      // 必须用浮点数
91      Pt inter(const Lt &a) const { return p + v * ((a.v ^ (p - a.p)) / (v ^ a.v)); } // 直线交点
92      T dis(const Pt &a) const { return abs(v ^ (a - p)) / v.len(); }                 // 点到直线距离
93      Pt pedal(const Pt &a) const { return p + v * ((v * (a - p)) / (v * v)); }       // 点在直线上的
```

```
    投影
};

// 线段
struct St {
    Pt a, b;
    bool operator<(const St &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
    int is_on(const Pt &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 Lt &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 St &s) const {
        if (is_on(s.a) || is_on(s.b) || s.is_on(a) || s.is_on(b)) return -1;
        const Lt 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;
    }
    // 计算线段上的网格点数量（包括端点）
    // 要求端点都在网格点上（必须用整数）
    i64 calc_pts_on_segment() const {
        i64 dx = (a.x - b.x), dy = (a.y - b.y);
        if (dx < 0) dx = -dx;
        if (dy < 0) dy = -dy;
        return gcd(dx, dy) + 1;
    }
    // 点到线段距离（必须用浮点数）
    T dis(const Pt &p) const {
        if ((p - a) * (b - a) < -eps || (p - b) * (a - b) < -eps) return min(p.dis(a), p.dis(b));
        const Lt l{a, b - a};
        return l.dis(p);
    }
    // 两线段间距离（必须用浮点数）
    T dis(const St &s) const {
        if (is_inter(s)) return 0;
        return min({dis(s.a), dis(s.b), s.dis(a), s.dis(b)});
    }
    // 计算线段的中垂线
    Lt midperp() const {
        Pt mid = (a + b) / 2; // 线段中点
        Pt vec = b - a;
        return {mid, Pt{-vec.y, vec.x}};
```

```
    }
};

// 多边形
struct Polygon {
    vector<Pt> p; // 以逆时针顺序存储
    int ne(const int i) const { return i == p.size() - 1 ? 0 : i + 1; }
    int pre(const int 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 Pt &a) const {
        int cnt = 0;
        for (int i = 0; i < p.size(); i++) {
            const Pt u = p[i], v = p[ne(i)];
            if (abs((a - u) ^ (a - v)) <= eps && (a - u) * (a - v) <= eps) return {true, 0};
            if (abs(u.y - v.y) <= eps) continue;
            const Lt 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};
    }
    // 计算多边形内（不包括边上）的网格点数（必须用整数）
    i64 calc_pts_in_polygon() const {
        i64 S = 0, C = 0;
        for (int i = 0; i < p.size(); i++) {
            C += St{p[i], p[ne(i)]}.calc_pts_on_segment() - 1;
        }
        // 注意这里 S 和 C 其实可能 有 .5 的
        // 只不过要有一起有，计算的时候就可以抵掉了。
        S = area() / 2;
        // 多边形面积 = 多边形内网格点数 + 多边形边上网格点数 / 2 - 1
        // 平行四边形格点仍然成立，三角形格点 等式右边 * 2
        // S = I + C / 2 - 1
        return S - C / 2 + 1;
    }
    // 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 (int i = 0; i < p.size(); i++) sum += p[i] ^ p[ne(i)];
        return sum;
    }
    // 多边形的周长
    T circ() const {
        T sum = 0;
        for (int i = 0; i < p.size(); i++) sum += p[i].dis(p[ne(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<St> e1(p.size()), e2(c.p.size()), edge(p.size() + c.p.size());
        vector<Pt> res;
        res.reserve(p.size() + c.p.size());
        const auto cmp = [](const St &u, const St &v) { return Argcmp()(u.b - u.a, v.b - v.a); };
        for (int i = 0; i < p.size(); i++) e1[i] = {p[i], p[this->ne(i)]};
        for (int i = 0; i < c.p.size(); i++) e2[i] = {c.p[i], c.p[c.ne(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<Pt> &res, const Pt &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};
    }
    // 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 Pt &u, const Pt &v, const Pt &w) { return (w - u) ^ (w - v); };
        T ans = 0;
        for (int i = 0, j = 1; i < p.size(); i++) {
            const auto nxti = this->ne(i);
            ans = max({ans, p[j].dis2(p[i]), p[j].dis2(p[nxti])});
            while (area(p[this->ne(j)], p[i], p[nxti]) >= area(p[j], p[i], p[nxti])) {
                j = this->ne(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 Pt &a) const {
        const auto &p = this->p;
        if (p.size() == 1) return a == p[0] ? -1 : 0;
        if (p.size() == 2) return St{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 Pt &u, const Pt &v) { return (u - p[0]).toleft(v - p[0]) == 1; };
        const int i = lower_bound(p.begin() + 1, p.end(), a, cmp) - p.begin();
        if (i == 1) return St{p[0], p[i]}.is_on(a) ? -1 : 0;
        if (i == p.size() - 1 && St{p[0], p[i]}.is_on(a)) return -1;
        if (St{p[i - 1], p[i]}.is_on(a)) return -1;
        return (p[i] - p[i - 1]).toleft(a - p[i - 1]) > 0;
    }
    // 求点在凸多边形上的两侧的点，逆时针给出下标
    // 必须保证 is_in(a) == -1
    pair<int, int> around_points(const Pt &a) const {
        const auto &p = this->p;
        if (a == p[0]) return {p.size() - 1, 1};
        if (St{p.back(), p[0]}.is_on(a) == 1) return {p.size() - 1, 0};
        if (St{p[0], p[1]}.is_on(a) == 1) return {0, 1};
        const auto cmp = [&](const Pt &u, const Pt &v) { return (u - p[0]).toleft(v - p[0]) == 1; };
        const int i = lower_bound(p.begin() + 1, p.end(), a, cmp) - p.begin();
        if (St{p[this->pre(i)], p[i]}.is_on(a) == -1) return {pre(i), ne(i)};
        return {this->pre(i), i};
    }
    // 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
    // 凸包点数 >= 3
    template <typename F>
    int extreme(const F &dir) const {
        const auto &p = this->p;
        const auto check = [&](const int i) { return dir(p[i]).toleft(p[this->ne(i)] - p[i]) >= 0; };
        const auto dir0 = dir(p[0]);
        const auto check0 = check(0);
        if (!check0 && check(p.size() - 1)) return 0;
        const auto cmp = [&](const Pt &v) {
            const int 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();
    }
```

```
292     // 过凸多边形外一点求凸多边形的切线，逆时针返回切点下标
293     // 复杂度 O(logn)
294     // 必须保证 点不在多边形内
295     // 凸多边形上返回两侧的点
296     // 凸包点数 >= 3
297     pair<int, int> tangent(const Pt &a) const {
298         if (is_in(a) == -1) return around_points(a);
299         const int i = extreme([&](const Pt &u) { return u - a; });
300         const int j = extreme([&](const Pt &u) { return a - u; });
301         return {i, j};
302     }
303     // 求平行于给定直线的凸多边形的切线，返回切点下标
304     // 复杂度 O(logn)
305     // 凸包点数 >= 3
306     pair<int, int> tangent(const Lt &a) const {
307         const int i = extreme([&](...) { return a.v; });
308         const int j = extreme([&](...) { return -a.v; });
309         return {i, j};
310     }
311     // 最小矩形覆盖 O(n)，返回面积和矩形端点（必须用浮点数）
312     // 矩形端点按逆时针给出。
313     pair<T, vector<Pt>> min_rectangle_cover() const {
314         const auto &p = this->p;
315         if (p.size() == 1) return {0, {}};
316         if (p.size() == 2) return {0, {}};
317         const auto dot = [&](const int u, const int v, const int w) { return (p[v] - p[u]) * (p[w]
    - p[u]); };
318         const auto cross = [&](const int u, const int v, const int w) { return (p[v] - p[u]) ^
    (p[w] - p[u]); };
319         int u = 1, d = 1;
320         T mu = INF, md = -INF;
321         for (int i = 0; i < p.size(); i++) {
322             if (mu >= dot(0, 1, i)) {
323                 mu = dot(0, 1, i);
324                 u = i;
325             }
326             if (md <= dot(0, 1, i)) {
327                 md = dot(0, 1, i);
328                 d = i;
329             }
330         }
331         T ans = INF;
332         vector<Pt> rec(4);
333         for (int l = 0, r = 1; l < p.size(); l++) {
334             const auto nxtl = this->ne(l);
335             while (cross(l, nxtl, r) <= cross(l, nxtl, this->ne(r))) r = this->ne(r);
336             while (dot(l, nxtl, u) >= dot(l, nxtl, this->ne(u))) u = this->ne(u);
337             while (dot(l, nxtl, d) <= dot(l, nxtl, this->ne(d))) d = this->ne(d);
338             Lt mid = St{p[l], p[nxtl]}.midperp();
339             Lt L = {p[l], p[nxtl] - p[l]};
340             Lt R = {p[r], p[nxtl] - p[l]};
```

```
            T res = L.dis(p[r]) * (mid.dis(p[u]) + mid.dis(p[d]));
            if (ans > res) {
                ans = res;
                rec[0] = L.pedal(p[u]);
                rec[1] = L.pedal(p[d]);
                rec[2] = R.pedal(p[d]);
                rec[3] = R.pedal(p[u]);
            }
        }
        return {ans, rec};
    }
};

// 圆 用浮点数
struct Circle {
    Pt c;
    T r;
    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 Pt &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 Lt &l) const {
        const T d = l.dis(c);
        if (d > r + eps) return 0;
        if (abs(d - r) <= eps) return 1;
        return 2;
    }
    // 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<Pt> inter(const Lt &l) const {
        const T d = l.dis(c);
        const Pt p = l.pedal(c);
        const int t = relation(l);
        if (t == 0) return vector<Pt>();
```

```
392         if (t == 1) return vector<Pt>{p};
393         const T k = sqrt(r * r - d * d);
394         return vector<Pt>{p - (l.v / l.v.len()) * k, p + (l.v / l.v.len()) * k};
395     }
396     // 圆与圆交点
397     vector<Pt> inter(const Circle &a) const {
398         const T d = c.dis(a.c);
399         const int t = relation(a);
400         if (t == -1 || t == 0 || t == 4) return vector<Pt>();
401         Pt e = a.c - c;
402         e = e / e.len() * r;
403         if (t == 1 || t == 3) {
404             if (r * r + d * d - a.r * a.r >= -eps) return vector<Pt>{c + e};
405             return vector<Pt>{c - e};
406         }
407         const T costh = (r * r + d * d - a.r * a.r) / (2 * r * d), sinth = sqrt(1 - costh *
    costh);
408         return vector<Pt>{c + e.rot(costh, -sinth), c + e.rot(costh, sinth)};
409     }
410     // 圆与圆交面积
411     T inter_area(const Circle &a) const {
412         const T d = c.dis(a.c);
413         const int t = relation(a);
414         if (t == -1) return area();
415         if (t < 2) return 0;
416         if (t > 2) return min(area(), a.area());
417         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);
418         const T sinth1 = sqrt(1 - costh1 * costh1), sinth2 = sqrt(1 - costh2 * costh2);
419         const T th1 = acos(costh1), th2 = acos(costh2);
420         return r * r * (th1 - costh1 * sinth1) + a.r * a.r * (th2 - costh2 * sinth2);
421     }
422     // 过圆外一点圆的切线
423     vector<Lt> tangent(const Pt &a) const {
424         const int t = is_in(a);
425         if (t == 1) return vector<Lt>();
426         if (t == -1) {
427             const Pt v = {-(a - c).y, (a - c).x};
428             return vector<Lt>{{a, v}};
429         }
430         Pt e = a - c;
431         e = e / e.len() * r;
432         const T costh = r / c.dis(a), sinth = sqrt(1 - costh * costh);
433         const Pt t1 = c + e.rot(costh, -sinth), t2 = c + e.rot(costh, sinth);
434         return vector<Lt>{{a, t1 - a}, {a, t2 - a}};
435     }
436     // 两圆的公切线
437     vector<Lt> tangent(const Circle &a) const {
438         const int t = relation(a);
439         vector<Lt> lines;
440         if (t == -1 || t == 4) return lines;
```

```
        if (t == 1 || t == 3) {
            const Pt 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 Pt e = (a.c - c) / (a.c - c).len();
        if (t <= 2) {
            const T costh = (r - a.r) / d, sinth = sqrt(1 - costh * costh);
            const Pt d1 = e.rot(costh, -sinth), d2 = e.rot(costh, sinth);
            const Pt 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 Pt d1 = e.rot(costh, -sinth), d2 = e.rot(costh, sinth);
            const Pt 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, Lt> inverse(const Lt &l) const {
        if (l.toleft(c) == 0) return l;
        const Pt v = l.toleft(c) == 1 ? Pt{l.v.y, -l.v.x} : Pt{-l.v.y, l.v.x};
        const T d = r * r / l.dis(c);
        const Pt p = c + v / v.len() * d;
        return Circle{(c + p) / 2, d / 2};
    }
    std::variant<Circle, Lt> inverse(const Circle &a) const {
        const Pt v = a.c - c;
        if (a.is_in(c) == -1) {
            const T d = r * r / (a.r + a.r);
            const Pt p = c + v / v.len() * d;
            return Lt{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 Pt p = c + v / v.len() * d1, q = c + v / v.len() * d2;
        return Circle{(p + q) / 2, p.dis(q) / 2};
    }
};

// 点集的凸包
// Andrew 算法，复杂度 O(nlogn)
Convex convexhull(vector<Pt> p) {
    vector<Pt> st;
```

```
    if (p.size() <= 2) return Convex{p};
    sort(p.begin(), p.end());
    const auto check = [](const vector<Pt> &st, const Pt &u) {
        const auto back1 = st.back(), back2 = *prev(st.end(), 2);
        return (back1 - back2).toleft(u - back1) <= 0;
    };
    for (const Pt &u : p) {
        while (st.size() > 1 && check(st, u)) st.pop_back();
        st.push_back(u);
    }
    int k = st.size();
    p.pop_back();
    reverse(p.begin(), p.end());
    for (const Pt &u : p) {
        while (st.size() > k && check(st, u)) st.pop_back();
        st.push_back(u);
    }
    st.pop_back();
    return Convex{st};
}

// 最小圆覆盖 | 期望 O(n) | 必须用浮点数
Circle min_circle_cover(vector<Pt> a) {
    mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
    // 两点确定一个圆
    auto get2 = [](const Pt &a, const Pt &b) -> Circle {
        return {(a + b) / 2, a.dis(b) / 2};
    };
    // 三点确定一个圆
    auto get3 = [](const Pt &a, const Pt &b, const Pt &c) -> Circle {
        Lt u = St{a, b}.midperp(), v = St{a, c}.midperp();
        Pt p = u.inter(v);
        return {p, a.dis(p)};
    };
    shuffle(a.begin(), a.end(), rng);
    Circle C{a[0], 0};
    for (int i = 1; i < a.size(); i++) {
        if (C.r < C.c.dis(a[i])) {
            C = {a[i], 0};
            for (int j = 0; j < i; j++) {
                if (C.r < C.c.dis(a[j])) {
                    C = get2(a[i], a[j]);
                    for (int k = 0; k < j; k++) {
                        if (C.r < C.c.dis(a[k])) {
                            C = get3(a[i], a[j], a[k]);
                        }
                    }
                }
            }
        }
    }
```

```
543     return C;
544 }
545 
546 // 半平面交
547 // 排序增量法，复杂度 O(nlogn)
548 // 输入与返回值都是用直线表示的半平面集合
549 vector<Lt> halfinter(vector<Lt> l, const T lim = 1e9) {
550     // 必须用浮点数，return <= 则会去掉冗余线
551     const auto check = [](const Lt &a, const Lt &b, const Lt &c) { return a.toleft(b.inter(c)) <
    0; };
552     // 无精度误差的方法，但注意取值范围会扩大到三次方
553     // const auto check = [](const Lt &a, const Lt &b, const Lt &c) {
554     //     const Pt 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);
555     //     return p.toleft(q) < 0;
556     // };
557     l.push_back({{-lim, 0}, {0, -1}});
558     l.push_back({{0, -lim}, {1, 0}});
559     l.push_back({{lim, 0}, {0, 1}});
560     l.push_back({{0, lim}, {-1, 0}});
561     sort(l.begin(), l.end());
562     deque<Lt> q;
563     for (int i = 0; i < l.size(); i++) {
564         if (i > 0 && l[i - 1].v.toleft(l[i].v) == 0 && l[i - 1].v * l[i].v > eps) continue;
565         while (q.size() > 1 && check(l[i], q.back(), q[q.size() - 2])) q.pop_back();
566         while (q.size() > 1 && check(l[i], q[0], q[1])) q.pop_front();
567         if (!q.empty() && q.back().v.toleft(l[i].v) <= 0) return vector<Lt>();
568         q.push_back(l[i]);
569     }
570     while (q.size() > 1 && check(q[0], q.back(), q[q.size() - 2])) q.pop_back();
571     while (q.size() > 1 && check(q.back(), q[0], q[1])) q.pop_front();
572     return vector<Lt>(q.begin(), q.end());
573 }
574 
575 // 平面最近点对
576 // 扫描线，复杂度 O(nlogn)
577 // 返回最近点对距离平方
578 T closest_pair(vector<Pt> pts) {
579     sort(pts.begin(), pts.end());
580     const auto cmpy = [](const Pt &a, const Pt &b) {
581         if (abs(a.y - b.y) <= eps) return a.x < b.x - eps;
582         return a.y < b.y - eps;
583     };
584     multiset<Pt, decltype(cmpy)> s{cmpy};
585     T ans = INF;
586     for (int i = 0, l = 0; i < pts.size(); i++) {
587         const T sqans = sqrtl(ans) + 1;
588         while (l < i && pts[i].x - pts[l].x >= sqans) s.erase(s.find(pts[l++]));
589         for (auto it = s.lower_bound(Pt{-INF, pts[i].y - sqans}); it != s.end() && it->y -
    pts[i].y <= sqans; it++) {
590             ans = min(ans, pts[i].dis2(*it));
```

```
        }
        s.insert(pts[i]);
    }
    return ans;
}

// 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<Pt> &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 (int i = 0; i < vec.size(); i++) {
        for (int 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 Pt 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<int> vx(vec.size()), pos(vec.size());
    for (int i = 0; i < vec.size(); i++) vx[i] = i;
    sort(vx.begin(), vx.end(), [&](int x, int y) { return vec[x] < vec[y]; });
    for (int i = 0; i < vx.size(); i++) pos[vx[i]] = i;
    for (auto [u, v] : evt) {
        const int i = pos[u], j = pos[v];
        const int l = min(i, j), r = max(i, j);
        const Pt 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};
}

// 判断多条线段是否有交点
// 扫描线，复杂度 O(nlogn)
bool segs_inter(const vector<St> &segs) {
    if (segs.empty()) return false;
    using seq_t = tuple<T, int, St>; // 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 St &u, const St &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<St, 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;
}

// 圆与多边形面积交
T area_inter(const Circle &circ, const Polygon &poly) {
    const auto cal = [](const Circle &circ, const Pt &a, const Pt &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 Lt 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 Pt 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 Pt 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 && St{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;
        }
        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 (int i = 0; i < poly.p.size(); i++) {
        const Pt a = poly.p[i], b = poly.p[poly.ne(i)];
        ans += cal(circ, a, b);
    }
    return ans;
}

// 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 int siz = polys.size();
    vector<vector<pair<Pt, Pt>>> segs(siz);
    const auto check = [](const Pt &u, const St &e) { return !((u < e.a && u < e.b) || (u > e.a &&
u > e.b)); };
    auto cut_edge = [&](const St &e, const int i) {
        const Lt le{e.a, e.b - e.a};
        vector<pair<Pt, int>> evt;
        evt.push_back({e.a, 0});
        evt.push_back({e.b, 0});
        for (int j = 0; j < polys.size(); j++) {
            if (i == j) continue;
            const auto &pj = polys[j];
            for (int k = 0; k < pj.p.size(); k++) {
                const St s = {pj.p[k], pj.p[pj.ne(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 Lt ls{s.a, s.b - s.a};
```

```
                    const Pt 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});
                }
            }
        }
        sort(evt.begin(), evt.end());
        if (e.a > e.b) reverse(evt.begin(), evt.end());
        int sum = 0;
        for (int i = 0; i < evt.size(); i++) {
            sum += evt[i].second;
            const Pt 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 (int i = 0; i < polys.size(); i++) {
        const auto &pi = polys[i];
        for (int k = 0; k < pi.p.size(); k++) {
            const St ei = {pi.p[k], pi.p[pi.ne(k)]};
            cut_edge(ei, i);
        }
    }
    vector<T> ans(siz);
    for (int i = 0; i < siz; i++) {
        T sum = 0;
        sort(segs[i].begin(), segs[i].end());
        int cnt = 0;
        for (int 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 int siz = circs.size();
    using arc_t = tuple<Pt, 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 int i) {
        vector<pair<T, int>> evt;
        evt.push_back({-PI, 0});
        evt.push_back({PI, 0});
        int init = 0;
        for (int 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 Pt 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 (int 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 * (sinl(r) - sinl(l)) - cc.y * cr * (cosl(r) - cosl(l));
    };
    for (int i = 0; i < circs.size(); i++) {
        const auto &ci = circs[i];
        cut_circ(ci, i);
    }
    vector<T> ans(siz);
    for (int i = 0; i < siz; i++) {
```

```
838         T sum = 0;
839         sort(arcs[i].begin(), arcs[i].end());
840         int cnt = 0;
841         for (int j = 0; j < arcs[i].size(); j++) {
842             if (j > 0 && eq(arcs[i][j], arcs[i][j - 1]))
843                 arcs[i + (++cnt)].push_back(arcs[i][j]);
844             else
845                 cnt = 0, sum += oint(arcs[i][j]);
846         }
847         ans[i] = sum / 2;
848     }
849     return ans;
850 }
851
```

动态凸包

```cpp
// https://codeforces.com/problemset/problem/70/D
#include <bits/stdc++.h>
using ld = long double;
using i64 = long long;
using namespace std;

// 有浮点数但是大部分都是 i64 时，可以选择用 pair<ld, ld> 替换必需内容
// 然后删去强制浮点数的函数，改用手写 pair<ld, ld> 替代。
using T = i64; // 全局数据类型

const T eps = 1e-12;
const T INF = numeric_limits<T>::max();
const T PI = acosl(-1);
#define setp(x) cout << fixed << setprecision(x)

// 点与向量
struct Pt {
    T x, y;
    bool operator==(const Pt &a) const { return (abs(x - a.x) <= eps && abs(y - a.y) <= eps); }
    bool operator!=(const Pt &a) const { return !(*this == a); }
    bool operator<(const Pt &a) const {
        if (abs(x - a.x) <= eps) return y < a.y - eps;
        return x < a.x - eps;
     }
     bool operator>(const Pt &a) const { return !(*this < a || *this == a); }
    Pt operator+(const Pt &a) const { return {x + a.x, y + a.y}; }
    Pt operator-(const Pt &a) const { return {x - a.x, y - a.y}; }
    Pt operator-() const { return {-x, -y}; }
    Pt operator*(const T k) const { return {k * x, k * y}; }
    Pt operator/(const T k) const { return {x / k, y / k}; }
    T operator*(const Pt &a) const { return x * a.x + y * a.y; } // 点积
    T operator^(const Pt &a) const { return x * a.y - y * a.x; } // 叉积，注意优先级
    int toleft(const Pt &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 Pt &a) const { return (a - (*this)).len2(); } // 两点距离的平方
    // 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
    int quad() const {
        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);
    }
};
```

```cpp

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

// 注意整数时需要全体坐标在外面 * 3，才能保证准确
// 注意要把 Argcmp 的 区分长度注释解除
struct DynamicConvex {
    Pt o;
    set<Pt, Argcmp> ss; // 注意 ss 中的点 已经 - o
    using iter = set<Pt, Argcmp>::iterator;
    DynamicConvex(const Pt &a, const Pt &b, const Pt &c) {
        o = (a + b + c) / 3;
        ss.insert(a - o);
        ss.insert(b - o);
        ss.insert(c - o);
    }
    iter pre(iter it) const { return it == ss.begin() ? --ss.end() : (--it); }
    iter nxt(iter it) const { return (++it) == ss.end() ? ss.begin() : it; }
    // -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(Pt x) const {
        x = x - o;
        auto it = ss.lower_bound(x);
        if (it != ss.end() && (*it ^ x) == 0) return x * x <= *it * *it;
        iter l = pre(it);
        iter r = (it == ss.end()) ? ss.begin() : it;
        T res = (*l - x ^ *r - x);
        if (abs(res) <= eps) return -1;
        return res > 0;
    }
    // 凸包中加入点
    void insert(Pt x) {
        if (is_in(x)) return;
        x = x - o;
        auto [it, _] = ss.insert(x);
        while (ss.size() >= 4 && (*it - (*nxt(it)) ^ (*nxt(nxt(it))) - *nxt(it)) >= 0)
ss.erase(nxt(it));
        while (ss.size() >= 4 && ((*pre(pre(it))) - *pre(it) ^ *it - (*pre(it))) >= 0)
ss.erase(pre(it));
    }
    // 判断加入了这个点后...
    bool tryToInsert(Pt x) {
        // 能否成功加入这个点
```

```
98          if (is_in(x)) return false;
99          x = x - o;
100         auto [it, _] = ss.insert(x);
101         // 是否会删除其他的点
102         if (ss.size() >= 4 && ((*it - (*nxt(it)) ^ (*nxt(nxt(it))) - *nxt(it)) >= 0 ||
    ((*pre(pre(it))) - *pre(it) ^ *it - (*pre(it))) >= 0)) {
103             ss.erase(x);
104             return 0;
105         }
106         ss.erase(x);
107         return 1;
108     }
109     // 强制擦除点
110     // 注意如果 erase 导致凸包重心转移，需要重新创建新的凸包
111     void erase(Pt x) {
112         x = x - o;
113         ss.erase(x);
114     }
115 };
116
117 int main() {
118     ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
119
120     int n;
121     cin >> n;
122     vector<int> op(n + 1);
123     vector<Pt> pts(n + 1);
124     for (int i = 1; i <= n; i++) {
125         cin >> op[i] >> pts[i].x >> pts[i].y;
126         pts[i] = pts[i] * 3;
127     }
128     DynamicConvex D(pts[1], pts[2], pts[3]);
129     for (int i = 4; i <= n; i++) {
130         if (op[i] == 1) { // 添加
131             D.insert(pts[i]);
132         } else { // 查询
133             if (D.is_in(pts[i])) {
134                 cout << "YES\n";
135             } else {
136                 cout << "NO\n";
137             }
138         }
139     }
140     return 0;
141 }
142
```

动态凸包重构

计算能构成凸包的连续区间数

```cpp
// 2025 HDU 多校 10 1005
#include <bits/stdc++.h>
using ld = long double;
using i64 = long long;
using namespace std;

// 有浮点数但是大部分都是 i64 时，可以选择用 pair<ld, ld> 替换必需内容
// 然后删去强制浮点数的函数，改用手写 pair<ld, ld> 替代。
using T = i64; // 全局数据类型

const T eps = 1e-12;
const T INF = numeric_limits<T>::max();
const T PI = acosl(-1);
#define setp(x) cout << fixed << setprecision(x)

// 点与向量
struct Pt {
    T x, y;
    bool operator==(const Pt &a) const { return (abs(x - a.x) <= eps && abs(y - a.y) <= eps); }
    bool operator!=(const Pt &a) const { return !(*this == a); }
    bool operator<(const Pt &a) const {
        if (abs(x - a.x) <= eps) return y < a.y - eps;
        return x < a.x - eps;
     }
     bool operator>(const Pt &a) const { return !(*this < a || *this == a); }
    Pt operator+(const Pt &a) const { return {x + a.x, y + a.y}; }
    Pt operator-(const Pt &a) const { return {x - a.x, y - a.y}; }
    Pt operator-() const { return {-x, -y}; }
    Pt operator*(const T k) const { return {k * x, k * y}; }
    Pt operator/(const T k) const { return {x / k, y / k}; }
    T operator*(const Pt &a) const { return x * a.x + y * a.y; } // 点积
    T operator^(const Pt &a) const { return x * a.y - y * a.x; } // 叉积，注意优先级
    int toleft(const Pt &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 Pt &a) const { return (a - (*this)).len2(); } // 两点距离的平方
    // 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
    int quad() const {
        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);
    }
};

// 极角排序
struct Argcmp {
```

```
52      bool operator()(const Pt &a, const Pt &b) const {
53          const int qa = a.quad(), qb = b.quad();
54          if (qa != qb) return qa < qb;
55          const auto t = a ^ b;
56          // 不同长度的向量需要分开
57          if (abs(t) <= eps) return a * a < b * b - eps;
58          return t > eps;
59      }
60  };
61
62  // 注意整数时需要全体坐标在外面 * 3，才能保证准确
63  // 注意要把 Argcmp 的 区分长度注释解除
64  struct DynamicConvex {
65      Pt o;
66      set<Pt, Argcmp> ss;
67      using iter = set<Pt, Argcmp>::iterator;
68      DynamicConvex(const Pt &a, const Pt &b, const Pt &c) {
69          o = (a + b + c) / 3;
70          ss.insert(a - o);
71          ss.insert(b - o);
72          ss.insert(c - o);
73      }
74      iter pre(iter it) const { return it == ss.begin() ? --ss.end() : (--it); }
75      iter nxt(iter it) const { return (++it) == ss.end() ? ss.begin() : it; }
76      // -1 点在多边形边上 | 0 点在多边形外 | 1 点在多边形内
77      int is_in(Pt x) const {
78          x = x - o;
79          auto it = ss.lower_bound(x);
80          if (it != ss.end() && (*it ^ x) == 0) return x * x <= *it * *it;
81          iter l = pre(it);
82          iter r = (it == ss.end()) ? ss.begin() : it;
83          T res = (*l - x ^ *r - x);
84          if (abs(res) <= eps) return -1;
85          return res > 0;
86      }
87      void insert(Pt x) {
88          // if (is_in(x) == 1) return;
89          x = x - o;
90          auto [it, _] = ss.insert(x);
91          while (ss.size() >= 4 && (*it - (*nxt(it)) ^ (*nxt(nxt(it))) - *nxt(it)) > 0)
    ss.erase(nxt(it));
92          while (ss.size() >= 4 && ((*pre(pre(it))) - *pre(it) ^ *it - (*pre(it))) > 0)
    ss.erase(pre(it));
93      }
94      bool tryToInsert(Pt x) {
95          if (is_in(x) == 1) return false;
96          x = x - o;
97          auto [it, _] = ss.insert(x);
98          if (ss.size() >= 4 && ((*it - (*nxt(it)) ^ (*nxt(nxt(it))) - *nxt(it)) > 0 ||
    ((*pre(pre(it))) - *pre(it) ^ *it - (*pre(it))) > 0)) {
99              ss.erase(x);
```

```
            return 0;
        }
        ss.erase(x);
        return 1;
    }
    // 注意如果 erase 导致凸包重心转移，需要重新创建新的凸包
    void erase(Pt x) {
        x = x - o;
        ss.erase(x);
    }
};

void solve(int tt) {
    int n;
    cin >> n;
    vector<Pt> pts(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> pts[i].x >> pts[i].y;
        pts[i] = pts[i] * 3;
    }
    // 最左边不共线的点
    vector<int> lf(n + 1);
    int fir = 0;
    for (int i = 3; i <= n; i++) {
        if ((pts[i - 2] - pts[i - 1] ^ pts[i] - pts[i - 1]) == 0) {
            lf[i] = lf[i - 1];
        } else {
            lf[i] = i - 2;
        }
        if (!fir && lf[i]) fir = i;
    }
    if (!fir) {
        cout << 1LL * n * (n + 1) / 2 << '\n';
        return;
    }
    DynamicConvex D(pts[1], pts[fir - 1], pts[fir]);
    for (int i = 1; i <= fir; i++) {
        D.insert(pts[i]);
    }
    i64 ans = 1LL * fir * (fir + 1) / 2;
    for (int mid = 1, L = 1, R = fir + 1; R <= n; R++) {
        while (L < R && !D.tryToInsert(pts[R])) {
            D.erase(pts[L]);
            L++;
            if (mid < L) { // rebuild
                D = DynamicConvex(pts[lf[R]], pts[R - 1], pts[R]);
                mid = lf[R];
                for (int i = lf[R] + 1; i <= R - 2; i++) {
                    D.insert(pts[i]);
                }
                int newL = lf[R];
```

```
                for (int i = lf[R] - 1; i >= L && D.tryToInsert(pts[i]); i--) {
                    D.insert(pts[i]);
                    newL = i;
                }
                L = newL;
                break;
            }
        }
        D.insert(pts[R]);
        // cout << L << ' ' << R << '\n';
        ans += R - L + 1;
    }
    cout << ans << '\n';
}

int main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int tt = 1;
    cin >> tt;
    for (int i = 1; i <= tt; i++) {
        solve(i);
    }
    return 0;
}
```

三维基础

```cpp
#include <bits/stdc++.h>
using ld = long double;
using i64 = long long;
using namespace std;

using T = ld; // 全局数据类型

const T eps = 1e-12;
const T INF = numeric_limits<T>::max();
const T PI = acosl(-1);
#define setp(x) cout << fixed << setprecision(x)

struct Pt {
    T x, y, z;
    bool operator==(const Pt &a) const { return (abs(x - a.x) <= eps && abs(y - a.y) <= eps) && abs(z - a.z) <= eps; }
    bool operator!=(const Pt &a) const { return !(*this == a); }
    bool operator<(const Pt &a) const {
        if (abs(x - a.x) <= eps && abs(y - a.y) <= eps) return z < a.z - eps;
        if (abs(x - a.x) <= eps) return y < a.y - eps;
        return x < a.x - eps;
    }
    bool operator>(const Pt &a) const { return !(*this < a || *this == a); }
    Pt operator+(const Pt &a) const { return {x + a.x, y + a.y, z + a.z}; }
    Pt operator-(const Pt &a) const { return {x - a.x, y - a.y, z - a.z}; }
    Pt operator-() const { return {-x, -y - z}; }
    Pt operator*(const T k) const { return {k * x, k * y, k * z}; }
    Pt operator/(const T k) const { return {x / k, y / k, z / k}; }
    T operator*(const Pt &a) const { return x * a.x + y * a.y + z * a.z; }
    // 注意三维叉积是向量
    Pt operator^(const Pt &a) const { return {y * a.z - z * a.y, z * a.x - x * a.z, x * a.y - y * a.x}; }
    T len2() const { return (*this) * (*this); }               // 向量长度的平方
    T dis2(const Pt &a) const { return (a - (*this)).len2(); } // 两点距离的平方
    // 必须用浮点数
    T len() const { return sqrtl(len2()); }
// 向量长度
    T dis(const Pt &a) const { return sqrtl(dis2(a)); }
// 两点距离
    T ang(const Pt &a) const { return acosl(max(-1.0L, min(1.0L, (*this) * a) / (len() * a.len()))); } // 向量夹角
    // 自身绕 向量v 逆时针旋转
    Pt rot(Pt v, const T a) const {
        v = v / v.len();         // 单位化轴
        Pt p1 = v * (v * *this); // 平行分量
        Pt p2 = *this - p1;      // 垂直分量 1
        if (p2.len() <= eps) return *this;
        Pt p3 = v ^ p2;                  // 垂直分量 2
```

```
        p3 = p3 * (p2.len() / p3.len()); // 调整成 p2 长度
        return p1 + p2 * cosl(a) - p3 * sinl(a);
    }
};

struct Plane {
    // 点 法向量
    Pt p, n;
    int tofront(const Pt &a) const { // 点在法向量方向
        T t = (a - p) * n;
        return (t > eps) - (t < -eps);
    }
    // 必须用浮点数
    T dis(const Pt &a) const { return abs((a - p) * n) / n.len(); }
    // 求所有点距离平面的最大值：
    // (所有点的法向量的点积的 max - min) / n.len()
};

```

三维凸包

```cpp
// O(n²) 增量法求三维凸包
#include <bits/stdc++.h>
using namespace std;
using ld = long double;

const int maxn = 2e3 + 3;
const ld eps = 1e-12; // 精度要求高一点

int n, m;           // n是总点数，m是凸包中平面的数量
bool g[maxn][maxn]; // g用来判断一条边被照到几次

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

// 用rand函数来生成一个非常小的随机数
double rand_eps() {
    // 用rand生成一个-0.5到0.5之间的数，再乘eps，就得到了一个非常小的随机数
    return (fmodl(1.0L * rng(), 1.0L) - 0.5L) * eps;
}

struct Pt {
    ld x, y, z;
    // 微小扰动，给每个坐标都加一个极小的随机数
    void shake() { x += rand_eps(), y += rand_eps(), z += rand_eps(); }
    Pt operator-(Pt t) { return {x - t.x, y - t.y, z - t.z}; }
    Pt operator^(Pt t) { return {y * t.z - t.y * z, t.x * z - x * t.z, x * t.y - y * t.x}; }
    ld operator*(Pt t) { return x * t.x + y * t.y + z * t.z; }
    ld len() { return sqrtl(x * x + y * y + z * z); }
} p[maxn]; // p来存所有点

struct Plane {  // 定义平面的结构体
    int v[3];   // 三个顶点
    Pt norm() { // 求法向量
        return (p[v[1]] - p[v[0]]) ^ (p[v[2]] - p[v[0]]);
    }
    bool above(Pt t) { // 判断一个点是否在平面上方
        return ((t - p[v[0]]) * norm()) >= 0;
    }
    ld area() { // 求三角形的面积
        return norm().len() / 2;
    }
} plane[maxn], tp[maxn];
// plane存凸包上的平面，tp用来更新凸包，
// 每次凸包上要留的平面和新加的平面都存进tp，
// 要删的平面不存，最后将tp在复制给plane，就实现了凸包的更新

void convex() {
    plane[m++] = {0, 1, 2};        // 初始化凸包，随便三个点存入，确定最开始的一个平面，这里取得是前三个点
```

```
48      plane[m++] = {2, 1, 0};        // 因为不知道第一个平面怎么样是逆时针，所以都存一遍，顺时针存的一会会被删掉
49      for (int i = 3; i < n; i++) { // 从第四个点开始循环每个点
50          int cnt = 0;
51          for (int j = 0; j < m; j++) {        // 循环每个平面
52              bool fg = plane[j].above(p[i]); // 判断这个点是否在该平面上方
53              if (!fg) tp[cnt++] = plane[j];  // 如果是下方的话，说明照不到，存进tp数组
54              for (int k = 0; k < 3; k++) {   // 循环该平面的三条边
55                  // ab边照不照得到情况赋值给g[a][b]
56                  g[plane[j].v[k]][plane[j].v[(k + 1) % 3]] = fg;
57              }
58          }
59          for (int j = 0; j < m; j++) { // 然后就循环每个平面的每条边
60              for (int k = 0; k < 3; k++) {
61                  int a = plane[j].v[k], b = plane[j].v[(k + 1) % 3];
62                  // 判断该边是否被照到了一次，即是否是交界线的边
63                  // 若是，加新平面abi，ab一定是逆时针的，i在后面
64                  if (g[a][b] && !g[b][a]) tp[cnt++] = {a, b, i};
65              }
66          }
67          m = cnt; // 将tp再赋值给plane
68          for (int j = 0; j < m; j++) plane[j] = tp[j];
69      }
70  }
71
72  int main() {
73      cin >> n;
74      for (int i = 0; i < n; i++) {
75          cin >> p[i].x >> p[i].y >> p[i].z; // 输入n个点
76          p[i].shake();                      // 微小扰动
77      }
78      convex();                   // 求三维凸包
79      ld ans = 0;                 // 求面积和
80      for (int i = 0; i < m; i++) // 循环最终凸包上的m个平面
81          ans += plane[i].area(); // 将平面的面积加和
82      cout << fixed << setprecision(3) << ans << '\n';
83
84      return 0;
85  }
86
```