计算几何

距离

1.曼哈顿距离

$$d(A,B)=|x_1-y_1|+|x_2-y_2|+///+|x_n-y_n|=\sum_{i=1}^{n}|x_i-y_i|$$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, x, y, minx = 0x7fffffff, maxx = 0, miny = 0x7fffffff, maxy = 0;
    scanf("%d", &n);
    for (int i = 1; i ≤ n; i++) {
        scanf("%d%d", &x, &y);
        minx = min(minx, x + y), maxx = max(maxx, x + y);
        miny = min(miny, x - y), maxy = max(maxy, x - y);
    }
    printf("%d\n", max(maxx - minx, maxy - miny));
    return 0;
}
```

2.切比雪夫距离

$$d(x,y)=\max||x_1-y_1|,|x_2-y_2|,\cdots,|x_n-y_n||=\max||x_i-y_i||(i\in[1,n])$$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, x, y, a, b, minx = 0x7fffffff, maxx = 0, miny = 0x7fffffff, maxy
= 0;
    scanf("%d", &n);
    for (int i = 1; i ≤ n; i++) {
        scanf("%d%d", &a, &b);
        x = a + b, y = a - b;
        minx = min(minx, x), maxx = max(maxx, x);
        miny = min(miny, y), maxy = max(maxy, y);
    }
    printf("%d\n", max(maxx - minx, maxy - miny));
    return 0;
}
```

pick定理

- 给定顶点均为整点的简单多边形，皮克定理说明了其面积A和内部格点数目i, 边上格点数目b的关系
- $$A = i + \frac{b}{2} - 1$$

  板子

```
#include <cmath>
#include <cstdio>
#include <iostream>
using namespace std;
const int MAXN = 110;

struct node {
    int x, y;
} p[MAXN];

int gcd(int x, int y) { return y == 0 ? x : gcd(y, x % y); }  // 求最大公约数

int area(int a, int b) { return p[a].x * p[b].y - p[a].y * p[b].x; }  // 求区域

int main() {
    int t, ncase = 1;
    scanf("%d", &t);
    while (t--) {
        int n, dx, dy, x, y, num = 0, sum = 0;
        scanf("%d", &n);
        p[0].x = 0, p[0].y = 0;
        for (int i = 1; i ≤ n; i++) {
            scanf("%d%d", &x, &y);
            p[i].x = x + p[i - 1].x, p[i].y = y + p[i - 1].y;
            dx = x, dy = y;
            if (x < 0) dx = -x;
            if (y < 0) dy = -y;
            num += gcd(dx, dy);
            sum += area(i - 1, i);
        }
        if (sum < 0) sum = -sum;
        printf("Scenario #%d:\n", ncase++);
        printf("%d %d %.1f\n\n", (sum - num + 2) >> 1, num, sum * 0.5);
    }
    return 0;
}
```

三角剖分

（我看不懂）

```
#include <algorithm>
#include <cmath>
#include <cstring>
#include <list>
#include <utility>
#include <vector>

const double EPS = 1e-8;
const int MAXV = 10000;

struct Point {
    double x, y;
    int id;

    Point(double a = 0, double b = 0, int c = -1) : x(a), y(b), id(c) {}

    bool operator<(const Point& a) const {
        return x < a.x || (fabs(x - a.x) < EPS && y < a.y);
    }

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

    double dist2(const Point& b) {
        return (x - b.x) * (x - b.x) + (y - b.y) * (y - b.y);
    }
};

struct Point3D {
    double x, y, z;

    Point3D(double a = 0, double b = 0, double c = 0) : x(a), y(b), z(c) {}

    Point3D(const Point& p) { x = p.x, y = p.y, z = p.x * p.x + p.y * p.y; }

    Point3D operator-(const Point3D& a) const {
        return Point3D(x - a.x, y - a.y, z - a.z);
    }

    double dot(const Point3D& a) { return x * a.x + y * a.y + z * a.z; }
};

struct Edge {
    int id;
```

```
    std::list<Edge>::iterator c;

    Edge(int id = 0) { this->id = id; }
};

int cmp(double v) { return fabs(v) > EPS ? (v > 0 ? 1 : -1) : 0; }

double cross(const Point& o, const Point& a, const Point& b) {
    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

Point3D cross(const Point3D& a, const Point3D& b) {
    return Point3D(a.y * b.z - a.z * b.y, -a.x * b.z + a.z * b.x,
        a.x * b.y - a.y * b.x);
}

int inCircle(const Point& a, Point b, Point c, const Point& p) {
    if (cross(a, b, c) < 0) std::swap(b, c);
    Point3D a3(a), b3(b), c3(c), p3(p);
    b3 = b3 - a3, c3 = c3 - a3, p3 = p3 - a3;
    Point3D f = cross(b3, c3);
    return cmp(p3.dot(f));  // check same direction, in: < 0, on: = 0, out:
> 0
}

int intersection(const Point& a, const Point& b, const Point& c,
    const Point& d) {  // seg(a, b) and seg(c, d)
    return cmp(cross(a, c, b)) * cmp(cross(a, b, d)) > 0 &&
        cmp(cross(c, a, d)) * cmp(cross(c, d, b)) > 0;
}

class Delaunay {
public:
    std::list<Edge> head[MAXV];  // graph
    Point p[MAXV];
    int n, rename[MAXV];

    void init(int n, Point p[]) {
        memcpy(this->p, p, sizeof(Point) * n);
        std::sort(this->p, this->p + n);
        for (int i = 0; i < n; i++) rename[p[i].id] = i;
        this->n = n;
        divide(0, n - 1);
    }

    void addEdge(int u, int v) {
        head[u].push_front(Edge(v));
        head[v].push_front(Edge(u));
        head[u].begin()->c = head[v].begin();
        head[v].begin()->c = head[u].begin();
```

```
    }

    void divide(int l, int r) {
        if (r - l ≤ 2) {  // #point ≤ 3
            for (int i = l; i ≤ r; i++)
                for (int j = i + 1; j ≤ r; j++) addEdge(i, j);
            return;
        }
        int mid = (l + r) / 2;
        divide(l, mid);
        divide(mid + 1, r);

        std::list<Edge>::iterator it;
        int nowl = l, nowr = r;

        for (int update = 1; update;) {
            // find left and right convex, lower common tangent
            update = 0;
            Point ptL = p[nowl], ptR = p[nowr];
            for (it = head[nowl].begin(); it ≠ head[nowl].end(); it++) {
                Point t = p[it→id];
                double v = cross(ptR, ptL, t);
                if (cmp(v) > 0 || (cmp(v) == 0 && ptR.dist2(t) <
ptR.dist2(ptL))) {
                    nowl = it→id, update = 1;
                    break;
                }
            }
            if (update) continue;
            for (it = head[nowr].begin(); it ≠ head[nowr].end(); it++) {
                Point t = p[it→id];
                double v = cross(ptL, ptR, t);
                if (cmp(v) < 0 || (cmp(v) == 0 && ptL.dist2(t) <
ptL.dist2(ptR))) {
                    nowr = it→id, update = 1;
                    break;
                }
            }
        }

        addEdge(nowl, nowr);  // add tangent

        for (int update = 1; true;) {
            update = 0;
            Point ptL = p[nowl], ptR = p[nowr];
            int ch = -1, side = 0;
            for (it = head[nowl].begin(); it ≠ head[nowl].end(); it++) {
                if (cmp(cross(ptL, ptR, p[it→id])) > 0 &&
                    (ch == -1 || inCircle(ptL, ptR, p[ch], p[it→id]) < 0))
{
```

```
                    ch = it->id, side = -1;
                }
            }
            for (it = head[nowr].begin(); it != head[nowr].end(); it++) {
                if (cmp(cross(ptR, p[it->id], ptL)) > 0 &&
                    (ch == -1 || inCircle(ptL, ptR, p[ch], p[it->id]) < 0))
{
                    ch = it->id, side = 1;
                }
            }
            if (ch == -1) break;  // upper common tangent
            if (side == -1) {
                for (it = head[nowl].begin(); it != head[nowl].end();) {
                    if (intersection(ptL, p[it->id], ptR, p[ch])) {
                        head[it->id].erase(it->c);
                        head[nowl].erase(it++);
                    }
                    else {
                        it++;
                    }
                }
                nowl = ch;
                addEdge(nowl, nowr);
            }
            else {
                for (it = head[nowr].begin(); it != head[nowr].end();) {
                    if (intersection(ptR, p[it->id], ptL, p[ch])) {
                        head[it->id].erase(it->c);
                        head[nowr].erase(it++);
                    }
                    else {
                        it++;
                    }
                }
                nowr = ch;
                addEdge(nowl, nowr);
            }
        }
    }

    std::vector<std::pair<int, int> > getEdge() {
        std::vector<std::pair<int, int> > ret;
        ret.reserve(n);
        std::list<Edge>::iterator it;
        for (int i = 0; i < n; i++) {
            for (it = head[i].begin(); it != head[i].end(); it++) {
                if (it->id < i) continue;
                ret.push_back(std::make_pair(p[i].id, p[it->id].id));
            }
        }
```

```
        return ret;
    }
};
```

凸包

Andrew算法求凸包

```
// stk[] 是整型，存的是下标
// p[] 存储向量或点
tp = 0;                          // 初始化栈
std::sort(p + 1, p + 1 + n);  // 对点进行排序
stk[++tp] = 1;
// 栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新
for (int i = 2; i ≤ n; ++i) {
    while (tp ≥ 2  // 下一行 * 操作符被重载为叉积
        && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) ≤ 0)
        used[stk[tp--]] = 0;
    used[i] = 1;  // used 表示在凸壳上
    stk[++tp] = i;
}
int tmp = tp;  // tmp 表示下凸壳大小
for (int i = n - 1; i > 0; --i)
if (!used[i]) {
    // ↓求上凸壳时不影响下凸壳
    while (tp > tmp && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]])
≤ 0)
        used[stk[tp--]] = 0;
    used[i] = 1;
    stk[++tp] = i;
}
for (int i = 1; i ≤ tp; ++i)  // 复制到新数组中去
h[i] = p[stk[i]];
int ans = tp - 1;
```

Graham扫描法

```
struct Point {
        double x, y, ang;
        Point operator-(const Point& p) const {
                return {x - p.x, y - p.y, 0};
        }
} p[MAX];
double dis(Point p1, Point p2) {
        return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y -
p2.y));
}
```

```
bool cmp(Point p1, Point p2) {
        if (p1.ang == p2.ang) {
                return dis(p1, p[1]) < dis(p2, p[1]);
        }
        return p1.ang < p2.ang;
}
double cross(Point p1, Point p2) {
        return p1.x * p2.y - p1.y * p2.x;
}
int main() {
        for (int i = 2; i ≤ n; ++i) {
                if (p[i].y < p[1].y || (p[i].y == p[1].y && p[i].x <
p[1].x)) {
                        std::swap(p[1], p[i]);
                }
        }
        for (int i = 2; i ≤ n; ++i) {
                p[i].ang = atan2(p[i].y - p[1].y, p[i].x - p[1].x);
        }
        std::sort(p + 2, p + n + 1, cmp);
        sta[++top] = 1;
        for (int i = 2; i ≤ n; ++i) {
                while (top ≥ 2 && cross(p[sta[top]] - p[sta[top - 1]], p[i]
- p[sta[top]]) < 0) {
                        top--;
                }
                sta[++top] = i;
        }
        return 0;
}
```

三维凸包

```
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;
const int N = 2010;
const double eps = 1e-9;
int n, cnt, vis[N][N];
double ans;

double Rand() { return rand() / (double)RAND_MAX; }

double reps() { return (Rand() - 0.5) * eps; }

struct Node {
```

```
    double x, y, z;

    void shake() {
        x += reps();
        y += reps();
        z += reps();
    }

    double len() { return sqrt(x * x + y * y + z * z); }

    Node operator-(Node A) { return { x - A.x, y - A.y, z - A.z }; }

    Node operator*(Node A) {
        return { y * A.z - z * A.y, z * A.x - x * A.z, x * A.y - y * A.x };
    }

    double operator&(Node A) { return x * A.x + y * A.y + z * A.z; }
} A[N];

struct Face {
    int v[3];

    Node Normal() { return (A[v[1]] - A[v[0]]) * (A[v[2]] - A[v[0]]); }

    double area() { return Normal().len() / 2.0; }
} f[N], C[N];

int see(Face a, Node b) { return ((b - A[a.v[0]]) & a.Normal()) > 0; }

void Convex_3D() {
    f[++cnt] = { 1, 2, 3 };
    f[++cnt] = { 3, 2, 1 };

    for (int i = 4, cc = 0; i ≤ n; i++) {
        for (int j = 1, v; j ≤ cnt; j++) {
            if (!(v = see(f[j], A[i]))) C[++cc] = f[j];

            for (int k = 0; k < 3; k++) vis[f[j].v[k]][f[j].v[(k + 1) % 3]]
= v;
        }

        for (int j = 1; j ≤ cnt; j++)
            for (int k = 0; k < 3; k++) {
                int x = f[j].v[k], y = f[j].v[(k + 1) % 3];

                if (vis[x][y] && !vis[y][x]) C[++cc] = { x, y, i };
            }

        for (int j = 1; j ≤ cc; j++) f[j] = C[j];
```

```
            cnt = cc;
            cc = 0;
        }
    }

    int main() {
        cin >> n;

        for (int i = 1; i ≤ n; i++) cin >> A[i].x >> A[i].y >> A[i].z,
    A[i].shake();

        Convex_3D();

        for (int i = 1; i ≤ cnt; i++) ans += f[i].area();

        printf("%.3f\n", ans);
        return 0;
    }
```

扫描线

Atlantis问题

在二维坐标系上，给出多个矩形的左下以及右上坐标，求出所有矩形构成的图形的面积

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#define maxn 300
using namespace std;

int lazy[maxn << 3];  // 标记了这条线段出现的次数
double s[maxn << 3];

struct node1 {
    double l, r;
    double sum;
} cl[maxn << 3];  // 线段树

struct node2 {
    double x, y1, y2;
    int flag;
} p[maxn << 3];  // 坐标

// 定义sort比较
bool cmp(node2 a, node2 b) { return a.x < b.x; }

// 上传
```

```
void pushup(int rt) {
    if (lazy[rt] > 0)
        cl[rt].sum = cl[rt].r - cl[rt].l;
    else
        cl[rt].sum = cl[rt * 2].sum + cl[rt * 2 + 1].sum;
}

// 建树
void build(int rt, int l, int r) {
    if (r - l > 1) {
        cl[rt].l = s[l];
        cl[rt].r = s[r];
        build(rt * 2, l, (l + r) / 2);
        build(rt * 2 + 1, (l + r) / 2, r);
        pushup(rt);
    }
    else {
        cl[rt].l = s[l];
        cl[rt].r = s[r];
        cl[rt].sum = 0;
    }
    return;
}

// 更新
void update(int rt, double y1, double y2, int flag) {
    if (cl[rt].l == y1 && cl[rt].r == y2) {
        lazy[rt] += flag;
        pushup(rt);
        return;
    }
    else {
        if (cl[rt * 2].r > y1) update(rt * 2, y1, min(cl[rt * 2].r, y2),
flag);
        if (cl[rt * 2 + 1].l < y2)
            update(rt * 2 + 1, max(cl[rt * 2 + 1].l, y1), y2, flag);
        pushup(rt);
    }
}

int main() {
    int temp = 1, n;
    double x1, y1, x2, y2, ans;
    while (scanf("%d", &n) && n) {
        ans = 0;
        for (int i = 0; i < n; i++) {
            scanf("%lf %lf %lf %lf", &x1, &y1, &x2, &y2);
            p[i].x = x1;
            p[i].y1 = y1;
            p[i].y2 = y2;
```

```
            p[i].flag = 1;
            p[i + n].x = x2;
            p[i + n].y1 = y1;
            p[i + n].y2 = y2;
            p[i + n].flag = -1;
            s[i + 1] = y1;
            s[i + n + 1] = y2;
        }
        sort(s + 1, s + (2 * n + 1));   // 离散化
        sort(p, p + 2 * n, cmp);   // 把矩形的边的横坐标从小到大排序
        build(1, 1, 2 * n);        // 建树
        memset(lazy, 0, sizeof(lazy));
        update(1, p[0].y1, p[0].y2, p[0].flag);
        for (int i = 1; i < 2 * n; i++) {
            ans += (p[i].x - p[i - 1].x) * cl[1].sum;
            update(1, p[i].y1, p[i].y2, p[i].flag);
        }
        printf("Test case #%d\nTotal explored area: %.2lf\n\n", temp++,
ans);
    }
    return 0;
}
```

B维正交范围

B维正交范围指在一个B维直角坐标系下，第i维坐标在一个整数范围[l,r]内部的点集
一般来说，一维正交范围简称区间，二维正交范围简称曲形，三维正交范围简称立方体

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

int n, m;
int x[500010], y[500010], ans[500010];
int ax[1500010], ay[1500010], tx, ty;  // 离散化

struct query {
    int a, b, c, d;
} q[500010];  // 保存查询操作方便离散化

struct ope {
    int type, x, y, id;

    inline ope(int type = 0, int x = 0, int y = 0, int id = 0) {
        this->type = type, this->x = x, this->y = y, this->id = id;
    }

    inline bool operator<(const ope& rhs) const {
        if (x == rhs.x) return type < rhs.type;
        return x < rhs.x;
```

```
    }
};

ope op[2500010];
int tot;  // 操作总数

int sum[1500010];  // 树状数组

int lowbit(int x) { return x & (-x); }

void add(int x, int k) {
    while (x ≤ 1500000) {
        sum[x] = sum[x] + k;
        x = x + lowbit(x);
    }
}

int getsum(int x) {
    int ret = 0;
    while (x > 0) {
        ret = ret + sum[x];
        x = x - lowbit(x);
    }
    return ret;
}

int main() {
    scanf("%d%d", &n, &m), tx = n, ty = n;
    for (int i = 1; i ≤ n; i++)
        scanf("%d%d", &x[i], &y[i]), ax[i] = x[i], ay[i] = y[i];
    for (int i = 1, l, r; i ≤ m; i++) {
        scanf("%d%d%d%d", &q[i].a, &q[i].b, &q[i].c, &q[i].d);
        ax[++tx] = q[i].a, ay[++ty] = q[i].b, ax[++tx] = q[i].c, ay[++ty] =
q[i].d;
    }
    std::sort(ax + 1, ax + tx + 1), std::sort(ay + 1, ay + ty + 1);
    tx = std::unique(ax + 1, ax + tx + 1) - ax - 1;
    ty = std::unique(ay + 1, ay + ty + 1) - ay - 1;
    for (int i = 1; i ≤ n; i++) {
        x[i] = std::lower_bound(ax + 1, ax + tx + 1, x[i]) - ax;
        y[i] = std::lower_bound(ay + 1, ay + ty + 1, y[i]) - ay;
        op[++tot] = ope(0, x[i], y[i], i);  // 加点操作
    }
    for (int i = 1; i ≤ m; i++) {
        q[i].a = std::lower_bound(ax + 1, ax + tx + 1, q[i].a) - ax;
        q[i].b = std::lower_bound(ay + 1, ay + ty + 1, q[i].b) - ay;
        q[i].c = std::lower_bound(ax + 1, ax + tx + 1, q[i].c) - ax;
        q[i].d = std::lower_bound(ay + 1, ay + ty + 1, q[i].d) - ay;
        op[++tot] = ope(1, q[i].c, q[i].d, i);  // 将查询差分
        op[++tot] = ope(1, q[i].a - 1, q[i].b - 1, i);
```

```
        op[++tot] = ope(2, q[i].a - 1, q[i].d, i);
        op[++tot] = ope(2, q[i].c, q[i].b - 1, i);
    }
    std::sort(op + 1, op + tot + 1);  // 将操作按横坐标排序，且优先执行加点操作
    for (int i = 1; i ≤ tot; i++) {
        if (op[i].type == 0)
            add(op[i].y, 1);
        else if (op[i].type == 1)
            ans[op[i].id] += getsum(op[i].y);
        else
            ans[op[i].id] -= getsum(op[i].y);
    }
    for (int i = 1; i ≤ m; i++) printf("%d\n", ans[i]);
    return 0;
}
```

旋转卡壳

求凸包直径

```
int sta[N], top;  // 将凸包上的节点编号存在栈里，第一个和最后一个节点编号相同
bool is[N];

ll pf(ll x) { return x * x; }

ll dis(int p, int q) { return pf(a[p].x - a[q].x) + pf(a[p].y - a[q].y); }

ll sqr(int p, int q, int y) { return abs((a[q] - a[p]) * (a[y] - a[q])); }

ll mx;

void get_longest() {  // 求凸包直径
    int j = 3;
    if (top < 4) {
        mx = dis(sta[1], sta[2]);
        return;
    }
    for (int i = 1; i ≤ top; ++i) {
        while (sqr(sta[i], sta[i + 1], sta[j]) ≤
            sqr(sta[i], sta[i + 1], sta[j % top + 1]))
            j = j % top + 1;
        mx = max(mx, max(dis(sta[i + 1], sta[j]), dis(sta[i], sta[j])));
    }
}
```

求最小矩形覆盖

```
void get_biggest() {
    int j = 3, l = 2, r = 2;
    double t1, t2, t3, ans = 2e10;
    for (int i = 1; i ≤ top; ++i) {
        while (sqr(sta[i], sta[i + 1], sta[j]) ≤
            sqr(sta[i], sta[i + 1], sta[j % top + 1]))
            j = j % top + 1;
        while (dot(sta[i + 1], sta[r % top + 1], sta[i]) ≥
            dot(sta[i + 1], sta[r], sta[i]))
            r = r % top + 1;
        if (i == 1) l = r;
        while (dot(sta[i + 1], sta[l % top + 1], sta[i]) ≤
            dot(sta[i + 1], sta[l], sta[i]))
            l = l % top + 1;
        t1 = sqr(sta[i], sta[i + 1], sta[j]);
        t2 = dot(sta[i + 1], sta[r], sta[i]) + dot(sta[i + 1], sta[l],
sta[i]);
        t3 = dot(sta[i + 1], sta[i + 1], sta[i]);
        ans = min(ans, t1 * t2 / t3);
    }
}
```