工具函数

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

const double EPS = 1e-8;

int sign(double val) {
    if (std::abs(val) < EPS) return 0;
    if (val > 0) return 1;
    return -1;
}

int dcmp(double val1, double val2) {
    return sign(val1 - val2);
}
```

点与线

```
struct Point {
    double x, y;

    Point(double a = 0, double b = 0) : x(a), y(b) {}

    friend Point operator+(Point a, Point b) {
        return Point(a.x + b.x, a.y + b.y);
    }

    friend Point operator-(Point a, Point b) {
        return Point(a.x - b.x, a.y - b.y);
    }

    friend Point operator/(Point a, double b) {
        return Point(a.x / b, a.y / b);
    }

    friend Point operator*(Point a, double b) {
        return Point(a.x * b, a.y * b);
    }

    friend Point operator*(double a, Point b) {
        return Point(b.x * a, b.y * a);
    }

    friend double operator*(Point a, Point b) {
        return a.x * b.x + a.y * b.y;
    }

    friend bool operator==(Point a, Point b) {
        return (!sign(a.x - b.x) && !sign(a.y - b.y));
    }

    friend bool operator!=(Point a, Point b) {
        return !(a == b);
    }

    // 用%代表叉乘,因为 ^ 优先级太低了
    friend double operator%(Point a, Point b) {
        return a.x * b.y - a.y * b.x;
```

```
    }

    double ddis() { return x * x + y * y; }

    double dis() { return sqrt(x * x + y * y); }
} INFP(1e20, 1e20);

// 点到点的距离
double dis(Point a, Point b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

// 点到点的距离（不开方）
double ddis(Point a, Point b) {
    return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
}

struct Seg {
    Point p1, p2;

    Seg() {}

    Seg(Point a, Point b) : p1(a), p2(b) {}

    friend double operator*(Seg a, Seg b) {
        return (a.p2 - a.p1) * (b.p2 - b.p1);
    }

    friend double operator%(Seg a, Seg b) {
        return (a.p2 - a.p1) % (b.p2 - b.p1);
    }

    friend double operator%(Seg a, Point b) {
        return (a.p2 - a.p1) % (b - a.p1);
    }

    friend double operator%(Point a, Seg b) {
        return (a - b.p1) % (b.p2 - b.p1);
    }

    friend bool operator==(Seg a, Seg b) {
        return (a.p1 == b.p1 && a.p2 == b.p2) || (a.p1 == b.p2 && a.p2 == b.p1);
    }
```

```
    void swap() {
        std::swap(p1, p2);
    }

    double len() { return dis(p1, p2); }
};

// 点到直线距离
double calPSDis1(Seg a, Point b) {
    return std::abs((a.p2 - a.p1) % (b - a.p1)) / a.len();
}

// 点到线段距离
double calPSDis2(Seg a, Point b) {
    int res1 = sign((b - a.p1) * (a.p2 - a.p1));
    int res2 = sign((b - a.p2) * (a.p1 - a.p2));
    if (res1 >= 0 && res2 >= 0) {
        return calPSDis1(a, b);
    }
    return std::min(dis(a.p1, b), dis(a.p2, b));
}

// 求点到线段/直线的垂足
Point calPSFeet(Seg a, Point b) {
    auto vec = a.p2 - a.p1;
    return a.p1 + vec * ((b - a.p1) * vec) / (vec.x * vec.x + vec.y * vec.y);
}

// 计算线段及其延长线/直线的交点
Point calSegCross(Seg a, Seg b) {
    auto res1 = (a.p2 - a.p1) % (b.p1 - a.p1);
    auto res2 = (a.p2 - a.p1) % (b.p2 - a.p1);
    auto x = (res2 * b.p1.x - res1 * b.p2.x) / (res2 - res1);
    auto y = (res2 * b.p1.y - res1 * b.p2.y) / (res2 - res1);
    return Point(x, y);
}

// 判断点是否在线段上（包括端点）
bool isPAtS(Seg a, Point b) {
    Point p1 = a.p1 - b, p2 = a.p2 - b;
    if (sign(p1 % p2) == 0 && sign(p1 * p2) <= 0) return true;
    return false;
```

```
}

// 判断线段是否相交
bool isSegCross(Seg a, Seg b) {
    double c1 = (a.p2 - a.p1) % (b.p1 - a.p1), c2 = (a.p2 - a.p1) % (b.p2 - a.p1);
    double c3 = (b.p2 - b.p1) % (a.p1 - b.p1), c4 = (b.p2 - b.p1) % (a.p2 - b.p1);

    // 允许线段在端点处相交则添加
    if (!sign(c1) || !sign(c2) || !sign(c3) || !sign(c4)) {
        bool f1 = isPAtS(a, b.p1);
        bool f2 = isPAtS(a, b.p2);
        bool f3 = isPAtS(b, a.p1);
        bool f4 = isPAtS(b, a.p2);
        bool f = (f1 | f2 | f3 | f4);
        return f;
    }
    return (sign(c1) * sign(c2) < 0 && sign(c3) * sign(c4) < 0);
}

// 计算平面最近点对欧式距离
double calPointMinDis(int l, int r, Point point[], int t[]) {
    /*
    开始递归前先按照 x 排序
    std::sort(point, point + n, [&](const Point &p1, const Point &p2) {
        return p1.x < p2.x;
    });
    */
    if (r - l == 0)
        return 1e15;
    if (r - l == 1)  // 如果递归完后直接输出距离
        return dis(point[l], point[r]);
    int mid = (l + r) >> 1;
    double ans = std::min(calPointMinDis(l, mid, point, t),
                          calPointMinDis(mid + 1, r, point, t));
    int cnt = 0;
    for (int i = l; i <= r; i++)
        // 还有一种情况是距离最小的两点刚好分在 mid 两端 ans 距离内的点
        if (point[i].x >= point[mid].x - ans && point[i].x <= point[mid].x + ans)
            t[++cnt] = i;
    std::sort(t + 1, t + cnt + 1, [&](int i, int j) {
        return point[i].y < point[j].y;
    }); // 以 y 坐标大小排序
    for (int i = 1; i <= cnt; i++)
```

```
        for (int j = i + 1; j <= cnt; j++) {
            if (point[t[j]].y >= point[t[i]].y + ans) break;
            // 两个点的垂直距离超过 ans 就不必计算了，显然不可能会成为新的 ans
            ans = std::min(ans, dis(point[t[i]], point[t[j]]));
        }
    return ans;
}
```

圆

```
struct Cir {
    Point o;
    double r;

    Cir(Point oo, double rr) : o(oo), r(rr) {}

    Cir(double x = 0, double y = 0, double rr = 0) { o.x = x, o.y = y, r = rr; }

    Point getPoint(double theta) { // 根据极角返回圆上一点的坐标
        double co = cos(theta) * r;
        double si = sin(theta) * r;
        if (sign(co) == 0) co = 0.0;
        if (sign(si) == 0) si = 0.0;
        return Point(o.x + co, o.y + si);
    }

    Point getPoint(Point p) { // 根据一点返回圆上一点的坐标
        auto vec = p - o;
        return o + vec * r / vec.dis();
    }
};

// 判断圆相交情况
int isCirCross(Cir a, Cir b) {
    // 返回值恰好为两个圆的公共切线数量
    int res = dcmp(dis(a.o, b.o), a.r + b.r);
    if (res > 0) return 4; // 两个圆不相交
    if (res == 0) return 3; // 两个圆外接
    res = dcmp(dis(a.o, b.o), std::abs(a.r - b.r));
    if (res > 0) return 2; // 两个圆相交
    if (res == 0) return 1; // 有一个圆内切另外一个圆
    return 0; // 一个圆在另一个圆内部
}

// 计算两个圆的交点
std::array<Point, 2> calCirCross(Cir c1, Cir c2) {
    Point vec12 = c2.o - c1.o; // 两圆圆心的向量
    double d = vec12.dis();    // 圆心距
    double a = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2.0 * c1.r * d));
    // vec12 与（c1 与一个交点）的夹角
```

```
    double t = atan2(vec12.y, vec12.x); // vec12 与x轴的夹角
    return {c1.getPoint(t + a), c1.getPoint(t - a)};
}

// 三角形内切圆
Cir calTrangleInnerCir(Point p1, Point p2, Point p3) {
    Cir c;
    auto d1 = dis(p1, p2), d2 = dis(p1, p3), d3 = dis(p2, p3);
    c.r = std::abs((p2.x - p1.x) * (p3.y - p1.y) -
            (p2.y - p1.y) * (p3.x - p1.x)) / (d1 + d2 + d3);
    c.o.x = (d3 * p1.x + d1 * p3.x + d2 * p2.x) / (d1 + d2 + d3);
    c.o.y = (d3 * p1.y + d1 * p3.y + d2 * p2.y) / (d1 + d2 + d3);
    return c;
}

// 三角形外接圆
Cir calTrangleOuterCir(Point p1, Point p2, Point p3) {
    Cir c;
    c.r = dis(p1, p2) * dis(p2, p3) * dis(p1, p3) / std::abs((p2 - p1) % (p3 - p1)) / 2.0;
    auto a11 = (p2.x - p1.x);
    auto a12 = (p2.y - p1.y);
    auto b1 = (p1.x * (p2.x - p1.x) + p1.y * (p2.y - p1.y) + 0.5 * ddis(p1, p2));
    auto a21 = (p3.x - p1.x);
    auto a22 = (p3.y - p1.y);
    auto b2 = (p1.x * (p3.x - p1.x) + p1.y * (p3.y - p1.y) + 0.5 * ddis(p1, p3));
    auto down = (a11 * a22 - a21 * a12);
    c.o.x = (b1 * a22 - b2 * a12) / down;
    c.o.y = (a11 * b2 - b1 * a21) / down;
    return c;
}

// 直线和圆交点个数
int isSegCrossCir(Seg s, Cir c) {
    int res = dcmp(calPSDis1(s, c.o), c.r);
    if (res == 0) return 1; // 外切
    if (res == 1) return 0; // 不相交
    return 2;
}

// 计算直线与圆交点
std::array<Point, 2> calSegCrossCir(Cir c, Seg s) {
    auto feet = calPSFeet(s, c.o);
    int sta = isSegCrossCir(s, c);
```

```
    if (sta == 0) return {INFP, INFP};
    if (sta == 1) return {feet, feet};
    double base = sqrt(c.r * c.r - (feet - c.o).ddis());
    auto vec = s.p2 - s.p1;
    Point e = vec / vec.dis();
    return {feet + e * base, feet - e * base};
}

std::array<Point, 2> calPointCirTangent(Cir C, Point p) {
    auto d = dis(p, C.o);
    int aa = dcmp(d, C.r);
    if (aa < 0) return {INFP, INFP};  // 点在圆内
    else if (aa == 0) return {p, p}; // 点在圆上，该点就是切点

    // 点在圆外，有两个切点
    double base = atan2(p.y - C.o.y, p.x - C.o.x);
    double ang = acos(C.r / d);
    return {C.getPoint(base - ang), C.getPoint(base + ang)};
}

// 计算 c1 与 c2 的所有切线中 c1 的所有切点
std::vector<Point> calCirTangents(Cir c1, Cir c2) {
    int CrossRes = isCirCross(c1, c2);
    std::vector<Point> res;
    auto d = dis(c1.o, c2.o);
    if (CrossRes == 0) return res;
    double base = atan2(c2.o.y - c1.o.y, c2.o.x - c1.o.x);
    // AB 向量的极角，c1 为大圆
    if (CrossRes == 1) { // 内切
        res.push_back(c1.getPoint(base));
        return res;
    }
    double ang1 = acos((c1.r - c2.r) / d); // 外公切线的图
    res.push_back(c1.getPoint(base + ang1));
    res.push_back(c1.getPoint(base - ang1));
    if (CrossRes == 3)
        res.push_back(c1.getPoint(base));
        // 一条内公切线，此时c2上的点为 c2.getPoint(base + pi)
    else if (CrossRes == 4) { // 两条内公切线，对应内公切线的图
        double ang2 = acos((c1.r + c2.r) / d);
        res.push_back(c1.getPoint(base + ang2));
        res.push_back(c1.getPoint(base - ang2));
    }
```

```
    return res;
}

// 最小圆覆盖
Cir calMinCoverCir(Point p[], int n) {
    std::random_device rd;
    std::mt19937 g(rd());
    std::shuffle(p, p + 1 + n, g);
    Cir c(p[0], 0);
    for (int i = 1; i < n; i++) {
        if (dcmp(dis(c.o, p[i]), c.r) > 0) {
            c.r = 0;
            c.o = p[i];
            for (int j = 0; j < i; j++) {
                if (dcmp(dis(c.o, p[j]), c.r) > 0) {
                    c.r = dis(p[j], p[i]) / 2;
                    c.o = {(p[j].x + p[i].x) / 2, (p[j].y + p[i].y) / 2};
                    for (int k = 0; k < j; k++) {
                        if (dcmp(dis(c.o, p[k]), c.r) > 0) {
                            c = calTrangleOuterCir(p[i], p[j], p[k]);
                        }
                    }
                }
            }
        }
    }
    return c;
}
```

多边形

```
// 多边形的重心(凹凸都可以)
Point calPolygonCentre(Point point[], int n) {
    double sum = 0.0, sumx = 0, sumy = 0;
    Point p1 = point[0], p2 = point[1], p3;
    for (int i = 2; i <= n - 1; i++) {
        p3 = point[i];
        double area = (p2 - p1) % (p3 - p2) / 2.0;
        sum += area;
        sumx += (p1.x + p2.x + p3.x) * area;
        sumy += (p1.y + p2.y + p3.y) * area;
        p2 = p3;
    }
    return Point(sumx / (3.0 * sum), sumy / (3.0 * sum));
}

// 判断点是否在多边形内部(凹凸都可以)
int isPointInPolygon(int n, Point point[], Point P) {
    int cnt = 0;
    Point P1, P2;
    for (int i = 0; i < n; i++) {
        P1 = point[i];
        P2 = point[(i + 1) % n];
        if (isPAtS(Seg(P1, P2), P)) return 1;  // 如果点在边上返回 1
        if ((sign(P1.y - P.y) > 0 != sign(P2.y - P.y) > 0) &&
            sign(P.x - (P.y - P1.y) * (P1.x - P2.x) / (P1.y - P2.y) - P1.x) < 0)
            cnt++;
    }
    if (cnt & 1) return 2; // 在内部返回 2
    else return 0; // 在外部返回 0
}

// 多边形面积(凹凸都可以)
double calPolygonArea(Point point[], int n) {
    double area = 0;
    for (int i = 0; i < n; i++)
        area += point[i] % point[(i + 1) % n];
    return std::abs(area) / 2.0;
}

// 计算凸多边形被直线 s 切割的面积
```

```
double calPolygonAreaCut(Point point[], int n, Seg s) {
    Point dir = s.p2 - s.p1, tmp[110];
    // 默认点的顺序为为逆时针
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        int res1 = sign((point[i] - s.p1) % dir);
        int res2 = sign((point[(i + 1) % n] - s.p1) % dir);
        if (res1 <= 0) tmp[cnt++] = point[i];
        // 求左侧面积，如果求右侧改成 >= 0
        if (res1 * res2 == -1) {
            tmp[cnt++] = calSegCross(s, Seg(point[i], point[(i + 1) % n]));
        }
    }
    return calPolygonArea(tmp, cnt);
}

//  求 n 个点之间最大的曼哈顿距离
//  曼哈顿距离:abs(x1 - x2) + abs(y1 - y2)
//  比雪夫距离:max(abs(x1 - x2), abs(y1 - y2))
//  曼哈顿坐标系是通过切比雪夫坐标系旋转 45度 后，再缩小到原来的一半得到的。
//  曼哈顿 -> 切比雪夫 (x, y) -> (x + y, x - y)
//  切比雪夫 -> 曼哈顿 (x, y) -> ((x + y) / 2, (x - y) / 2)
void solve() {
    int minx = 1e10, miny = 1e10, maxx = -1e10, maxy = -1e10;
    int n;
    std::cin >> n;
    for (int i = 1; i <= n; i++) {
        int a, b;
        std::cin >> a >> b;
        int x = a + b, y = a - b;
        maxx = std::max(x, maxx), maxy = std::max(y, maxy);
        minx = std::min(x, minx), miny = std::min(y, miny);
    }
    std::cout << std::max(maxx - minx, maxy - miny);
}

// 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 + b / 2 - 1（没有取整,可能是小数）
// 取格点的组成图形的面积为一单位.在平行四边形格点,皮克定理依然成立。
// 套用于任意三角形格点，皮克定理则是 A = 2 * i + b - 2
std::array<int, 2> pick(int n, Point point[]) {
```

```
    int sum = 0, num = 0;
    for (int i = 0; i < n; i++) {
        int x = std::abs(point[(i + 1) % n].x - point[i].x) + 0.5;
        int y = std::abs(point[(i + 1) % n].y - point[i].y) + 0.5;
        num += std::__gcd(abs(x), abs(y));
        sum += point[(i + 1) % n] % point[i];
    }
    sum = abs(sum);// sum 为两倍的面积
    return {(sum - num) / 2 + 1, num};
}

// andrew 找凸包
int andrew(int n, Point point[], Point res[]) {
    // n >= 2
    // point 和 res 下标从 0 开始,方便取模
    std::sort(point, point + n, [&](const Point &a, const Point &b) {
        if (std::abs(a.x - b.x) < EPS) return a.y < b.y;
        return a.x < b.x;
    });
    int m = 0;
    for (int i = 0; i < n; i++) {
        while (m > 1 && (res[m - 1] - res[m - 2]) % (point[i] - res[m - 2]) <= 0) --m;
        // 如果想保留凸包边上的点,可以把 <= 换成 <
        // 凸包为逆时针方向
        res[m++] = point[i];
    }
    int k = m;
    for (int i = n - 2; i >= 0; i--) {
        while (m > k && (res[m - 1] - res[m - 2]) % (point[i] - res[m - 2]) <= 0) --m;
        res[m++] = point[i];
    }
    --m;
    // point[0] 被重复记录了,删去
    return m;
}

// 点c到线段ab的距离
double segdis(Point a, Point b, Point c) {
    return std::abs((b - a) % (c - a)) / dis(a, b);
}

// 旋转卡壳求凸包直径
double rotate(int n, Point res[]) {
```

```
    // 虽然 n >= 2 时即认为凸包存在，但是旋转卡壳的时候要求 n >= 3
    if (n == 2) return dis(res[1], res[0]);
    int cur = 0;
    double ans = 0;
    for (int i = 0; i < n; i++) {
        while (segdis(res[i], res[(i + 1) % n], res[cur]) <=
               segdis(res[i], res[(i + 1) % n], res[(cur + 1) % n]))
            cur = (cur + 1) % n;
        ans = std::max(ans, dis(res[i], res[cur]));
        ans = std::max(ans, dis(res[(i + 1) % n], res[cur]));
    }
    return ans;
}

// 判断凸包
bool isConvex(int n, Point point[]) {
    // 默认为顺时针
    // 如果输入点的方式为逆时针输入,则将 < 改成 >
    // 认为凸包边上可以有 点 则加上 =
    for (int i = 0; i < n; i++) {
        if ((point[i] - point[(i + 1) % n]) % (point[(i + 2) % n] - point[(i + 1) % n]) > 0)
            return false;
    }
    return true;
}
```