计算几何

常数定义相关

- 一般定义精度$eps = 10^{-8}$，根据题意可以适当改大或者改小
- 在精度要求较高的题目需要使用$long\ double$

```
//long double 的输入输出
scanf("%Lf" , &a);
printf("%.10Lf" , a);
//常用函数:fabsl(a),cosl(a).....
//即在末尾加上了字母l
//常数定义
const double eps = 1e-8;
const double PI = acos(-1.0);

int sgn(double x)//符号函数，eps使用最多的地方
{
    if (fabs(x) < eps)
        return 0;
    if (x < 0)
        return -1;
    else
        return 1;
}
```

点类及其相关操作

点类

- point类需要包含的基本功能
  - 向量的加减、向量的叉积点积运算
  - 为sort重载的<运算符
  - eps意义下2向量的=判断
  - 向量旋转Rotate
- 可以加入$ang = atan2(y, x)$点与x正向的夹角

```
struct Point
{
    double x, y;
    Point() {}
    Point(double _x, double _y) : x(_x), y(_y) {}
    Point operator-(const Point &b) const { return Point(x - b.x, y - b.y); }
    Point operator+(const Point &b) const { return Point(x + b.x, y + b.y); }

    double operator^(const Point &b) const { return x * b.y - y * b.x; } //叉积
    double operator*(const Point &b) const { return x * b.x + y * b.y; } //点积

    bool operator<(const Point &b) const { return x < b.x || (x == b.x && y < b.y); }
    bool operator==(const Point &b) const { return sgn(x - b.x) == 0 && sgn(y - b.y) == 0; }

    Point Rotate(double B, Point P) //绕着点P，逆时针旋转角度B(弧度)
    {
        Point tmp;
        tmp.x = (x - P.x) * cos(B) - (y - P.y) * sin(B) + P.x;
        tmp.y = (x - P.x) * sin(B) + (y - P.y) * cos(B) + P.y;
        return tmp;
    }
};
```

点间距离及向量的长度

```
double dist(Point a, Point b) { return sqrt((a - b) * (a - b)); } //两点间距离
double len(Point a){return sqrt(a.x * a.x + a.y * a.y);}//向量的长度
```

直线类及其相关操作

直线类

```
struct Line
{
    Point s, e;
    Line() {}
    Line(Point _s, Point _e) : s(_s), e(_e) {}

    //两直线相交求交点
    //第一个值为0表示直线重合，为1表示平行,为2是相交
    //只有第一个值为2时，交点才有意义

    pair<int, Point> operator&(const Line &b) const
    {
        Point res = s;
        if (sgn((s - e) ^ (b.s - b.e)) == 0)
        {
            if (sgn((s - b.e) ^ (b.s - b.e)) == 0)
                return make_pair(0, res); //重合
            else
                return make_pair(1, res); //平行
        }
        double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e));
        res.x += (e.x - s.x) * t;
        res.y += (e.y - s.y) * t;
        return make_pair(2, res);
    }
};
```

相交关系判断

• 判断线段是否相交

- 返回1表示相交,0表示不相交

```
bool inter(Line l1, Line l2)
{
    return max(l1.s.x, l1.e.x) >= min(l2.s.x, l2.e.x) &&
           max(l2.s.x, l2.e.x) >= min(l1.s.x, l1.e.x) &&
           max(l1.s.y, l1.e.y) >= min(l2.s.y, l2.e.y) &&
           max(l2.s.y, l2.e.y) >= min(l1.s.y, l1.e.y) &&
           sgn((l2.s - l1.e) ^ (l1.s - l1.e)) * sgn((l2.e - l1.e) ^ (l1.s - l1.e)) <= 0 &&
           sgn((l1.s - l2.e) ^ (l2.s - l2.e)) * sgn((l1.e - l2.e) ^ (l2.s - l2.e)) <= 0;
}
```

- 判断直线L1和线段L2是否相交
- 返回1表示相交,0表示不相交

```
bool Seg_inter_line(Line l1, Line l2)
{
    return sgn((l2.s - l1.e) ^ (l1.s - l1.e)) * sgn((l2.e - l1.e) ^ (l1.s - l1.e)) <= 0;
}
```

距离相关计算

- 求点到直线的距离
- 返回result最近的点，垂足

```
Point PointToLine(Point P, Line L)
{
    Point result;
    double t = ((P - L.s) * (L.e - L.s)) / ((L.e - L.s) * (L.e - L.s));
    result.x = L.s.x + (L.e.x - L.s.x) * t;
    result.y = L.s.y + (L.e.y - L.s.y) * t;
    return result;
}
```

- 求点到线段的距离
- 返回点到线段上最近的点

```
Point NearestPointToLineSeg(Point P, Line L)
{
    Point result;
    double t = ((P - L.s) * (L.e - L.s)) / ((L.e - L.s) * (L.e - L.s));
    if (t >= 0 && t <= 1)
    {
        result.x = L.s.x + (L.e.x - L.s.x) * t;
        result.y = L.s.y + (L.e.y - L.s.y) * t;
    }
    else
    {
        if (dist(P, L.s) < dist(P, L.e))
            result = L.s;
        else
            result = L.e;
    }
    return result;
}
```

点和直线相关

```
//计算多边形面积,点的编号从0~n-1
double CalcArea(Point p[], int n)
{
    double res = 0;
    for (int i = 0; i < n; i++)
        res += (p[i] ^ p[(i + 1) % n]) / 2;
    return fabs(res);
}
//*判断点在线段上
bool OnSeg(Point P, Line L)
{
    return sgn((L.s - P) ^ (L.e - P)) == 0 &&
            sgn((P.x - L.s.x) * (P.x - L.e.x)) <= 0 &&
            sgn((P.y - L.s.y) * (P.y - L.e.y)) <= 0;
}
```

凸包相关

求凸包Andrew算法

• 参数说明
o p为点的编号0…n−1,n为点的数量
o ch为生成的凸包上的点
o 返回凸包大小m,编号0…m−1

```
int ConvexHull(Point *p, int n, Point *ch) //求凸包
{
    sort(p, p + n);
    n = unique(p, p + n) - p; //去重
    int m = 0;
    for (int i = 0; i < n; ++i)
    {
        while (m > 1 && sgn((ch[m - 1] - ch[m - 2]) ^ (p[i] - ch[m - 1])) <= 0)
            --m;
        ch[m++] = p[i];
    }
    int k = m;
    for (int i = n - 2; i >= 0; i--)
    {
        while (m > k && sgn((ch[m - 1] - ch[m - 2]) ^ (p[i] - ch[m - 1])) <= 0)
            --m;
        ch[m++] = p[i];
    }
    if (n > 1)
        m--;
    return m;
}
```

极角排序

• 叉积：对于 tmp = a x b
o 如果b在a的逆时针(左边):tmp > 0
o 顺时针(右边): tmp < 0
o 同向: tmp = 0

• 相对于原点的极角排序
o 如果是相对于某一点x,只需要把x当作原点即可

```
bool mycmp(Point a, Point b)
{
    if (atan2(a.y, a.x) != atan2(b.y, b.x))
        return atan2(a.y, a.x) < atan2(b.y, b.x);
    else
        return a.x < b.x;
}
```

点和多边形的位置关系

判断点在凸多边形内

• 要求
o 点形成一个凸包，而且按逆时针排序
o 如果是顺时针把里面的<0改为>0
o 点的编号:0~n-1
• 返回值:
o -1:点在凸多边形外
o 0:点在凸多边形边界上
o 1:点在凸多边形内

```
int inConvexPoly(Point a, Point p[], int n)
{
    for (int i = 0; i < n; i++)
    {
        if (sgn((p[i] - a) ^ (p[(i + 1) % n] - a)) < 0)
            return -1;
        else if (OnSeg(a, Line(p[i], p[(i + 1) % n])))
            return 0;
    }
    return 1;
}
```

判断点是否在凸包内

```
bool inConvex(Point A, Point *p, int tot)
{
    int l = 1, r = tot - 2, mid;
    while (l <= r)
    {
        mid = (l + r) >> 1;
        double a1 = (p[mid] - p[0]) ^ (A - p[0]);
        double a2 = (p[mid + 1] - p[0]) ^ (A - p[0]);
        if (a1 >= 0 && a2 <= 0)
        {
            if (((p[mid + 1] - p[mid]) ^ (A - p[mid])) >= 0)
                return true;
            return false;
        }
        else if (a1 < 0)
            r = mid - 1;
        else
            l = mid + 1;
    }
    return false;
}
```

判断点在任意多边形内

- 射线法，poly[]的顶点数要大于等于3,点的编号0~n-1
- 返回值
  - -1:点在凸多边形外
  - 0:点在凸多边形边界上
  - 1:点在凸多边形内

```
int inPoly(Point p, Point poly[], int n)
{
    int cnt;
    Line ray, side;
    cnt = 0;
    ray.s = p;
    ray.e.y = p.y;
    ray.e.x = -100000000000.0; //-INF,注意取值防止越界

    for (int i = 0; i < n; i++)
    {
        side.s = poly[i];
        side.e = poly[(i + 1) % n];

        if (OnSeg(p, side))
            return 0;

        //如果平行轴则不考虑
        if (sgn(side.s.y - side.e.y) == 0)
            continue;

        if (OnSeg(side.s, ray))
        {
            if (sgn(side.s.y - side.e.y) > 0)
                cnt++;
        }
        else if (OnSeg(side.e, ray))
        {
            if (sgn(side.e.y - side.s.y) > 0)
                cnt++;
        }
        else if (inter(ray, side))
            cnt++;
    }
    if (cnt % 2 == 1)
        return 1;
    else
        return -1;
}
```

判断凸多边形

- 允许共线边
- 点可以是顺时针给出也可以是逆时针给出
  - 但是乱序无效
- 点的编号0…n−1

```
bool isconvex(Point poly[], int n)
{
    bool s[3];
    memset(s, false, sizeof(s));
    for (int i = 0; i < n; i++)
    {
        s[sgn((poly[(i + 1) % n] - poly[i]) ^ (poly[(i + 2) % n] - poly[i])) + 1] = true;
        if (s[0] && s[2])
            return false;
    }
    return true;
}
```

判断凸包是否相离

- 凸包a：n个点,凸包b：m个点
  - 凸包上的点不能出现在另一个凸包内
  - 凸包上的线段两两不能相交

```
bool isConvexHullSeparate(int n, int m, Point a[], Point b[])
{
    for (int i = 0; i < n; i++)
        if (inPoly(a[i], b, m) != -1)
            return false;

    for (int i = 0; i < m; i++)
        if (inPoly(b[i], a, n) != -1)
            return false;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            Line l1 = Line(a[i], a[(i + 1) % n]);
            Line l2 = Line(b[j], b[(j + 1) % m]);
            if (inter(l1, l2))
                return false;
        }
    }
    return true;
}
```

闵可夫斯基和

- 返回凸包A和B的闵可夫斯基和的凸包M上的点的数量num
- 凸包编号 0…num−1
- 由于可能三点共线，需要对M再求一次凸包(主函数内进行)

```
const int MAX = 2e5 + 5;
Point s1[MAX], s2[MAX];

int Minkowski(Point A[], int n, Point B[], int m, Point M[])
{
    int tot = 0;
    for (int i = 0; i < n; i++)
        s1[i] = A[(i + 1) % n] - A[i];

    for (int i = 0; i < m; i++)
        s2[i] = B[(i + 1) % m] - B[i];

    M[tot] = A[0] + B[0];
    int p1 = 0, p2 = 0;

    while (p1 < n && p2 < m)
        ++tot, M[tot] = M[tot - 1] + ((s1[p1] ^ s2[p2]) >= 0 ? s1[p1++] : s2[p2++]);
    while (p1 < n)
        ++tot, M[tot] = M[tot - 1] + s1[p1++];
    while (p2 < m)
        ++tot, M[tot] = M[tot - 1] + s2[p2++];
    return tot + 1;
}
```

模板 from daoqi

```
namespace computation_geometry {
    using namespace std;
    //计算几何
    //点
    struct Point {
        double x, y;
        Point() {};
        Point(double x, double y) :x(x), y(y) {};

        //减法(a-b)
        Point operator-(Point& b) {
            return { this->x - b.x,this->y - b.y };
        }
        //数乘
        Point operator*(double t) const {
            return { this->x * t,this->y * t };
        }
        //向量加法
        Point operator+(Point& b) {
            return Point{ this->x + b.x,this->y + b.y };
        }
        Point operator/(double t) {
            return { this->x / t,this->y / t };
        }
    };

    //求点积(x1*x2+y1*y2)
    double dot(Point a, Point b) {
        return a.x * b.x + a.y * b.y;
    }
    //求模长
    double len(Point a) {
        return sqrt(a.x * a.x + a.y * a.y);
    }
    //求夹角
    double angle(Point a, Point b) {
        return acos(dot(a, b) / len(a) / len(b));
    }
    //求叉积(a*b==x1*y2-x2*y1),b在a的逆时针方向值为正，否则为负
    double cross(Point b, Point a, Point c = { 0,0 }) {
```

```
    return ((a - c).x * (b - c).y) - ((a - c).y * (b - c).x);
}
//两点之间的距离
double dis(Point a, Point b) {
    return sqrt(1.0 * (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
//判断线线的位置关系
//1.直线ab与线段cd无交点：cross(a,b,c)*cross(a,b,d)>0
//2.直线ab与线段cd有交点：cross(a,b,c)*cross(a,b,d)<= 0
//计算直线交点au与bv
Point getNode(Point a, Point u, Point b, Point v) {
    double t = cross((a - b), v) / cross(v, u);
    u.x *= t, u.y *= t;
    return a + u;
}
double eps = 1e-6;Point o = { 0,0 };
double PI = acos(-1), R;
Point p[4];
//三角剖分，计算多边形与圆的相交面积
Point rotate(Point a, double b) {//将线段旋转一定的角度
    return Point(a.x * cos(b) - a.y * sin(b), a.x * sin(b) + a.y * cos(b));
}
Point norm(Point a) {//单位向量
    return a / len(a);
}
bool onSegment(Point p, Point a, Point b) {//p在ab线段上
    return fabs(cross(a - p, b - p) < eps) && (dot(a - p, b - p) <= 0);
}
double getDP2(Point a, Point b, Point& pa, Point& pb) {
    Point e = getNode(a, b - a, o, rotate(b - a, PI / 2));//找到圆心与ab的垂足
    double d = dis(o, e);
    if (!onSegment(e, a, b)) d = std::min(dis(o, a), dis(o, b));
    if (R <= d) return d;
    double len = sqrt(R * R - dis(o, e) * dis(o, e));

    pa = (norm(a - b) * len) + e;
    pb = (norm(b - a) * len) + e;
    return d;//d:线段到圆心的距离；pa,pb:直线与圆的交点
}
double sector(Point a, Point b) {//扇形面积
    double angle = acos(dot(a, b) / len(a) / len(b));//[0,pi]
    if (cross(a, b) < 0) angle = -angle;
    return R * R * angle / 2;
```

```
}

double getArea(Point a, Point b) {//面积的交
    if (fabs(cross(a, b)) < eps) return 0;//ab与圆心共线
    double da = dis(o, a), db = dis(o, b);
    if (R >= da && R >= db) return cross(a, b) / 2;//ab在圆内，R为直径
    Point pa, pb;
    double d = getDP2(a, b, pa, pb);
    if (R <= d) return sector(a, b);//ab在圆外
    if (R >= da) return cross(a, pb) / 2 + sector(pb, b);//a在圆外
    if (R >= db) return sector(a, pa) + cross(pa, b) / 2;//b在圆内
    return sector(a, pa) + cross(pa, pb) / 2 + sector(pb, b);//ab是割线
}


//凸包算法:Andrew算法
Point st[N], s[N];
int n;
int top;
double Andrew() {
    sort(s + 1, s + n + 1, [&](Point a, Point b) {
        return a.x != b.x ? a.x < b.x : a.y < b.y;
        });
    for (int i = 1;i <= n;i++) {//下凸包
        while (top > 1 && cross(st[top], s[i], st[top - 1]) <= 0) top--;
        st[++top] = s[i];
    }
    int t = top;
    for (int i = n - 1;i >= 1;i--) {//上凸包
        while (top > t && cross(st[top], s[i], st[top - 1]) <= 0) top--;
        st[++top] = s[i];
    }
    double res = 0;//周长
    for (int i = 1;i < top;i++) {
        res += dis(st[i], st[i + 1]);
    }
    n = top - 1;//为了配合下面的旋转卡壳，只求凸包不用加
    return res;
}

//旋转卡壳问题实例
//1.给定个点，求最远点对的距离
//先用Androw算法，求出凸包上的点数即n=top-1;
```

```
    double rotating_calipers() {//旋转卡壳
        double res = 0;
        for (int i = 1, j = 2;i <= n;i++) {
            while (cross(st[i + 1], st[j], st[i]) < cross(st[i + 1], st[j + 1], st[i])) j = j %
            res = max(res, max(dis(st[i], st[j]), dis(st[i + 1], s[j])));
        }
        return res;
    }
    double rotating_calipers2() {//求最大四边行的面积
        double ans = 0;
        for (int i = 1;i <= n;i++) {
            int a = i, b = i + 1;//a为i到j之间的点，b为j到i之间的点
            for (int j = i + 1;j <= n;j++) {
                while (cross(st[j], st[a + 1], st[i]) < cross(st[j], st[a], st[i])) a = a % n +
                while (cross(st[j], st[b + 1], st[i]) > cross(st[j], st[b], st[i])) b = b % n +
                ans = max(ans, -cross(st[j], st[a], st[i]) + cross(st[j], st[b], st[i]));
            }
        }
        return ans / 2;
    }
    void solve() {
        std::cin >> n;
        for (int i = 1;i <= n;++i)std::cin >> s[i].x >> s[i].y;
        Andrew();
        auto res = rotating_calipers();
        int ans = res * res;
        std::cout << ans;
    }
}
```