目录

1.平面扫描

平面扫描是用来降低算法的复杂度的方法，通过扫描线在平面上按照给定的轨迹移动的同时，不断根据扫描线扫过的部分更新信息，从而得到整体所要求的结果，既可以从左向右平移与y轴平行的直线，也可以固定射线的端点逆时针转动。

> 平面上有N NN个两两没有公共点的圆，i ii号圆的圆心在(x_i, y_i),半径为r_i。求所有最外层的，即不包含与其他圆内部的圆。

> 利用垂直于x轴的线去从左往右扫描，观察点为圆的左右端点。因为圆是没有公共点的，这使得包含的判断变得简单。如果某个圆被包含那么一定是被最近的两个外圈的其中一个圆包含（y轴上），假设这个圆已经被包含，如果远的圆如果想包含这个圆必须把两个圆都包住，那么第一层包裹已经不再是外层。所以如果存在被包括在某个外层里，一定是被包裹在最近的两个外层的其中一个。

```
typedef long long ll;
typedef pair<double, int>PDL;
const int N = 50007, M = 5000007, INF = 0x3f3f3f3f;
set<PDL>out;
vector<int>ans;
int n, m;
int cnt, num;
PDL p[N * 2];
double x[N], y[N], r[N];
bool inside(int i, int j)
{
    double dx = x[i] - x[j];
    double dy = y[i] - y[j];
    return dx * dx + dy * dy <= r[j] * r[j];
}

int main()
{
    scanf("%d", &n);

    for(int i = 1; i <= n; ++ i){
        scanf("%lf%lf%lf", &r[i], &x[i], &y[i]);
        p[++num].first = x[i] - r[i];//左端点
        p[num].second = i;
        p[++num].first = x[i] + r[i];//右端点
        p[num].second = i + n;
    }
    sort(p + 1, p + 1 + num);

    for(int i = 1; i <= num; ++ i){
        int id = p[i].second;
        if(id <= n){//左端点
            set<PDL> :: iterator it = out.lower_bound(make_pair(y[id], id));
            if(it != out.end() && inside(id, it->second))continue;//被包含，不是答案，跳过
            if(it != out.begin() && inside(id, (--it)->second))continue;
            ans.push_back(id);
            out.insert(make_pair(y[id], id));//外圈
        }
        else {//到了这个圆的右端点该删除了，已经没用了，留着会影响答案
```

```
            id -= n;
            out.erase(make_pair(y[id], id));
        }
    }
    printf("%d\n", ans.size());
    sort(ans.begin(),ans.end());
    for(int i = 0; i < ans.size(); ++i)
        printf("%d ", ans[i]);

    return 0;
}
```

2.凸包

过多边形的任意一边做一条直线，如果其他各个顶点都在这条直线的同侧，则把这个多边形叫做凸多边形。

凸包求解算法的基础便是凸多边形的定义与性质。

假设平面上n个点，过某些点作一个多边形，使这个多边形能把所有点都“包”起来。当这个多边形是凸多边形的时候，我们就叫它“凸包”。

假设每种颜料都拥有(R,G,B)三种属性，表示该种颜料红色，绿色，与蓝色的化学成分所占的比重

给你若干种已有的不限量的颜料，问是否能够勾兑出目标颜料(R0,G0,B0)

将每一种颜料映射为二维欧氏空间中的一个点，我们可以将已经给定的颜料与目的颜料在空间中标定出来

经过观察与思考，我们可以发现，一个颜料能够被勾兑出来当且仅当该颜料对应的点，位于以给定颜料所构成的凸包之中

Andrew算法

判断这样连是否为凸多边形的一部分：
如果p2-p3的斜率小于p1-p3，那么我们就没必要选择p2这个点，而是直接走p1~p3即可，也就是说我们把已经放进去的p2踢掉

```
//计算凸包，输入点数组p，个数为p输出点数组ch，函数返回凸包顶点个数。
//输入不能有重复的点，函数执行完后的输入点的顺序将被破坏（因为要排序，可以加一个数组存原来的id）
//如果不希望在凸包边上有输入点，把两个≤改成<即可
int ConvexHull(Point* p, int n, Point* ch)
{
    sort(p, p + n);
    int m = 0;
    for(int i = 0; i < n; ++ i){//下凸包
        //如果叉积≤0说明新边斜率小说明已经不是凸包边了，赶紧踢走
        while(m > 1 && Cross(ch[m - 1] - ch[m - 2], p[i] - ch[m - 2]) ≤ 0)m -- ;
        ch[m ++ ] = p[i];
    }
    int k = m;
    for(int i = n - 2; i ≥ 0; -- i){//上凸包
        while(m > k && Cross(ch[m - 1] - ch[m - 2], p[i] - ch[m - 2]) ≤ 0)m -- ;
        ch[m ++ ] = p[i];
    }
    if(n > 1) m -- ;
    return m;
}
```

直线两点式转为一般式使用公式O(1)计算点到直线距离

> 平面上有n个点，求一条直线，使得这n个点都在这条直线上或同侧，且每个点到该直线的距离之和尽量小。

首先，一条直线不分割这n个点当且仅当不分割这n个点的凸包。并且为了使这n个点到该直线的距离最小，这条直线应是凸包上某一条边所在直线。
由几何知识可得，对于点$P(x_0,y_0)$和直线$Ax+By+C=0$之间的距离

$dis=\frac{|Ax_0+By_0+C|}{\sqrt{A^2+B^2}}$

又因为这n个点在这条直线的同侧，所以对于所有的点Ax_0+By_0+C符号相同，所以预处理出所有点x坐标和y坐标的和就 **可以$\Theta(1)$算出每个点到该直线的距离之和。**

还有一个问题，如何将直线的两点式
$l_{PQ},P(x_P,y_P),Q(x_Q,y_Q)$转化为一般式$Ax+By+C=0$

简单计算可得一个解为
$A=y_Q-y_P$
$B=x_P-x_Q$
$C=-x_P*A-y_P*B$

注意千万不能用除法，否则会计算结果会特别大或NaN导致WA.

```
//Ax + By + C = 0;
double get(double A, double B, double C) {
    double k = fabs(A*sumx + B*sumy + n*C);
    double v = sqrt(A*A + B*B);//v != 0;
    return k/v;
}

double getDist(const point &a, const point &b) {
    double A = a.y-b.y;
    double B = b.x-a.x;
    double C = a.x*b.y - a.y*b.x;
    return get(A, B, C);
}

int main()
{
    int counter = 0;
    int T;
    point t;
    scanf("%d", &T);
    while(T--) {
        init();
        scanf("%d", &n);
        for(int i = 0; i < n; i++) {
            scanf("%lf%lf", &t.x, &t.y);
            sumx += t.x; sumy += t.y;
            tmp.push_back(t);
        }
        polygon_convex tres = convex_hull(tmp);
        int Size = (int)tres.P.size();
        printf("Case #%d: ", ++counter);
        if(Size == 2 || Size == 1) { //刚开始wa一次，看了看题目n>0.又把n=1考虑一下。
            printf("0.000\n");
            continue;
        }
```

```
        for(int i = 0; i < Size; i++) {
            double temp = getDist(tres.P[i], tres.P[(i+1)%Size]);
            ans = min(ans, temp);
        }
        printf("%.3lf\n", ans/n);
    }
    return 0;
}
```

判断点是否在凸包内

先判定和边界的关系
然后找到与其极角相邻的两点，凭此判断
须保证A[1]=(0,0)

```
ll in(Node a)
{
    if(a*A[1]>0||A[tot]*a>0) return 0;
    ll ps=lower_bound(A+1,A+tot+1,a,cmp2)-A-1;
    return (a-A[ps])*(A[ps%tot+1]-A[ps])<=0;
}
```

3.旋转卡壳

利用凸包上一些奇妙的单调性，求解

- 多边形直径
- 多边形宽度
- 最小面积矩形覆盖

复杂度$O(n)$

```
for(int i=1,p=1;i<=n;i++)
{
    //这份代码只卡了一个点，还可以同时卡多个点
    while(H(A[i],A[i+1],A[p%n+1])>=H(A[i],A[i+1],A[p])) p=p%n+1;
    Ans=max(Ans,max((A[p]-A[i]).dis(),(A[p]-A[i+1]).dis()));
}
```

> 给定平面上 n 个点，求凸包直径。
> 第一行一个正整数 n。接下来 n 行，每行两个整数 x,y，表示一个点的坐标。输出一行一个整数，表示答案的平方。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 50007, M = 50007, INF = 0x3f3f3f3f;
const double DINF = 12345678910, eps = 1e-10, PI = acos(-1);
struct Point{
    int x, y;
    Point(double x = 0, double y = 0):x(x), y(y){ }//构造函数
};
typedef Point Vector;
Vector operator + (Vector A, Vector B){return Vector(A.x + B.x, A.y + B.y);}
Vector operator - (Point A, Point B){return Vector(A.x - B.x, A.y - B.y);}
Vector operator * (Vector A, double p){return Vector(A.x * p, A.y * p);}
Vector operator / (Vector A, double p){return Vector(A.x / p, A.y / p);}
```

```
bool operator < (const Point& a, Point& b) {return a.x < b.x || (a.x == b.x && a.y < b. y);}
int dcmp(double x){
    if(fabs(x) < eps) return 0;
    else return x < 0 ? -1 : 1;
}
bool operator == (const Point& a, const Point& b){return !dcmp(a.x - b.x) && !dcmp(a.y - b.y);}
double Polar_angle(Vector A){return atan2(A.y, A.x);}
inline double D_to_R(double D)//角度转弧度
{ return PI/180*D; }
double Cross(Vector A, Vector B){return A.x * B.y - B.x * A.y;}
Vector Rotate(Vector A, double rad){
    return Vector(A.x * cos(rad) - A.y * sin(rad), A.x * sin(rad) + A.y * cos(rad));
}
Point Get_line_intersection(Point P,Vector v,Point Q,Vector w)
{
    Vector u = P - Q;
    double t = Cross(w, u) / Cross(v, w);
    return P + v * t;
}
double convex_polygon_area(Point* p, int n)
{
    double area = 0;
    for(int i = 1; i ≤ n - 2; ++ i)
        area += Cross(p[i] - p[0], p[i + 1] - p[0]);
    return area / 2;
}
int ConvexHull(Point* p, int n, Point* ch)
{
    sort(p, p + n);
    int m = 0;
    for(int i = 0; i < n; ++ i){//下凸包
        while(m > 1 && Cross(ch[m - 1] - ch[m - 2], p[i] - ch[m - 2]) ≤ 0)m -- ;
        ch[m ++ ] = p[i];
    }
    int k = m;
    for(int i = n - 2; i ≥ 0; -- i){//上凸包
        while(m > k && Cross(ch[m - 1] - ch[m - 2], p[i] - ch[m - 2]) ≤ 0)m -- ;
        ch[m ++ ] = p[i];
    }
    if(n > 1) m -- ;
    return m;
}
int get_dist (const Point &x){return x.x * x.x + x.y * x.y;}
Point p[N], con[N];
int con_num;
double Rotating_calipers()
{
    int op = 1, ans = 0;
    for(int i = 0; i < con_num; ++ i){
        while(Cross((con[i] - con[op]), (con[i + 1] - con[i])) < Cross((con[i] - con[op + 1]), (con[i + 1] - con[i])))
        //（写成≤会被两个点的数据卡掉，所以必须写成<）
            op = (op + 1) % con_num;
            ans = max(ans, max(get_dist(con[i] - con[op]), get_dist(con[i + 1] - con[op])));
    }
```

```
    printf("%d\n", ans);
    return ans;
}
int n ;
int main()
{
    scanf("%d", &n);
    for(int i = 0; i < n; ++ i){
        scanf("%d%d", &p[i].x, &p[i].y);
    }
    con_num = ConvexHull(p, n, con);
    double res = Rotating_calipers();
    return 0;
}

```

4.半平面交

定义一个半平面为一个向量的左侧，半平面交即为若干个半平面的交集。

有向直线

```

//有向直线。它的左边就是对应的半平面
struct Line
{
    Point P;//直线上任意一点
    Vector v;//方向向量。左边就是对应的半平面
    double deg;//极角
    Line(){}
    Line(Point P, Vector v):P(P), v(v){deg = atan2(v.y, v.x);}
    bool operator < (const Line& L)const {//排序时使用的比较运算符
        return deg < L.deg;
    }
};
```

$O(nlogn)$的半平面交

```
//半平面交一般是一个凸多边形，但是有时候会是一个无界多边形
//甚至会是一个直线、线段、点，但是结果一定是凸的

//点p在有向直线L的左边（线上的不算）（叉积大于0a在b的左侧，小于0在右侧[sin夹角]）
bool on_left(Line L, Point P){return Cross(L.v, P - L.P) > 0;}
//两个有向直线的交点/假定交点唯一存在
Point get_intersection(Line a, Line b)
{
    Vector u = a.P - b.P;
    double t = Cross(b.v, u) / Cross(a.v, b.v);
    return a.P + a.v * t;
}
//半平面交的主过程
//L数组存所有有向直线，n为有向直线个数，半平面交的交点存在poly数组中
```

```cpp
int half_plane_intersection(Line* L, int n, Point* poly)
{
    sort(L, L + n);//按照极角排序

    int first, last;//双端队列
    Point *p = new Point[n];//p[i]为q[i]和q[i + 1]的交点
    Line *q = new Line[n];//手写的Line类型的双端队列（数组）
    q[first = last = 0] = L[0];//双端队列初始化的时候只有一个半平面L[0]
    for(int i = 1; i < n; ++ i){
        while(first < last && !on_left(L[i], p[last - 1]))last -- ;
        while(first < last && !on_left(L[i], p[first]))first ++ ;
        q[++ last] = L[i];//新的点是一定要放进去的
        if(fabs(Cross(q[last].v, q[last - 1].v)) < eps){
        //相邻的两个向量平行且同向，取内侧的那一个
            last -- ;//如果新的向量上的一个点在老的向量的左侧就取新的
            if(on_left(q[last], L[i].P))q[last] = L[i];
        }
        if(first < last)p[last - 1] = get_intersection(q[last - 1], q[last]);
    }
    while(first < last && !on_left(q[first], p[last - 1]))last -- ;
    //删除无用的平面

    if(last - first <= 1)return 0;//空集
    p[last] = get_intersection(q[last], q[first]);//计算首尾两个半平面交（环状）
    //从手写deque中复制答案到输出数组中
    int m = 0;
    for(int i = first ; i <= last; ++ i)poly[m ++ ] = p[i];
    return m;
}
```

> 逆时针给出n个凸多边形的顶点坐标，求它们交的面积。

二分 + 半平面交

> 在大海的中央，有一个凸n边形的小岛，求出岛上离海边最远的一个点到海边的距离，保留6位小数。

```cpp
//有向直线。它的左边就是对应的半平面
struct Line
{
    Point P;//直线上任意一点
    Vector v;//方向向量。左边就是对应的半平面
    double deg;//极角
    Line(){}
    Line(Point P, Vector v):P(P), v(v){deg = atan2(v.y, v.x);}
    bool operator < (const Line& L)const {//排序时使用的比较运算符
        return deg < L.deg;
    }
};
bool on_left(Line L, Point P){return Cross(L.v, P - L.P) > 0;}
//两个有向直线的交点/假定交点唯一存在
Point get_intersection(Line a, Line b)
{
    Vector u = a.P - b.P;
    double t = Cross(b.v, u) / Cross(a.v, b.v);
    return a.P + a.v * t;
}
```

```
//半平面交的主过程
int half_plane_intersection(Line* L, int n, Point* poly)
{
    sort(L, L + n);//按照极角排序

    int first, last;//双端队列
    Point *p = new Point[n];//p[i]为q[i]和q[i + 1]的交点
    Line *q = new Line[n];//手写的Line类型的双端队列（数组）
    q[first = last = 0] = L[0];//双端队列初始化的时候只有一个半平面L[0]
    for(int i = 1; i < n; ++ i){
        while(first < last && !on_left(L[i], p[last - 1]))last -- ;
        while(first < last && !on_left(L[i], p[first]))first ++ ;
        q[++ last] = L[i];//新的点是一定要放进去的
        if(fabs(Cross(q[last].v, q[last - 1].v)) < eps){
        //相邻的两个向量平行且同向，取内侧的那一个
            last -- ;//如果新的向量上的一个点在老的向量的左侧就取新的
            if(on_left(q[last], L[i].P))q[last] = L[i];
        }
        if(first < last)p[last - 1] = get_intersection(q[last - 1], q[last]);
    }
    while(first < last && !on_left(q[first], p[last - 1]))last -- ;
    //删除无用的平面

    if(last - first <= 1)return 0;//空集
    p[last] = get_intersection(q[last], q[first]);//计算首尾两个半平面交（环状）
    //从手写deque中复制答案到输出数组中
    int m = 0;
    for(int i = first ; i <= last; ++ i)poly[m ++ ] = p[i];
    return m;
}
Point p[N], poly[N];
Line L[N];
Vector v[M], v2[M];
int n;

int main()
{
    while(scanf("%d", &n) != EOF && n)
    {
        int m, x, y;
        for(int i = 0; i < n; ++i){
            scanf("%d%d", &x, &y);
            p[i] = Point(x, y);
        }
        for(int i = 0; i < n; ++ i){
            v[i] = p[(i + 1) % n] - p[i];
            v2[i] = Normal(v[i]);//单位法向量
        }
        double l = 0, r = 20000;
        while(r - l > eps){
            double mid = l + (r - l) / 2;
            for(int i = 0; i < n; ++ i)
                L[i] = Line(p[i] + v2[i] * mid, v[i]);
          //收缩/放大整个凸多边形
          //点+单位向量*长度=平移后的点
```

```
            m = half_plane_intersection(L, n, poly);
            if(!m)r = mid;
            else l = mid;
        }
        printf("%.6f\n", l);
    }
    return 0;
}

```

5.闵可夫斯基和

> 定义p+q=(p.x+q.x,p.y+q.y)，给定两个点集，求{pi+qj}的凸包（凸壳）的问题

```
Vector V1[N],V2[N];
inline int Mincowski(Point *P1,Re n,Point *P2,Re m,Vector *V){//【闵可夫斯基和】求两个凸包{P1},{P2}的向量集合{V}={P1+P2}构成的凸包
    for(Re i=1;i<=n;++i)V1[i]=P1[i<n?i+1:1]-P1[i];
    for(Re i=1;i<=m;++i)V2[i]=P2[i<m?i+1:1]-P2[i];
    Re t=0,i=1,j=1;V[++t]=P1[1]+P2[1];
    while(i<=n&&j<=m)++t,V[t]=V[t-1]+(dcmp(Cro(V1[i],V2[j]))>0?V1[i++]:V2[j++]);
    while(i<=n)++t,V[t]=V[t-1]+V1[i++];
    while(j<=m)++t,V[t]=V[t-1]+V2[j++];
    return t;
}
```

6.平面区域

当平面上有很多线段时，组成的图形往往不止一个多边形，而是一个平面直线图（PSLG），它代表一个平面区域划分，其中一个区域是一个多边形。

如果只有点和边的信息，如何找出所有区域呢？为方便起见，我们把每条边u-v拆成两条半边 u-v 和 v-u.并且每条半边只与它左边的面相邻。接下来，我们从一条半边出发遍历，每次像卷包裹算法那样找一个逆时针转的尽量多的边作为下一条边，直到回到出发的那条半边。

程序实现上可以把边表扩大一倍，让编号为i的半边的反向边的编号为i^1.

首先看一下边的存储结构定义：

```
//边
struct Edge
{
    int from, to; //起点，终点
    double ang;   //极角
};
```

平面直线图：

```
//平面直线图。
struct PSLG
{
    int n, m;                //顶点数 边数
    int face_cnt;            //面数
    double x[maxn];          //顶点
```

```
    double y[maxn];
    vector<Edge> edges;
    vector<int> G[maxn];
    bool vis[maxn*2];       // 每条边是否被访问过
    int left[maxn*2];        //左面的编号
    int prev[maxn*2];        //相同起点的上一条边，即顺时针旋转碰到的下一条边的编号
    vector<Polygon> faces;  //面
    double areas[maxn];     //每个polygon的面积

    void init(int n)
    {
        this->n = n;
        edges.clear();
        faces.clear();
        for(int i = 0; i < n; i++)
            G[i].clear();
    }

    //有向线段from-to的极角
    double getAngle(int from, int to)
    {
        return atan2(y[to]-y[from], x[to]-x[from]);
    }

    void AddEdge(int from, int to)
    {
        edges.push_back((Edge){from, to, getAngle(from, to)});
        edges.push_back((Edge){to, from, getAngle(to, from)});
        m = edges.size();
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }

    //找出faces并计算面积
    void Build()
    {
        int u, i, j, d, from, e;
        //给从u开始个各条边按照极角排序
        for(u = 0; u < n; u++)
        {
            d = G[u].size();
            for(i = 0; i < d; i++)
                for(j = i+1; j < d; j++)   //这里假设从u出发的线段不会太多
                    if(edges[G[u][i]].ang > edges[G[u][j]].ang)
                        swap(G[u][i], G[u][j]);
            for(i = 0; i < d; i++)
                prev[G[u][(i+1)%d]] = G[u][i];
        }

        face_cnt = 0;
        memset(vis, false, sizeof(vis));
        for(u = 0; u < n; u++)
            for(i = 0; i < G[u].size(); i++)
            {
                e = G[u][i];
```

```
                if(!vis[e])                                    //逆时针找圈
                {
                    face_cnt++;                                //找到一个新的面
                    Polygon poly;
                    for(;;)
                    {
                        vis[e] = true;
                        left[e] = face_cnt;
                        from = edges[e].from;
                        poly.push_back(Point(x[from], y[from]));  //把新的点加入面中
                        e = prev[e ^ 1];                        //e^1为反向边，然后prev就是需要走的下一条边
                        if(e == G[u][i])                        //回到了起始边
                            break;
                    }
                    faces.push_back(poly);
                }
            }
        //对于连通图，惟一一个面积小于零的面是无限面
        //对于内部区域来说，无限面多边形的各个顶点是顺时针的
        for(i = 0; i < faces.size(); i++)                      //计算各个面的面积
            areas[i] = PolygonArea(faces[i]);
    }
};
```

7.平面最近点对

> 给定平面上n个点，找出其中的一对点的距离，使得在这n个点的所有点对中，该距离为所有点对中最小的

```
struct Point
{
    double x, y;
    Point(double x = 0, double y = 0):x(x), y(y){ }
};
Point p[N];
int tmp[N], n;
bool cmp(const Point &a, const Point &b){
        if(a.x == b.x)return a.y < b.y;
        return a.x < b.x;
}
bool cmps(const int &a, const int &b){
    return p[a].y < p[b].y;
}
double get_dist(int i, int j)
{
    return sqrt((p[i].x - p[j].x) * (p[i].x - p[j].x) + (p[i].y - p[j].y) * (p[i].y - p[j].y));
}
double merge(int l, int r)
{
    double d = DINF;
    if(l == r)return d;
    if(l + 1 == r)return get_dist(l, r);
    int mid = l + r >> 1;
    double d1 = merge(l, mid), d2 = merge(mid + 1, r);
```

```
    d = min(d1, d2);
    int tot = 0;
    for(int i = l; i <= r; ++ i)
        if(fabs(p[mid].x - p[i].x) <= d)
            tmp[++ tot] = i;
    sort(tmp + 1, tmp + 1 + tot, cmps);

    for(int i = 1; i <= tot; ++ i)
        for(int j = i + 1; j <= tot; ++ j){
            if(fabs(p[tmp[j]].y - p[tmp[i]].y) > d)break;
            d = min(d, get_dist(tmp[i], tmp[j]));
        }
    return d;
}
int main()
{
    scanf("%d", &n);
    for(int i = 1; i <= n; ++ i)
        scanf("%lf%lf", &p[i].x, &p[i].y);
    sort(p + 1, p + 1 + n, cmp);
    double ans = merge(1, n);
    printf("%.4f\n", ans);
    return 0;
}

```

> 题意实际上就是给定长度为 n 的一串序列$a_1, a_2, \dots, a_n$，找到两个正整数$i, j \in [1, n]$，求$(i-j)^2+(\sum_{k=i+1}^{j} a_k)^2$的最小值
>
> 分析
> 将原式中的$\sum_{k=i+1}^{j} a_k$用前缀和$S_j - S_i$替代，则原式变换为$(i-j)^2+(S_j-S_i)^2$
>
> 不难看出变换后的式子为两点之间欧几里德距离的平方，于是原题转化为求平面上的最近点对距离的平方。
>
> 关于最近点对距离的求解可使用二分法

注意本题的数据会卡普通的$O(nlogn)$的分治求平面最近点对算法，我们需要加一些玄学优化，比如在筛y轴的时候一旦大于d就直接break，因为是排过序的，当前的y都大了，说明接下来的所有点都不能用了。

```
#include<cstdio>
#include<algorithm>
#include<cmath>
#define R register
using namespace std;
const int N = 200007;

typedef long long ll;

struct Point {
    ll x, y;
    bool operator < (const Point& t)const {
        return y < t.y;
    }
};

Point v[N], tmp[N];
int n;

```

```
ll get_dist(Point A, Point B)
{
    return (ll)(A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y);
}

ll solve(const int l, const int r){//求平面最近点对的分治算法
    if(l == r)return 0x3f3f3f3f3f3f;
    if(l + 1 == r)return get_dist(v[l], v[r]);//分治到了一个点对，直接返回答案
    int mid = (l + r) >> 1;
    ll d1 = solve(l, mid), d2 = solve(mid + 1, r);
    ll d = min(d1, d2);
    int tot = 0;
    //先筛选x
    for(int i = l; i <= r; ++ i){
        if((v[mid].x - v[i].x) * (v[mid].x - v[i].x) <= d)tmp[++tot] = v[i];
    }
    sort(tmp + 1, tmp + tot + 1);//按y排序
    //再筛选y
    for(int i = 1; i <= tot; ++ i){
        for(int j = i + 1; j <= tot; ++ j){
            if((tmp[i].y - tmp[j].y) * (tmp[i].y - tmp[j].y) > d)break;//剪枝优化，不加过不了本题
            d = min(d, get_dist(tmp[i], tmp[j]));
        }
    }
    return d;
}

int main()
{
    scanf("%d", &n);
    for(int i = 1; i <= n; ++ i){
        int x;
        scanf("%d", &x);
        v[i].y = v[i - 1].y + x;//y就是前缀和，我们通过公式推出来的
        v[i].x = i;
    }
    ll minv = solve(1, n);
    printf("%lld\n", minv);
    return 0;
}

```

更快的神仙操作

```
@3A17K巨佬的神仙操作！
我们充分发扬人类智慧：
将所有点全部绕原点旋转同一个角度，然后按 x 坐标排序
根据数学直觉，在随机旋转后，答案中的两个点在数组中肯定不会离得太远
所以我们只取每个点向后的5个点来计算答案
这样速度快得飞起，在 n=1000000 时都可以在1s内卡过
注意旋转θ角时坐标变换
x'=xcosθ-ysinθ
y'=xsinθ+ycosθ
代码如下：
#include<cstdio>
#include<cmath>
```

```cpp
#include<iostream>
#include<algorithm>
using namespace std;
const int N=2e5+50;
#define D double
struct spot{
    D a[4];
}p[N];
D x,y,x_,y_,z,w,ans;
int n;
bool mmp(const spot &u,const spot &v){
    return u.a[0]<v.a[0];
}
int main(){
    scanf("%d",&n);
    z=sin(1),w=cos(1);  //旋转1弧度≈57°
    for(int i=1;i<=n;i++){
        scanf("%lf%lf",&x,&y);
        x_=x*w-y*z;
        y_=x*z+y*w;   //计算旋转后的坐标
        p[i].a[0]=x_;
        p[i].a[1]=y_;
        p[i].a[2]=x;
        p[i].a[3]=y;   //存下来
    }
    sort(p+1,p+n+1,mmp);   //排序
    for(int i=n+1;i<=n+10;i++)
    p[i].a[0]=p[i].a[1]=-N-0.01;  //边界处理
    ans=2e9+0.01;  //初始化答案
    for(int i=1;i<=n;i++)
    for(int j=1;j<=5;j++){  //枚举
        x=p[i].a[2];y=p[i].a[3];
        x_=p[i+j].a[2];y_=p[i+j].a[3];
        z=sqrt((x-x_)*(x-x_)+(y-y_)*(y-y_));  //计算距离
        if(ans>z)ans=z;   //更新答案
    }
    printf("%.4lf\n",ans);  //输出
}
```