**1.计算几何的基础函数**

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cmath>

#include <algorithm>

#include <string>

#include <stack>

#include <queue>

using namespace std;

typedef long long ll;

const int N = 10009;

const int M = 1000;

const int INF = 0x3fffffff;

const double eps = 1e-8;

const double PI = acos(-1.0);

int sgn(double x)

{

if(fabs(x)<eps) return 0;

if(x<0) return -1;

else return 1;

}

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);

}

//叉积

double operator ^ (const Point&b)const{

return x\*b.y - y\*b.x;

}

//点积

double operator \* (const Point&b)const{

return x\*b.x + y\*b.y;

}

};

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

//只有第一个值是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);

else return make\_pair(1,res);

}

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);

}

};

//两个点距离

double dist(Point a,Point b)

{

return sqrt((a-b)\*(a-b));

}

//点到直线的距离,返回result，是点到直线最近的点

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;

}

//点到线段的距离，返回点到线段最近的点

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;

}

//判断直线l1与线段l2是否相交

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;

}

//判断线段相交

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;

}

//判断点在凸多边形内

//点形成一个凸包，而且按逆时针排序(如果是顺时针把里面的<0改为>0)

//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:点在凸变形内

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:点在多边形外

//0:点在多边形边界上

//1:点在多边形内

int inPoly(Point p,Point poly[],int n)

{

int cnt = 0;

Line ray,side;

ray.s = p;

ray.e.y = p.y;

ray.e.x = -INF;

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;

}

//点到多边形的距离

//点在多边形上则距离为0

//否则，距离为点到多变形上所有边的距离中的最小值

double PointToPoly(Point p,Point poly[],int n)

{

if( inPoly(p,poly,n) >= 0 ) return 0.0;

double ans = INF;

for( int i = 0 ; i < n ; ++i ){

Point s = NearestPointToLineSeg(p,Line(poly[i],poly[(i+1)%n]));

ans = min(ans,dist(p,s));

}

return ans;

}

//返回线段的交点

Point SegInter(Line l1,Line l2)

{

Point res = l1.s;

double t = ((l1.s-l2.e)^(l2.s-l2.e))/((l1.s-l1.e)^(l2.s-l2.e));

res.x += t\*(l1.e.x-l1.s.x);

res.y += t\*(l1.e.y-l1.s.y);

return res;

}

//给定一条直线和x值，返回一个直线上的点

Point getPointX(Line L,int x)

{

double a,b,c;

a = L.s.y-L.e.y;

b = L.e.x-L.s.x;

c = L.s.x\*L.e.y-L.e.x\*L.s.y;

if(sgn(b) == 0) return Point(x,0.0);

else return Point(x,(-c-a\*x)/b);

}

//给定一条直线和y值，返回一个直线上的点

Point getPointY(Line L,int y)

{

double a,b,c;

a = L.s.y-L.e.y;

b = L.e.x-L.s.x;

c = L.s.x\*L.e.y-L.e.x\*L.s.y;

if(sgn(a) == 0) return Point(0.0,y);

else return Point((-c-b\*y)/a,y);

}

//线段与多边形的关系

//返回值:

//1:线段与多边形相交

//0:不相交

int SegOnPoly(Line L,Point poly[],int n)

{

for( int i = 0 ; i < n ; ++i ){

if(inter(L,Line(poly[i],poly[(i+1)%n]))) return 1;

}

if( inPoly(L.s,poly,n) || inPoly(L.e,poly,n) ) return 1;

return 0;

}

//线段到多边形的距离

//线段与多变形相交，距离为0

//否则是线段两个端点到多边形距离的最小值

double SegToPoly(Line L,Point poly[],int n)

{

if( SegOnPoly(L,poly,n) >= 0 ) return 0.0;

return min(PointToPoly(L.s,poly,n),PointToPoly(L.e,poly,n));

}

//多边形与多边形的关系

//返回值:

//1:相交；0:不相交

int PolyOnPoly(Point poly1[],Point poly2[],int n1,int n2)

{

//某条线段与多边形相交

for( int i = 0 ; i < n1 ; ++i ){

if(SegOnPoly(Line(poly1[i],poly1[(i+1)%n1]),poly2,n2)) return 1;

}

//多边形嵌套

int flag = 0;

for( int i = 0 ; i < n1 ; ++i ){

if(!inPoly(poly1[i],poly2,n2)) flag = 1;

}

if( !flag ) return 1;

flag = 0;

for( int i = 0 ; i < n2 ; ++i ){

if(!inPoly(poly2[i],poly1,n1)) flag = 1;

}

if( !flag ) return 1;

return 0;

}

//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

//点的编号1~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;

}

//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);

}

Point poly[109];

int main()

{

return 0;

}

**2.平面最小点对**

#include <iostream>

#include <cstdio>

#include <cstring>

using namespace std;

typedef long long ll;

const int N = 100090;

const int M = 1000;

const double INF = 1e20;

const double eps = 1e-8;

const double PI = acos(-1.0);

//用分冶法求平面上的最小点对

//先按x从小到大的顺序对点排序，假设左区间为[L,M](最小距离为d1),右区间为[M+1,R](最小距离为d2)

//如果最小点对全在左区间或右区间，则d=min(d1,d2),但可能一个点在左区间，一个点在右区间产生最小值

//计算左区间每个点到右区间每个点的距离，得到最小值与d1和d2比较，但最坏的时间复杂度为0(n^2)

//所以对此进行优化：

//优化1:若左右区间内的某个点的x到区间中点x的距离大于d，则可以不用计算该点到其他点的距离

//统计这部分符合条件的点，求距离，但可能一个点也筛不掉，时间复杂度任然很高

//优化2:将取出来的那部分点按y坐标大小进行排序，如果p.y-q.y < d,那么q后面的点都不要算了

//根据某个理论，q前面的点最多6个，所以这一步为常数时间复杂度，所以总的时间复杂度为0(nlogn)

struct Point

{

double x,y;

};

double dist(Point a,Point b)

{

return sqrt((a.x-b.x)\*(a.x-b.x)+(a.y-b.y)\*(a.y-b.y));

}

Point p[N],tmpt[N];

bool cmpxy(Point a,Point b)

{

if(a.x != b.x) return a.x < b.x;

else return a.y < b.y;

}

bool cmpy(Point a,Point b)

{

return a.y < b.y;

}

double Closest\_Pair(int left,int right)

{

double d = INF;

if(left == right) return d;

if(left+1 == right) return dist(p[left],p[right]);

int mid = (left+right)>>1;

double d1 = Closest\_Pair(left,mid);

double d2 = Closest\_Pair(mid+1,right);

d = min(d1,d2);

int k = 0;

//优化1

for( int i = left ; i <= right ; ++i ){

if(fabs(p[mid].x-p[i].x) <= d)

tmpt[k++] = p[i];

}

//优化2

sort(tmpt,tmpt+k,cmpy);

for( int i = 0 ; i < k ; ++i ){

for( int j = i+1 ; j < k && tmpt[j].y-tmpt[i].y < d ; ++j ){

d = min(d,dist(tmpt[i],tmpt[j]));

}

}

return d;

}

int main()

{

int n;

while(~scanf("%d",&n)&&n){

for( int i = 0 ; i < n ; ++i ){

scanf("%lf%lf",&p[i].x,&p[i].y);

}

sort(p,p+n,cmpxy);

printf("%.2lf\n",Closest\_Pair(0,n-1)/2);

}

return 0;

}

**3.旋转卡壳**

//求凸包，Graham算法

//先找出最左下角的点p，将其他点按照与p点的极角从小到大排序，极角相同，按到p点的距离从小到大排序

//用一个栈存储凸包上的点，如果点的个数n>3,Stack[0]=c1,Stack[1]=c2(点的编号)

//循环找点q使Stack[top-1],Stack[top-2]能够"左转"，即(Stack[top-1]-Stack[top-2])^(q-Stack[i-1])<=0

//点的编号0~n-1

Point List[N];

int Stack[N],top;

//相对于list[0]的极角排序

bool \_cmp(Point p1,Point p2)

{

double tmp = (p1-List[0])^(p2-List[0]);

if(sgn(tmp)>0) return true;

else if(sgn(tmp)==0 && sgn(dist(p1,List[0])-dist(p2,List[0])) <= 0)

return true;

else return false;

}

void Graham(int n)

{

Point p0;

int k = 0;

p0 = List[0];

//找最下边的一个点

for( int i = 1 ; i < n ; ++i ){

if( (p0.y > List[i].y) || (p0.y == List[i].y && p0.x > List[i].x) ){

p0 = List[i];

k = i;

}

}

swap(List[k],List[0]);

sort(List+1,List+n,\_cmp);

if(n == 1){

top = 1;

Stack[0] = 0;

return;

}

if(n == 2){

top = 2;

Stack[0] = 0;

Stack[1] = 1;

return;

}

Stack[0] = 0;

Stack[1] = 1;

top = 2;

for( int i = 2 ; i < n ; ++i ){

while(top > 1 &&

sgn((List[Stack[top-1]]-List[Stack[top-2]])^(List[i]-List[Stack[top-2]])) <= 0)

top--;

Stack[top++] = i;

}

}

//旋转卡壳，求两点间距离的平方的最大值

//对任意一条边，逆时针看与该边相连的点，总是先增后减，找到那个极值点p，求出dist

//然后旋转该边到相邻的边L，此时极值点也会跟着进行旋转

//即能在常数时间内逆时针找到到边L的极值点

//所以时间复杂度为0(n)

//int rotating\_calipers(Point p[],int n)

//{

// int ans = 0;

// Point v;

// int cur = 1;

// for( int i = 0 ; i < n ; ++i ){

// v = p[i]-p[(i+1)%n];

// while((v^(p[(cur+1)%n]-p[cur]))<0)

// cur = (cur+1)%n;

//可能平行，所以要把dist(p[(i+1)%n],p[(cur+1)%n])的距离也算出来

// ans = max(ans,max(dist(p[i],p[cur]),dist(p[(i+1)%n],p[(cur+1)%n])));

// }

// return ans;

//}

//旋转卡壳，求平面点集最大的三角形面积

//枚举i,j,极值点k会跟着一起旋转，能在常数时间内确定k，所以时间复杂度为0(n^2)

//旋转卡壳，求平面点集最大的三角形面积

double rotating\_calipers(Point p[],int n)

{

double ans = 0;

int i,j,k;

for( i = 0 ; i < n ; ++i ){

for( j = (i+1)%n,k = (j+1)%n ; (j+1)%n != i ; j = (j+1)%n ){

while( sgn((p[(k+1)%n]-p[k])^(p[j]-p[i])) < 0 )

k = (k+1)%n;

ans = max(ans,fabs((p[k]-p[j])^(p[j]-p[i])));

}

}

return ans;

}

**4.hdu6055（统计正方形的个数）**

int graph[M][M];

vector<PA>st;

//平面区域内给n个整点

//求一个平面区域内的正方形个数

bool judge( int x , int y )

{

//忘了判断是否越界，wa了无数次

if( x+205 < 0 || y+205 < 0 ) return false;

if( graph[x+205][y+205] ) return true;

else return false;

}

int main()

{

int n;

while( ~scanf("%d",&n) ){

memset( graph , 0 , sizeof(graph) );

st.clear();

int a,b;

for( int i = 0 ; i < n ; ++i ){

scanf("%d%d",&a,&b);

graph[a+205][b+205] = 1;

st.push\_back(PA(a,b));

}

int x1,y1,x2,y2,cnt = 0;

for( int i = 0 ; i < n ; ++i ){

for( int j = 0 ; j < n ; ++j ){

if( i == j ) continue;

x1 = st[i].first;

y1 = st[i].second;

x2 = st[j].first;

y2 = st[j].second;

if( y1 == y2 ) continue;

if( x1>=x2&&y1>=y2&&judge(x1-(y1-y2),y1+(x1-x2))

&&judge(x2-(y1-y2),y2+(x1-x2)) ) cnt++;

}

}

printf("%d\n",cnt);

}

return 0;

}