首先是什么叫计算几何：就是利用计算机建立数学模型解决几何问题

一般常见的提醒有两种，平面的和三维的

首先我们要了解一些前置的技能，大部分都是高中数学的知识

首先我们从比较常见的二维开始说：

点：一般用坐标表示（x，y）

向量：和点差不多，一般也是（x，y）只不过这个时候表示的是矢量

线：一般用一个点和一个向量表示，点确定位置，矢量代表线的方向

特殊的是线段，用两个点就可以确定

点乘：

这个不说了，很常见。。。

```
double Dot(double x1,double y1,double x2,double y2)  //计算点积
{
    return (x1*y2+x2*y1);
}
```

叉乘：

关于叉积：叉积=0是指两向量平行（重合）；叉积>0，则向量a在向量b的顺时针方向（粗略的理解为在a在b的下方）；叉积<0，则向量a在向量b的逆时针方向，可以理解为在a在b的上方

```
double cross(double x1,double y1,double x2,double y2)  //计算叉积
{
    return (x1*y2-x2*y1);
}
```

多边形：开数组然后按照顺时针或者逆时针的顺序存储点坐标即可，矩形特殊可以只存一条对角线

曲线：常见的是圆，一般记录圆心和半径

基本公式：

正弦定理：$$\frac{a}{\sin A}=\frac{b}{\sin B}=\frac{c}{\sin C}=2R$$

余弦定理：$$\begin{aligned} a^2&=b^2+c^2-2bc\cos A\\ b^2&=a^2+c^2-2ac\cos B\\ c^2&=a^2+b^2-2ab\cos C \end{aligned}$$

其中a,b,c表示角A，B，C分别对应的边

基本操作：

点与线的位置关系：有一条过P点且知道方向矢量的线，问点Q在这条线的哪边（外内）

我们可以利用向量叉乘的性质$\overrightarrow{PQ} \times \mathbf{v}$，判断此时的正负，如果为负，则Q在直线的上方，为0，则Q在直线上，为正，Q在直线下方

快速排斥实验和跨立实验

判断线段相交：

首先特殊情况，先判断平行，直接看两条线段所在直线的斜率是否相等即可

部分重合或者全部重合，看是否有三点以上的点共线

两条线交点为一条线端点：也是判断是否有三点共线

剩下的情况：

规定一条线段占用的区域是以这条线段为对角线，各边均与某一坐标轴平行的矩形所占用的区域，可以发现，只要两条线段的区域没有重合，则两条线段一定不相交。

由此

可以快速的判断出两条线段不相交，上诉情况被称为**未通过快速排斥实验**

未通过快速排斥实验室两条限度那无交点的充分不必要条件，我们还需要进一步的判断。

如果a，b两条线段相交，b线段的端点一定分布在a线段所在直线的两端，同理a的两个端点也分布在b线段所在直线的两端，我们可以直接判断一条线段的端点对另一条线段所在直线的位置关系。

此为**跨立实验**，如果通过了跨立实验（特殊情况共线也可），（所以）结合快速排斥实验我们可以解决这个问题。

求任意多边形的周长和面积

求周长久直接算长度就可以

面积：

考虑向量积的模的几何意义，我们可以利用向量积完成。

讲多边形上的点逆时针表级为P1，P2-----Pn，再任选一个辅助点O，记向量Vi=Pi-O，则

$$S = \frac{1}{2}\left|\sum_{i=1}^{n} \mathbf{v}_i \times \mathbf{v}_{i \bmod n+1}\right|$$

极角序

首先得知道极坐标（人教版高中数学选修 4-4），在平面内取一点O，叫做极点选定一个长度单位和角度的正方向（一般是逆时针）

对于平面内的任何一点M，（ρ，θ）是一个有序对，ρ表示的是M到O的长度（一般叫做极径），θ叫做M的极角。

那么给定平面上的一些点，把它们按照一个选定的中心点排成顺/逆时针。通称极角排序。

这里给两种方法：

首先是存储

```
struct point//存储点
{
    double x,y;
};

double cross(double x1,double y1,double x2,double y2)  //计算叉积
{
    return (x1*y2-x2*y1);
}

double compare(point a,point b,point c)//计算极角
{
    return cross((b.x-a.x),(b.y-a.y),(c.x-a.x),(c.y-a.y));
}
```

第一种是用叉乘的：

```
bool cmp2(point a,point b)
{
    point c;//原点
    c.x = 0;
    c.y = 0;
    if(compare(c,a,b)==0)//计算叉积，函数在上面有介绍，如果叉积相等，按照x从小到大排序
        return a.x<b.x;
    else return compare(c,a,b)>0;
}
```

第二种是用atan2()函数的：返回值是弧度

在头文件cmath里面，atan2(double y,double x)其中y表示已知点Y坐标，同理x,返回值是此点与远点连先与x轴正方向的夹角，可处理-180~180度。

```
bool cmp1(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;
}
```

时间：相较于计算叉积，利用atan2时间快，这个时间会快一点

精度： atan2精度不如叉积高

凸多边形

所有内角大小都在0~Pi范围内的简单多边形

凸包

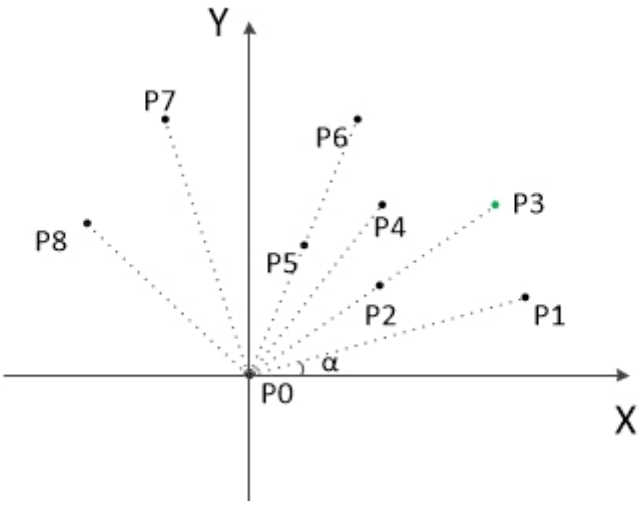
在平面上能包含所有给定点的最小凸多边形叫做凸包。

定义为，对于给定点集X，包含X种所有元素的交集S被称为X的凸包。

这里我们介绍两种方法求凸包：Graham动态扫描法和Andrew算法

first：Graham扫描法，复杂度是nlog(n)

先找到凸包上的一个点，然后从那个店开始按照逆时针方向逐个找图报上的点，实际上就是进行极角排序，然后对其查询使用。



步骤：

1.把所有点放在二维坐标系中，纵坐标最小的店一定是凸包上的点，如上图P0

2.把P0视作原点（平移）

3.计算各个点相对于P0的极角α，从小到大的顺序排序，当α相同的时候，ρ小的排在前面，排出P1到P8，显而易见P1和P8一定是凸包上的点。

由此我们已知凸包上的第一个和第二个点，我们用栈存储，现在按极角排序时候得到的P1后面那个点当做**当前点**，即为P2，接下来找第三个点：

4.链接**栈顶的前一个点**和**栈顶那个点**，得到直线L，看当前点是在直线L的左边还是右边，如果在直线的右边，则进行step5，左否则（无论是在线上还是在左边），就执行step6

5.在右边，说明栈顶那个元素不是凸包上的点，把栈顶元素弹出，返回step4（比如）

6.**当前点**是凸包上的点，把**当前点**压入栈，进入step7

7.检查当前的点P2书不是step3那个结果的最后一个元素，是的话就结束，不是的话就把P2后面点当做当前点，进入step4.

最后栈中的元素就是凸包上的点。

下面是个板板。。。

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<cmath>
#define PI 3.1415926535
using namespace std;
struct node
{
    int x,y;
};
node vex[1000];//存入的所有的点
node stackk[1000];//凸包中所有的点
int xx,yy;
bool cmp1(node a,node b)//排序找第一个点
{
    if(a.y==b.y)
        return a.x<b.x;
    else
        return a.y<b.y;
}
int cross(node a,node b,node c)//计算叉积
{
    return (b.x-a.x)*(c.y-a.y)-(c.x-a.x)*(b.y-a.y);
}
double dis(node a,node b)//计算距离
{
    return sqrt((a.x-b.x)*(a.x-b.x)*1.0+(a.y-b.y)*(a.y-b.y));
}
bool cmp2(node a,node b)//极角排序另一种方法，速度快
{
    if(atan2(a.y-yy,a.x-xx)!=atan2(b.y-yy,b.x-xx))
        return (atan2(a.y-yy,a.x-xx))<(atan2(b.y-yy,b.x-xx));
    return a.x<b.x;
}
bool cmp(node a,node b)//极角排序
{
    int m=cross(vex[0],a,b);
    if(m>0)
        return 1;
```

```
    else if(m==0&&dis(vex[0],a)-dis(vex[0],b)<=0)
        return 1;
    else return 0;
    /*if(m==0)
        return dis(vex[0],a)-dis(vex[0],b)<=0?true:false;
    else
        return m>0?true:false;*/
}
int main()
{
    int t,L;
    while(~scanf("%d",&t),t)
    {
        int i;
        for(i=0; i<t; i++)
        {
            scanf("%d%d",&vex[i].x,&vex[i].y);
        }
        if(t==1)
            printf("%.2f\n",0.00);
        else if(t==2)
            printf("%.2f\n",dis(vex[0],vex[1]));
        else
        {
            memset(stackk,0,sizeof(stackk));
            sort(vex,vex+t,cmp1);
            stackk[0]=vex[0];
            xx=stackk[0].x;
            yy=stackk[0].y;
            sort(vex+1,vex+t,cmp2);//cmp2是更快的，cmp更容易理解
            stackk[1]=vex[1];//将凸包中的第两个点存入凸包的结构体中
            int top=1;//最后凸包中拥有点的个数
            for(i=2; i<t; i++)
            {
                while(i>=1&&cross(stackk[top-1],stackk[top],vex[i])<0)    //对使用
极角排序的i>=1有时可以不用，但加上总是好的
                    top--;
                stackk[++top]=vex[i];                                     //控制<0
或<=0可以控制重点，共线的，具体视题目而定。
            }
            double s=0;
            //for(i=1; i<=top; i++)//输出凸包上的点
            //cout<<stackk[i].x<<" "<<stackk[i].y<<endl;
            for(i=1; i<=top; i++)   //计算凸包的周长
                s+=dis(stackk[i-1],stackk[i]);
            s+=dis(stackk[top],vex[0]);//最后一个点和第一个点之间的距离
            /*s+=2*PI*L;
            int ans=s+0.5;//四舍五入
            printf("%d\n",ans);*/
            printf("%.2lf\n",s);
        }
    }
}
```

下面介绍Andrew算法，复杂度为O（n）

是Graham的改进算法。

规则来说是按X从小到大进行排序，如果x相同按Y从小到大排序

显然排序后最小的元素和最大的元素一定在凸包上，因为是凸多边形，我们如果从一个点出发逆时针走，轨迹总是"左拐"的，一旦出现右拐，就说明这一段不在凸包上，所以我们可以简单的用一个单调栈来维护凸包的上下凸壳

预处理

```
// 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;
```

因为从左向右看，上下凸壳所旋转的方向不同，为了让单调栈起作用，我们首先升序枚举出下凸壳，然后降序求出上凸壳。找到最后一个点时，凸包的半个壳就出来了，再反过来判断一边，下半个壳也就出来了，这就是Andrew的核心步骤.

然后我们可以对P1,P2,.....Pn这n个有序点进行扫描，具体方法：

求凸壳时，一旦发现即将进栈的点（P）和栈顶的两个点（S1,S2，S1是栈顶）的正方向向右旋转，即叉积等于0，$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$，则弹出栈顶，回到上一步，继续检测，直到$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} \geq 0$；或者栈内只有一个元素为止。

通常情况下不需要保存位于凸包边上的点，因此上段中，$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$的符号<看情况可以改成<=，同时后面一个条件应改为>

```
//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*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;
```

最后凸包上面有ans个元素，（额外存储了1号点，h数组中有ans+1个元素），并且按照逆时针方向排序

周长为

$$\sum_{i=1}^{ans} \left| \overrightarrow{h_i h_{i+1}} \right|$$

后面这种比较常用。。。

旋转卡壳

求凸多边形直径：

我们将一个多边形上任意两个点间的距离的最大值定义为多边形的直径。

显然，确认一个凸多边形P直径的点不可能在P内部，所以搜索应该在边界上进行，事实上，由于直径是由多边形的平行切线的最远距离决定的，我们只需要查询对踵点。

这里有一个O（n）的算法。

算法描述

1. 计算多边形y方向的端点，我们称之为ymin和ymax
2. 通过yminymin和ymaxymax 构造两条水平切线。由于他们已经是一对对踵点，计算他们之间的距离并维护一个当前的最大值。
3. 同时旋转两条线直到其中一条与多边形的一边重合。
4. 一个新的对踵点对此时产生。计算新的距离，并和当前的最大值比较，大于当前最大值则更新。
5. 重复步骤3和步骤4的过程直到再次产生对踵点对。
6. 输出确定最大直径的对踵点对。

这个没整板子。。。。但也挺好理解的QAQ

张口就是老随机了

看似暴力，实际睿智

随机增量法

该算法时间复杂度低，应用范围大

增量法的思想和数学的归纳法类似，本质是将一个问题化为规模小一层的子问题，解决子问题后当做已知条件加入当前问题。

$$T(n) = T(n-1) + g(n)$$

增量法往往会结合随机化，以避免最坏情况的出现

最小圆覆盖：

在一个平面上有 n个点，求一个半径最小的圆，能覆盖所有的点。

假设圆O是前i-1个点的最小覆盖圆重复以上过程依次加入第j个点，若第j个点在圆外，则最小覆盖圆必经过第j个点，重复以上步骤（因为最多需要三个点来确定这个最小覆盖圆，所以要重复三次）

O（n）的时间复杂度

具体步骤：

1.将所有点随机排布（保证算法的复杂度）

2.初始随机找到两个点，设置为P1，P2，以P1,P2围巾直径得到初始圆，设为C2，Ci为包含 前i个点的最小圆

3.按照次序加点，设当前点为Pi，若Pi在当前圆Ci-1内，则Ci=Ci-1，否则进行step4

4.我们需要在新构造一个圆，显然Pi一定在新圆的边界上，所以我们可以先以P1Pi为直径简单的得到一个Ci

5.新找到的Ci不一定能包含1~i中所有的点，我们找到**不在**Ci中的一点Pj（j<i），那么PiPj一定在更新的圆的边界上，现在为止，我们能确定有两个点（Pi，Pj）在更新的圆的边界上，因此，我们可以以PiPj为直径构造出一个Cj

6.同样的，新得到的Cj不一定能包含1~j中所有的点，我们找不在Cj中的一点Pk（k<j<i）那么PiPjPk一定在更新的圆的边界上，现在我们能确定有三个点（Pi，Pj，Pk）一定在更新的圆的边界上。

（这里面的j和k都是遍历区间所有的）

因为三点确定一个圆，所以Pi，Pj，Pk，构成了新的圆，一定能覆盖前i个点

```
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>

using namespace std;

int n;
double r;

struct point {
  double x, y;
} p[100005], o;

inline double sqr(double x) { return x * x; }

inline double dis(point a, point b) {
  return sqrt(sqr(a.x - b.x) + sqr(a.y - b.y));
}
```

```
inline bool cmp(double a, double b) { return fabs(a - b) < 1e-8; }//相差在1e-8内

point geto(point a, point b, point c) {
  double a1, a2, b1, b2, c1, c2;
  point ans;
  a1 = 2 * (b.x - a.x), b1 = 2 * (b.y - a.y),
  c1 = sqr(b.x) - sqr(a.x) + sqr(b.y) - sqr(a.y);
  a2 = 2 * (c.x - a.x), b2 = 2 * (c.y - a.y),
  c2 = sqr(c.x) - sqr(a.x) + sqr(c.y) - sqr(a.y);
  if (cmp(a1, 0)) {
    ans.y = c1 / b1;
    ans.x = (c2 - ans.y * b2) / a2;
  } else if (cmp(b1, 0)) {
    ans.x = c1 / a1;
    ans.y = (c2 - ans.x * a2) / b2;
  } else {
    ans.x = (c2 * b1 - c1 * b2) / (a2 * b1 - a1 * b2);
    ans.y = (c2 * a1 - c1 * a2) / (b2 * a1 - b1 * a2);
  }
  return ans;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%lf%lf", &p[i].x, &p[i].y);
  for (int i = 1; i <= n; i++) swap(p[rand() % n + 1], p[rand() % n + 1]);//随机
打乱顺序
  o = p[1];
  for (int i = 1; i <= n; i++) {
    if (dis(o, p[i]) < r || cmp(dis(o, p[i]), r)) continue;//在圆内或者几乎在圆上
    //更新圆
    o.x = (p[i].x + p[1].x) / 2;
    o.y = (p[i].y + p[1].y) / 2;
    r = dis(p[i], p[1]) / 2;
    for (int j = 2; j < i; j++) {
      if (dis(o, p[j]) < r || cmp(dis(o, p[j]), r)) continue;
      o.x = (p[i].x + p[j].x) / 2;
      o.y = (p[i].y + p[j].y) / 2;
      r = dis(p[i], p[j]) / 2;
      for (int k = 1; k < j; k++) {
        if (dis(o, p[k]) < r || cmp(dis(o, p[k]), r)) continue;
        o = geto(p[i], p[j], p[k]);
        r = dis(o, p[i]);
      }
    }
  }
  printf("%.10lf\n%.10lf %.10lf", r, o.x, o.y);
  return 0;
}
```

时间复杂度分析：虽然目测是O（n^3）,实际上是O（n）的，第二个for循环中，第j个点有3/j的概率不在前j-1，这种情况下要进行一个O（j）的第三层循环，所以实际上的复杂度是O（1），如果Pj在园内，也是O（1），所以三层for循环复杂度一共是O（n）

补充：为啥是3/j?

先随机生成i个点，求出最小圆覆盖后，我们可以认为这个圆是由现在边界上的三个点来确定的，也就是说大明湖随机生成的最后一个点不是那三个点时，新生成的点就在圆内，否则在圆外，所以第j个点在前j-1个点最小圆覆盖外的概率是3/j.