目录

前置知识：

```
inline double R_to_D(double rad)//弧度转角度
{ return 180/Pi*rad; }
inline double D_to_R(double D)//角度转弧度
{ return Pi/180*D; }
```

弧长计算公式：
$L = n^\circ \times r \times \frac{\pi}{180^\circ}$，$L = \alpha \times r$。

其中n是圆心角度数（角度制），r是半径，L是圆心角弧长，α是圆心角度数（弧度制） 。

> 实际上就是有圆心角，转化成弧度制，然乘起来就是弧长。
>
> 因为一个圆的总周长：C=2πR 然后就是那一段弧长L所对圆心角α与2π的比值 ， 然后就可以推出来了，注意我们定义的 π 是弧度制的度数，$\pi = 180^\circ$，所以$n^\circ \times \frac{\pi}{180^\circ} = n^\circ \times 1$实际上就是一个表达形式的转换。

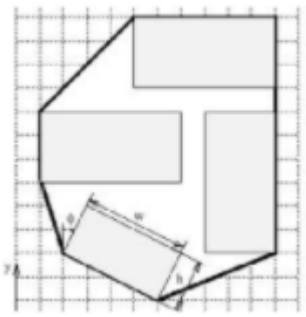
需要注意的是:点加减向量为点、点减点为向量

点加减向量为点：

点加减向量的运算实际上就是这个点，朝着这个向量的方向，移动这个向量的长度 ， 就得到了新的点的坐标

实例：

我们根据长方形的中心坐标求得长方形四个端点的坐标。



输入每行5个实数 x，y，w，h，j，其中（x，y）是木板（长方形）中心的坐标，w是宽，h是高，j是木板顺时针旋转的角度（j = 0表示不旋转，此时长度为w的那条边应该是水平方向）。

思路：

枚举四个方向（左上、右上、左下、右下），分别将对应的向量（$\pm\frac{w}{2}$，$\pm\frac{h}{2}$进行旋转输入的木板偏移的角度，然后加上原先点的坐标即可（w, h分别为宽和高（水平放置时横着的那条边和竖着的那条边））。

```
    Point o(x, y);//!
//要注意看题，这里给的角度j是顺时针的角度，然而我们的Rotate函数是取的逆时针的角度，所以一定要取相反数
    deg = - D_to_R(j);//角度转换成弧度，还要记得取相反数，注意这里是负的！！！
    p[tot ++ ] = o + Rotate(Vector(-w / 2, -h / 2), deg);
    p[tot ++ ] = o + Rotate(Vector(w / 2, -h / 2), deg);
    p[tot ++ ] = o + Rotate(Vector(-w / 2, h / 2), deg);
    p[tot ++ ] = o + Rotate(Vector(w / 2, h / 2), deg);
```

所以我如果想要得到一个新点的话，我们就可以通过点加上新位置的方向以及长度构成的向量（一个以原点为起点的向量）

1.基本运算

```
#include<bits/stdc++.h>
using namespace std;

const int N = 5007, M = 50007, INF = 0x3f3f3f3f;
const double DINF = 1e18, eps = 1e-8;
```

```cpp
struct Point{
    double x, y;
    Point(double x = 0, double y = 0):x(x), y(y){ }//构造函数
};

//!注意区分点和向量
typedef Point Vector;
//向量平移之后还是那个向量，所以只需要原点和向量的终点即可
//!向量 + 向量 = 向量，点 + 向量 = 向量
Vector operator + (Vector A, Vector B){return Vector(A.x + B.x, A.y + B.y);}
//!点 - 点 = 向量(向量BC = C - B)
Vector operator - (Point A, Point B){return Vector(A.x - B.x, A.y - B.y);}
//!向量 * 数 = 向量
Vector operator * (Vector A, double p){return Vector(A.x * p, A.y * p);}
//!向量 / 数= 向量
Vector operator / (Vector A, double p){return Vector(A.x / p, A.y / p);}

//!点/向量的比较函数
bool operator < (const Point& a, const Point& b) {return a.x < b.x || (a.x == b.x && a.y < b.y);}
//!求极角//在极坐标系中，平面上任何一点到极点的连线和极轴的夹角叫做极角。
//单位弧度rad
double Polar_angle(Vector A){return atan2(A.y, A.x);}
```

向量的数乘（除法也可以理解为数乘）$\frac{\vec{a}}{\lambda} = \frac{1}{\lambda}\vec{a} = (\frac{1}{\lambda}x, \frac{1}{\lambda}y)$

1.1 判断正负函数(sgn)

```cpp
//!三态函数sgn用于判断相等，减少精度误差问题
int sgn(double x){
    if(fabs(x) < eps)
        return 0;
    if(x < 0)
        return -1;
    return 1;
}

//重载等于运算符
bool operator == (const Point& a, const Point& b){return !sgn(a.x - b.x) && !sgn(a.y - b.y);}

```

传统意义	修正写法1	修正写法2
a==b	sgn(a-b) ==0	fabs(a -b) <eps
a!=b	sgn(a-b) !=0	fabs(a-b)>eps
a<b	sgn(a-b)<0	a-b<-eps
a<=b	sgn(a-b)<=0	a-b<eps
a>b	sgn(a-b)>0	a-b>eps
a>=b	sgn(a-b)>=0	a-b>-eps

1.2 点积（数量积、内积）(Dot)

$\alpha \cdot \beta = |\alpha||\beta|cos\theta$

对加法满足分配律(满足交换律)

```
//!点积(满足交换律)
double Dot(Vector A, Vector B){return A.x * B.x + A.y * B.y;}
```

夹角 θ 与点积大小的关系：

(1). 若 $\theta = 0^\circ, \vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}|$ 。

(2). 若 $\theta = 180^\circ, \vec{a} \cdot \vec{b} = -|\vec{a}||\vec{b}|$ 。

(3). 若 $\theta < 90^\circ, \vec{a} \cdot \vec{b} > 0$ 。

(4). 若 $\theta = 90^\circ, \vec{a} \cdot \vec{b} = 0$ 。

(5). 若 $\theta > 90^\circ, \vec{a} \cdot \vec{b} < 0$ 。

1.3 向量积，叉积(Cross)

$\alpha \times \beta = |\alpha||\beta| sin\theta$

θ表示向量α旋转到向量β所经过的夹角

对加法满足分配律(不满足交换律)

几何意义

向量α与β所张成的平行四边形的有向面积

判断外积的符号

右手定则

$\alpha \times \beta$

若β在α的逆时针方向，则为正值、顺时针则为负值、两向量共线则为0

```
//!向量的叉积(不满足交换律)
//等于两向量有向面积的二倍(从v的方向看,w在左边,叉积>0,w在右边,叉积<0,共线,叉积=0)
//cross(x, y) = -cross(y, x)
//cross(x, y) : xAyB - xByA
double Cross(Vector A, Vector B){return A.x *  B.y - B.x * A.y;}
```

1.3.1 判断向量bc是不是向ab的逆时针方向（左边）转(ToLeftTest)

也可以看作一个点c是否在向量ab的左边
凸包构造时将会频繁用到此公式

```
bool ToLeftTest(Point a, Point b, Point c){
    return Cross(b - a, c - b) > 0;
}
```

1.4 取模（模长，求长度）(Length)

```
double Length(Vector A){return sqrt(Dot(A, A));}
```

1.5 计算两向量夹角(Angle)

返回值为弧度制下的夹角

```cpp
double Angle(Vector A, Vector B){return acos(Dot(A, B) / Length(A) / Length(B));}
```

1.6 计算两向量构成的平行四边形有向面积(Area2)

```cpp
//!三个点确定两个向量,(交点为A的两个向量AB和AC)然后求这两个向量的叉积(叉乘)
double Area2(Point A, Point B, Point C){return Cross(B - A, C - A);}
```

1.7 计算向量逆时针旋转九十度之后的单位法线（法向量）(Normal)

```cpp
//!向量的单位法线实际上就是将该向量向左旋转90°
//因为是单位法线所以长度归一化调用前请确保A不是零向量
Vector Normal(Vector A) {
    double L = Length(A);
    return Vector(-A.y / L, A.x / L);
}
```

```cpp
inline Vector Format(const Vector &A) {
    double L = Length(A);
    return Vector(A.x / L,A.y / L);
}
```

1.8 计算向量逆时针旋转后的向量(Rotate)

旋转 θ 角时点的坐标变换
$x' = xcos\theta - ysin\theta$
$y' = xsin\theta + ycos\theta$

对于点 $P = (x, y)$ 或向量 $\vec{a} = (x, y)$，将其顺时针旋转 θ 角度（点：关于原点，向量：关于起点）：

$$\begin{vmatrix} x & y \end{vmatrix} \times \begin{vmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{vmatrix} = \begin{vmatrix} xcos\theta + ysin\theta & -xsin\theta + ycos\theta \end{vmatrix}$$

```cpp
//!一个向量旋转rad弧度之后的新向量（点也行）
//!x' = xcosa - ysina, y' = xsina + ycosa
//rad:弧度 且为逆时针旋转的角
Vector Rotate(Vector A, double rad){
    return Vector(A.x * cos(rad) - A.y * sin(rad), A.x * sin(rad) + A.y * cos(rad));
}
```

1.9 点绕着 p 点逆时针旋转 angle(rotate)

```cpp
//绕着 p 点逆时针旋转 angle
Point rotate(Point p,double angle){
    Point v = (*this) - p;
    double c = cos(angle), s = sin(angle);
    return Point(p.x + v.x*c - v.y*s,p.y + v.x*s + v.y*c);
 }
```

将点 $A(x, y)$ 绕点 $B(x_0, y_0)$ 顺时针旋转 θ 角度：

$$\begin{vmatrix} (x-x_0)cos\theta + (y-y_0)sin\theta + x_0 & -(x-x_0)sin\theta + (y-y_0)cos\theta + y_0 \end{vmatrix}$$

```cpp
inline Point turn_PP(Point a, Point b, double theta){//【将点A绕点B顺时针旋转theta(弧度)】
    double x = (a.x-b.x) * cos(theta) + (a.y-b.y) * sin(theta) + b.x;
    double y = - (a.x-b.x) * sin(theta) + (a.y-b.y) * cos(theta) + b.y;
    return Point(x,y);
}
```

1.10 复数表示

```cpp
#include <complex>
using namespace std;
typedef complex<double> Point;
typedef Point Vector;//复数定义向量后，自动拥有构造函数、加减法和数量积
const double eps = 1e-9;
int sgn(double x){
    if(fabs(x) < eps)
        return 0;
    if(x < 0)
        return -1;
    return 1;
}
double Length(Vector A){
    return abs(A);
}
double Dot(Vector A, Vector B){//conj(a+bi)返回共轭复数a-bi
    return real(conj(A)*B);
}
double Cross(Vector A, Vector B){
    return imag(conj(A)*B);
}
Vector Rotate(Vector A, double rad){
    return A*exp(Point(0, rad));//exp(p)返回以e为底复数的指数
}
```

2.点与线

- 一般式$ax+by+c=0$
- 点向式：直线上一点(x_0,y_0)和方向向量(u,v)即可使用
  $\frac{(x-x_0)}{u}=\frac{(y-y_0)}{v},(u\neq 0,v\neq 0)$
- 斜截式$y=kx+b$
  计算机中常用点向式表示直线，即参数方程形式表示

直线可以用直线上的一个点 P_0 和方向向量v表示

$P=P_0+vt$其中 t 为参数（可以通过限制参数来表示线段和射线，长度）

2.1 直线的实现(Line)

```cpp
struct Line{//直线定义
    Vector v;
    Point p;
    Line(Vector v, Point p):v(v), p(p) {}
    Point get_point_in_line(double t){//返回直线上一点P = v + (p - v)*t
        return v + (p - v)*t;
    }
};
```

2.2 判断点和直线关系(relation)

- 利用三点共线的等价条件$\alpha \times \beta = 0$
- 直线上取两不同点与待测点构成向量求叉积是否为零来判断点是否在直线上（叉积为0互相平行）

右手定则

```cpp
//点和直线关系
//1 在左侧
//-1 在右侧
//0 在直线上 //直线上两点s和e
//A, B:直线上一点,C:待判断关系的点
int relation(Point A, Point B, Point C)
{
    // 1 left -1 right 0 in
    int c = sgn(Cross((B - A), (C - A)));
    if(c < 0) return 1;
    else if(c > 0) return -1;
    return 0;
}
```

2.3 计算两直线交点(Get_line_intersection)

```cpp
//调用前要确保两直线p + tv 和 Q + tw之间有唯一交点，当且仅当Corss(v, w) ≠ 0;（t是参数）
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;
}

//!直线的参数式 直线 AB:A + tv(v为向量AB, t为参数)
```

2.4 计算点到直线的距离(Distance_point_to_line)

```cpp
double Distance_point_to_line(Point P, Point A, Point B)
{
    Vector v1 = B - A, v2 = P - A;
    return fabs(Cross(v1, v2) / Length(v1));//如果不取绝对值，那么得到的是有向距离
}
```

2.5 计算点到线段的距离(Distance_point_to_segment)

```cpp
//垂线距离或者PA或者PB距离
double Distance_point_to_segment(Point P, Point A, Point B)
{
    if(A == B) return Length(P - A);//（如果重合那么就是两个点之间的距离，直接转成向量求距离即可）
    Vector v1 = B - A, v2 = P - A, v3 = P - B;
    if(dcmp(Dot(v1, v2)) < 0) return Length(v2);//A点左边
    if(dcmp(Dot(v1, v3)) > 0)return Length(v3);//B点右边
    return fabs(Cross(v1, v2) / Length(v1));//垂线的距离
}
```

2.6 求点在直线上的投影点(Get_line_projection)

```cpp
//!点在直线上的投影
//点积满足分配率:两直线垂直，点积为0，向量v,Q = A + t0v,
//点P,v与直线PQ垂直:
//Dot(v, p - (A + t0v)) = 0 -> Dot(v, P - A) - t0 * Dot(v, v) = 0;
Point Get_line_projection(Point P, Point A, Point B)
{
    Vector v = B - A;
    return A + v * (Dot(v, P - A) / Dot(v, v));
}
```

2.7 计算点 P 到直线 AB 的垂足

~~垂足和投影好像是一个东西~~

```cpp
inline Point FootPoint(Point p, Point a, Point b){//【点P到直线AB的垂足】
    Vector x = p - a, y = p - b, z = b - a;
    double len1 = Dot(x, z) / Length(z), len2 = - 1.0 * Dot(y, z) / Length(z);//分别计算AP,BP在AB,BA上的投影
    return a + z * (len1 / (len1 + len2));//点A加上向量AF
}
```

2.8 计算点到直线的对称点

```cpp
inline Point Symmetry_PL(Point p, Point a, Point b){//【点P关于直线AB的对称点】
    return p + (FootPoint(p, a, b) - p) * 2;
    //实际上就是求垂足之后延长一倍得到的向量，与原来的点加起来就行了
}
```

2.8 判断点是否在线段上(OnSegment)

```cpp
bool OnSegment(Point p, Point a1, Point a2){
    return sgn(Cross(a1-p, a2-p)) == 0 && sgn(Dot(a1-p, a2-p)) < 0;
}
```

2.8 判断两线段是否相交（不算在端点处相交）(segment_proper_intersection)

```cpp
bool segment_proper_intersection(Point a1, Point a2, Point b1, Point b2)
{
    double c1 = Cross(a2 - a1, b1 - a1), c2 = Cross(a2 - a1, b2 - a1);
    double c3 = Cross(b2 - b1, a1 - b1), c4 = Cross(b2 - b1, a2 - b1);
    return dcmp(c1) * dcmp(c2) < 0 && dcmp(c3) * dcmp(c4) < 0;
    //如果算在端点处相交就改成 ≤ 即可，也可以用下面的那个长模板
}
```

2.9 判断两线段是否相交（包含端点处相交）(Segment_proper_intersection)

```cpp
bool Segment_proper_intersection(Point a1, Point a2, Point b1, Point b2){
    double c1 = Cross(a2-a1, b1-a1), c2 = Cross(a2-a1, b2-a1);
    double c3 = Cross(b2-b1, a1-b1), c4 = Cross(b2-b1, a2-b1);
    //if判断控制是否允许线段在端点处相交，根据需要添加
    if(!sgn(c1) || !sgn(c2) || !sgn(c3) || !sgn(c4)){
        bool f1 = OnSegment(b1, a1, a2);
        bool f2 = OnSegment(b2, a1, a2);
        bool f3 = OnSegment(a1, b1, b2);
        bool f4 = OnSegment(a2, b1, b2);
        bool f = (f1|f2|f3|f4);
        return f;
    }
    return (sgn(c1)*sgn(c2) < 0 && sgn(c3)*sgn(c4) < 0);
}
```

2.10 判断点是否在一条线段上(不含端点)(on_segment)

```cpp
bool on_segment(Point P, Point a1, Point a2)
{
    return dcmp(Cross(a1 - P, a2 - P)) == 0 && dcmp(Dot(a1 - P, a2 - P)) < 0;
}

```

2.11 两向量的关系(parallel)

```cpp
//直线v和直线Line(s, e);
bool parallel(Line v){
    return sgn((e-s)^(v.e-v.s)) == 0;
}
```

2.12 两直线关系(linecrossline)

```cpp
//两直线关系
//0 平行
//1 重合
//2 相交
int linecrossline(Line v){
    if((*this).parallel(v))
    return v.relation(s)==3;
    return 2;
}
```

2.13 求两线段最小距离的平方

> 题目大意：给两条线段求他们间的最小距离的平方（以分数形式输出）。

```cpp
/*LA 4973异面线段*/
#include<cstdio>
#include<cmath>
#include<algorithm>
using namespace std;

struct Point3 {
  int x, y, z;
  Point3(int x=0, int y=0, int z=0):x(x),y(y),z(z) { }
};

typedef Point3 Vector3;

Vector3 operator + (const Vector3& A, const Vector3& B) { return Vector3(A.x+B.x, A.y+B.y, A.z+B.z); }
Vector3 operator - (const Point3& A, const Point3& B) { return Vector3(A.x-B.x, A.y-B.y, A.z-B.z); }
Vector3 operator * (const Vector3& A, int p) { return Vector3(A.x*p, A.y*p, A.z*p); }

bool operator == (const Point3& a, const Point3& b) {
  return a.x==b.x && a.y==b.y && a.z==b.z;
}

Point3 read_point3() {
  Point3 p;
  scanf("%d%d%d", &p.x, &p.y, &p.z);
  return p;
}

int Dot(const Vector3& A, const Vector3& B) { return A.x*B.x + A.y*B.y + A.z*B.z; }
int Length2(const Vector3& A) { return Dot(A, A); }
Vector3 Cross(const Vector3& A, const Vector3& B) { return Vector3(A.y*B.z - A.z*B.y, A.z*B.x - A.x*B.z, A.x*B.y - A.y*B.x); }

typedef long long LL;

LL gcd(LL a, LL b) { return b ? gcd(b, a%b) : a; }
LL lcm(LL a, LL b) { return a / gcd(a,b) * b; }

struct Rat {
  LL a, b;
  Rat(LL a=0):a(a),b(1) { }
  Rat(LL x, LL y):a(x),b(y) {
    if(b < 0) a = -a, b = -b;
    LL d = gcd(a, b); if(d < 0) d = -d;
    a /= d; b /= d;
  }
};

Rat operator + (const Rat& A, const Rat& B) {
  LL x = lcm(A.b, B.b);
  return Rat(A.a*(x/A.b)+B.a*(x/B.b), x);
```

```
}

Rat operator - (const Rat& A, const Rat& B) { return A + Rat(-B.a, B.b); }
Rat operator * (const Rat& A, const Rat& B) { return Rat(A.a*B.a, A.b*B.b); }

void updatemin(Rat& A, const Rat& B) {
  if(A.a*B.b > B.a*A.b) A.a = B.a, A.b = B.b;
}

// 点P到线段AB的距离的平方
Rat Rat_Distance2ToSegment(const Point3& P, const Point3& A, const Point3& B) {
  if(A == B) return Length2(P-A);
  Vector3 v1 = B - A, v2 = P - A, v3 = P - B;
  if(Dot(v1, v2) < 0) return Length2(v2);
  else if(Dot(v1, v3) > 0) return Length2(v3);
  else return Rat(Length2(Cross(v1, v2)), Length2(v1));
}

// 求异面直线p1+su和p2+tv的公垂线对应的s。如果平行/重合，返回false
bool Rat_LineDistance3D(const Point3& p1, const Vector3& u, const Point3& p2, const Vector3& v, Rat& s) {
  LL b = (LL)Dot(u,u)*Dot(v,v) - (LL)Dot(u,v)*Dot(u,v);
  if(b == 0) return false;
  LL a = (LL)Dot(u,v)*Dot(v,p1-p2) - (LL)Dot(v,v)*Dot(u,p1-p2);
  s = Rat(a, b);
  return true;
}

void Rat_GetPointOnLine(const Point3& A, const Point3& B, const Rat& t, Rat& x, Rat& y, Rat& z) {
  x = Rat(A.x) + Rat(B.x-A.x) * t;
  y = Rat(A.y) + Rat(B.y-A.y) * t;
  z = Rat(A.z) + Rat(B.z-A.z) * t;
}

Rat Rat_Distance2(const Rat& x1, const Rat& y1, const Rat& z1, const Rat& x2, const Rat& y2, const Rat& z2) {
  return (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)+(z1-z2)*(z1-z2);
}

int main() {
  int T;
  scanf("%d", &T);
  LL maxx = 0;
  while(T--) {
    Point3 A = read_point3();
    Point3 B = read_point3();
    Point3 C = read_point3();
    Point3 D = read_point3();
    Rat s, t;
    bool ok = false;
    Rat ans = Rat(1000000000);
    if(Rat_LineDistance3D(A, B-A, C, D-C, s))
      if(s.a > 0 && s.a < s.b && Rat_LineDistance3D(C, D-C, A, B-A, t))
        if(t.a > 0 && t.a < t.b) {
```

```
            ok = true; // 异面直线/相交直线
            Rat x1, y1, z1, x2, y2, z2;
            Rat_GetPointOnLine(A, B, s, x1, y1, z1);
            Rat_GetPointOnLine(C, D, t, x2, y2, z2);
            ans = Rat_Distance2(x1, y1, z1, x2, y2, z2);
        }
        if(!ok) { // 平行直线/重合直线
          updatemin(ans, Rat_Distance2ToSegment(A, C, D));
          updatemin(ans, Rat_Distance2ToSegment(B, C, D));
          updatemin(ans, Rat_Distance2ToSegment(C, A, B));
          updatemin(ans, Rat_Distance2ToSegment(D, A, B));
        }
        printf("%lld %lld\n", ans.a, ans.b);
    }
    return 0;
}
```

3.多边形

普通多边形

通常按照 **逆时针** 储存所有顶点

定义

多边形

由在同一平面且不在同一直线上的多条线段首位顺次连接且不相交所组成的图形叫做多边形

简单多边形

简单多边形是除相邻边外其它边不相交的多边形

凸多边形

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（所有的正多边形都是凸多边形，所有的三角形都是凸多边形）

任意凸多边形外角和均为360°

任意凸多边形内角和为(n−2) * 180°

3.0.三角形

三角形面积

- 利用两条边叉积除以二取绝对值
- 海伦公式

$$S=\sqrt{p(p-a)(p-b)(p-c)}, p=\frac{(a+b+c)}{2}$$

- $S=\frac{absinC}{2}$

3.0.1 三角形四心

- **外心**：三边中垂线交点，到三角形三个顶点距离相同（外接圆圆心）
- **内心**：角平分线的交点，到三角形三边的距离相同（内切圆圆心）
- **垂心**：三条垂线的交点
- **重心**：三条中线的交点，到三角形三顶点距离的平方和最小的点，三角形内到三边距离之积最大的点

三角形重心

三点各坐标的平均值$(\frac{x1+x2+x3}{3},\frac{y1+y2+y3}{3})$

3.0.2向量求三角形垂心(getcircle)

```
inline Circle getcircle(Point A,Point B,Point C){//【三点确定一圆】向量垂心法
    Point P1=(A+B)*0.5,P2=(A+C)*0.5;
    Point O=cross_LL(P1,P1+Normal(B-A),P2,P2+Normal(C-A));
    return Circle(O,Len(A-O));
}
```

3.1.0 正多边形的一些性质和概念

外接圆

把圆分为n(n≥3)等份，依次连接各分点所得的多边形就是这个圆的内接正n边形，也就是正n边形的外接圆。

内切圆

把圆分为m(m≥3)等份，经过各分点作圆的切线，以相邻切线的交点为顶点的多边形就是这个圆的外切正m边形，也就是正m边形的内切圆。

内角

正n边形的内角和度数为：（n - 2)×180°;
正n边形的一个内角是(n-2)×180°÷n.
正n边形内固定一个边，按角度从小到大遍历所有点，每个角的角度为(180.0-deg)/2*(i-2)，deg是一个内角，i是所选点的标号，从1~n顺时针编号可由内角公式推出

外角

正n边形外角和等于n·180° - (n - 2)·180°=360°
所以正n边形的一个外角为：360°÷n.
所以正n边形的一个内角也可以用这个公式：180°-360°÷n.

中心角

任何一个正多边形，都可作一个外接圆，多边形的中心就是所作外接圆的圆心，所以每条边的中心角，实际上就是这条边所对的弧的圆心角，因此这个角就是360度÷边数。
正多边形中心角：360°÷n
因此可证明，正n边形中，外角=中心角=360°÷n对角线
在一个正多边形中，所有的顶点可以与除了他相邻的两个顶点的其他顶点连线，就成了顶点数减2（2是那两个相邻的点）个三角形。三角形内角和：180度，所以把边数减2乘上180度，就是这个正多边形的内角和。对角线数量的计算公式：n(n-3)÷2。

面积

设正n边形的半径为R，边长为an，中心角为αn，边心距为rn，则αn=360°÷n，an=2Rsin(180°÷n)，rn=Rcos(180°÷n)，R^2=r n2+(an÷2)2，周长pn=n×an，面积Sn=pn×rn÷2。

对称轴

正多边形的对称轴——
奇数边：连接一个顶点和顶点所对的边的中点的线段所在的直线，即为对称轴；
偶数边：连接相对的两个边的中点，或者连接相对称的两个顶点的线段所在的直线，都是对称轴。
正N边形边数、角数、对称轴数都为N。

3.1 求多边形面积(convex_polygon_area)

我们可以从第一个顶点除法把凸多边形分成n-2个三角形(三角剖分)，然后把面积加起来，最后返回值说为有向面积更贴近本质

```cpp
//!求凸多边形的有向面积
//叉积的几何意义就是三角形有向面积的二倍，所以这里要除以二
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;
    //return fabs(area / 2);//不加的话求的是有向面积，逆时针为负，顺时针为正
}
//!求非凸多边形的有向面积
//我们叉积求得的三角形面积是有向的，在外面的面积可以正负抵消掉，
//因此非凸多边形也适用，可以从任意点出发划分
//可以取原点为起点，减少叉乘次数
double polyg_on_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;
}
```

3.2 判断点在多边形内(is_point_in_polygon)

有射线法与转角法。
转角法的基本思想是看多边形相对于这个点转了多少度
如果是三百六十度，说明点在多边形内
如果是零度，说明点在多边形外
如果是一百八十度，说明点在多边形边界上
如果直接按照定义来算，则需要计算大量反三角函数，不仅速度慢，而且容易产生精度问题
因此我们采用winding number绕数来计算

```cpp
//判断点是否在多边形内，若点在多边形内返回1，在多边形外部返回0，在多边形上返回-1
int is_point_in_polygon(Point p, vector<Point> poly){//待判断的点和该多边形的所有点的合集
    int wn = 0;
    int n = poly.size();
    for(int i = 0; i < n; ++i){
        if(OnSegment(p, poly[i], poly[(i+1)%n])) return -1;
        int k = sgn(Cross(poly[(i+1)%n] - poly[i], p - poly[i]));
        int d1 = sgn(poly[i].y - p.y);
        int d2 = sgn(poly[(i+1)%n].y - p.y);
        if(k > 0 && d1 ≤ 0 && d2 > 0) wn++;
        if(k < 0 && d2 ≤ 0 && d1 > 0) wn--;
    }
    if(wn ≠ 0)
        return 1;
    return 0;
}
```

3.3 判断点在凸多边形内

只需要判断点是否在所有边的左边（按逆时针顺序排列的顶点集）可以使用ToLeftTest，$O(n)$

判断折线（向量）bc是不是向（向量）ab的逆时针方向（左边）转向(ToLeftTest)

凸包构造时将会频繁用到此公式

```
bool ToLeftTest(Point a, Point b, Point c){
    return Cross(b - a, c - b) > 0;
}
```

3.4 求多边形重心(barycenter)

```
point barycenter(int n,point* p){
    point ret,t;
    double t1=0,t2;
    int i;
    ret.x=ret.y=0;
    for (i=1;i<n-1;i++)
        if (fabs(t2=xmult(p[0],p[i],p[i+1]))>eps){
            t=barycenter(p[0],p[i],p[i+1]);
            ret.x+=t.x*t2;
            ret.y+=t.y*t2;
            t1+=t2;
        }
    if (fabs(t1)>eps)
        ret.x/=t1,ret.y/=t1;
    return ret;
}
```

3.5 判定凸多边形(is_convex)

3.6 判点在凸多边形内或多边形边上(inside_convex)

3.7 判点在任意多边形内(inside_polygon)

3.8 判线段在任意多边形内(inside_polygon)

```
#include <stdlib.h>
#include <math.h>
#define MAXN 1000
#define offset 10000
#define eps 1e-8
#define zero(x) (((x)>0?(x):-(x))<eps)
#define _sign(x) ((x)>eps?1:((x)<-eps?2:0))
struct point{double x,y;};
struct line{point a,b;};


double xmult(point p1,point p2,point p0){
    return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}


//判定凸多边形,顶点按顺时针或逆时针给出,允许相邻边共线
int is_convex(int n,point* p){
```

```
    int i,s[3]={1,1,1};
    for (i=0;i<n&&s[1]|s[2];i++)
        s[_sign(xmult(p[(i+1)%n],p[(i+2)%n],p[i]))]=0;
    return s[1]|s[2];
}


//判定凸多边形,顶点按顺时针或逆时针给出,不允许相邻边共线
int is_convex_v2(int n,point* p){
    int i,s[3]={1,1,1};
    for (i=0;i<n&&s[0]&&s[1]|s[2];i++)
        s[_sign(xmult(p[(i+1)%n],p[(i+2)%n],p[i]))]=0;
    return s[0]&&s[1]|s[2];
}


//判点在凸多边形内或多边形边上,顶点按顺时针或逆时针给出
int inside_convex(point q,int n,point* p){
    int i,s[3]={1,1,1};
    for (i=0;i<n&&s[1]|s[2];i++)
        s[_sign(xmult(p[(i+1)%n],q,p[i]))]=0;
    return s[1]|s[2];
}


//判点在凸多边形内,顶点按顺时针或逆时针给出,在多边形边上返回0
int inside_convex_v2(point q,int n,point* p){
    int i,s[3]={1,1,1};
    for (i=0;i<n&&s[0]&&s[1]|s[2];i++)
        s[_sign(xmult(p[(i+1)%n],q,p[i]))]=0;
    return s[0]&&s[1]|s[2];
}


//判点在任意多边形内,顶点按顺时针或逆时针给出
//on_edge表示点在多边形边上时的返回值,offset为多边形坐标上限
int inside_polygon(point q,int n,point* p,int on_edge=1){
    point q2;
    int i=0,count;
    while (i<n)
        for (count=i=0,q2.x=rand()+offset,q2.y=rand()+offset;i<n;i++)
            if (zero(xmult(q,p[i],p[(i+1)%n]))&&(p[i].x-q.x)*(p[(i+1)%n].x-q.x)<eps&&(p[i].y-
q.y)*(p[(i+1)%n].y-q.y)<eps)
                return on_edge;
            else if (zero(xmult(q,q2,p[i])))
                break;
            else if (xmult(q,p[i],q2)*xmult(q,p[(i+1)%n],q2)←
eps&&xmult(p[i],q,p[(i+1)%n])*xmult(p[i],q2,p[(i+1)%n])←eps)
                count++;
    return count&1;
}


inline int opposite_side(point p1,point p2,point l1,point l2){
    return xmult(l1,p1,l2)*xmult(l1,p2,l2)←eps;
```

```
}


inline int dot_online_in(point p,point l1,point l2){
    return zero(xmult(p,l1,l2))&&(l1.x-p.x)*(l2.x-p.x)<eps&&(l1.y-p.y)*(l2.y-p.y)<eps;
}


//判线段在任意多边形内,顶点按顺时针或逆时针给出,与边界相交返回1
int inside_polygon(point l1,point l2,int n,point* p){
    point t[MAXN],tt;
    int i,j,k=0;
    if (!inside_polygon(l1,n,p)||!inside_polygon(l2,n,p))
        return 0;
    for (i=0;i<n;i++)
        if (opposite_side(l1,l2,p[i],p[(i+1)%n])&&opposite_side(p[i],p[(i+1)%n],l1,l2))
            return 0;
        else if (dot_online_in(l1,p[i],p[(i+1)%n]))
            t[k++]=l1;
        else if (dot_online_in(l2,p[i],p[(i+1)%n]))
            t[k++]=l2;
        else if (dot_online_in(p[i],l1,l2))
            t[k++]=p[i];
    for (i=0;i<k;i++)
        for (j=i+1;j<k;j++){
            tt.x=(t[i].x+t[j].x)/2;
            tt.y=(t[i].y+t[j].y)/2;
            if (!inside_polygon(tt,n,p))
                return 0;
        }
    return 1;
}


point intersection(line u,line v){
    point ret=u.a;
    double t=((u.a.x-v.a.x)*(v.a.y-v.b.y)-(u.a.y-v.a.y)*(v.a.x-v.b.x))
            /((u.a.x-u.b.x)*(v.a.y-v.b.y)-(u.a.y-u.b.y)*(v.a.x-v.b.x));
    ret.x+=(u.b.x-u.a.x)*t;
    ret.y+=(u.b.y-u.a.y)*t;
    return ret;
}


point barycenter(point a,point b,point c){
    line u,v;
    u.a.x=(a.x+b.x)/2;
    u.a.y=(a.y+b.y)/2;
    u.b=c;
    v.a.x=(a.x+c.x)/2;
    v.a.y=(a.y+c.y)/2;
    v.b=b;
    return intersection(u,v);
}

```

```
127
128 //多边形重心
129 point barycenter(int n,point* p){
130     point ret,t;
131     double t1=0,t2;
132     int i;
133     ret.x=ret.y=0;
134     for (i=1;i<n-1;i++)
135         if (fabs(t2=xmult(p[0],p[i],p[i+1]))>eps){
136             t=barycenter(p[0],p[i],p[i+1]);
137             ret.x+=t.x*t2;
138             ret.y+=t.y*t2;
139             t1+=t2;
140         }
141     if (fabs(t1)>eps)
142         ret.x/=t1,ret.y/=t1;
143     return ret;
144 }
```

3.9 多边形切割(常用于半平面交)

```
1  //多边形切割
2  //可用于半平面交
3  #define MAXN 100
4  #define eps 1e-8
5  #define zero(x) (((x)>0?(x):-(x))<eps)
6  struct point{double x,y;};
7
8
9  double xmult(point p1,point p2,point p0){
10     return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
11 }
12
13
14 int same_side(point p1,point p2,point l1,point l2){
15     return xmult(l1,p1,l2)*xmult(l1,p2,l2)>eps;
16 }
17
18
19 point intersection(point u1,point u2,point v1,point v2){
20     point ret=u1;
21     double t=((u1.x-v1.x)*(v1.y-v2.y)-(u1.y-v1.y)*(v1.x-v2.x))
22              /((u1.x-u2.x)*(v1.y-v2.y)-(u1.y-u2.y)*(v1.x-v2.x));
23     ret.x+=(u2.x-u1.x)*t;
24     ret.y+=(u2.y-u1.y)*t;
25     return ret;
26 }
27
28
29 //将多边形沿l1,l2确定的直线切割在side侧切割,保证l1,l2,side不共线
30 void polygon_cut(int& n,point* p,point l1,point l2,point side){
31     point pp[MAXN];
32     int m=0,i;
33     for (i=0;i<n;i++){
34         if (same_side(p[i],side,l1,l2))
```

```
            pp[m++]=p[i];
        if (!same_side(p[i],p[(i+1)%n],l1,l2)&&!
(zero(xmult(p[i],l1,l2))&&zero(xmult(p[(i+1)%n],l1,l2))))
            pp[m++]=intersection(p[i],p[(i+1)%n],l1,l2);
    }
    for (n=i=0;i<m;i++)
        if (!i||!zero(pp[i].x-pp[i-1].x)||!zero(pp[i].y-pp[i-1].y))
            p[n++]=pp[i];
    if (zero(p[n-1].x-p[0].x)&&zero(p[n-1].y-p[0].y))
        n--;
    if (n<3)
        n=0;
}
```

3.10 判断四边形类型

判定方法如下：

平行四边形： 对角线互相平分或者用定义。

梯形： 有一组对边平行，一组对边不平行。

在已知平行四边形的情况下

矩形： 对角线长度相等或用定义。

菱形： 对角线互相垂直（数量积为0）或用定义。

正方形： 即是矩形又是菱形。

题目只是给你几个点的坐标，并不知道那些点能组成对角线。所以枚举组合加上两线段是否相交的判定即可。

找到那两条对角线后，就可以随便乱搞用上面的判定了。

4.圆

圆的问题一般直接联立解方程即可。

计算机中储存圆通常记录圆心坐标与半径即可

弧长公式：$L=\alpha\times r$，弧长等于半径*圆心角

定义

```
struct Circle
{
    Point c;
    double r;
    Circle(Point c=Point(),double r=0):c(c),r(r){}
    inline Point point(double a)//通过圆心角求坐标
    { return Point(c.x+cos(a)*r,c.y+sin(a)*r); }
};
inline Circle read_circle()
{
    Circle C;
    scanf("%lf%lf%lf",&C.c.x,&C.c.y,&C.r);
    return C;
}
```

4.1 圆与直线交点(getLineCircleIntersection)

```cpp
//求圆与直线交点
int getLineCircleIntersection(Line L, Circle C, double& t1, double& t2, vector<Point>& sol){
    double a = L.v.x, b = L.p.x - C.c.x, c = L.v.y, d = L.p.y - C.c.y;
    double e = a*a + c*c, f = 2*(a*b + c*d), g = b*b + d*d - C.r*C.r;
    double delta = f*f - 4*e*g;//判别式
    if(sgn(delta) < 0)//相离
        return 0;
    if(sgn(delta) == 0){//相切
        t1 = -f /(2*e);
        t2 = -f /(2*e);
        sol.push_back(L.point(t1));//sol存放交点本身
        return 1;
    }
    //相交
    t1 = (-f - sqrt(delta))/(2*e);
    sol.push_back(L.point(t1));
    t2 = (-f + sqrt(delta))/(2*e);
    sol.push_back(L.point(t2));
    return 2;
}
```

4.2 求两圆交点(get_circle_circle_intersection)

```cpp
//两圆相交保存所有交点返回交点个数（至多两个）
int get_circle_circle_intersection(Circle c1, Circle c2, vector<Point>& sol)
{
    double d = Length(c1.c - c2.c);
    if(dcmp(d) == 0){
        if(dcmp(c1.r - c2.r) == 0)return -1;//两圆重合
        return 0;
    }
    if(dcmp(c1.r + c2.r - d) < 0)return 0;//相离
    if(dcmp(fabs(c1.r - c2.r) - d) > 0)return 0;//在另一个圆的内部

    double a = angle(c2.c - c1.c);//向量c1c2的极角
    double da = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2 * c1.r * d));
    //c1c2到c1p1的角
    Point p1 = c1.point(a - da), p2 = c1.point(a + da);
    sol.push_back(p1);
    if(p1 == p2)return 1;
    sol.push_back(p2);
    return 2;
}

```

4.3 点到圆的切线(get_tangents)

```cpp

//过点p到圆c的切线，v[i]是第i条切线， 返回切线的条数
int get_tangents(Point p, Circle C, Vector* v){
    Vector u = C.c - p;
    double dist = Length(u);
```

```
    if(dist < C.r)return 0;//点在内部，没有切线
    else if(dcmp(dist - C.r) == 0){//p在圆上，只有一条切线
        v[0] = Rotate(u, PI / 2);//切线就是垂直嘛
        return 1;
    }
    else {//否则是两条切线
        double ang = asin(C.r / dist);
        v[0] = Rotate(u, - ang);
        v[1] = Rotate(u, +ang);
        return 2;
    }
}
```

4.4 两圆的公切线(get_tangents)

两圆的公切线。
根据两圆的圆心距从小到大排列，一共有6种情况。
情况一：两圆完全重合。有无数条公切线。
情况二：两圆内含，没有公共点。没有公切线。
情况三：两圆内切。有1条外公切线。
情况四：两圆相交。有2条外公切线。
情况五：两圆外切。有3条公切线，其中一条内公切线，两条外公切线。
情况六：两圆相离。有4条公切线，其中内公切线两条，外公切线两条。
可以根据圆心距和半径的关系辨别出这6种情况，然后逐一求解。
情况一和情况二没什么需要求的，情况三和情况五中的内公切线都对应于“过圆上一点求圆的切线”，只需连接圆心和切点，旋转90°后即可知道切线的方向向量。这样，问题的关键是求出情况四、五中的外公切线和情况六中的内公切线。

```
//返回切线的条数，-1表示无穷条切线
//a[i]和b[i] 分别是第i条切线在圆A和圆B上的切点
int get_tangents(Circle A, Circle B, Point* a, Point* b)
{
    int cnt = 0;
    if(A.r < B.r)swap(A, B), swap(a, b);
    int d2 = (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y);
    int rdiff = A.r - B.r;
    int rsum = A.r + B.r;
    if(d2 < rdiff * rdiff)return 0;//内含
    double base = atan2(B.y - A.y, B.x - A.x);
    if(d2 == 0 && A.r == B.r)return -1;//无限多条切线
    if(d2 == rdiff * rdiff){//内切，1条切线
        a[cnt] = A.point(base);
        b[cnt] = B.point(base); cnt ++ ;
        return 1;
    }
    //有外共切线
    double ang = acos((A.r - B.r) / sqrt(d2));
    a[cnt] = A.point(base + ang);b[cnt] = B.point(base + ang);cnt ++;
    a[cnt] = A.point(base - ang);b[cnt] = B.point(base - ang);cnt ++;
    if(d2 == rsum * rsum){//一条内公切线
        a[cnt] = A.point(base);b[cnt] = B.point(PI + base); cnt ++ ;
    }
    else if(d2 > rsum * rsum){//两条内公切线
        double ang = acos((A.r + B.r) / sqrt(d2));
```

```
        a[cnt] = A.point(base + ang); b[cnt] = B.point(PI + base + ang); cnt ++ ;
        a[cnt] = A.point(base - ang); b[cnt] = B.point(PI + base - ang); cnt ++ ;
    }
    return cnt;
}
```

4.5 两圆相交面积(AreaOfOverlap)

通过计算两个圆相交所构成的两个扇形面积和减去其构成的筝形的面积

```
double AreaOfOverlap(Point c1, double r1, Point c2, double r2){
    double d = Length(c1 - c2);
    if(r1 + r2 < d + eps)
        return 0.0;
    if(d < fabs(r1 - r2) + eps){
        double r = min(r1, r2);
        return pi*r*r;
    }
    double x = (d*d + r1*r1 - r2*r2)/(2.0*d);
    double p = (r1 + r2 + d)/2.0;
    double t1 = acos(x/r1);
    double t2 = acos((d - x)/r2);
    double s1 = r1*r1*t1;
    double s2 = r2*r2*t2;
    double s3 = 2*sqrt(p*(p - r1)*(p - r2)*(p - d));
    return s1 + s2 - s3;
}
```

4.6 模板合集

```
#include <math.h>
#define eps 1e-8
struct point{double x,y;};

double xmult(point p1,point p2,point p0){
    return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}

double distance(point p1,point p2){
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}

double disptoline(point p,point l1,point l2){
    return fabs(xmult(p,l1,l2))/distance(l1,l2);
}

point intersection(point u1,point u2,point v1,point v2){
    point ret=u1;
    double t=((u1.x-v1.x)*(v1.y-v2.y)-(u1.y-v1.y)*(v1.x-v2.x))
            /((u1.x-u2.x)*(v1.y-v2.y)-(u1.y-u2.y)*(v1.x-v2.x));
    ret.x+=(u2.x-u1.x)*t;
    ret.y+=(u2.y-u1.y)*t;
    return ret;
}
```

4.7 判直线和圆相交(intersect_line_circle)

```
//判直线和圆相交,包括相切
int intersect_line_circle(point c,double r,point l1,point l2){
    return disptoline(c,l1,l2)<r+eps;
}
```

4.8 判线段和圆相交(intersect_seg_circle)

```
//判线段和圆相交,包括端点和相切
int intersect_seg_circle(point c,double r,point l1,point l2){
    double t1=distance(c,l1)-r,t2=distance(c,l2)-r;
    point t=c;
    if (t1<eps||t2<eps)
        return t1>-eps||t2>-eps;
    t.x+=l1.y-l2.y;
    t.y+=l2.x-l1.x;
    return xmult(l1,c,t)*xmult(l2,c,t)<eps&&disptoline(c,l1,l2)-r<eps;
}
```

4.9 判圆和圆相交(intersect_circle_circle)

```
//判圆和圆相交,包括相切
int intersect_circle_circle(point c1,double r1,point c2,double r2){
    return distance(c1,c2)<r1+r2+eps&&distance(c1,c2)>fabs(r1-r2)-eps;
}

```

4.10 计算圆上到点p最近点(dot_to_circle)

```
//计算圆上到点p最近点,如p与圆心重合,返回p本身
point dot_to_circle(point c,double r,point p){
    point u,v;
    if (distance(p,c)<eps)
        return p;
    u.x=c.x+r*fabs(c.x-p.x)/distance(c,p);
    u.y=c.y+r*fabs(c.y-p.y)/distance(c,p)*((c.x-p.x)*(c.y-p.y)<0?-1:1);
    v.x=c.x-r*fabs(c.x-p.x)/distance(c,p);
    v.y=c.y-r*fabs(c.y-p.y)/distance(c,p)*((c.x-p.x)*(c.y-p.y)<0?-1:1);
    return distance(u,p)<distance(v,p)?u:v;
}

```

4.11 计算直线/线段与圆的交点（intersection_line_circle）

计算线段与圆的交点可用这个函数求得交点之后再判点是否在线段上

```
//计算直线与圆的交点,保证直线与圆有交点
void intersection_line_circle(point c,double r,point l1,point l2,point& p1,point& p2){
    point p=c;
    double t;
    p.x+=l1.y-l2.y;
    p.y+=l2.x-l1.x;
```

```
    p=intersection(p,c,l1,l2);
    t=sqrt(r*r-distance(p,c)*distance(p,c))/distance(l1,l2);
    p1.x=p.x+(l2.x-l1.x)*t;
    p1.y=p.y+(l2.y-l1.y)*t;
    p2.x=p.x-(l2.x-l1.x)*t;
    p2.y=p.y-(l2.y-l1.y)*t;
}

```

4.12 计算圆与圆的交点（intersection_circle_circle）

```
//计算圆与圆的交点,保证圆与圆有交点,圆心不重合
void intersection_circle_circle(point c1,double r1,point c2,double r2,point& p1,point& p2){
    point u,v;
    double t;
    t=(1+(r1*r1-r2*r2)/distance(c1,c2)/distance(c1,c2))/2;
    u.x=c1.x+(c2.x-c1.x)*t;
    u.y=c1.y+(c2.y-c1.y)*t;
    v.x=u.x+c1.y-c2.y;
    v.y=u.y-c1.x+c2.x;
    intersection_line_circle(c1,r1,u,v,p1,p2);
}


//将向量p逆时针旋转angle角度
Point Rotate(Point p,double angle) {
    Point res;
    res.x=p.x*cos(angle)-p.y*sin(angle);
    res.y=p.x*sin(angle)+p.y*cos(angle);
    return res;
}
```

4.13 求圆外一点对圆的两个切点(TangentPoint_PC)

```
//求圆外一点对圆(o,r)的两个切点result1和result2
void TangentPoint_PC(Point poi,Point o,double r,Point &result1,Point &result2) {
    double line=sqrt((poi.x-o.x)*(poi.x-o.x)+(poi.y-o.y)*(poi.y-o.y));
    double angle=acos(r/line);
    Point unitvector,lin;
    lin.x=poi.x-o.x;
    lin.y=poi.y-o.y;
    unitvector.x=lin.x/sqrt(lin.x*lin.x+lin.y*lin.y)*r;
    unitvector.y=lin.y/sqrt(lin.x*lin.x+lin.y*lin.y)*r;
    result1=Rotate(unitvector,-angle);
    result2=Rotate(unitvector,angle);
    result1.x+=o.x;
    result1.y+=o.y;
    result2.x+=o.x;
    result2.y+=o.y;
    return;
}
```

4.14 求三角形的外接圆(get_circumcircle)

```cpp
//get_circumcircle
Circle WaiJieYuan(Point p1,Point p2,Point p3)
{
    double Bx=p2.x-p1.x,By=p2.y-p1.y;
    double Cx=p3.x-p1.x,Cy=p3.y-p1.y;
    double D=2*(Bx*Cy-By*Cx);
    double ansx=(Cy*(Bx*Bx+By*By)-By*(Cx*Cx+Cy*Cy))/D+p1.x;
    double ansy=(Bx*(Cx*Cx+Cy*Cy)-Cx*(Bx*Bx+By*By))/D+p1.y;
    Point p(ansx,ansy);
    return Circle(p,Length(p1-p));
}
```

4.15 求三角形的内接圆(get_incircle)

```cpp
//get_incircle
Circle NeiJieYuan(Point p1,Point p2,Point p3)
{
    double a=Length(p2-p3);
    double b=Length(p3-p1);
    double c=Length(p1-p2);
    Point p=(p1*a+p2*b+p3*c)/(a+b+c);
    return Circle(p,DistanceToLine(p,p1,p2));
}
```

4.16 二维几何110_2合一！

具体问题见蓝书（《算法竞赛入门经典训练指南》P267）

```cpp
const char* q1="CircumscribedCircle";
const char* q2="InscribedCircle";
const char* q3="TangentLineThroughPoint";
const char* q4="CircleThroughAPointAndTangentToALineWithRadius";
const char* q5="CircleTangentToTwoLinesWithRadius";
const char* q6="CircleTangentToTwoDisjointCirclesWithRadius";
char cmd[100];
void output(vector<double> res)
{
    sort(res.begin(),res.end());
    printf("[");
    for(int i=0;i<res.size();i++)
    {
        if(i) printf(",");
        printf("%.6lf",res[i]);
    }
    printf("]\n");
}
void output(vector<Point> res)
{
    sort(res.begin(),res.end());
    printf("[");
    for(int i=0;i<res.size();i++)
    {
        if(i) printf(",");
        printf("(%.6lf,%.6lf)",res[i].x,res[i].y);
```

```
    }
    printf("]\n");
}
int main()
{
    while(scanf("%s",cmd)==1){
        if(strcmp(cmd,q1)==0){
            Point p1,p2,p3;
            p1=read_point();
            p2=read_point();
            p3=read_point();
            Circle C=WaiJieYuan(p1,p2,p3);
            printf("(%.6lf,%.6lf,%.6lf)\n",C.c.x,C.c.y,C.r);
        }
        if(strcmp(cmd,q2)==0){
            Point p1,p2,p3;
            p1=read_point();
            p2=read_point();
            p3=read_point();
            Circle C=NeiJieYuan(p1,p2,p3);
            printf("(%.6lf,%.6lf,%.6lf)\n",C.c.x,C.c.y,C.r);
        }
        if(strcmp(cmd,q3)==0){
            Circle C=read_circle();
            Point p=read_point();
            vector<Point> v;
            vector<double> res;
            GetTangents(p,C,v);
            for(int i=0;i<v.size();i++)
            {
                double tmp=R_to_D(Angle(v[i]-p));
                if(tmp<0) tmp+=180;
                if(tmp≥180) tmp-=180;
                res.push_back(tmp);
            }
            output(res);
        }
        if(strcmp(cmd,q4)==0){
            Point p=read_point();
            Point A=read_point();
            Point B=read_point();
            double r;
            scanf("%lf",&r);
            Circle C(p,r);
            Vector v=Normal(B-A)*r;//向两侧平移r
            Point A1=A+v,B1=B+v;
            Point A2=A-v,B2=B-v;
            vector<Point> res;
            GetLineCircleIntersection(A1,B1,C,res);
            GetLineCircleIntersection(A2,B2,C,res);
            output(res);
        }
        if(strcmp(cmd,q5)==0){
            Point p1=read_point();
            Point p2=read_point();
```

```
            Point p3=read_point();
            Point p4=read_point();
            double r;
            scanf("%lf",&r);
            Vector v=Normal(p1-p2)*r;
            Point A1=p1+v,B1=p2+v;
            Point A2=p1-v,B2=p2-v;
            v=Normal(p3-p4)*r;
            Point A3=p3+v,B3=p4+v;
            Point A4=p3-v,B4=p4-v;//向两侧平移r
            vector<Point> res;
            res.push_back(GetLineIntersection(A1,B1,A3,B3));
            res.push_back(GetLineIntersection(A1,B1,A4,B4));
            res.push_back(GetLineIntersection(A2,B2,A3,B3));
            res.push_back(GetLineIntersection(A2,B2,A4,B4));
            output(res);
        }
        if(strcmp(cmd,q6)==0){
            Circle c1=read_circle();
            Circle c2=read_circle();
            double r;
            scanf("%lf",&r);
            c1.r+=r;c2.r+=r;//向外膨胀r
            vector<Point> res;
            GetCircleCircleIntersection(c1,c2,res);
            output(res);
        }
    }
    return 0;
}

```

4.17 经纬度转换为空间坐标

蓝书P269

4.18 球面距离

蓝书P270

4.19 三点确定一圆

设 $x^2+y^2+Dx+Ey+F=0$，圆心为 O，半径为 r，带入三点 $A(x_1,y_1),B(x_2,y_2),C(x_3,y_3)$，解得：

$$\begin{cases} D=\frac{(x_2^2+y_2^2-x_3^2-y_3^2)(y_1-y_2)-(x_1^2+y_1^2-x_2^2-y_2^2)(y_2-y_3)}{(x_1-x_2)(y_2-y_3)-(x_2-x_3)(y_1-y_2)} \\ E=\frac{x_1^2+y_1^2-x_2^2-y_2^2+D(x_1-x_2)}{y_2-y_1} \\ F=-(x_1^2+y_1^2+Dx_1+Ey_1) \\ O=(-\frac{D}{2},-\frac{E}{2}) \\ r=\frac{D^2+E^2-4F}{4} \end{cases}$$

> 给你不共线的三个点的座标，你的任务是算出通过这三个点的唯一圆的方程式并以下列2种方式输出：
> $(x-h)2+(y-k)2=r2x2+y2+cx+dy-e=0$

```
#include<bits/stdc++.h>
#define pi  3.141592653589793
using namespace std;
double pf(double a) {//平方
    return a*a;
};
//两点距离
double func1(double a1,double a2,double b1,double b2){
    double delta_x = a1-b1;
    double delta_y = a2-b2;
    double t1=pf(delta_x);
    double t2=pf(delta_y);
    return sqrt(t1+t2);
};
//三阶行列式求解//a1 a2 a3 竖着写
double func2(double a1,double a2,double a3,double b1,double b2,double b3,double c1,double c2,double c3){
    return  a1*b2*c3+b1*c2*a3+c1*a2*b3-a3*b2*c1-b3*c2*a1-c3*a2*b1;
}
int main()
{
    double a1,a2,b1,b2,c1,c2;
    while (cin>>a1>>a2>>b1>>b2>>c1>>c2) {
        //先求出三边的边长
        double a,b,c;
        a=func1(a1, a2, b1, b2);
        b=func1(c1, c2, b1, b2);
        c=func1(a1, a2, c1, c2);
        //用海伦公式求面积
        //先求半周长
        double p = (a+b+c)/2;
        //求S
        double s = sqrt(p*(p-a)*(p-b)*(p-c));
        //外接圆半径 R= abc/(4S)
        double r = a*b*c / (4*s);
        //cout<<r<<endl;
        //由公式求圆心坐标
        double x = 0.5 * func2(1,1,1,pf(a1)+pf(a2),pf(b1)+pf(b2),pf(c1)+pf(c2),a2,b2,c2) / func2(1,1,1,a1,b1,c1,a2,b2,c2);
```

```
        //cout<<x<<endl;
        double y = 0.5 * func2(1,1,1,a1,b1,c1,pf(a1)+pf(a2),pf(b1)+pf(b2),pf(c1)+pf(c2)) /
func2(1,1,1,a1,b1,c1,a2,b2,c2);
       // cout<<y<<endl;
        //输出//加fabs，浮点数的绝对值函数，防止-0和0出现
        if(x≥0) printf("(x - %.3lf)^2 +",fabs(x));
        else printf("(x + %.3lf)^2 +",fabs(x));
        if(y≥0) printf(" (y - %.3lf)^2",fabs(y));
        else printf(" (y + %.3lf)^2",fabs(y));
        printf(" = %.3lf^2\n",r);
        printf("x^2 + y^2");
        if(x≥0) printf(" - %.3lfx",2*fabs(x));
        else printf(" + %.3lfx",2*fabs(x));
        if(y≥0) printf(" - %.3lfy",2*fabs(y));
        else printf(" + %.3lfy",2*fabs(y));
        double temp = pf(x) + pf(y) - pf(r);
        if(temp≥0) printf(" + %.3lf = 0",fabs(temp));
        else printf(" - %.3lf = 0",fabs(temp));
        cout<<endl<<endl;
    }
    return 0;
}
```

4.20 两个圆的关系(relationcircle)

```
//两圆的关系
//5 相离
//4 外切
//3 相交
//2 内切
//1 内含
//需要 Point 的 distance //测试：UVA12304
int relationcircle(circle v){
    double d = p.distance(v.p);
    if(sgn(d-r-v.r) > 0)return 5;
    if(sgn(d-r-v.r) == 0)return 4;
    double l = fabs(r-v.r);
    if(sgn(d-r-v.r)<0 && sgn(d-l)>0)return 3;
    if(sgn(d-l)==0)return 2;
    if(sgn(d-l)<0)return 1;
}
```

5. 网格

5.1 多边形上的网格点个数

5.2 多边形内的网格点个数

```
#define abs(x) ((x)>0?(x):-(x))
struct point{int x,y;};
int gcd(int a,int b){return b?gcd(b,a%b):a;}
//多边形上的网格点个数
```

```
int grid_onedge(int n,point* p){
    int i,ret=0;
    for (i=0;i<n;i++)
        ret+=gcd(abs(p[i].x-p[(i+1)%n].x),abs(p[i].y-p[(i+1)%n].y));
    return ret;
}
//多边形内的网格点个数
int grid_inside(int n,point* p){
    int i,ret=0;
    for (i=0;i<n;i++)
        ret+=p[(i+1)%n].y*(p[i].x-p[(i+2)%n].x);
    return (abs(ret)-grid_onedge(n,p))/2+1;
}
```

6.一些函数/定理/应用

6.1 欧拉定理

> 题目大意：给出一个平面上n-1个点的回路，第n个顶点与第1个顶点相同，求它把整个平面分成了几个部分（包括内部围起来的部分和外面的无限大的区域）。

欧拉定理：设平面图的顶点数、边数和面数分别为V，E，F，则V+F-E=2。

> 那么我们先求所有的交点(不算n个端点,端点另外再加,所以这些交点一定是在线段内部的,即刘汝佳所说的规范相交): 由于一共n条线段,且任意线段不会部分重叠,所以任意两条线段要不平行(不相交)要不就规范相交. 所以我们只需要枚举所有的线段,然后求出他们规范相交的节点保存在数组中即可.
>
> 然后我们再把原本的n-1个顶点(起点与终点相同不用都加)加进去,然后去重.
>
> 下面我们要知道该平面一共有多少条线段,那么我们只要知道原来的每条线段到底被分成了多少段即可. 对于原来的每条线段,我们用上面求得的所有点去判断,当前这个点是不是在该线段内(不包含端点),如果该点在线段内,那么就说明该线段被该点截断,多出来1段.所以总的线段数目在n的基础上要+1.

```
//上述均为模板全部省略
Point p[N], v[N * N];
int n;
int kcase;
int main()
{
    while(~scanf("%d", &n) && n){
        for(int i = 0; i < n; ++ i)
            scanf("%lf %lf", &p[i].x, &p[i].y), v[i] = p[i];
        n -- ;
        int c = n, e = n;
        for(int i = 0; i < n; ++ i)
            for(int j = i + 1; j < n; ++ j)
            if(segment_proper_intersection(p[i], p[i + 1], p[j], p[j + 1]))
            //方向向量和一个点确定一个直线P =  A + tv
            v[c ++ ] = Get_line_intersection(p[i], p[i + 1] - p[i], p[j], p[j + 1] - p[j]);
        sort(v, v + c);
        c = unique(v, v + c) - v;//去重防止三点共线
        //原来有n个点，有n-1个线段，新增点以后，
        //我们看有多少个新增的点在原来的线段上，
        //每有一个在原来线段的上面就会把这个线段割开成两个线段，答案+1
        for(int i = 0; i < c; ++ i)
            for(int j = 0; j < n; ++ j)
            if(on_segment(v[i], p[j], p[j + 1]))e ++ ;
```

```
        printf("Case %d: There are %d pieces.\n", ++ kcase, e + 2 - c);
    }
    return 0;
}

```

6.2 Pick定理（根据点在多边形内和边界的个数求面积）

格点就是横纵坐标相等的点

皮克定理是指一个计算点阵中顶点在格点上的多边形面积公式，该公式可以表示为

$2S = 2a + b - 2$

其中a表示多边形内部的点数，b表示多边形边界上的点数，S表示多边形的面积。

常用形式

$S = a + \frac{b}{2} - 1$

常用计算

给你多边形的顶点，问多边形内部有多少点

$a = S - \frac{b}{2} + 1$

```
//A = b / 2 + i -1  其中 b 与 i 分别表示在边界上及内部的格子点之个数
typedef struct TPoint
{
    int x;
    int y;
}TPoint;
typedef struct TLine
{
    int a, b, c;
}TLine;
int triangleArea(TPoint p1, TPoint p2, TPoint p3)
{
    //已知三角形三个顶点的坐标，求三角形的面积
    int k = p1.x * p2.y + p2.x * p3.y + p3.x * p1.y
      - p2.x * p1.y - p3.x * p2.y - p1.x * p3.y;
    if(k < 0) return -k;
    else return k;
}
TLine lineFromSegment(TPoint p1, TPoint p2)
{
    //线段所在直线,返回直线方程的三个系统
    TLine tmp;
    tmp.a = p2.y - p1.y;
    tmp.b = p1.x - p2.x;
    tmp.c = p2.x * p1.y - p1.x * p2.y;
    return tmp;
}
void swap(int &a, int &b)
{
    int t;
    t = a;
    a = b;
    b = t;
```

```
}
int Count(TPoint p1, TPoint p2)
{
    int i, sum = 0, y;
    TLine l1 =  lineFromSegment(p1, p2);
    if(l1.b == 0) return abs(p2.y - p1.y) + 1;
    if(p1.x > p2.x) swap(p1.x, p2.x); //这里没有交换WA两次
    for(i = p1.x;i ≤ p2.x;i++){
        y = -l1.c - l1.a * i;
        if(y % l1.b == 0) sum++;
    }
    return sum;
}
int main()
{
    //freopen("in.in", "r", stdin);
    //freopen("OUT.out", "w", stdout);
    TPoint p1, p2, p3;
    while(scanf("%d%d%d%d%d%d", &p1.x, &p1.y, &p2.x, &p2.y, &p3.x, &p3.y) ≠ EOF){
        if(p1.x == 0 && p1.y == 0 && p2.x == 0 && p2.y == 0 && p3.x == 0 && p3.y == 0) break;
        int  A = triangleArea(p1, p2, p3);//A为面积的两倍
        int b = 0;
        int i;
        b = Count(p1, p2) + Count(p1, p3) + Count(p3, p2) - 3;//3个顶点多各多加了一次
        //i = A / 2- b / 2 + 1;
        i = (A - b) / 2 + 1;
        printf("%d\n", i);
    }
    return 0;
}
```

6.3 判断四点共面（混合积）

```
struct  point
{
    double x, y, z;
    point  operator - (point &o)
    {
        point  ans;
        ans.x = this→x - o.x;
        ans.y = this→y - o.y;
        ans.z = this→z - o.z;
        return ans;
    }
};

double  dot_product(const point &a, const point &b)
{
    return a.x * b.x + a.y * b.y + a.z * b.z;
}

point cross_product(const point &a, const point &b)
{
```

```
    point  ans;
    ans.x = a.y * b.z - a.z * b.y;
    ans.y = a.z * b.x - a.x * b.z;
    ans.z = a.x * b.y - a.y * b.x;
    return ans;
}

int main()
{
    point p[4];
    int T;
    for (scanf("%d", &T); T--;)
    {
        for (int i = 0; i < 4; ++i)
        {
            scanf("%lf%lf%lf", &p[i].x, &p[i].y, &p[i].z);
        }
        puts(dot_product(p[3] - p[0], cross_product(p[2] - p[0], p[1] - p[0])) == 0.0 ? "Yes\n"
    : "No\n");
    }
    return 0;
}
```