计算几何

1. 向量的基本运算

1.1 点和向量的表示

在平面直角坐标系中，任意一点的坐标可以用一个有序数对 (x, y) 表示，向量也是如此

```
struct Point//点或向量
{
    double x, y;
    Point() {}
    Point(double x, double y) :x(x), y(y) {}
};
typedef Point Vector;
```

1.2 基本向量运算

设向量 $v_1 = (x_1, y_1), v_2 = (x_2, y_2)$，定义如下运算

1.2.1 向量加法

$$v_1 + v_2 = (x_1 + x_2, y_1 + y_2)$$

1.2.2 向量减法

$$v_1 - v_2 = (x_1 - x_2, y_1 - y_2)$$

若 $P = (x_1, y_1), Q = (x_2, y_2)$，则 $\overrightarrow{PQ} = Q - P = (x_2 - x_1, y_2 - y_1)$

1.2.3 向量模长

$$|v_1| = \sqrt{x_1^2 + y_1^2}$$

向量模长可以用来求两点间的距离

1.2.4 向量数乘

$$av_1 = (ax_1, ay_1), a \in \mathbb{R}$$

向量数乘可以实现向量的长度伸缩

1.2.5 向量内积（点积）

$$v_1 \cdot v_2 = |v_1||v_2| \cos < v_1, v_2 >= x_1x_2 + y_1y_2$$

$v_1 \cdot v_2 = 0$ 当且仅当 $v_1 \perp v_2$

向量内积可以用来求向量间的夹角

1.2.6 向量外积（叉积）

这个定义可能来自张量（Tensor）代数

$$v_1 \times v_2 = \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} = x_1 y_2 - x_2 y_1$$

$$|v_1 \times v_2| = |v_1||v_2| \sin < v_1, v_2 >$$

外积是很重要的一个概念，有很多应用

外积可以用来求面积，以 v_1, v_2 为邻边的平行四边形面积为 $|v_1 \times v_2|$

$v_1 \times v_2 = 0$ 当且仅当 $v_1 \parallel v_2$

外积可以用来判断向量间的位置关系，若 v_1 旋转到 v_2 的方向为顺时针，则 $v_1 \times v_2 < 0$，反之 $v_1 \times v_2 > 0$

1.2.7 向量旋转

向量 v_1 逆时针旋转 θ 后的坐标满足

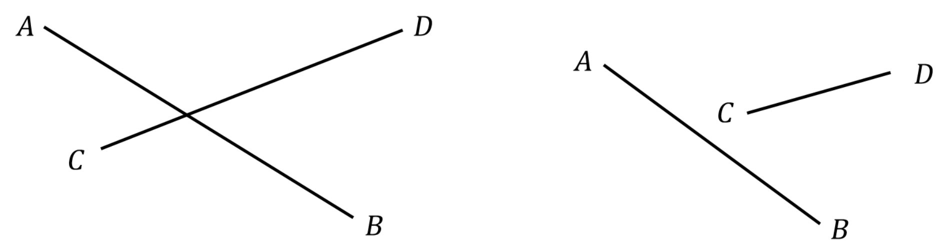
$$\begin{cases} x' = x_1 \cos\theta - y_1 \sin\theta \\ y' = x_1 \sin\theta + y_1 \cos\theta \end{cases}$$

```
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-6;//eps用于控制精度
const double pi = acos(-1.0);//pi
struct Point//点或向量
{
    double x, y;
    Point() {}
    Point(double x, double y) :x(x), y(y) {}
};
typedef Point Vector;
Vector operator + (Vector a, Vector b)//向量加法
{
    return Vector(a.x + b.x, a.y + b.y);
}
Vector operator - (Vector a, Vector b)//向量减法
{
    return Vector(a.x - b.x, a.y - b.y);
}
Vector operator * (Vector a, double p)//向量数乘
{
    return Vector(a.x*p, a.y*p);
}
Vector operator / (Vector a, double p)//向量数除
{
    return Vector(a.x / p, a.y / p);
}
int dcmp(double x)//精度三态函数(>0,<0,=0)
{
    if (fabs(x) < eps)return 0;
    else if (x > 0)return 1;
    return -1;
}
bool operator == (const Point &a, const Point &b)//向量相等
{
    return dcmp(a.x - b.x) == 0 && dcmp(a.y - b.y) == 0;
}
double Dot(Vector a, Vector b)//内积
```

```
{
    return a.x*b.x + a.y*b.y;
}
double Length(Vector a)//模
{
    return sqrt(Dot(a, a));
}
double Angle(Vector a, Vector b)//夹角,弧度制
{
    return acos(Dot(a, b) / Length(a) / Length(b));
}
double Cross(Vector a, Vector b)//外积
{
    return a.x*b.y - a.y*b.x;
}
Vector Rotate(Vector a, double rad)//逆时针旋转
{
    return Vector(a.x*cos(rad) - a.y*sin(rad), a.x*sin(rad) + a.y*cos(rad));
}
double Distance(Point a, Point b)//两点间距离
{
    return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}
double Area(Point a, Point b, Point c)//三角形面积
{
    return fabs(Cross(b - a, c - a) / 2);
}
```

2. 直线与线段

2.1 线段相交问题



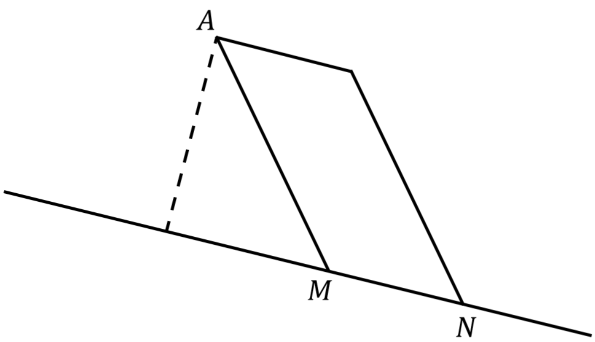
线段 AB 与 CD 相交（不考虑端点）的充分必要条件是

$$(\overrightarrow{CA} \cdot \overrightarrow{CB})(\overrightarrow{DA} \cdot \overrightarrow{DB}) < 0, (\overrightarrow{AC} \cdot \overrightarrow{AD})(\overrightarrow{BC} \cdot \overrightarrow{BD}) < 0$$

```
bool Intersect(Point A, Point B, Point C, Point D)//线段相交（不包括端点）
{
    double t1 = Cross(C - A, D - A)*Cross(C - B, D - B);
    double t2 = Cross(A - C, B - C)*Cross(A - D, B - D);
    return dcmp(t1) < 0 && dcmp(t2) < 0;
}
bool StrictIntersect(Point A, Point B, Point C, Point D) //线段相交（包括端点）
{
    return
        dcmp(max(A.x, B.x) - min(C.x, D.x)) >= 0
```

```
        && dcmp(max(C.x, D.x) - min(A.x, B.x)) >= 0
        && dcmp(max(A.y, B.y) - min(C.y, D.y)) >= 0
        && dcmp(max(C.y, D.y) - min(A.y, B.y)) >= 0
        && dcmp(Cross(C - A, D - A)*Cross(C - B, D - B)) <= 0
        && dcmp(Cross(A - C, B - C)*Cross(A - D, B - D)) <= 0;
}
```

2.2 点到直线的距离



如图所示，要计算点A到直线MN的距离，可以构建以AM，MN为邻边的平行四边形，其面积

$$S = |\overrightarrow{MA} \times \overrightarrow{MN}|$$

平行四边形的面积为底乘高，选取MN为底，高为

$$d = \frac{S}{\left|\overrightarrow{MN}\right|}$$

即为所求的A到直线MN的距离

```
double DistanceToLine(Point A, Point M, Point N)//点A到直线MN的距离,Error:MN=0
{
    return fabs(Cross(A - M, A - N) / Distance(M, N));
}
```

2.3 两直线交点

在实际应用中，通常的已知量是直线上某一点的坐标和直线的方向向量，对于两直线 l_1, l_2 ,设 $P\left(x_1, y_1\right)$, $\mathrm{Q}\left(x_2, y_2\right)$ 分别在 l_1 ， l_2 上， l_1 ， l_2 的方向向量分别为 $v=\left(a_1, b_1\right)$, $w=\left(a_2, b_2\right)$,由此可以得到两直线的方程

$$l_1:\left(x-x_1, y-y_1\right) \times\left(a_1, b_1\right)=0$$

$$l_2:\left(x-x_2, y-y_2\right) \times\left(a_2, b_2\right)=0$$

即

$$l_1: a_1 x-b_1 y=a_1 x_1-b_1 y_1$$

$$l_2: a_2 x-b_2 y=a_2 x_2-b_2 y_2$$

联立两直线的方程，由克拉默法则得，方程组的解为

$$\begin{cases} x = \dfrac{\begin{vmatrix} a_1x_1-b_1y_1 & -b_1 \\ a_2x_2-b_2y_2 & -b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & -b_1 \\ a_2 & -b_2 \end{vmatrix}} \\ y = \dfrac{\begin{vmatrix} a_1 & a_1x_1-b_1y_1 \\ a_2 & a_2x_2-b_2y_2 \end{vmatrix}}{\begin{vmatrix} a_1 & -b_1 \\ a_2 & -b_2 \end{vmatrix}} \end{cases}$$

进一步进行化简，得到

$$(x,y) = P + v \cdot \frac{w \times u}{v \times w}$$

其中 $u = -\overrightarrow{PQ}$

```
Point GetLineIntersection(Point P, Vector v, Point Q, Vector w)//两直线的交点
{
    Vector u = P - Q;
    double t = Cross(w, u) / Cross(v, w);
    return P + v * t;
}
```

3. 多边形

3.1 点和多边形的位置关系

设有（凸）$n(n \geq 3)$ 边形 $P_0P_2\dots P_{n-1}$，点的顺序为顺时针或逆时针，以及点A，记

$$\theta_i = \begin{cases} < \overrightarrow{AP_i}, \overrightarrow{AP_{i+1}} >, i < n-1 \\ < \overrightarrow{AP_{n-1}}, \overrightarrow{AP_0} >, i = n-1 \end{cases}$$

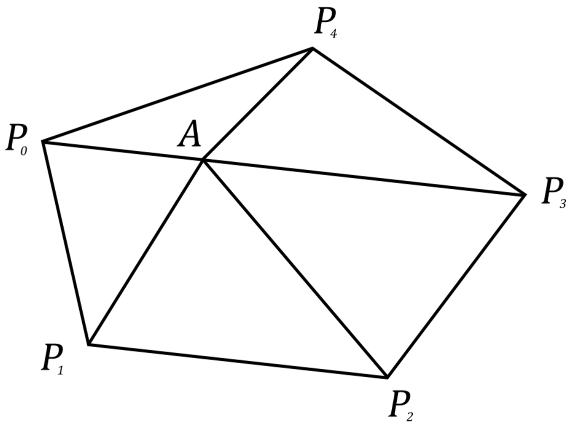
点在多边形内等价于

$$\sum_{i=0}^{n-1} \theta_i = 2\pi$$

```
/*模板说明：P[]为多边形的所有顶点，下标为0~n-1，n为多边形边数*/
Point P[1005];
int n;
bool InsidePolygon (Point A) //判断点是否在凸多边形内（角度和判别法）
{
    double alpha = 0;
    for (int i = 0; i < n; i++)
        alpha += fabs(Angle(P[i] - A, P[(i + 1) % n] - A));
    return dcmp(alpha - 2 * pi) == 0;
}
```

3.2 多边形的面积

设有（凸）$n(n \geq 3)$ 边形 $P_0P_2\dots P_{n-1}$ ，点的顺序为顺时针或逆时针，以及多边形内一点A，把多边形切割成如下所示n个三角形



多边形的面积等于所有三角形（有向）面积之和，代入坐标 $P_i(x_i, y_i), i = 0, 1, \ldots, n-1$ 计算得

$$S = \left| \frac{1}{2} \sum_{i=0}^{n-2} (x_i y_{i+1} - x_{i+1} y_i) + \frac{1}{2}(x_{n-1} y_0 - x_0 y_{n-1}) \right|$$

与A的坐标无关，因此A可任取，甚至可取在多边形外，通常为计算方便，取A为坐标原点

```
/*模板说明：P[]为多边形的所有顶点，下标为0~n-1，n为多边形边数*/
Point P[1005];
int n;
double PolygonArea()//求多边形面积（叉积和计算法）
{
    double sum = 0;
    Point O = Point(0, 0);
    for (int i = 0; i < n; i++)
        sum += Cross(P[i] - O, P[(i + 1) % n] - O);
    if (sum < 0)sum = -sum;
    return sum / 2;
}
```

3.3 凸包

在一个实向量空间 V 中，对于给定集合 X ，所有包含 X 的凸集的交集 S 称为 X 的凸包

$$S = \cap_{X \subset K \subset V, K \text{ is convex}} K$$

3.3.1 Graham's scan算法

第一步：找到最下边的点，如果有多个点纵坐标相同的点都在最下方，则选取最左边的，记为点A。这一步只需要扫描一遍所有的点即可，时间复杂度为 $O(n)$

第二步：将所有的点按照 AP_i 的极角大小进行排序，极角相同的按照到点A的距离排序。时间复杂度为 $O(nlogn)$

第三步：维护一个栈，以保存当前的凸包。按第二步中排序得到的结果，依次将点加入到栈中，如果当前点与栈顶的两个点不是“向左转”的，就表明当前栈顶的点并不在凸包上，而我们需要将其弹出栈，重复这一个过程直到当前点与栈顶的两个点是“向左转”的。这一步的时间复杂度为 $O(n)$

```
//求凸包
/*模板说明：n为所有点的个数，top为栈顶，P[]为所有点，下标为0~n-1，result[]为凸包上的点，下标为0~top，包含凸包边上的点,Error:有重复点*/
int n, top;
Point P[10005], result[10005];
```

```
bool cmp(Point A, Point B)
{
    double ans = Cross(A - P[0], B - P[0]);
    if (dcmp(ans) == 0)
        return dcmp(Distance(P[0], A) - Distance(P[0], B)) < 0;
    else
        return ans > 0;
}
void Graham()//Graham凸包扫描算法
{
    for (int i = 1; i < n; i++)//寻找起点
        if (P[i].y < P[0].y || (dcmp(P[i].y - P[0].y) == 0 && P[i].x < P[0].x))
            swap(P[i], P[0]);
    sort(P + 1, P + n, cmp);//极角排序，中心为起点
    result[0] = P[0];
    result[1] = P[1];
    top = 1;
    for (int i = 2; i < n; i++)
    {
        while (top >= 1 && Cross(result[top] - result[top - 1], P[i] -
result[top - 1]) < 0)
            top--;
        result[++top] = P[i];
    }
}
```

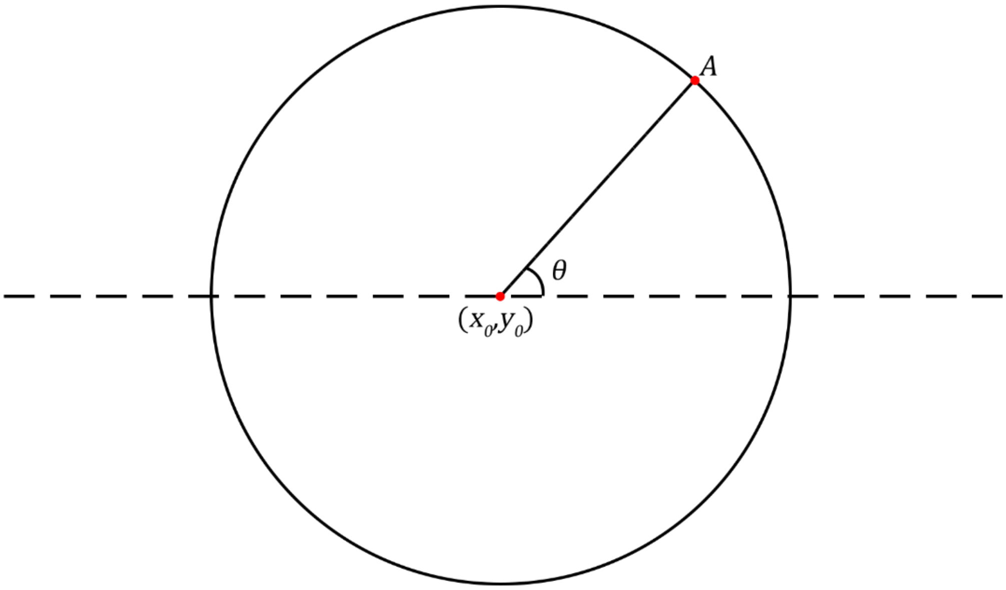
3.3.2 Andrew's monotone chain 算法

原理与Graham's scan算法相似，但上下凸包是分开维护的

```
namespace ConvexHull{
    bool cmp1(Point a,Point b){
        if(fabs(a.x-b.x)<eps)return a.y<b.y;
        return a.x<b.x;
    }
    //从左下角开始逆时针排列，去除凸包边上的点
    vector<Point> Andrew_s_monotone_chain(vector<Point> P){
        int n=P.size(),k=0;
        vector<Point> H(2*n);
        sort(P.begin(),P.end(),cmp1);
        for(int i=0;i<n;i++){
            while(k>=2 && Cross(H[k-1]-H[k-2],P[i]-H[k-2])<eps)k--;
            H[k++]=P[i];
        }
        int t=k+1;
        for(int i=n-1;i>0;i--){
            while(k>=t && Cross(H[k-1]-H[k-2],P[i-1]-H[k-2])<eps)k--;
            H[k++]=P[i-1];
        }
        H.resize(k-1);
        return H;
    }
}
```

4. 圆

4.1 圆的参数方程

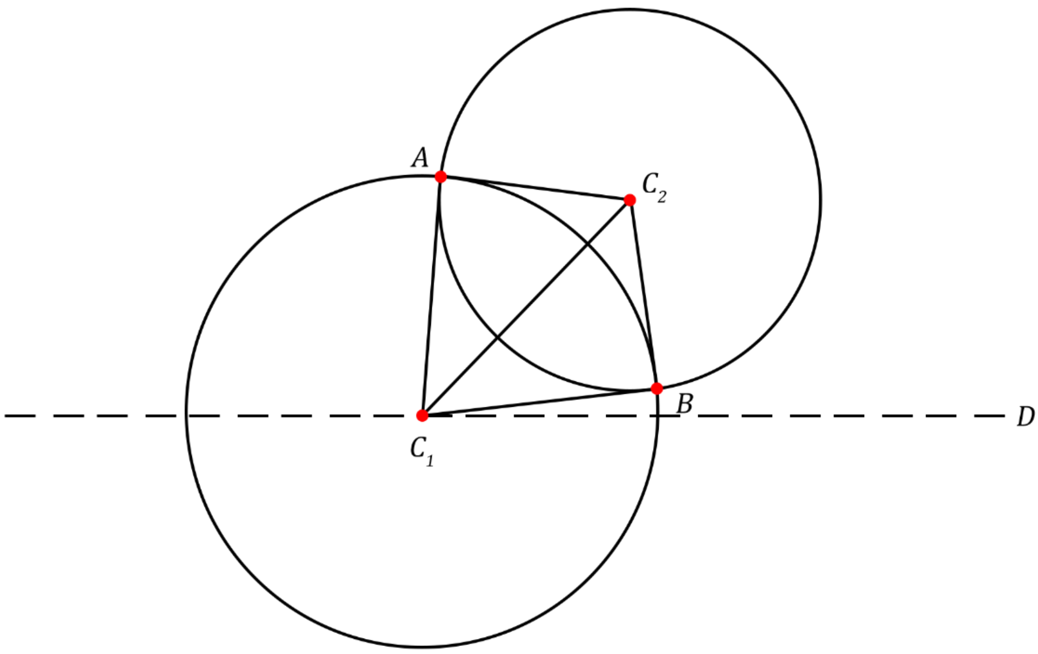


以(x_0, y_0)为圆心，r为半径的圆的参数方程为

$$\begin{cases} x = x_0 + r\cos\theta \\ y = y_0 + r\sin\theta \end{cases}$$

根据圆上一点和圆心连线与x轴正向的夹角可求得该点的坐标

4.2 两圆交点



设两圆C_1, C_2，其半径为$r_1, r_2 (r_1 \geq r_2)$，圆心距为d，则有

①两圆重合$\Longleftrightarrow d = 0 \ \ r_1 = r_2$

②两圆外离$\Longleftrightarrow d > r_1 + r_2$

③两圆外切$\Longleftrightarrow d = r_1 + r_2$

④两圆相交$\Longleftrightarrow r_1 - r_2 < d < r_1 + r_2$

⑤两圆内切$\Longleftrightarrow d=r_1-r_2$

⑥两圆内含$\Longleftrightarrow d<r_1-r_2$

对于情形④，如下图所示，要求A与B的坐标，只需求$\angle AC_1D$与$\angle BC_1D$，进而通过圆的参数方程即可求得

$$\angle AC_1D=\angle C_2C_1D+\angle AC_1C_2$$

$$\angle BC_1D=\angle C_2C_1D-\angle AC_1C_2$$

$\angle C_2C_1D$可以通过C_1,C_2的坐标求得，而$\angle AC_1C_2$可以通过ΔAC_1C_2上的余弦定理求得

对于情形③和情形⑤，上述方法求得的两点坐标是相同的，即为切点的坐标

```
struct Circle
{
    Point c;
    double r;
    Point point(double a)//基于圆心角求圆上一点坐标
    {
        return Point(c.x + cos(a)*r, c.y + sin(a)*r);
    }
};
double Angle(Vector v1)
{
    if (v1.y >= 0)return Angle(v1, Vector(1.0, 0.0));
    else return 2 * pi - Angle(v1, Vector(1.0, 0.0));
}
int GetCC(Circle C1, Circle C2)//求两圆交点
{
    double d = Length(C1.c - C2.c);
    if (dcmp(d) == 0)
    {
        if (dcmp(C1.r - C2.r) == 0)return -1;//重合
        else return 0;
    }
    if (dcmp(C1.r + C2.r - d) < 0)return 0;
    if (dcmp(fabs(C1.r - C2.r) - d) > 0)return 0;

    double a = Angle(C2.c - C1.c);
    double da = acos((C1.r*C1.r + d * d - C2.r*C2.r) / (2 * C1.r*d));
    Point p1 = C1.point(a - da), p2 = C1.point(a + da);
    if (p1 == p2)return 1;
    else return 2;
}
```