```
#include <iostream>
#include <cstdio>
#include <cmath>
#include <algorithm>


using namespace std;
const double PI = acos(-1.0);
const double eps = 1e-10;

/***************常用函数***************/
//判断ta与tb的大小关系
int sgn( double ta, double tb)
{
    if(fabs(ta-tb)<eps)return 0;
    if(ta<tb)   return -1;
    return 1;
}

//点
class Point
{
public:

    double x, y;

    Point(){}
    Point( double tx, double ty){ x = tx, y = ty;}

    bool operator < (const Point &_se) const
    {
        return x<_se.x || (x==_se.x && y<_se.y);
    }
    friend Point operator + (const Point &_st,const Point &_se)
    {
        return Point(_st.x + _se.x, _st.y + _se.y);
    }
    friend Point operator - (const Point &_st,const Point &_se)
    {
        return Point(_st.x - _se.x, _st.y - _se.y);
    }
    //点位置相同(double类型)
    bool operator == (const Point &_off)const
    {
```

```
        return  sgn(x, _off.x) == 0 && sgn(y, _off.y) == 0;
    }

};

/***************常用函数***************/
//点乘
double dot(const Point &po,const Point &ps,const Point &pe)
{
    return (ps.x - po.x) * (pe.x - po.x) + (ps.y - po.y) * (pe.y - po.y);
}
//叉乘
double xmult(const Point &po,const Point &ps,const Point &pe)
{
    return (ps.x - po.x) * (pe.y - po.y) - (pe.x - po.x) * (ps.y - po.y);
}
//两点间距离的平方
double getdis2(const Point &st,const Point &se)
{
    return (st.x - se.x) * (st.x - se.x) + (st.y - se.y) * (st.y - se.y);
}
//两点间距离
double getdis(const Point &st,const Point &se)
{
    return sqrt((st.x - se.x) * (st.x - se.x) + (st.y - se.y) * (st.y - se.y));
}

//两点表示的向量
class Line
{
public:

    Point s, e;//两点表示，起点[s]，终点[e]
    double a, b, c;//一般式,ax+by+c=0
    double angle;//向量的角度，[-pi,pi]

    Line(){}
    Line( Point ts, Point te):s(ts),e(te){}//get_angle();}
    Line(double _a,double _b,double _c):a(_a),b(_b),c(_c){}

    //排序用
    bool operator < (const Line &ta)const
    {
        return angle<ta.angle;
```

```
    }
    //向量与向量的叉乘
    friend double operator / ( const Line &_st, const  Line &_se)
    {
        return (_st.e.x - _st.s.x) * (_se.e.y - _se.s.y) - (_st.e.y - _st.s.y)
* (_se.e.x - _se.s.x);
    }
    //向量间的点乘
    friend double operator *( const Line &_st, const  Line &_se)
    {
        return (_st.e.x - _st.s.x) * (_se.e.x - _se.s.x) - (_st.e.y - _st.s.y)
* (_se.e.y - _se.s.y);
    }
    //从两点表示转换为一般表示
    //a=y2-y1,b=x1-x2,c=x2*y1-x1*y2
    bool pton()
    {
        a = e.y - s.y;
        b = s.x - e.x;
        c = e.x * s.y - e.y * s.x;
        return true;
    }
    //半平面交用
    //点在向量左边（右边的小于号改成大于号即可,在对应直线上则加上=号）
    friend bool operator < (const Point &_Off, const Line &_Ori)
    {
        return (_Ori.e.y - _Ori.s.y) * (_Off.x - _Ori.s.x)
            < (_Off.y - _Ori.s.y) * (_Ori.e.x - _Ori.s.x);
    }
    //求直线或向量的角度
    double get_angle( bool isVector = true)
    {
        angle = atan2( e.y - s.y, e.x - s.x);
        if(!isVector && angle < 0)
            angle += PI;
        return angle;
    }

    //点在线段或直线上 1:点在直线上 2 点在 s,e 所在矩形内
    bool has(const Point &_Off, bool isSegment = false) const
    {
        bool ff = sgn( xmult( s, e, _Off), 0) == 0;
        if( !isSegment) return ff;
        return ff
```

```
            && sgn(_Off.x - min(s.x, e.x), 0) >= 0 && sgn(_Off.x - max(s.x, 
e.x), 0) <= 0
            && sgn(_Off.y - min(s.y, e.y), 0) >= 0 && sgn(_Off.y - max(s.y, 
e.y), 0) <= 0;
    }

    //点到直线/线段的距离
    double dis(const Point &_Off, bool isSegment = false)
    {
        ///化为一般式
        pton();
        //到直线垂足的距离
        double td = (a * _Off.x + b * _Off.y + c) / sqrt(a * a + b * b);
        //如果是线段判断垂足
        if(isSegment)
        {
            double xp = (b * b * _Off.x - a * b * _Off.y - a * c) / ( a * a 
+ b * b);
            double yp = (-a * b * _Off.x + a * a * _Off.y - b * c) / (a * a 
+ b * b);
            double xb = max(s.x, e.x);
            double yb = max(s.y, e.y);
            double xs = s.x + e.x - xb;
            double ys = s.y + e.y - yb;
           if(xp > xb + eps || xp < xs - eps || yp > yb + eps || yp < ys - 
eps)
                td = min( getdis(_Off,s), getdis(_Off,e));
        }
        return fabs(td);
    }

    //关于直线对称的点
    Point mirror(const Point &_Off)
    {
        ///注意先转为一般式
        Point ret;
        double d = a * a + b * b;
        ret.x = (b * b * _Off.x - a * a * _Off.x - 2 * a * b * _Off.y - 2 
* a * c) / d;
        ret.y = (a * a * _Off.y - b * b * _Off.y - 2 * a * b * _Off.x - 2 
* b * c) / d;
        return ret;
    }
    //计算两点的中垂线
```

```
static Line ppline(const Point &_a,const Point &_b)
{
    Line ret;
    ret.s.x = (_a.x + _b.x) / 2;
    ret.s.y = (_a.y + _b.y) / 2;
    //一般式
    ret.a = _b.x - _a.x;
    ret.b = _b.y - _a.y;
    ret.c = (_a.y - _b.y) * ret.s.y + (_a.x - _b.x) * ret.s.x;
    //两点式
    if(fabs(ret.a) > eps)
    {
        ret.e.y = 0.0;
        ret.e.x = - ret.c / ret.a;
        if(ret.e == ret. s)
        {
            ret.e.y = 1e10;
            ret.e.x = - (ret.c - ret.b * ret.e.y) / ret.a;
        }
    }
    else
    {
        ret.e.x = 0.0;
        ret.e.y = - ret.c / ret.b;
        if(ret.e == ret. s)
        {
            ret.e.x = 1e10;
            ret.e.y = - (ret.c - ret.a * ret.e.x) / ret.b;
        }
    }
    return ret;
}

//------------直线和直线（向量）-------------
//向量向左边平移 t 的距离
Line& moveLine( double t)
{
    Point of;
    of = Point( -( e.y - s.y), e.x - s.x);
    double dis = sqrt( of.x * of.x + of.y * of.y);
    of.x= of.x * t / dis, of.y = of.y * t / dis;
    s = s + of, e = e + of;
    return *this;
}
```

```
    //直线重合
    static bool equal(const Line &_st,const Line &_se)
    {
        return _st.has( _se.e) && _se.has( _st.s);
    }
    //直线平行
    static bool parallel(const Line &_st,const Line &_se)
    {
        return sgn( _st / _se, 0) == 0;
    }
    //两直线（线段）交点
    //返回-1 代表平行，0 代表重合，1 代表相交
    static bool crossLPt(const Line &_st,const Line &_se, Point &ret)
    {
        if(parallel(_st,_se))
        {
            if(Line::equal(_st,_se)) return 0;
            return -1;
        }
        ret = _st.s;
        double t = ( Line(_st.s,_se.s) / _se) / ( _st / _se);
        ret.x += (_st.e.x - _st.s.x) * t;
        ret.y += (_st.e.y - _st.s.y) * t;
        return 1;
    }
    //------------线段和直线（向量）----------
    //直线和线段相交
    //参数：直线[_st],线段[_se]
    friend bool crossSL( Line &_st, Line &_se)
    {
        return sgn( xmult( _st.s, _se.s, _st.e) * xmult( _st.s, _st.e, _se.e),
0) >= 0;
    }

    //判断线段是否相交(注意添加 eps)
    static bool isCrossSS( const Line &_st, const  Line &_se)
    {
        //1.快速排斥试验判断以两条线段为对角线的两个矩形是否相交
        //2.跨立试验（等于 0 时端点重合）
        return
            max(_st.s.x, _st.e.x) >= min(_se.s.x, _se.e.x) &&
            max(_se.s.x, _se.e.x) >= min(_st.s.x, _st.e.x) &&
            max(_st.s.y, _st.e.y) >= min(_se.s.y, _se.e.y) &&
            max(_se.s.y, _se.e.y) >= min(_st.s.y, _st.e.y) &&
```

```
            sgn( xmult( _se.s, _st.s, _se.e) * xmult( _se.s, _se.e, _st.s),
0) >= 0 &&
            sgn( xmult( _st.s, _se.s, _st.e) * xmult( _st.s, _st.e, _se.s),
0) >= 0;
    }
};

//寻找凸包的 graham 扫描法所需的排序函数
Point gsort;
bool gcmp( const Point &ta, const Point &tb)/// 选取与最后一条确定边夹角最
小的点，即余弦值最大者
{
    double tmp = xmult( gsort, ta, tb);
    if( fabs( tmp) < eps)
        return getdis( gsort, ta) < getdis( gsort, tb);
    else if( tmp > 0)
        return 1;
    return 0;
}

class Polygon
{
public:
    const static int maxpn = 5e4+7;
    Point pt[maxpn];//点（顺时针或逆时针）
    Line dq[maxpn]; //求半平面交打开注释
    int n;//点的个数


    //求多边形面积，多边形内点必须顺时针或逆时针
    double area()
    {
        double ans = 0.0;
        for(int i = 0; i < n; i ++)
        {
            int nt = (i + 1) % n;
            ans += pt[i].x * pt[nt].y - pt[nt].x * pt[i].y;
        }
        return fabs( ans / 2.0);
    }
    //求多边形重心，多边形内点必须顺时针或逆时针
    Point gravity()
    {
        Point ans;
```

```
        ans.x = ans.y = 0.0;
        double area = 0.0;
        for(int i = 0; i < n; i ++)
        {
            int nt = (i + 1) % n;
            double tp = pt[i].x * pt[nt].y - pt[nt].x * pt[i].y;
            area += tp;
            ans.x += tp * (pt[i].x + pt[nt].x);
            ans.y += tp * (pt[i].y + pt[nt].y);
        }
        ans.x /= 3 * area;
        ans.y /= 3 * area;
        return ans;
    }
    //判断点是否在任意多边形内[射线法]，O(n)
    bool ahas( Point &_Off)
    {
        int ret = 0;
        double infv = 1e20;//坐标系最大范围
        Line l = Line( _Off, Point( -infv ,_Off.y));
        for(int i = 0; i < n; i ++)
        {
            Line ln = Line( pt[i], pt[(i + 1) % n]);
            if(fabs(ln.s.y - ln.e.y) > eps)
            {
                Point tp = (ln.s.y > ln.e.y)? ln.s: ln.e;
                if( ( fabs( tp.y - _Off.y) < eps && tp.x < _Off.x + eps) ||
Line::isCrossSS( ln, l))
                    ret++;
            }
            else if( Line::isCrossSS( ln, l))
                ret++;
        }
        return ret&1;
    }

    //判断任意点是否在凸包内，O(logn)
    bool bhas( Point & p)
    {
        if( n < 3)
            return false;
        if( xmult( pt[0], p, pt[1]) > eps)
            return false;
        if( xmult( pt[0], p, pt[n-1]) < -eps)
```

```
            return false;
        int l = 2,r = n-1;
        int line = -1;
        while( l <= r)
        {
            int mid = ( l + r) >> 1;
            if( xmult( pt[0], p, pt[mid]) >= 0)
                line = mid,r = mid - 1;
            else l = mid + 1;
        }
        return xmult( pt[line-1], p, pt[line]) <= eps;
    }



    //凸多边形被直线分割
    Polygon split( Line &_Off)
    {
        //注意确保多边形能被分割
        Polygon ret;
        Point spt[2];
        double tp = 0.0, np;
        bool flag = true;
        int i, pn = 0, spn = 0;
        for(i = 0; i < n; i ++)
        {
            if(flag)
                pt[pn ++] = pt[i];
            else
                ret.pt[ret.n ++] = pt[i];
            np = xmult( _Off.s, _Off.e, pt[(i + 1) % n]);
            if(tp * np < -eps)
            {
                flag = !flag;
                Line::crossLPt(  _Off,  Line(pt[i],  pt[(i  +  1)  %  n]),
spt[spn++]);
            }
            tp = (fabs(np) > eps)?np: tp;
        }
        ret.pt[ret.n ++] = spt[0];
        ret.pt[ret.n ++] = spt[1];
        n = pn;
        return ret;
    }
```

```
/** 卷包裹法求点集凸包，_p 为输入点集，_n 为点的数量 **/
void ConvexClosure( Point _p[], int _n)
{
    sort( _p, _p + _n);
    n = 0;
    for(int i = 0; i < _n; i++)
    {
        while( n > 1 && sgn( xmult( pt[n-2], pt[n-1], _p[i]), 0) <= 0)
            n--;
        pt[n++] = _p[i];
    }
    int _key = n;
    for(int i = _n - 2; i >= 0; i--)
    {
        while( n > _key && sgn( xmult( pt[n-2], pt[n-1], _p[i]), 0) <=
0)
            n--;
        pt[n++] = _p[i];
    }
    if(n>1)   n--;//除去重复的点，该点已是凸包凸包起点
}
/****** 寻找凸包的 graham 扫描法********************/
/****** _p 为输入的点集,_n 为点的数量****************/

void graham( Point _p[], int _n)
{
    int cur=0;
    for(int i = 1; i < _n; i++)
        if( sgn( _p[cur].y, _p[i].y) > 0 || ( sgn( _p[cur].y, _p[i].y)
== 0 && sgn( _p[cur].x, _p[i].x) > 0) )
            cur = i;
    swap( _p[cur], _p[0]);
    n = 0, gsort = pt[n++] = _p[0];
    if( _n <= 1)   return;
    sort( _p + 1, _p+_n ,gcmp);
    pt[n++] = _p[1];
    for(int i = 2; i < _n; i++)
    {
        while(n>1 && sgn( xmult( pt[n-2], pt[n-1], _p[i]), 0) <= 0)// 当
凸包退化成直线时需特别注意 n
            n--;
        pt[n++] = _p[i];
```

```
        }
    }
    //凸包旋转卡壳(注意点必须顺时针或逆时针排列)
    //返回值凸包直径的平方（最远两点距离的平方）
    pair<Point,Point> rotating_calipers()
    {
        int i = 1 % n;
        double ret = 0.0;
        pt[n] = pt[0];
        pair<Point,Point>ans=make_pair(pt[0],pt[0]);
        for(int j = 0; j < n; j ++)
        {
            while( fabs( xmult( pt[i+1], pt[j], pt[j + 1])) > fabs( xmult( pt[i], pt[j], pt[j + 1])) + eps)
                i = (i + 1) % n;
            //pt[i]和 pt[j],pt[i + 1]和 pt[j + 1]可能是对踵点
            if(ret < getdis2(pt[i],pt[j]))  ret = getdis2(pt[i],pt[j]), ans = make_pair(pt[i],pt[j]);
            if(ret < getdis2(pt[i+1],pt[j+1])) ret = getdis(pt[i+1],pt[j+1]), ans = make_pair(pt[i+1],pt[j+1]);
        }
        return ans;
    }

    //凸包旋转卡壳(注意点必须逆时针排列)
    //返回值两凸包的最短距离
    double rotating_calipers( Polygon &_Off)
    {
        int i = 0;
        double ret = 1e10;//inf
        pt[n] = pt[0];
        _Off.pt[_Off.n] = _Off.pt[0];
        //注意凸包必须逆时针排列且 pt[0]是左下角点的位置
        while( _Off.pt[i + 1].y > _Off.pt[i].y)
            i = (i + 1) % _Off.n;
        for(int j = 0; j < n; j ++)
        {
            double tp;
            //逆时针时为 >,顺时针则相反
            while((tp = xmult(_Off.pt[i + 1],pt[j], pt[j + 1]) - xmult(_Off.pt[i], pt[j], pt[j + 1])) > eps)
                i = (i + 1) % _Off.n;
            //(pt[i],pt[i+1])和(_Off.pt[j],_Off.pt[j + 1])可能是最近线段
            ret = min(ret, Line(pt[j], pt[j + 1]).dis(_Off.pt[i], true));
```

```
            ret = min(ret, Line(_Off.pt[i], _Off.pt[i + 1]).dis(pt[j + 1],
true));
            if(tp > -eps)//如果不考虑 TLE 问题最好不要加这个判断
            {
                ret = min(ret, Line(pt[j], pt[j + 1]).dis(_Off.pt[i + 1],
true));
                ret = min(ret, Line(_Off.pt[i], _Off.pt[i + 1]).dis(pt[j],
true));
            }
        }
        return ret;
    }

    //-----------半平面交-------------
    //复杂度:O(nlog2(n))
    //获取半平面交的多边形（多边形的核）
    //参数：向量集合[l]，向量数量[ln];(半平面方向在向量左边)
    //函数运行后如果 n[即返回多边形的点数量]为 0 则不存在半平面交的多边形(不存在
区域或区域面积无穷大)
    int judege( Line &_lx, Line &_ly, Line &_lz)
    {
        Point tmp;
        Line::crossLPt(_lx,_ly,tmp);
        return sgn(xmult(_lz.s,tmp,_lz.e),0);
    }
    int halfPanelCross(Line L[], int ln)
    {
        int i, tn, bot, top;
        for(int i = 0; i < ln; i++)
            L[i].get_angle();
        sort(L, L + ln);
        //平面在向量左边的筛选
        for(i = tn = 1; i < ln; i ++)
            if(fabs(L[i].angle - L[i - 1].angle) > eps)
                L[tn ++] = L[i];
        ln = tn, n = 0, bot = 0, top = 1;
        dq[0] = L[0], dq[1] = L[1];
        for(i = 2; i < ln; i ++)
        {
            while(bot < top &&  judege(dq[top],dq[top-1],L[i]) > 0)
                top --;
            while(bot < top &&  judege(dq[bot],dq[bot+1],L[i]) > 0)
                bot ++;
            dq[++ top] = L[i];
```

```
        }
        while(bot < top && judege(dq[top],dq[top-1],dq[bot]) > 0)
            top --;
        while(bot < top && judege(dq[bot],dq[bot+1],dq[top]) > 0)
            bot ++;
        //若半平面交退化为点或线
        //          if(top <= bot + 1)
        //              return 0;
        dq[++top] = dq[bot];
        for(i = bot; i < top; i ++)
            Line::crossLPt(dq[i],dq[i + 1],pt[n++]);
        return n;
    }
};


class Circle
{
public:
    Point c;//圆心
    double r;//半径
    double db, de;//圆弧度数起点， 圆弧度数终点(逆时针 0-360)

    //-------圆---------

    //判断圆在多边形内
    bool inside( Polygon &_Off)
    {
        if(_Off.ahas(c) == false)
            return false;
        for(int i = 0; i < _Off.n; i ++)
        {
            Line l = Line(_Off.pt[i], _Off.pt[(i + 1) % _Off.n]);
            if(l.dis(c, true) < r - eps)
                return false;
        }
        return true;
    }

    //判断多边形在圆内（线段和折线类似）
    bool has( Polygon &_Off)
    {
        for(int i = 0; i < _Off.n; i ++)
            if( getdis2(_Off.pt[i],c) > r * r - eps)
```

```
            return false;
    return true;
}

//-------圆弧-------
//圆被其他圆截得的圆弧，参数：圆[_Off]
Circle operator-(Circle &_Off) const
{
    //注意圆必须相交，圆心不能重合
    double d2 = getdis2(c,_Off.c);
    double d = getdis(c,_Off.c);
    double ans = acos((d2 + r * r - _Off.r * _Off.r) / (2 * d * r));
    Point py = _Off.c - c;
    double oans = atan2(py.y, py.x);
    Circle res;
    res.c = c;
    res.r = r;
    res.db = oans + ans;
    res.de = oans - ans + 2 * PI;
    return res;
}
//圆被其他圆截得的圆弧，参数：圆[_Off]
Circle operator+(Circle &_Off) const
{
    //注意圆必须相交，圆心不能重合
    double d2 = getdis2(c,_Off.c);
    double d = getdis(c,_Off.c);
    double ans = acos((d2 + r * r - _Off.r * _Off.r) / (2 * d * r));
    Point py = _Off.c - c;
    double oans = atan2(py.y, py.x);
    Circle res;
    res.c = c;
    res.r = r;
    res.db = oans - ans;
    res.de = oans + ans;
    return res;
}

//过圆外一点的两条切线
//参数：点[_Off](必须在圆外),返回：两条切线(切线的 s 点为_Off,e 点为切点)
pair<Line, Line>  tangent( Point &_Off)
{
    double d = getdis(c,_Off);
    //计算角度偏移的方式
```

```
        double angp = acos(r / d), ango = atan2(_Off.y - c.y, _Off.x - c.x);
        Point pl = Point(c.x + r * cos(ango + angp), c.y + r * sin(ango + angp)),
            pr = Point(c.x + r * cos(ango - angp), c.y + r * sin(ango - angp));
        return make_pair(Line(_Off, pl), Line(_Off, pr));
    }

    //计算直线和圆的两个交点
    //参数：直线[_Off](两点式)，返回两个交点，注意直线必须和圆有两个交点
    pair<Point, Point> cross(Line _Off)
    {
        _Off.pton();
        //到直线垂足的距离
        double td = fabs(_Off.a * c.x + _Off.b * c.y + _Off.c) / sqrt(_Off.a * _Off.a + _Off.b * _Off.b);

        //计算垂足坐标
        double xp = (_Off.b * _Off.b * c.x - _Off.a * _Off.b * c.y - _Off.a * _Off.c) / ( _Off.a * _Off.a + _Off.b * _Off.b);
        double yp = (- _Off.a * _Off.b * c.x + _Off.a * _Off.a * c.y - _Off.b * _Off.c) / (_Off.a * _Off.a + _Off.b * _Off.b);

        double ango = atan2(yp - c.y, xp - c.x);
        double angp = acos(td / r);

        return make_pair(Point(c.x + r * cos(ango + angp), c.y + r * sin(ango + angp)),
            Point(c.x + r * cos(ango - angp), c.y + r * sin(ango - angp)));
    }
};

class triangle
{
public:
    Point a, b, c;//顶点
    triangle(){}
    triangle(Point a, Point b, Point c): a(a), b(b), c(c){}

    //计算三角形面积
    double area()
    {
        return fabs( xmult(a, b, c)) / 2.0;
    }
```

```
    //计算三角形外心
    //返回：外接圆圆心
    Point circumcenter()
    {
        double pa = a.x * a.x + a.y * a.y;
        double pb = b.x * b.x + b.y * b.y;
        double pc = c.x * c.x + c.y * c.y;
        double ta = pa * ( b.y - c.y) - pb * ( a.y - c.y) + pc * ( a.y - b.y);
        double tb = -pa * ( b.x - c.x) + pb * ( a.x - c.x) - pc * ( a.x -
b.x);
        double tc = a.x * ( b.y - c.y) - b.x * ( a.y - c.y) + c.x * ( a.y
- b.y);
        return Point( ta / 2.0 / tc, tb / 2.0 / tc);
    }

    //计算三角形内心
    //返回：内接圆圆心
    Point incenter()
    {
        Line u, v;
        double m, n;
        u.s = a;
        m = atan2(b.y - a.y, b.x - a.x);
        n = atan2(c.y - a.y, c.x - a.x);
        u.e.x = u.s.x + cos((m + n) / 2);
        u.e.y = u.s.y + sin((m + n) / 2);
        v.s = b;
        m = atan2(a.y - b.y, a.x - b.x);
        n = atan2(c.y - b.y, c.x - b.x);
        v.e.x = v.s.x + cos((m + n) / 2);
        v.e.y = v.s.y + sin((m + n) / 2);
        Point ret;
        Line::crossLPt(u,v,ret);
        return ret;
    }

    //计算三角形垂心
    //返回：高的交点
    Point perpencenter()
    {
        Line u,v;
        u.s = c;
        u.e.x = u.s.x - a.y + b.y;
        u.e.y = u.s.y + a.x - b.x;
```

```
        v.s = b;
        v.e.x = v.s.x - a.y + c.y;
        v.e.y = v.s.y + a.x - c.x;
        Point ret;
        Line::crossLPt(u,v,ret);
        return ret;
    }

    //计算三角形重心
    //返回：重心
    //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
    Point barycenter()
    {
        Line u,v;
        u.s.x = (a.x + b.x) / 2;
        u.s.y = (a.y + b.y) / 2;
        u.e = c;
        v.s.x = (a.x + c.x) / 2;
        v.s.y = (a.y + c.y) / 2;
        v.e = b;
        Point ret;
        Line::crossLPt(u,v,ret);
        return ret;
    }

    //计算三角形费马点
    //返回：到三角形三顶点距离之和最小的点
    Point fermentPoint()
    {
        Point u, v;
        double step = fabs(a.x) + fabs(a.y) + fabs(b.x) + fabs(b.y) + fabs(c.x)
+ fabs(c.y);
        int i, j, k;
        u.x = (a.x + b.x + c.x) / 3;
        u.y = (a.y + b.y + c.y) / 3;
        while (step > eps)
        {
            for (k = 0; k < 10; step /= 2, k ++)
            {
                for (i = -1; i <= 1; i ++)
                {
                    for (j =- 1; j <= 1; j ++)
                    {
```

```
                        v.x = u.x + step * i;
                        v.y = u.y + step * j;
                        if  (getdis(u,a)  +  getdis(u,b)  +  getdis(u,c)  >
getdis(v,a) + getdis(v,b) + getdis(v,c))
                            u = v;
                    }
                }
            }
        }
        return u;
    }
};
```