计算几何基础

0.公式

三角形:

1. 半周长 $P=(a+b+c)/2$
2. 面积 $S=aH_a/2=absin(C)/2=sqrt(P(P-a)(P-b)(P-c))$
3. 中线 $M_a=sqrt(2(b^2+c^2)-a^2)/2=sqrt(b^2+c^2+2bccos(A))/2$
4. 角平分线 $T_a=sqrt(bc((b+c)^2-a^2))/(b+c)=2bccos(A/2)/(b+c)$
5. 高线 $H_a=bsin(C)=csin(B)=sqrt(b^2-((a^2+b^2-c^2)/(2a))^2)$
6. 内切圆半径 $r=S/P=asin(B/2)sin(C/2)/sin((B+C)/2)$
   $=4Rsin(A/2)sin(B/2)sin(C/2)=sqrt((P-a)(P-b)(P-c)/P)$
   $=Ptan(A/2)tan(B/2)tan(C/2)$
7. 外接圆半径 $R=abc/(4S)=a/(2sin(A))=b/(2sin(B))=c/(2sin(C))$

四边形:

D1,D2为对角线,M对角线中点连线,A为对角线夹角

1. $a^2+b^2+c^2+d^2=D_1^2+D_2^2+4M^2$
2. $S=D_1D_2sin(A)/2$
   (以下对圆的内接四边形)
3. $ac+bd=D_1D_2$
4. $S=sqrt((P-a)(P-b)(P-c)(P-d))$,P为半周长

正n边形:

R为外接圆半径,r为内切圆半径

1. 中心角 $A=2PI/n$
2. 内角 $C=(n-2)PI/n$
3. 边长 $a=2sqrt(R^2-r^2)=2Rsin(A/2)=2rtan(A/2)$
4. 面积 $S=nar/2=nr^2tan(A/2)=nR^2sin(A)/2=na^2/(4tan(A/2))$

圆:

1. 弧长 $l=rA$

2. 弦长 $a = 2sqrt(2hr - h^2) = 2rsin(A/2)$
3. 弓形高 $h = r - sqrt(r^2 - a^2/4) = r(1 - cos(A/2)) = atan(A/4)/2$
4. 扇形面积 $S1 = rl/2 = r^2A/2$
5. 弓形面积 $S2 = (rl - a(r - h))/2 = r^2(A - sin(A))/2$

棱柱:

1. 体积 $V = Ah$,A为底面积,h为高
2. 侧面积 $S = lp$,l为棱长,p为直截面周长
3. 全面积 $T = S + 2A$

棱锥:

1. 体积 $V = Ah/3$,A为底面积,h为高
   (以下对正棱锥)
2. 侧面积 $S = lp/2$,l为斜高,p为底面周长
3. 全面积 $T = S + A$

棱台:

1. 体积 $V = (A_1 + A_2 + sqrt(A_1A_2))h/3$,A1.A2为上下底面积,h为高
   (以下为正棱台)
2. 侧面积 $S = (p1 + p2)l/2, p1.p2$为上下底面周长,l为斜高
3. 全面积 $T = S + A_1 + A_2$

圆柱:

1. 侧面积 $S = 2PIrh$
2. 全面积 $T = 2PIr(h + r)$
3. 体积 $V = PIr^2h$

圆锥:

1. 母线 $l = sqrt(h^2 + r^2)$
2. 侧面积 $S = PIrl$
3. 全面积 $T = PIr(l + r)$
4. 体积 $V = PIr^2h/3$

圆台:

1. 母线 $l = sqrt(h^2 + (r1 - r2)^2)$
2. 侧面积 $S = PI(r1 + r2)l$
3. 全面积 $T = PIr1(l + r1) + PIr2(l + r2)$
4. 体积 $V = PI(r1^2 + r2^2 + r1r2)h/3$

球:

1. 全面积 $T = 4PIr^2$
2. 体积 $V = 4PIr^3/3$

球台:

1. 侧面积 $S = 2PIrh$
2. 全面积 $T = PI(2rh + r1^2 + r2^2)$
3. 体积 $V = PIh(3(r1^2 + r2^2) + h^2)/6$

球扇形:

1. 全面积 $T = PIr(2h + r0)$,h为球冠高,r0为球冠底面半径
2. 体积 $V = 2PIr^2h/3$

1.常量、基础函数

```
#include<iostream>
#include<cmath>
#include<cstdio>
using namespace std;
const double pi = 3.1415926535898;
const double eps = 1e-8;

inline int fcmp(double x,double y) //浮点比较
{
  if(fabs(x-y) < eps) return 0;
  else return x > y ? 1 : -1;
}
inline double sqr(double x) //浮点平方
{
  return x * x;
}
inline double Sqrt(double x)
{
  return x <= 0 ? 0 : sqrt(x);
}
inline int abs(int x)
{
  return x >= 0 ? x : -x;
}

int main()
{
  double x;
  int fx = floor(x);  //向下取整
```

```
    int cx = ceil(x);    //向上取整
    int rx = round(x);   //四舍五入
}
```

2.二维点、向量

2.1 二维点类

```
struct Point
{
  double x, y, ang;
  Point(){}
  Point(double a, double b): x(a), y(b) {}
  friend Point operator + (const Point &a, const Point &b){
    return Point(a.x + b.x, a.y + b.y);
  }
  friend Point operator - (const Point &a, const Point &b){
    return Point(a.x - b.x, a.y - b.y);
  }
  friend bool operator == (const Point &a, const Point &b){
    return fcmp(a.x,b.x) == 0 && fcmp(a.y,b.y) == 0;
  }
  friend Point operator * (const Point &a, const double &b){
    return Point(a.x * b, a.y * b);
  }
  friend Point operator * (const double a,const Point &b){
    return Point(a * b.x, a * b.y);
  }
  friend Point operator / (const Point &a, const double &b){
    return Point(a.x / b, a.y / b);
  }
  friend bool operator < (const Point &a,const Point &b){
    if(a.x == b.x)  return a.y < b.y;
    return a.x < b.x;
  }
  friend double norm(const Point &a){
    return sqrt(sqr(a.x) + sqr(a.y));
  }
  void calcangle() {ang = atan2(y,x);}
};
```

2.2 向量

- 既有大小又有方向的量叫向量
- 通常用坐标表示

鉴于以上两点，向量可以用二维点类表示。
$\overrightarrow{AB} = B - A$

2.2.1向量运算

---

1. 内积

- 向量夹角为锐角，内积为正
- 向量夹角为钝角，内积为负
- 向量夹角为直角，内积为0

```
inline double dot(const Point &a,const Point &b) //内积
{
  return a.x * b.x + a.y * b.y;
}
```

2. 叉积

$A \times B = |A||B| sin\theta$

根据右手定则

- 若$A \times B < 0$ B在A的顺时针方向
- 若$A \times B = 0$ B与A共线
- 若$A \times B > 0$ B在A的逆时针方向

```
inline double det(const Point &a,const Point &b)  //外积
{
  return a.x * b.y - a.y * b.x;
}
```

3.两点间距离(向量长度)

```
inline double dist(const Point &a,const Point &b)  //两点间距离
{
  return (a - b).norm();
}
```

4.两向量夹角(弧度)

```
inline double angle(const Point &a,const Point &b)  //向量夹角
{
  return acos(dot(a,b) / norm(a) / norm(b));
}
```

5.向量绕原点逆时针旋转A(弧度)

```
inline Point rotate_point(const Point &p,double A)  //计算点p绕原点逆时针旋转A(弧度)
{
  return Point(p.x * cos(A) - p.y * sin(A),p.x * sin(A) - p.y * cos(A));
}
```

6.计算两向量构成的平行四边形有向面积

```
inline double area2(const Point a,const Point &b)  //计算两向量构成的有向面积
{
  return det(a,b);
}
```

2.3 点与线

2.3.1 直线线段类

```
struct Line
{
  Point s, t;
  double ang; //直线极角
  Line(){};
  Line(Point a,Point b) : s(a) , t(b) {}
  Line(Point p,double ang) //根据一个点和一个角(弧度制, 0 <= ang < pi) 确定直线
  {
    s = p;
    if(sgn(ang - pi / 2) == 0)  t = (s + Point(0,1));
    else t = s + Point(1,tan(ang));
  }
  Line(double a,double b,double c)  // ax + by + c = 0
  {
    if(sgn(a) == 0){s = Point(0,-c/b); t = Point(1,-c/b);}
    else if(sgn(b) == 0){s = Point(-c/a,0); t = Point(-c/a,1);}
    else{s = Point(0,-c/b); t = Point(1,(-c-a)/b);}
```

```
    }
    void calcangle()  {ang = atan2((t - s).y, (t - s).x);}  //计算直线极角
};
```

2.3.2 点与线关系计算

1.点到直线距离

```
inline double dis_point_line(const Point &p,const Point s,const Point &t)
    //计算点到直线距离
{
    return fabs(det(s-p,t-p) / dist(s,t));
}
```

2. 点到线段距离

若投影不在线段内，则距离为点到端点距离

```
inline double dis_point_segment(const Point &p,const Point &s,const Point
 &t)  //计算点到线段距离
{
    if(sgn(dot(p-s,t-s)) < 0)  return norm(p-s);
    if(sgn(dot(p-t,s-t)) < 0)  return norm(p-t);
    return fabs(det(s-p,t-p) / dist(s,t));
}
```

3.判断点相对直线的方向

```
inline int point_on_line(const Point &p,const Point &s,const Point &t)
//判断p位于直线的哪一边
{
    //位于s的逆时针方向返回-1
    //位于s的顺时针方向返回1
    //在直线上返回0
    return sgn(det(p-s,t-s));
}
```

4.判断点p是否位于线段上(包括端点)

```
inline bool point_on_segment(const Point &p,const Point &s,const Point &
t)
{
    return (sgn(det(p-s,t-s)) == 0 && sgn(dot(p-s,p-t)) <= 0);
}
```

5.计算点p于直线的投影,计算两点中垂线

- 定比分点

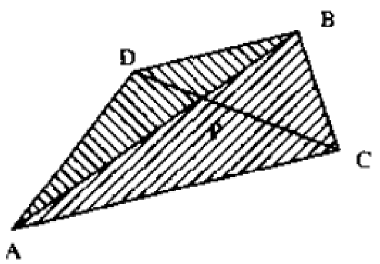
```
inline Point point_proj_line(const Point &p,const Point &s,const Point &t)
{
  double r = dot((t-s),(p-s)) / dot(t-s,t-s);
  return s + r * (t - s);
}

inline Line get_mid_line(const Point &a, const Point &b)  //计算线段ab的中垂线
{
  Point mid = (a + b) / 2;
  Point tp = b-a;
  return Line(mid,mid+Point(-tp.y,tp.x));
}
```

6.判断两直线是否平行

- 若平行，两直线向量叉乘结果为0

```
inline bool parallel(const Line &A,const Line &B)  //判断两直线是否平行
{
  return !sgn(det(A.s-A.t,B.s-B.t));
}
```

7.计算两直线交点，若存在，保存在res中



如图，如何求得直线 AB 与直线 CD 的交点P？

$$\frac{|DP|}{|CP|} = \frac{S_{\Delta ADB}}{S_{\Delta ACB}} = \frac{|\overrightarrow{AD} \times \overrightarrow{AB}|}{|\overrightarrow{AC} \times \overrightarrow{AB}|}$$

$$x_p = \frac{S_{\Delta ABD} \cdot x_C + S_{\Delta ABC} \cdot x_D}{S_{\Delta ABD} + S_{\Delta ABC}} = \frac{Area2(A,B,D) \cdot x_C - Area2(A,B,C) \cdot x_D}{Area2(A,B,D) - Area2(A,B,C)}$$

用到了定比分点公式的坐标形式

坐标形式

编辑

公式

在平面直角坐标系内，已知两点 $A(x_1,y_1)$，$B(x_2,y_2)$；在两点连线上有一点P,设它的坐标为(x,y)，且 $\vec{AP}:\vec{PB}=\lambda$，那么我们说P分有向线段 $\overline{AB}$ 的比为 λ。我们将

$$\frac{\vec{AP}}{\vec{PB}}=\lambda$$

$$x=\frac{x_1+\lambda x_2}{1+\lambda}$$

$$y=\frac{y_1+\lambda y_2}{1+\lambda}$$

称为有向线段 $\overline{AB}$ 的定比分点P的坐标公式。

当P为内分点时，$\lambda>0$；当P为外分点时，$\lambda<0(\lambda\neq-1)$；当P与A重合时，$\lambda=0$；当P与B重合时，λ 不存在。

推导过程

对 ΔOAB 及其上一点P应用定比分点公式的向量形式，即证。

```
inline bool line_make_point(const Line &A,const Line &B,Point &res)  //两直线交点
{
  if(parallel(A,B)) return false;
  double s1 = det(A.s-B.s,B.t-B.s);
  double s2 = det(A.t-B.s,B.t-B.s);
  res = (s1 * A.t - s2 * A.s) / (s1 - s2);
  return true;
}
```

9.判断直线A与线段B是否相交

```
inline int line_make_point(const Line &A,const Line &B) //判断直线A与线段B是否相交
{
  // 0表示不相交
  // 1表示规范相交
  // 2表示不规范相交(存在一个交点为端点)
  //判断两直线是否有交点时，需要先用上面的判断是否有交点，再用这个判断是否重合
  int d1,d2;
  d1 = sgn(det(A.t-A.s,B.s-A.s));
  d2 = sgn(det(A.t-A.s,B.t-A.s));
  if(d1 * d2 < 0) return 1;
  if(d1 == 0 || d2 == 0)  return 2;
  return 0;
}
```

10.判断线段A与线段B是否相交

- 进行两次跨立实验

```
inline int segment_make_point(const Line &A,const Line &B)
{
  double d1 = det(A.t-A.s,B.s-A.s),d2 = det(A.t-A.s,B.t-A.s);
  double d3 = det(B.t-B.s,A.s-B.s),d4 = det(B.t-B.s,A.t-B.s);
  if(!sgn(d1) || !sgn(d2) || !sgn(d3) || !sgn(d4)) //注意可能有不相交的情况
  {
    bool f1 = point_on_segment(A.s,B.s,B.t);
    bool f2 = point_on_segment(A.t,B.s,B.t);
    bool f3 = point_on_segment(B.s,A.s,A.t);
    bool f4 = point_on_segment(B.t,A.s,A.t);
    if(f1 || f2 || f3 || f4)  return 2; //判断某一端点是否位于另一线段上
  }
  if(sgn(d1) * sgn(d2) < 0 && sgn(d3) * sgn(d4) < 0) return 1;
  return 0;
}
```

11.极角排序

```
inline bool clk_cmp(const Point &a,const Point &b)  //顺时针极角
{
  if(!sgn(a.ang-b.ang)) return det(a,b) < 0;
  else return a.ang > b.ang;
}

inline bool l_clk_cmp(const Line &a,const Line &b)  //顺时针直线极角
{
  if(!sgn(a.ang - b.ang)) return det(a.t-a.s,b.t-a.s) >= 0; //极角相等时，位置越靠左则排越后面
  else return a.ang > a.ang;
}

inline bool aclk_cmp(const Point &a,const Point &b)  //逆时针极角
{
  if(!sgn(a.ang-b.ang)) return det(a,b) > 0;
  else return a.ang < b.ang;
}

inline bool l_aclk_cmp(const Line &a,const Line &b)  //逆时针直线极角
{
  if(!sgn(a.ang - b.ang)) return det(a.t-a.s,b.t-a.s) >= 0; //极角相等时，位置越靠右则排越前面
  else return a.ang < b.ang;
}
```

2.4 多边形

2.4.1 多边形类

- 多边形中的点应该以逆时针/顺时针排序

```
struct Polygon
{
  int n; //多边形点数
  Point a[1024];
  Polygon();
}
```

2.4.2 多边形函数

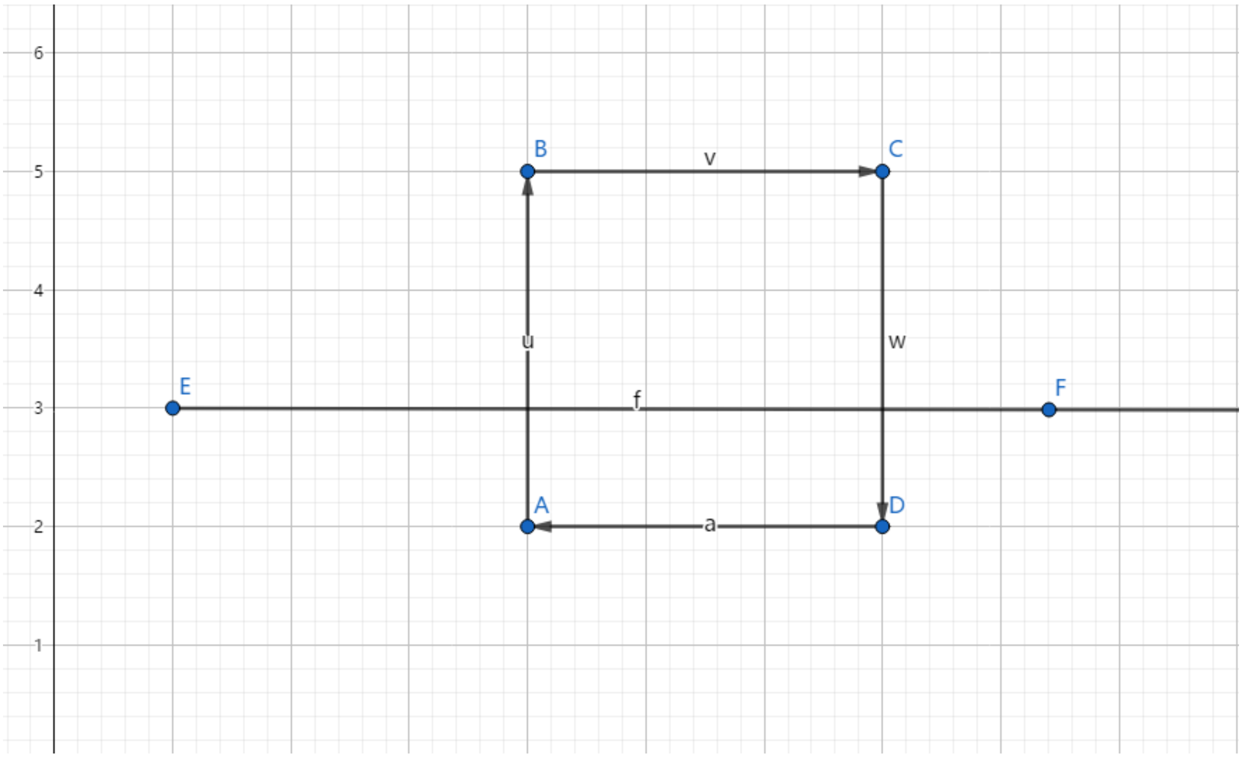
1.多边形面积

- 对于顺时针逆时针均适用

```
double area()
{
  double res = 0;
  a[n] = a[0];
  for(int i=0;i<n;i++)  res += det(a[i+1],a[i]);
  return fabs(res / 2);
}
```

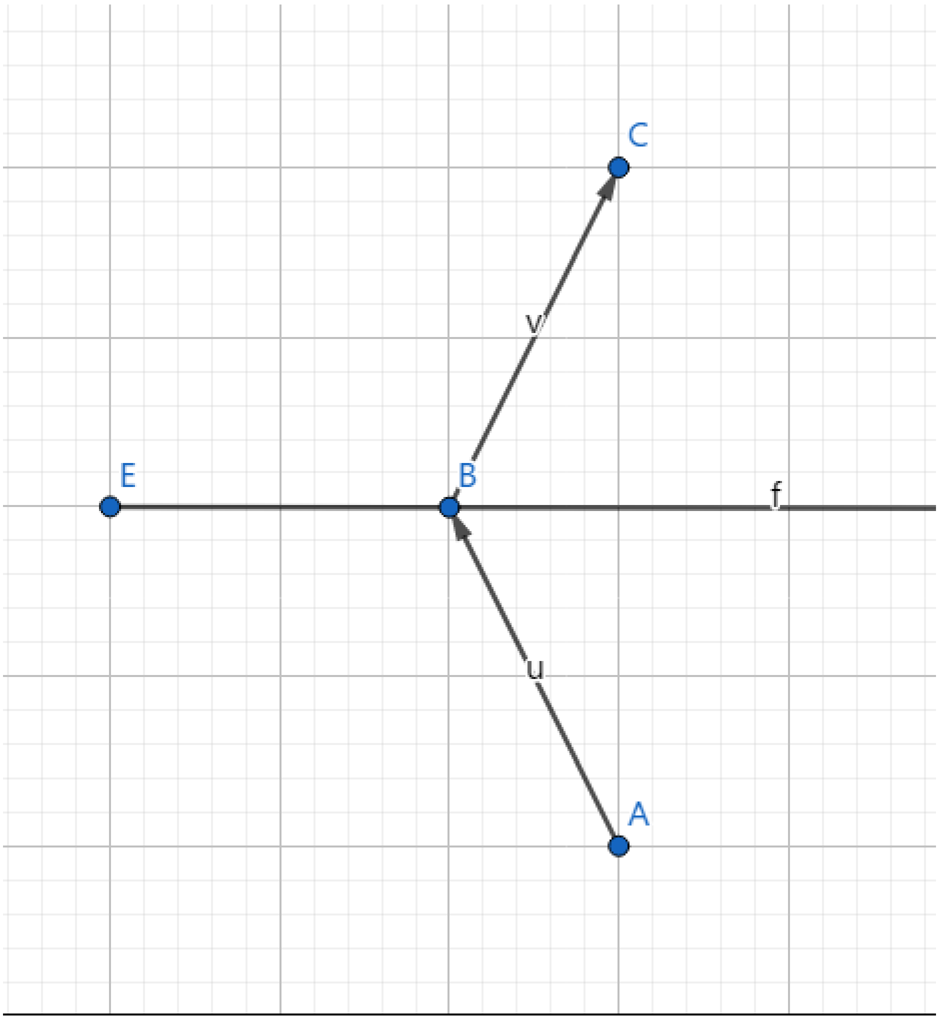
2.判断点是否在多边形内

- 采用射线法，首先按照顺时针/逆时针顺序排序多边形的顶点

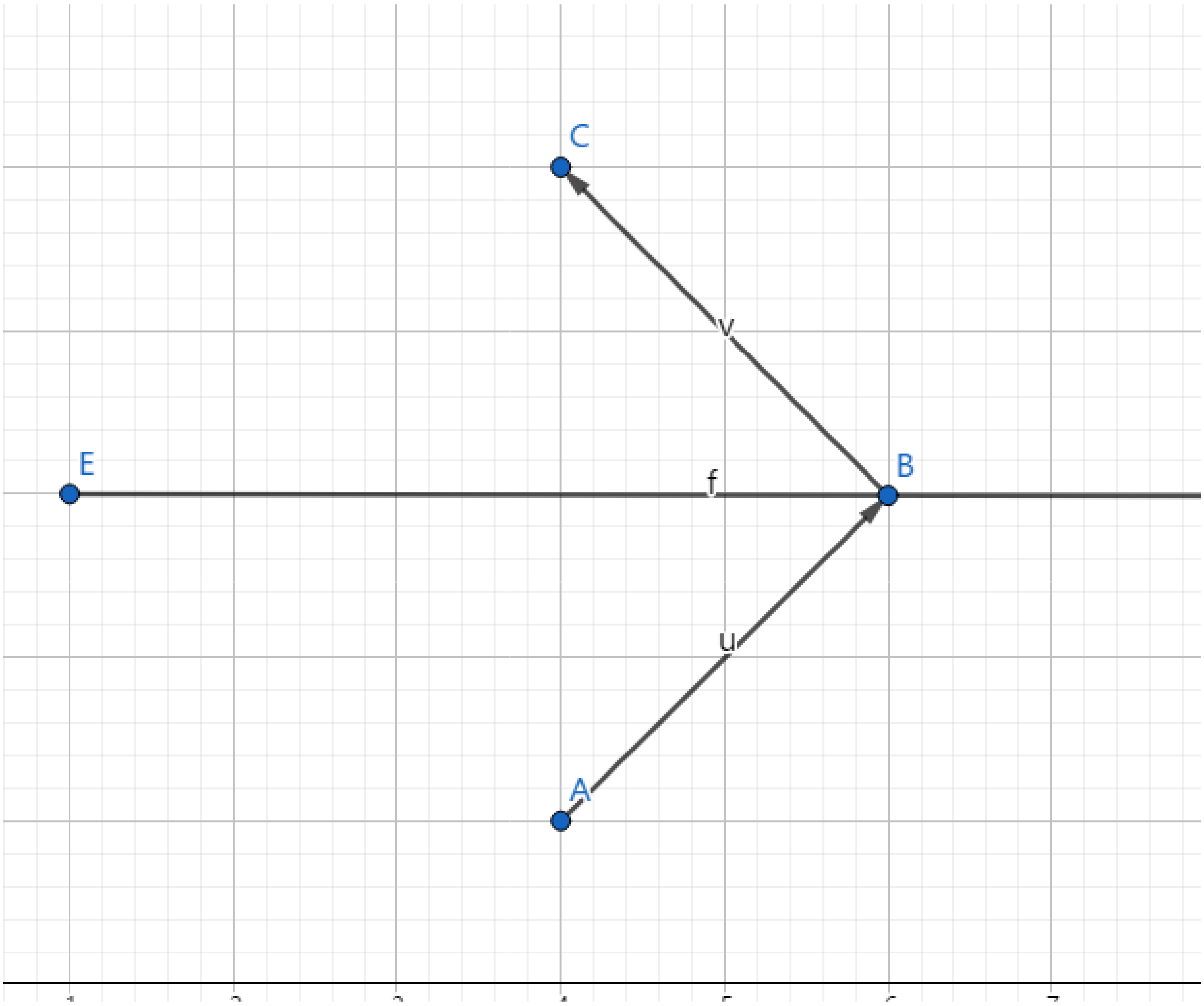


我们可以发现，对于所有被射线穿过的线段，有$k > 0$和$k < 0$两种情况，分别对应穿入和穿出。若穿入和传出的次数相等，则点在多边形外，反之。

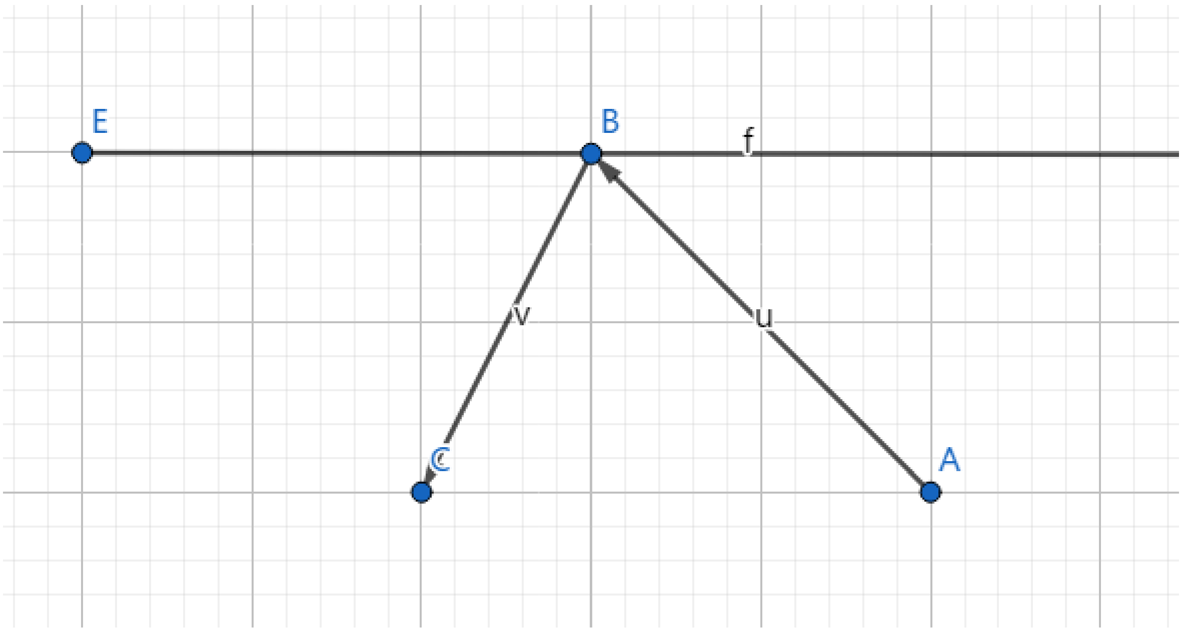
考虑四种边界情况：



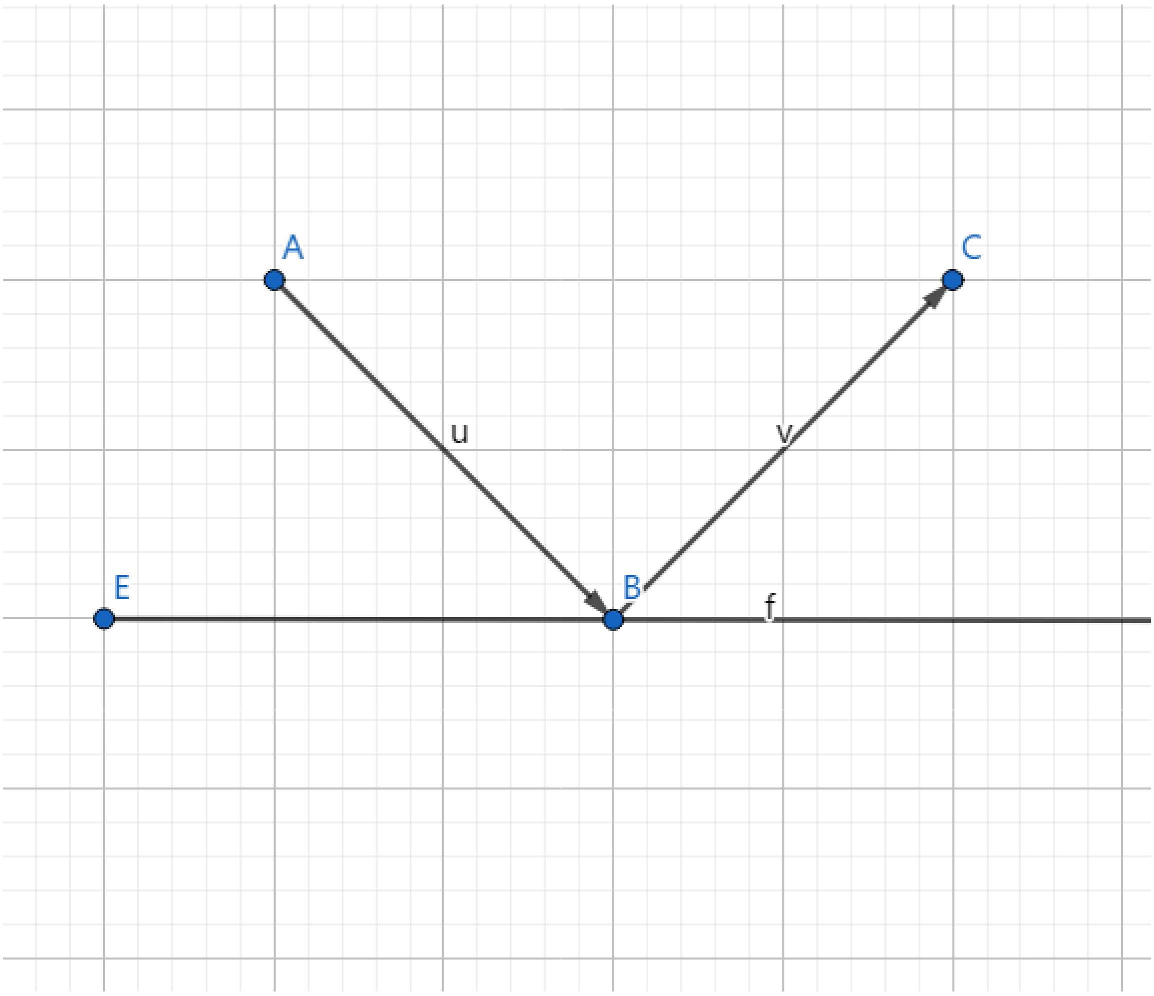
从顶点穿入



从顶点穿出



上方穿过一个顶点

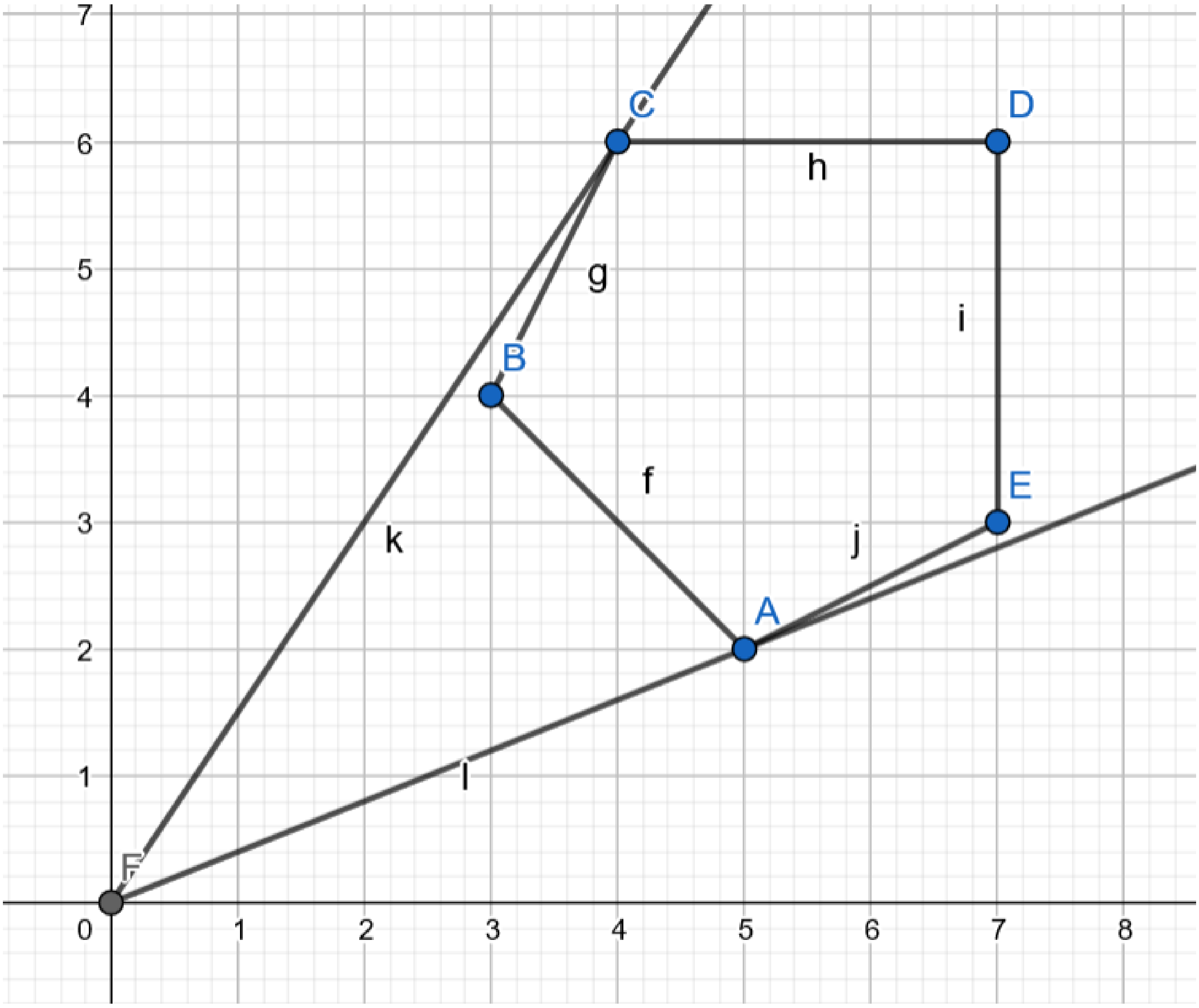


下方穿过一个顶点

四种情况都可以用d1和d2的取值来特判。

```
int point_in(Point t)  //射线法判断点是否在多边形内，0表示在多边形外，1表示在内，2表示在边界
  {
    int cnt = 0,i,d1,d2,k;
    a[n] = a[0];
    for(int i=0;i<n;i++)
    {
      if(point_on_segment(t,a[i],a[i+1])) return 2; //点在边上
      k = sgn(det(a[i+1]-a[i],t-a[i]));
      d1 = sgn(a[i].y - t.y);
      d2 = sgn(a[i+1].y - t.y);
      if(k > 0 && d1 <= 0 && d2 > 0)  cnt++;
      if(k < 0 && d2 <= 0 && d1 > 0)  cnt--;
    }
    return cnt != 0;
  }
```

3.从原点发出光源，计算多边形对于原点的张角




```
int n;
double k,h,x[maxn],y[maxn];

double calc(double x0,double y0,double x1,double y1)
{
  double a = atan2(y0,x0);
  double b = atan2(y1,x1);
  if(b - a > pi) a += 2 * pi;   //顺时针扫过x负半轴
  if(a - b > pi) b += 2 * pi;   //逆时针扫过x负半轴
  return a - b;
}

int main()
{
  scanf("%lf%lf%d",&k,&h,&n);
  for(int i=0;i<n;i++)  scanf("%lf%lf",&x[i],&y[i]);
  x[n] = x[0],y[n] = y[0];
  double summ = 0,maxx = 0,minn = 0;
  for(int i=0;i<n;i++) //规定方向，遍历每一条边
  {
    double tmp = calc(x[i],y[i],x[i+1],y[i+1]);
    summ += tmp;
    if(summ < minn) minn = summ;
    if(summ > maxx) maxx = summ;
```

```
    if(maxx - minn >= 2 * pi)
    {
      maxx = minn + 2 * pi;
      break;
    }
  }
  printf("%.2f\n",k * h * (maxx - minn));
  return 0;
}
```

2.5 凸多边形

2.5.1 凸多边形函数

1.Graham扫描法求凸包 O(nlogn)

1. 将点按x坐标从小到大排序，那么最左侧的点和最右侧的点一定在凸包中
2. 从左侧点开始，扫出凸包下半部分
3. 从右侧点开始，扫出凸包上半部分

```
Polygon_convex convec_hull(vector<Point> a)  //用a中的点求凸包
{
  Polygon_convex res(2 * a.size() + 5);
  sort(a.begin(),a.end(),less_cmp);
  a.erase(unique(a.begin(),a.end()),a.end());
  int m = 0;
  for(int i=0;i<int(a.size());i++)
  {
    while(m > 1 && sgn(det(res.p[m - 1] - res.p[m - 2], a[i] - res.p[m -
2])) >= 0) m--; //回溯栈内不合法点
    res.p[m++] = a[i];
  }
  int k = m;
  for(int i=int(a.size())-2;i>=0;i--)
  {
    while(m > k && sgn(det(res.p[m - 1] - res.p[m-2], a[i] - res.p[m-2]))
 >= 0) m--;
    res.p[m++] = a[i];
  }
  res.p.resize(m);
  if(a.size() > 1)  res.p.resize(m - 1);
  return res;
}
```

2.O(logn)判断点在凸包内

- 将凸包划分为几个三角形，二分判断点是否在三角形中

```
  bool point_in_convex(const Point &b)  //判断点b是否在凸包a内部，a必须为凸包，
且p[0]为最左侧点
  {
    if(sgn(det(b - a[0], a[1] - a[0])) > 0 || sgn(det(a[n-1] - a[0], b -
 a[0]) > 0))  return 0;
    int l = 1, r = n - 2, mid, res = 1;
    while(l <= r)
    {
      mid = (l + r) >> 1;
      if(sgn(det(a[mid] - a[0],b - a[0])) > 0)  {res = mid;l = mid + 1;}
      else r = mid - 1;
    }
    return sgn(det(a[res + 1] - a[res],b - a[res])) >= 0;
  }
```

3.1 点线综合

```
inline int sgn(double x)
{
  if(fabs(x) < eps) return 0;
  else return x > 0 ? 1 : -1;
}

inline int fcmp(double x,double y) //浮点比较
{
  if(fabs(x-y) < eps) return 0;
  else return x > y ? 1 : -1;
}

inline double sqr(double x) //浮点平方
{
  return x * x;
}
inline double Sqrt(double x)
{
  return x <= 0 ? 0 : sqrt(x);
}
inline int abs(int x)
{
  return x >= 0 ? x : -x;
}

struct Point
{
  double x, y;
  Point(){}
```

```
    Point(double a, double b): x(a), y(b) {}
    friend Point operator + (const Point &a, const Point &b){
      return Point(a.x + b.x, a.y + b.y);
    }
    friend Point operator - (const Point &a, const Point &b){
      return Point(a.x - b.x, a.y - b.y);
    }
    friend bool operator == (const Point &a, const Point &b){
      return fcmp(a.x,b.x) == 0 && fcmp(a.y,b.y) == 0;
    }
    friend Point operator * (const Point &a, const double &b){
      return Point(a.x * b, a.y * b);
    }
    friend Point operator * (const double a,const Point &b){
      return Point(a * b.x, a * b.y);
    }
    friend Point operator / (const Point &a, const double &b){
      return Point(a.x / b, a.y / b);
    }
    friend bool operator < (const Point &a,const Point &b){
      if(a.x == b.x)  return a.y < b.y;
      return a.x < b.x;
    }
    friend double norm(const Point &a){
      return sqrt(sqr(a.x) + sqr(a.y));
    }
  };

  inline double dot(const Point &a,const Point &b)  //内积
  {
    return a.x * b.x + a.y * b.y;
  }
  inline double det(const Point &a,const Point &b)  //外积
  {
    return a.x * b.y - a.y * b.x;
  }
  inline double dist(const Point &a,const Point &b)  //两点间距离
  {
    return norm((b - a));
  }
  inline double angle(const Point &a,const Point &b)  //向量夹角
  {
    return acos(dot(a,b) / norm(a) / norm(b));
  }
  inline Point rotate_point(const Point &p,double k)  //计算点p绕原点逆时针旋转
  k(弧度)
  {
    return Point(p.x * cos(k) - p.y * sin(k),p.x * sin(k) - p.y * cos(k));
  }

  //直线线段类
  struct Line
```

```
{
  Point s,t;
  Line(){};
  Line(Point a,Point b) : s(a) , t(b) {}
  Line(Point p,double ang) //根据一个点和一个角(弧度制, 0 <= ang < pi) 确定直线
  {
    s = p;
    if(sgn(ang - pi / 2) == 0)  t = (s + Point(0,1));
    else t = s + Point(1,tan(ang));
  }
  Line(double a,double b,double c)  // ax + by + c = 0
  {
    if(sgn(a) == 0){s = Point(0,-c/b); t = Point(1,-c/b);}
    else if(sgn(b) == 0){s = Point(-c/a,0); t = Point(-c/a,1);}
    else{s = Point(0,-c/b); t = Point(1,(-c-a)/b);}
  }
};

inline double dis_point_line(const Point &p,const Point s,const Point &t)
  //计算点到直线距离
{
  return fabs(det(s-p,t-p) / dist(s,t));
}

inline double dis_point_segment(const Point &p,const Point &s,const Point
 &t)  //计算点到线段距离
{
  if(sgn(dot(p-s,t-s)) < 0)  return norm(p-s);
  if(sgn(dot(p-t,s-t)) < 0)  return norm(p-t);
  return fabs(det(s-p,t-p) / dist(s,t));
}

inline int point_on_line(const Point &p,const Point &s,const Point &t)
//判断p位于直线的哪一边
{
  //位于上侧返回-1
  //位于下侧返回1
  //在直线上返回0
  return sgn(det(p-s,t-s));
}

inline bool point_on_segment(const Point &p,const Point &s,const Point &
t)  //判断p是否位于线段上(包括端点)
{
  return (sgn(det(p-s,t-s)) == 0 && sgn(dot(p-s,p-t)) <= 0);
}

inline Point point_proj_line(const Point &p,const Point &s,const Point &
t)  //计算点p于直线的投影
{
  double r = dot((t-s),(p-s)) / dot(t-s,t-s);
  return s + r * (t - s);
```

```
}

inline Line get_mid_line(const Point &a, const Point &b)  //计算线段ab的中垂线
{
  Point mid = (a + b) / 2;
  Point tp = b-a;
  return Line(mid,mid+Point(-tp.y,tp.x));
}

inline bool parallel(const Line &A,const Line &B)  //判断两直线是否平行
{
  return !sgn(det(A.s-A.t,B.s-B.t));
}

inline bool line_make_point(const Line &A,const Line &B,Point &res)  //两直线交点
{
  if(parallel(A,B)) return false;
  double s1 = det(A.s-B.s,B.t-B.s);
  double s2 = det(A.t-B.s,B.t-B.s);
  res = (s1 * A.t - s2 * A.s) / (s1 - s2);
  return true;
}

inline int line_make_point(const Line &A,const Line &B) //判断直线A与线段B是否相交
{
  // 0表示不相交
  // 1表示规范相交
  // 2表示不规范相交(存在一个交点为端点)
  //判断两直线是否有交点时，需要先用上面的判断是否有交点，再用这个判断是否重合
  int d1,d2;
  d1 = sgn(det(A.t-A.s,B.s-A.s));
  d2 = sgn(det(A.t-A.s,B.t-A.s));
  if(d1 * d2 < 0) return 1;
  if(d1 == 0 || d2 == 0)  return 2;
  return 0;
}

inline int segment_make_point(const Line &A,const Line &B)
{
  double d1 = det(A.t-A.s,B.s-A.s),d2 = det(A.t-A.s,B.t-A.s);
  double d3 = det(B.t-B.s,A.s-B.s),d4 = det(B.t-B.s,A.t-B.s);
  if(!sgn(d1) || !sgn(d2) || !sgn(d3) || !sgn(d4)) //注意可能有不相交的情况
  {
    bool f1 = point_on_segment(A.s,B.s,B.t);
    bool f2 = point_on_segment(A.t,B.s,B.t);
    bool f3 = point_on_segment(B.s,A.s,A.t);
    bool f4 = point_on_segment(B.t,A.s,A.t);
    if(f1 || f2 || f3 || f4)  return 2; //判断某一端点是否位于另一线段上
  }
```

```
    if(sgn(d1) * sgn(d2) < 0 && sgn(d3) * sgn(d4) < 0) return 1;
    return 0;
  }

  inline bool clk_cmp(const Point &a,const Point &b)  //顺时针极角
  {
    double x = atan2(a.y,a.x),y = atan2(b.y,b.x);
    if(!sgn(x-y)) return det(a,b) < 0;
    else return x > y;
  }

  inline bool clk_cmp(const Line &a,const Line &b)  //顺时针直线极角
  {
    Point ta = a.t - a.s,tb = b.t - b.s;
    double x = atan2(ta.y,ta.x),y = atan2(tb.y,tb.x);
    if(!sgn(x - y)) return det(ta,tb) < 0;
    else return x > y;
  }

  inline bool aclk_cmp(const Point &a,const Point &b)  //逆时针极角s
  {
    double x = atan2(a.y,a.x),y = atan2(b.y,b.x);
    if(!sgn(x-y)) return det(a,b) > 0;
    else return x < y;
  }

  inline bool aclk_cmp(const Line &a,const Line &b)  //逆时针直线极角
  {
    Point ta = a.t - a.s,tb = b.t - b.s;
    double x = atan2(ta.y,ta.x),y = atan2(tb.y,tb.x);
    if(!sgn(x - y)) return det(ta,tb) > 0;
    else return x < y;
  }
```

4 完整代码(实时更新)

```
/*
/*
注意输出%.2f是否要加 eps !!
注意数组的数据范围 !!
注意 -0.0 的情况 !!
注意多边形上的三点共线
*/

#include<bits/stdc++.h>
#define pi 3.1415926535898
using namespace std;
typedef long long LL;
const double eps = 1e-8;
```

```cpp
const int N = 1e4 + 10; //预设点的数量，注意更改
inline LL gcd(LL a,LL b){ return b==0? a: gcd(b,a%b); }

inline int sgn(double x)
{
  if(fabs(x) < eps) return 0;
  else return x > 0 ? 1 : -1;
}

inline int fcmp(double x,double y) //浮点比较
{
  if(fabs(x-y) < eps) return 0;
  else return x > y ? 1 : -1;
}

inline double sqr(double x) //浮点平方
{
  return x * x;
}
inline double Sqrt(double x)
{
  return x <= 0 ? 0 : sqrt(x);
}


/********************************
             点相关
********************************/

struct Point
{
  double x, y, ang;
  Point(){}
  Point(double a, double b): x(a), y(b) {}
  friend Point operator + (const Point &a, const Point &b){
    return Point(a.x + b.x, a.y + b.y);
  }
  friend Point operator - (const Point &a, const Point &b){
    return Point(a.x - b.x, a.y - b.y);
  }
  friend bool operator == (const Point &a, const Point &b){
    return fcmp(a.x,b.x) == 0 && fcmp(a.y,b.y) == 0;
  }
  friend Point operator * (const Point &a, const double &b){
    return Point(a.x * b, a.y * b);
  }
  friend Point operator * (const double a,const Point &b){
    return Point(a * b.x, a * b.y);
  }
  friend Point operator / (const Point &a, const double &b){
    return Point(a.x / b, a.y / b);
  }
```

```
  friend bool operator < (const Point &a,const Point &b){
    if(a.x == b.x)  return a.y < b.y;
    return a.x < b.x;
  }
  friend double norm(const Point &a){
    return sqrt(sqr(a.x) + sqr(a.y));
  }
  void calcangle() {ang = atan2(y,x);}
};

inline double dot(const Point &a,const Point &b)  //内积
{
  return a.x * b.x + a.y * b.y;
}
inline double det(const Point &a,const Point &b)  //外积
{
  return a.x * b.y - a.y * b.x;
}
inline double dist(const Point &a,const Point &b)  //两点间距离
{
  return norm((b - a));
}

inline Point trunc(const Point &a,double r)  //缩放向量长度到r
{
  double l = norm(a);
  if(!sgn(l)) return a;
  r /= l;
  return Point(a.x * r,a.y * r);
}

inline double angle(const Point &a,const Point &b)  //向量
{
  return acos(dot(a,b) / norm(a) / norm(b));
}

inline Point rotate_left(const Point &a)  //逆时针旋转90度
{
    return Point(-a.y,a.x);
}

inline Point rotate_right(const Point &a) //顺时针旋转90度
{
    return Point(a.y,-a.x);
}

inline Point rotate_point(const Point &p,double k)  //计算点p绕原点逆时针旋转
k(弧度)
{
  return Point(p.x * cos(k) - p.y * sin(k),p.x * sin(k) - p.y * cos(k));
}
```

```
/*******************************
            点线相关
*******************************/

//定比分点公式:
/*
  存在向量 AB
  向量上存在一点 P,满足AP = kPB
  则 P 的坐标为(xA + kxB / (1 + k), yA + kyB / (1 + k) )
  若 k 为负数,则 P 为外比分点
*/

//直线线段类
struct Line
{
  Point s, t;
  double ang; //直线极角
  Line(){};
  Line(Point a,Point b) : s(a) , t(b) {}
  Line(Point p,double ang) //根据一个点和一个角(弧度制, 0 <= ang < pi) 确定直线
  {
    s = p;
    if(sgn(ang - pi / 2) == 0)  t = (s + Point(0,1));
    else t = s + Point(1,tan(ang));
  }
  Line(double a,double b,double c)  // ax + by + c = 0
  {
    if(sgn(a) == 0){s = Point(0,-c/b); t = Point(1,-c/b);}
    else if(sgn(b) == 0){s = Point(-c/a,0); t = Point(-c/a,1);}
    else{s = Point(0,-c/b); t = Point(1,(-c-a)/b);}
  }
  void calcangle()  {ang = atan2((t - s).y, (t - s).x);}  //计算直线极角
};

inline double dis_point_line(const Point &p,const Line &l)  //计算点到直线距
离
{
  return fabs(det(l.s-p,l.t-p) / dist(l.s,l.t));
}

inline double dis_point_segment(const Point &p,const Line &l)  //计算点到线
段距离
{
  if(sgn(dot(p-l.s,l.t-l.s)) < 0)  return norm(p-l.s);
  if(sgn(dot(p-l.t,l.s-l.t)) < 0)  return norm(p-l.t);
  return fabs(det(l.s-p,l.t-p) / dist(l.s,l.t));
}

inline int point_on_line(const Point &p,const Line &l)  //判断p位于直线的哪
一边
{
```

```
    //位于上侧返回-1
    //位于下侧返回1
    //在直线上返回0
    return sgn(det(p-l.s,l.t-l.s));
}

inline bool point_on_segment(const Point &p,const Line &l)  //判断p是否位于线段上(包括端点)
{
    return (sgn(det(p-l.s,l.t-l.s)) == 0 && sgn(dot(p-l.s,p-l.t)) <= 0);
}

inline Point point_proj_line(const Point &p,const Line &l)  //计算点p于直线的投影
{
    double r = dot((l.t-l.s),(p-l.s)) / dot(l.t-l.s,l.t-l.s);
    return l.s + r * (l.t - l.s);
}

inline Line get_mid_line(const Point &a,const Point &b)  //计算线段ab的中垂线
{
    Point mid = (a + b) / 2;
    Point tp = b-a;
    return Line(mid,mid+Point(-tp.y,tp.x));
}

inline Line line_trans(const Line &a,const double b)  //直线向左侧平移 b 的距离
{
    Point d = rotate_left(a.t-a.s) / norm(a.t-a.s) * b;
    return Line(a.s+d,a.t+d);
}

inline bool parallel(const Line &A,const Line &B)  //判断两直线是否平行
{
    return !sgn(det(A.s-A.t,B.s-B.t));
}

inline bool line_make_point(const Line &A,const Line &B,Point &res)  //两直线交点
{
    if(parallel(A,B)) return false;
    double s1 = det(A.s-B.s,B.t-B.s);
    double s2 = det(A.t-B.s,B.t-B.s);
    res = (s1 * A.t - s2 * A.s) / (s1 - s2);
    return true;
}

inline int line_make_point(const Line &A,const Line &B) //判断直线A与线段B是否相交
{
```

```
  // 0表示不相交
  // 1表示规范相交
  // 2表示不规范相交(存在一个交点为端点)
  //判断两直线是否有交点时，需要先用上面的判断是否有交点，再用这个判断是否重合
  int d1,d2;
  d1 = sgn(det(A.t-A.s,B.s-A.s));
  d2 = sgn(det(A.t-A.s,B.t-A.s));
  if(d1 * d2 < 0) return 1;
  if(d1 == 0 || d2 == 0)  return 2;
  return 0;
}

inline int segment_make_point(const Line &A,const Line &B)
{
  double d1 = det(A.t-A.s,B.s-A.s),d2 = det(A.t-A.s,B.t-A.s);
  double d3 = det(B.t-B.s,A.s-B.s),d4 = det(B.t-B.s,A.t-B.s);
  if(!sgn(d1) || !sgn(d2) || !sgn(d3) || !sgn(d4)) //注意可能有不相交的情况
  {
    bool f1 = point_on_segment(A.s,Line(B.s,B.t));
    bool f2 = point_on_segment(A.t,Line(B.s,B.t));
    bool f3 = point_on_segment(B.s,Line(A.s,A.t));
    bool f4 = point_on_segment(B.t,Line(A.s,A.t));
    if(f1 || f2 || f3 || f4)  return 2; //判断某一端点是否位于另一线段上
  }
  if(sgn(d1) * sgn(d2) < 0 && sgn(d3) * sgn(d4) < 0) return 1;
  return 0;
}

inline bool clk_cmp(const Point &a,const Point &b)  //顺时针极角
{
  if(!sgn(a.ang-b.ang)) return det(a,b) < 0;
  else return a.ang > b.ang;
}

inline bool l_clk_cmp(const Line &a,const Line &b)  //顺时针直线极角
{
  if(!sgn(a.ang - b.ang)) return det(a.t-a.s,b.t-a.s) >= 0; //极角相等时，位
置越靠左则排越后面
  else return a.ang > a.ang;
}

inline bool aclk_cmp(const Point &a,const Point &b)  //逆时针极角
{
  if(!sgn(a.ang-b.ang)) return det(a,b) > 0;
  else return a.ang < b.ang;
}

inline bool l_aclk_cmp(const Line &a,const Line &b)  //逆时针直线极角
{
  if(!sgn(a.ang - b.ang)) return det(a.t-a.s,b.t-a.s) >= 0; //极角相等时，位
置越靠右则排越前面
  else return a.ang < b.ang;
```

```
}

/*******************************
            多边形相关
*******************************/

struct Polygon  //多边形类，点需要逆时针排序
{
  int n; //多边形点数
  Point a[N];
  Polygon(){};

  double area()  //返回多边形的有向面积,未除2
  {
    double res = 0;
    a[n] = a[0];
    for(int i=0;i<n;i++)  res += det(a[i],a[i+1]);
    return res;
  }

  int point_in(Point t)  //射线法判断点是否在多边形内，0表示在多边形外，1表示在
内，2表示在边界
  {
    int cnt = 0,i,d1,d2,k;
    a[n] = a[0];
    if(point_on_segment(t,Line(a[i],a[i+1]))) return 2; //点在边上
    k = sgn(det(a[i+1]-a[i],t-a[i]));
    d1 = sgn(a[i].y - t.y);
    d2 = sgn(a[i+1].y - t.y);
    if(k > 0 && d1 <= 0 && d2 > 0)  cnt++;
    if(k < 0 && d2 <= 0 && d1 > 0)  cnt--;
    return cnt != 0;
  }

  int border_int_point_num()  //计算多边形边界格点数目，已测试
  {
    int res = 0;
    a[n] = a[0];
    for(int i=0;i<n;i++)
       res += gcd(abs(int(a[i + 1].x - a[i].x)), abs(int(a[i + 1].y - a
[i].y)));
    return res;
  }

  int Inside_Int_Point_Num() // 多边形内的格点个数，已测试
  {
    return (abs(int(area())) + 2 - border_int_point_num()) / 2;
  }

  Point mass_center()  //返回多边形重心
  {
    Point res = Point(0,0);
```

```
    if(sgn(area()) == 0)  return res;
    a[n] = a[0];
    for(int i=0;i<n;i++)  res = res + (a[i] + a[i+1]) * det(a[i],a[i+1]);
    return res / area() / 6.0;
  }

  bool is_convex() //判断多边形是否为凸
  {
    bool s[3];
    memset(s,0,sizeof(s));
    a[n] = a[0],a[n+1] = a[1];
    for(int i=0;i<n;i++)
    {
      s[sgn(det(a[i+1]-a[i],a[i+2]-a[i])) + 1] = true;
      if(s[0] && s[2])  return false;
    }
    return true;
  }

  bool point_in_convex(const Point &b)  //判断点b是否在凸包a内部，a必须为凸包，且p[0]为最左侧点
  {
    if(sgn(det(b - a[0], a[1] - a[0])) > 0 || sgn(det(a[n-1] - a[0], b - a[0]) > 0))  return 0;
    int l = 1, r = n - 2, mid, res = 1;
    while(l <= r)
    {
      mid = (l + r) >> 1;
      if(sgn(det(a[mid] - a[0],b - a[0])) > 0)  {res = mid;l = mid + 1;}
      else r = mid - 1;
    }
    return sgn(det(a[res + 1] - a[res],b - a[res])) >= 0;
  }
};

bool less_cmp(const Point &a,const Point &b) //x坐标小优先，y坐标小其次
{
  if(sgn(a.x-b.x) == 0) return sgn(a.y-b.y) < 0;
  else return sgn(a.x - b.x) < 0;
}

Polygon convex_hull(Point *a,int n)  //用a中的点求凸包,n = 1 和 n = 2 应特判，已测试
{
  Polygon res;
  sort(a,a+n,less_cmp);
  unique(a,a+n);
  int m = 0;
  for(int i=0;i<n;i++)
  {
    while(m > 1 && sgn(det(res.a[m - 1] - res.a[m - 2], a[i] - res.a[m - 2])) <= 0) --m;
```

```
    //回溯栈内不合法点，注意可能有凸包上三点共线的情况，此时改成小于号则保留这种点
    res.a[m++] = a[i];
  }
  int k = m;
  for(int i=n-2;i>=0;i--)
  {
    while(m > k && sgn(det(res.a[m - 1] - res.a[m-2], a[i] - res.a[m-2]))
 <= 0) --m;
    res.a[m++] = a[i];
  }
  res.n = m > 1 ? m - 1 : m;
  return res;
}

/*******************************
            半平面相关
*******************************/

//单个半平面用Line类，有效空间是向量 s -> t 的逆时针方向(左侧)

Polygon convex_cut(const Polygon &con,const Line &hp)  //返回凸多边形与半平面
的交
//凸多边形逆时针，在半平面左侧
//返回的凸多边形内可能没有点，也可能是一个点或者一条线，注意特判
{
  Polygon res;
  int n = con.n;
  for(int i=0;i<n;i++)
  {
    Point tmp,p1 = con.a[i],p2 = con.a[(i + 1) % n];
    int d1 = sgn(det(hp.t - hp.s, p1 - hp.s)), d2 = sgn(det(hp.t - hp.s,
 p2 - hp.s));
    if(d1 >= 0) res.a[res.n++] = p1;
    if(d1 * d2 < 0)
    {
      line_make_point(Line(p1,p2),hp,tmp);
      res.a[res.n++] = tmp;
    }
  }
  return res;
}

struct Halfplanes  //半平面集合
{
  int n;
  Line hp[N];
  Point p[N];
  int que[N];
  int st,ed;

  Halfplanes()  {n = st = ed = 0;}
  void push(Line a) {hp[n++] = a;}
```

```
  void clear()  {n = st = ed = 0;}
  void unique() //在去重之前要保证极角有序
  {
    int m = 0;
    for(int i=0;i<n-1;i++)
    {
      if(sgn(hp[i].ang-hp[i+1].ang) == 0) continue;
      hp[m++] = hp[i];
    }
    hp[m++] = hp[n-1];
    n = m;
  }

  bool halfplane_intersection() //半平面交，需要保证点按照逆时针排列!
  //通过测试: POJ-3335,POJ-3130
  {
    if(n < 3) return false; //三个以上半平面才能组成多边形
    for(int i=0;i<n;i++)  hp[i].calcangle();
    sort(hp,hp+n,l_aclk_cmp);unique();
    que[st = 0] = 0,que[ed = 1] = 1;
    if(parallel(hp[0],hp[1])) return false;
    line_make_point(hp[0],hp[1],p[1]);
    for(int i=2;i<n;i++)
    {
      while(st < ed && sgn(det(hp[i].t - hp[i].s, p[ed] - hp[i].s) < 0))
 ed--; //上一个交点在新直线的右侧
      while(st < ed && sgn(det(hp[i].t - hp[i].s, p[st + 1] - hp[i].s)) <
 0)  st++;
      que[++ed] = i;
      if(parallel(hp[i],hp[que[ed-1]])) return false;
      line_make_point(hp[i],hp[que[ed-1]],p[ed]);
    }
    while(st < ed && sgn(det(hp[que[st]].t - hp[que[st]].s, p[ed] - hp[qu
e[st]].s)) < 0)  ed--; //维护前后合理性
    while(st < ed && sgn(det(hp[que[ed]].t - hp[que[ed]].s, p[st+1] - hp
[que[ed]].s)) < 0)  st++;
    if(st + 1 >= ed)  return false;
    return true;
  }

  void get_convex(Polygon &con)  //需要先调用 halfplane_intersection(),并返
回 true
  //通过测试: POJ-1279
  {
    line_make_point(hp[que[st]],hp[que[ed]],p[st]);
    con.n = ed - st + 1;
    for(int j=st,i=0;j<=ed;i++,j++) con.a[i] = p[j];
  }
};

/*********************************
            圆平面相关
```

```
*******************************/
struct Circle
{
  Point p;
  double r;
  Circle(){};
  Circle(Point _p,double _r) {p = _p;r = _r;}
  Circle(double x,double y,double _r) {p = Point(x,y);r = _r;}
  Circle(Point a,Point b,Point c) //三角形外接圆
  {
    Line l3 = get_mid_line(a,b),l4 = get_mid_line(a,c);
    line_make_point(l3,l4,p);
    r = dist(p,a);
  }
  bool operator == (const Circle &a) const{
    return (p == a.p) && sgn(r - a.r) == 0;
  }
  bool operator < (const Circle &a)  const{
    return (p < a.p) || ((p == a.p) && (sgn(r - a.r)) < 0);
  }
  double area() {return 2 * pi * r;}
};

int circle_point_relation(const Circle &a,const Point &b) //返回点和圆的关系
{
  //0:圆外 1:圆上 2:圆内
  double l = dist(b,a.p);
  if(sgn(l - a.r) < 0)  return 2;
  else if(sgn(l - a.r) == 0)  return 1;
  else return 0;
}

int circle_line_relation(const Circle &a,const Line &b)
{
  //比较圆的半径于圆心到直线的距离关系
  //2:交于圆 1:切于圆 0:与圆无交点
  double d = dis_point_line(a.p,b);  //圆心到直线的距离
  if(sgn(d - a.r) < 0)  return 2;
  else if(sgn(d - a.r) == 0)  return 1;
  return 0;
}

int circle_seg_relation(const Circle &a,const Line &b)
{
  //比较圆的半径于圆心到线段的距离关系
  double d = dis_point_segment(a.p,b);  //圆心到线段的距离
  if(sgn(d - a.r) < 0)  return 2;
  else if(sgn(d - a.r) == 0)  return 1;
  return 0;
}

int circle_circle_relation(const Circle &a,const Circle &b)  //返回两圆关系
```

```
{
  //5:相离 4:相切 3:相交 2:内切 1:内含
  double d = dist(a.p,b.p);
  if(sgn(d - a.r - b.r) > 0)  return 5;
  if(sgn(d - a.r - b.r) == 0)  return 4;
  double l = fabs(a.r - b.r);
  if(sgn(d - a.r - b.r) < 0 && sgn(d - l) > 0)  return 3;
  if(sgn(d - l) == 0) return 2;
  if(sgn(d - l) < 0)  return 1;
  return 0;
}

int circle_make_point(const Circle &a,const Circle &b,Point &p1,Point &p
2)
{
  //两圆交点，返回交点数量
  //UVA 12304
  int rel = circle_circle_relation(a,b);
  if(rel == 1 || rel == 5)  return 0;
  double d = dist(a.p,b.p);
  double l = (d * d + a.r * a.r - b.r * b.r) / (2 * d);  //一交点到圆心连线
投影到一圆心的距离
  double h = Sqrt(a.r * a.r - l * l);  //一交点到圆心连线的距离
  Point tmp = a.p + trunc(b.p - a.p,l); //一交点到圆心连线的投影
  p1 = tmp + trunc(rotate_left(b.p - a.p),h);
  p2 = tmp + trunc(rotate_right(b.p - a.p),h);
  if(rel == 2 || rel == 4)  return 1;
  return 2;
}

int circle_cross_line(const Circle &a,const Line &b,Point &p1,Point &p2)
{
   //圆交直线,返回交点数量
   //注意 p1 是 ab 方向延伸出去与圆的交点, p2 是 ba 方向延伸出去与圆的交点
  if(!circle_line_relation(a,b))  return 0;
  Point tmp = point_proj_line(a.p,b);  //圆心到直线的投影
  double d = dist(a.p,tmp); //圆心到直线的距离
  if(sgn(d) == 0) {p1 = tmp,p2 = tmp;return 1;} //直线与圆相切
  p1 = tmp + trunc(b.t-b.s,d);
  p2 = tmp - trunc(b.t-b.s,d);
  return 2;
}

int point_get_circle(const Point &p1,const Point &p2,double r,Circle &a,C
ircle &b)
{
  //得到过p1,p2两点，半径为r的两个圆，返回数量
  Circle t1(p1,r),t2(p2,r);
  int t = circle_make_point(t1,t2,a.p,b.p);
  if(!t)  return 0;
  a.r = b.r = r;
  return t;
```

```
}

int tagent_line(const Circle &a,const Point &b,Line &l1,Line &l2)
{
  //过一点做圆的两条切线
  int t = circle_point_relation(a,b);
  if(t == 2)  return 0;
  if(t == 1)  {l1 = Line(b,b + rotate_left((b - a.p)));l2 = l1;}
  double d = dist(a.p,b);
  double l = a.r * a.r / d;
  double h = Sqrt(a.r * a.r - l * l);
  l1 = Line(b,a.p + trunc(b - a.p,l) + trunc(rotate_left(b - a.p),h));
  l2 = Line(b,a.p + trunc(b - a.p,l) + trunc(rotate_right(b - a.p),h));
  return 2;
}

double circle_cross_circle(Circle &a,Circle &b) //求圆交面积
{
  //两个扇形减四边形
  int rel = circle_circle_relation(a,b);
  if(rel >= 4)  return 0.0;
  if(rel <= 2)  return min(a.area(),b.area());
  double d = dist(a.p,b.p);
  double hf = (a.r + b.r + d) / 2.0;
  double ss = 2 * Sqrt(hf * (hf - a.r) * (hf - b.r) * (hf - d));
  double a1 = acos((a.r * a.r + d * d - b.r * b.r) / (2.0 * a.r * d));
  a1 = a1 * a.r * a.r;
  double a2 = acos((b.r * b.r + d * d - a.r * a.r) / (2.0 * b.r * d));
  a2 = a2 * b.r * b.r;
  return a1 + a2 - ss;
}

double circle_cross_triangle(const Circle &c,const Point &a,const Point &
b)
{
  //求三角形pab的面积
  if(sgn(det(a - c.p,b - c.p)) == 0) return 0.0; //圆心与两点共线
  Point q[5],p1,p2;
  int len = 0;q[len++] = a;
  if(circle_cross_line(c,Line(a,b),p1,p2) == 2)  //ab连线与圆有两个交点
  {
    if(sgn(dot(a - p1,b - p1)) < 0) q[len++] = p1; //只有夹在ab连线内的交点才
对面积有影响
    if(sgn(dot(a - p2,b - p2)) < 0) q[len++] = p2;
  }
  q[len++] = b;
  //如果存在四个交点, 则 q[1] 与 a 一定被 b 分开,需要交换一下位置,方便计算
  if(len == 4 && sgn(dot(q[0] - q[1],q[2] - q[1])) > 0) swap(q[1],q[2]);
  double res = 0;
  for(int i=0;i<len-1;i++)
  {
```

```
    if(circle_point_relation(c,q[i]) == 0 ||
circle_point_relation(c,q[i+1]) == 0)
    {
      double ang = angle(q[i] - c.p,q[i+1] - c.p);
      res += c.r * c.r * ang / 2.0;
    }
    else res += fabs(det(q[i] - c.p,q[i+1] - c.p)) / 2.0;
  }
  return res;
}

double circle_cross_polygon(const Circle &a,const Polygon &b)
{
  double res = 0;
  for(int i=0;i<b.n;i++)
  {
    int j = (i + b.n) % b.n;
    if(sgn(det(b.a[i]-a.p,b.a[j]-a.p)) >= 0)  res += circle_cross_triangl
e(a,b.a[i],b.a[j]);
    else res -= circle_cross_triangle(a,b.a[i],b.a[j]);
  }
  return fabs(res);
}
```