

# 北京大学暑期课《ACM/ICPC竞赛训练》

北京大学信息学院 郭炜

[guo\\_wei@PKU.EDU.CN](mailto:guo_wei@PKU.EDU.CN)

<http://weibo.com/guoweiofpku>

课程网页：[http://acm.pku.edu.cn/summerschool/pku\\_acm\\_train.htm](http://acm.pku.edu.cn/summerschool/pku_acm_train.htm)

# 计算几何

---

北京大学  
林舒 / 郭炜 / 杜宇飞

# 内容

- 概述
- 基础——点、线、面
- 进阶——多边形、半平面

# 概述

---

走近计算几何

# 什么是计算几何

- 计算几何 Computational Geometry
- 研究几何形体的计算机表示、分析与综合
- “计算几何” ——以计算为主的几何

# 计算几何有何用



# 计算几何题的特点

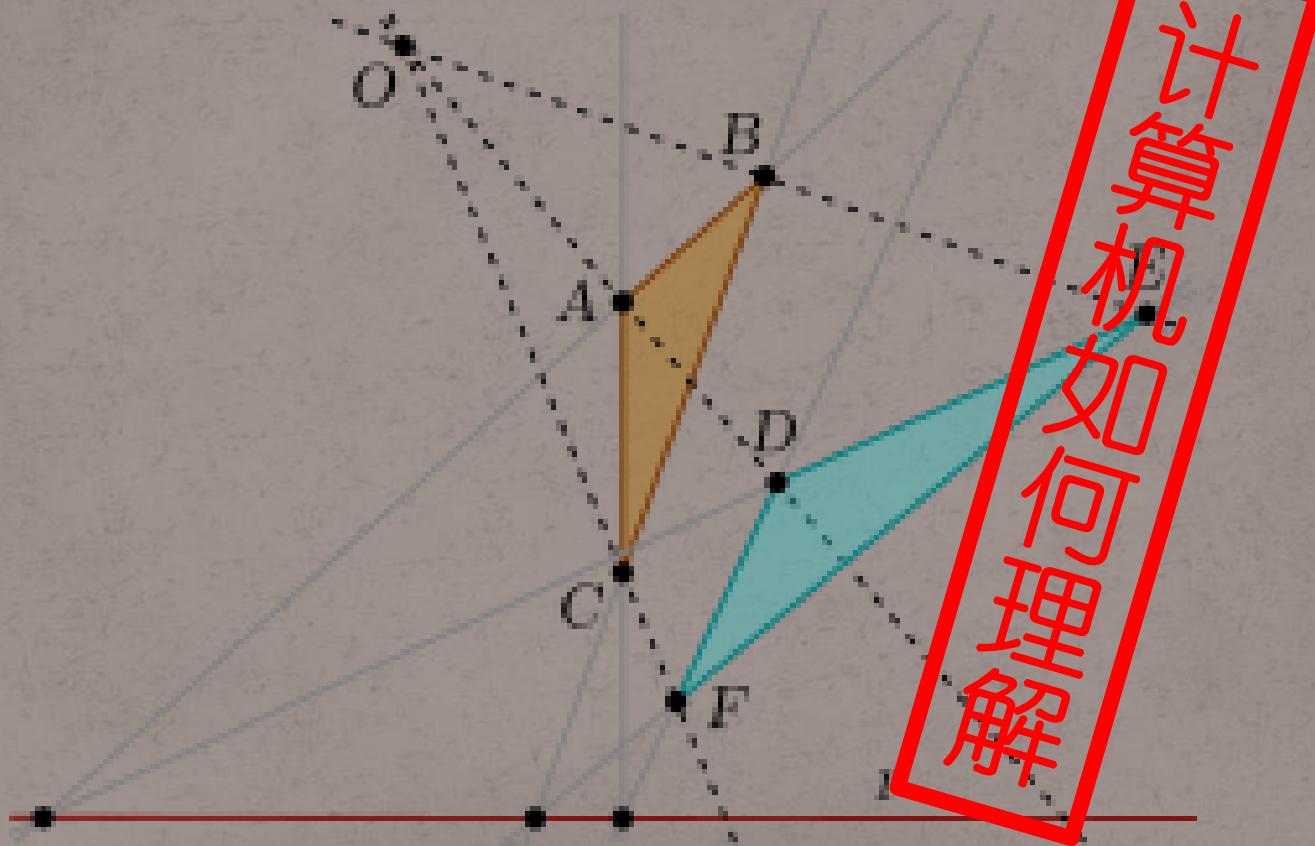
- 题目比较长
- 图形抽象，需要良好的数学基础和空间想象能力
- 有许多容易忽视的特殊情况，而且往往需要单独处理，代码量大
- 需要考虑浮点运算时产生的精度误差
- 可以与其他类型的题目结合，从而更加复杂
- 常作为压轴题目出现在程序设计竞赛中

# 基础——点、线、面

---

用矢量描述计算几何中的基本元素

# 几何图形的表示



# 几何图形的表示

- 沿用解析几何中的表示方法？
- 点  $P(x, y, z)$
- 线  $x=a_x t+b_x, y=a_y t+b_y, z=a_z t+b_z$
- 面  $ax+by+cz+d=0$



有没有更好的办法？

# 矢量法

- 表示简单
- 功能强大
- 特殊情况少，思维难度较低
- 函数可重复利用（即所谓的“模版”）
- 尽可能避免除法和三角函数，精度高，效率高

# 矢量表示

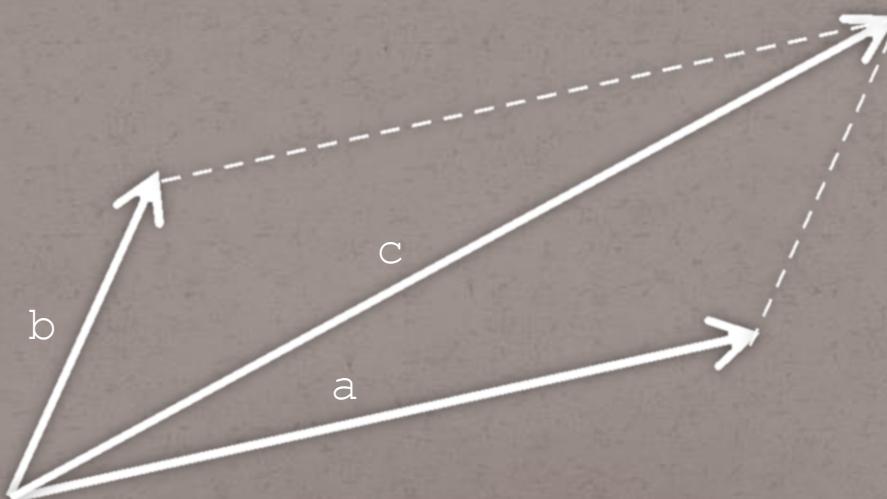
```
class CVector {  
    double x, y;  
};
```

表示从0点到  $(x, y)$  的矢量。对矢量只关心方向和长度，不关心（位置）起点终点

# 矢量的基本运算

```
CVector operator +(CVector p, CVector q) {  
    return CVector(p.x + q.x, p.y + q.y);  
}
```

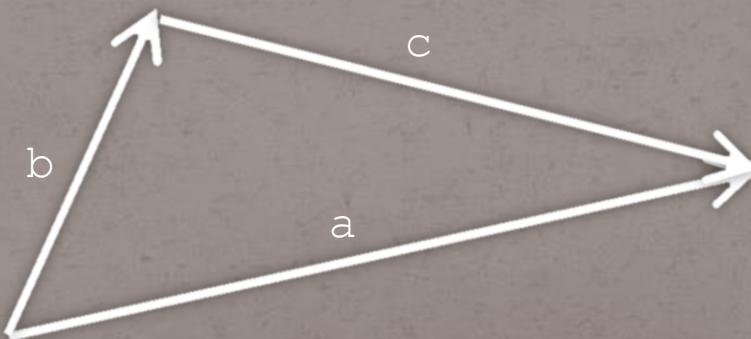
用法：  $c = a + b;$  //a,b,c都是CVector对象



# 矢量的基本运算

```
CVector operator -(CVector p, CVector q) {  
    return CVector(p.x - q.x, p.y - q.y);  
}
```

用法：  $c = a - b;$  //a,b,c都是CVector对象



# 矢量的基本运算

```
CVector operator *(double k, CVector p) {  
    return CVector(k * p.x, k * p.y);  
}
```

用法： c = f \* a; //a是CVector对象, f是double

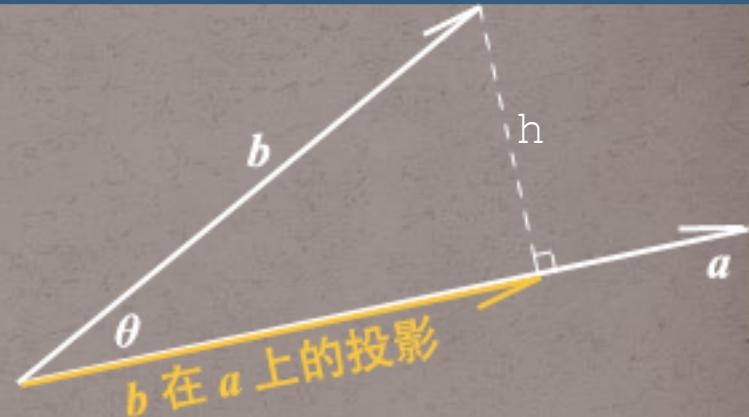
c方向与a相同， |c| = f \* |a|

# 矢量的点积

- 性质： $\mathbf{p} \cdot \mathbf{q} = |\mathbf{p}| |\mathbf{q}| \cos \langle \mathbf{p}, \mathbf{q} \rangle$

```
double operator *(CVector p, CVector q) {  
    return p.x * q.x + p.y * q.y;  
}
```

a与b的点积，就是a的模乘以b在a上投影的模。若投影与a方向相反则为负值



- 功能：求同向还是异向；求投影；求出投影后用勾股定理求点到直线距离；

# 矢量的点积

```
double operator *(CVector p, CVector q) {  
    return p.x * q.x + p.y * q.y;  
}
```

用法： double c = a \* b;

//b,c都是CVector对象

- 若  $a \cdot b = 0$ , 则 a 和 b 垂直

# 矢量模长

- 用矢量与自身点积求模

```
double length(CVector p) {  
    return sqrt(p * p);  
} //求矢量的模
```

# 矢量单位化

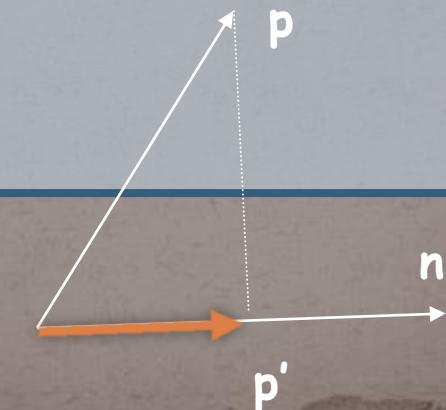
- 将矢量除以自身的长度以得到同方向的单位矢量

```
CVector unit(CVector p) {  
    return 1 / length(p) * p;  
}
```

# 矢量的投影长度

- 矢量与该方向单位矢量的点积
- 注意：负数表示反方向

```
double project(CVector p, CVector n) {  
    return dot(p, unit(n)); // 点积  
}  
double dot(CVector p, CVector q) {  
    return p.x*q.x+p.y*q.y;  
}
```



## 矢量的点积

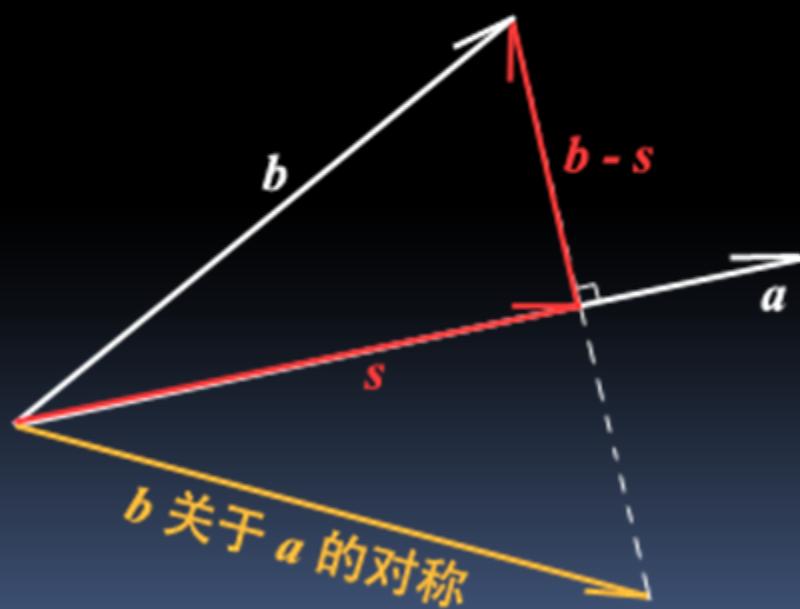
$\mathbf{b}$  在  $\mathbf{a}$  上的投影的“模”是  $\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}|}$ 。

所以  $\mathbf{b}$  在  $\mathbf{a}$  上的投影即为  $\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}|} \frac{\mathbf{a}}{|\mathbf{a}|} = \mathbf{a} \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a}^2}$ 。

# 矢量的点积

记  $\mathbf{b}$  在  $\mathbf{a}$  上的投影为  $\mathbf{s} = \mathbf{a} \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a}^2}$ 。

则  $\mathbf{b}$  关于  $\mathbf{a}$  的对称为  $\mathbf{b} - 2(\mathbf{b} - \mathbf{s}) = 2\mathbf{s} - \mathbf{b} = 2\mathbf{a} \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a}^2} - \mathbf{b}$ 。



# 矢量的叉积

- 性质：在二维情况下，  
 $|p \times q| = |p| |q| \sin\langle p, q \rangle$
- 功能：求面积；求顺时针方向还是逆时针方向；判断是否在半平面上

```
double operator ^(CVector p, CVector q) {  
    return p.x * q.y - q.x * p.y;  
}
```

用法：c = a ^ b; //a,b,c都是CVector对象

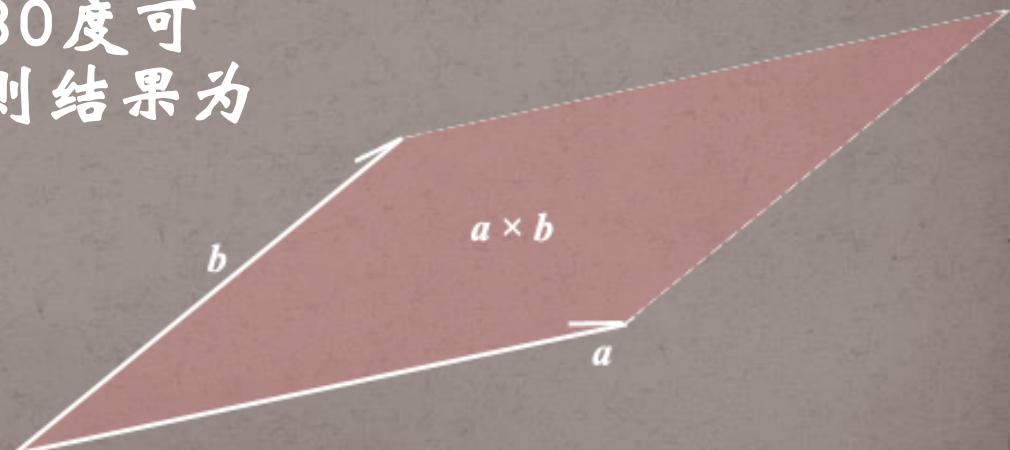
# 矢量的叉积

- 性质：在二维情况下，

$$|\mathbf{p} \times \mathbf{q}| = |\mathbf{p}| |\mathbf{q}| \sin \langle \mathbf{p}, \mathbf{q} \rangle$$

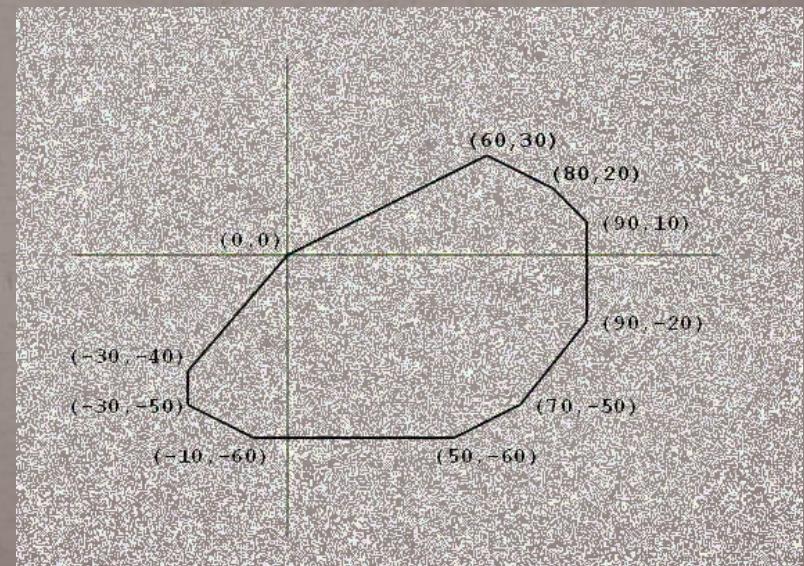
- $\mathbf{a} \times \mathbf{b}$  为有向面积，可正可负。

若  $\mathbf{a}$  逆时针旋转小于 180 度可到  $\mathbf{b}$ ，则结果为正，否则结果为负



# Scrambled Polygon

- POJ 2007
- 乱序给出凸多边形的顶点坐标，要求按逆时针顺序输出各顶点。给的第一个点一定是 $(0, 0)$ ，没有其他点在坐标轴上，没有三点共线的情况。
- 利用叉积排序



```
//POJ 2007 Scrambled Polygon    给定凸多边形顶点，要求按反时针序输出。  
//叉积排序  By Guo Wei  
#include <iostream>  
#include <cmath>  
#include <algorithm>  
using namespace std;  
struct Vector  
{  
    double x, y;  
    Vector(int xx, int yy) :x(xx), y(yy) {}  
    Vector() {}  
    double operator ^ (const Vector & v) const {  
        return x*v.y - v.x*y;  
    }  
};  
#define Point Vector
```

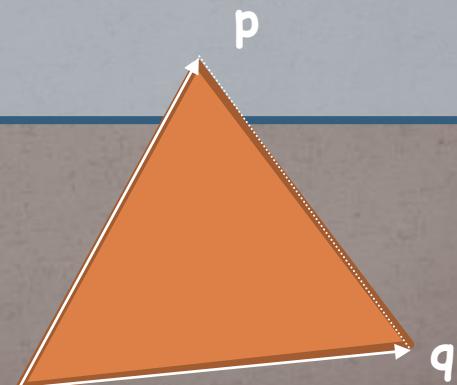
```
Vector operator - (const Point & p1, const Point & p2)
{ //从A点指向B点的矢量AB可用B-A来表示
    return Vector(p1.x - p2.x, p1.y - p2.y);
    //矢量从 p2指向p1
}
bool operator < (const Point & p1, const Point & p2)
{
    //如果p1^p2 > 0, 说明p1经逆时针旋转<180度可以到p2, 则 p1 < p2
    if ((Vector(p2 - Point(0, 0)) ^
        Vector(p1 - Point(0, 0))) > 0)
        return true;
    return false;
}
```

```
Point ps[60];
int main()
{
    int x, y;
    int n = 0;
    while (cin >> ps[n].x >> ps[n].y)
        ++n;
    sort(ps + 1, ps + n);
    cout << "(0,0)" << endl;
    for (int i = n - 1; i > 0; --i)
        cout << "(" << ps[i].x << ", "
              << ps[i].y << ")" << endl;
    return 0;
}
```

# 两个矢量所围成的三角形面积

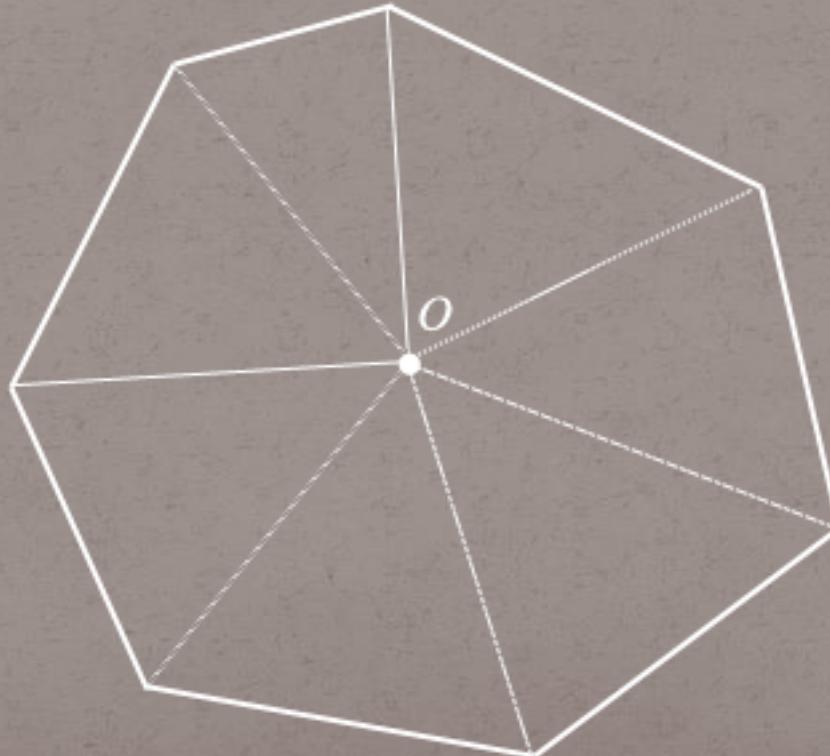
- 两个矢量的叉积的一半
- 注意：得到的面积为有向面积，可能为负

```
double area(CVector p, CVector q) {  
    return p^q / 2;  
}
```



# 求多边形的面积

- 基本的思路是进行三角剖分。



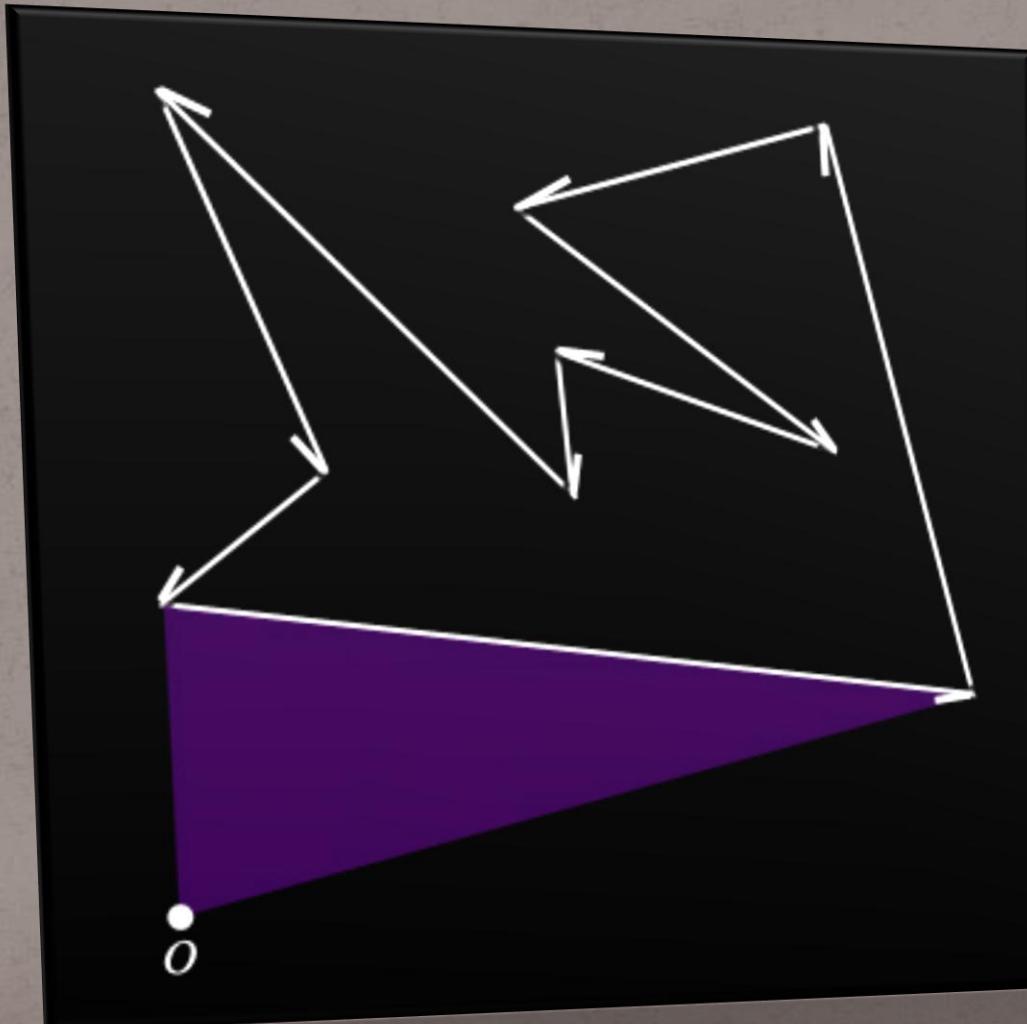
# 求多边形的面积

- 如果用普通面积的累加来计算的话，三角剖分就变得十分复杂。
- 但是使用有向面积的话，这个过程就变得简单自然通用。

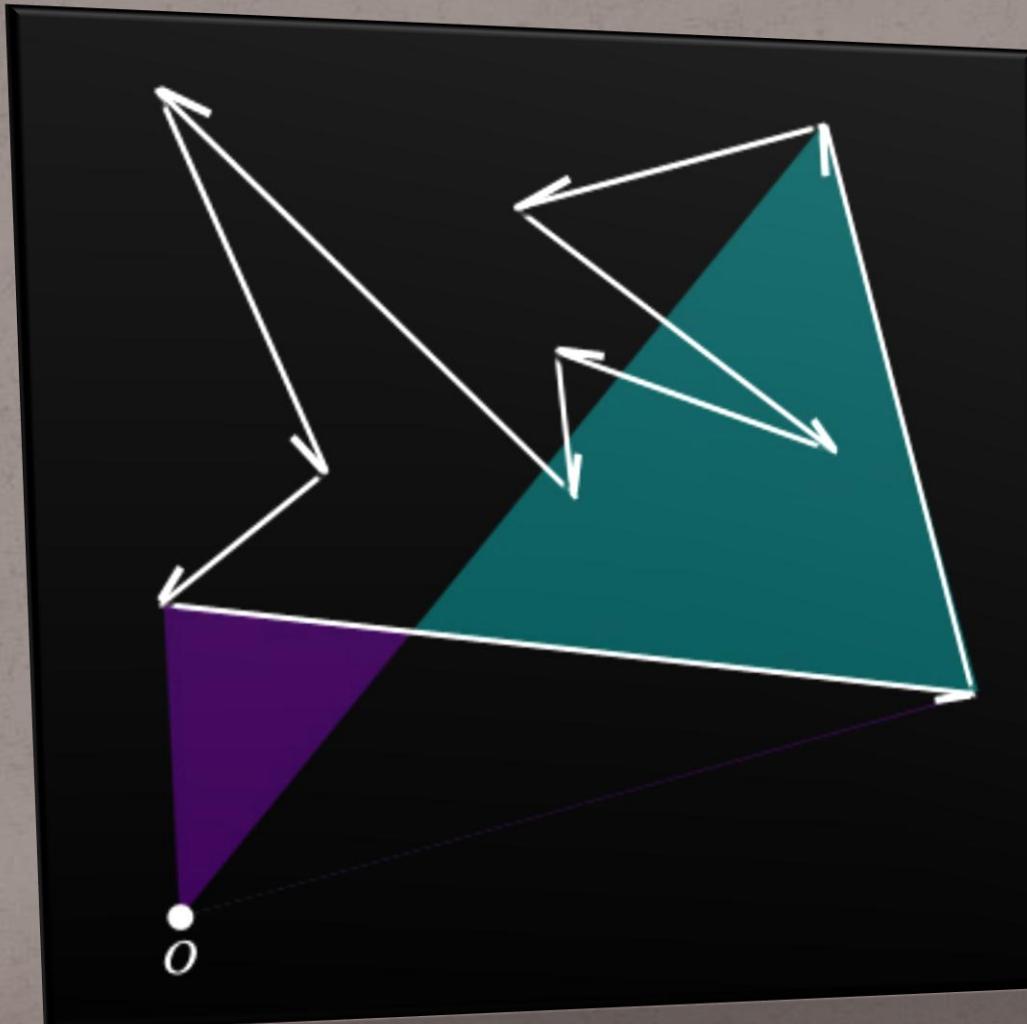
# 算法

- 按逆时针方向顺次为各边指定方向。
- 对于每条边  $\overrightarrow{AB}$ , 累加  $\frac{A \times B}{2}$  的值。
- 最后得到的结果即为多边形的面积。
- 当然也可以累加  $A \times B$  的值, 最后再除以 2。

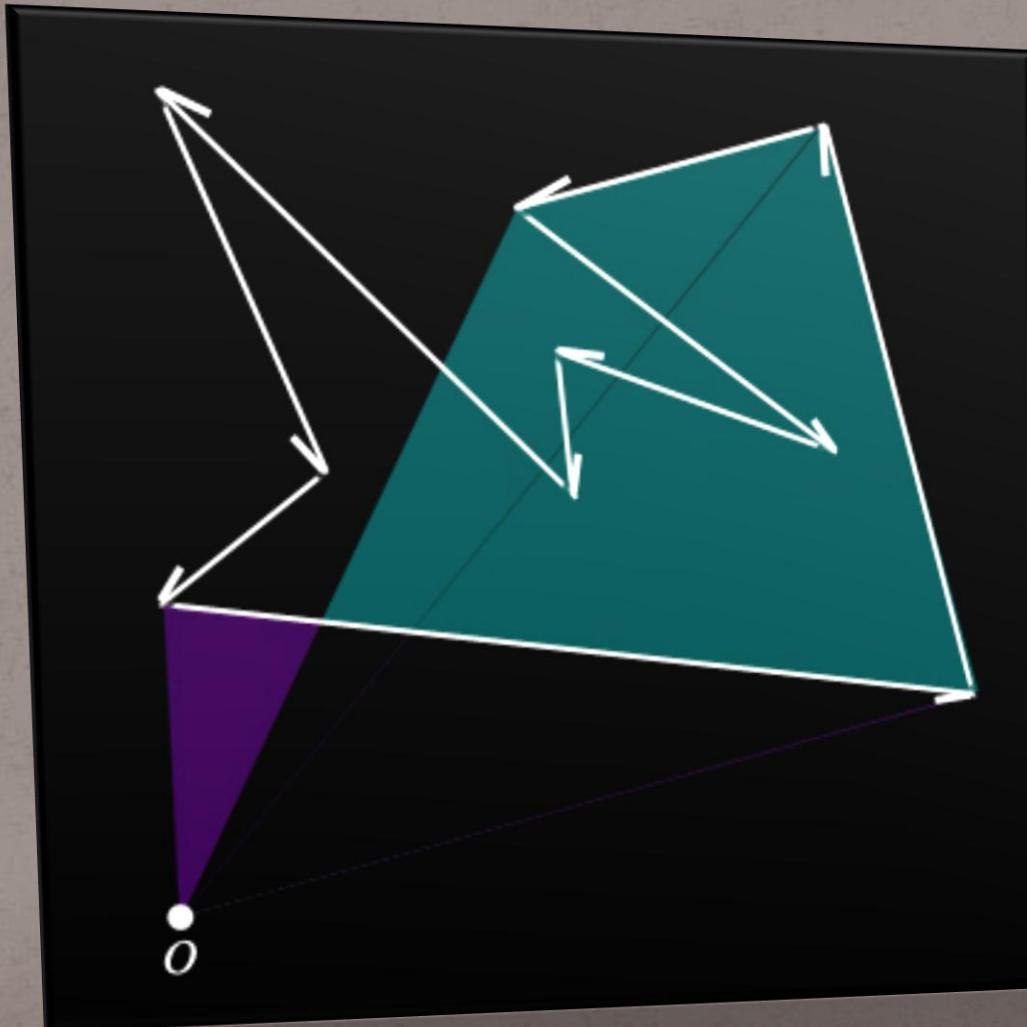
# 算法演示



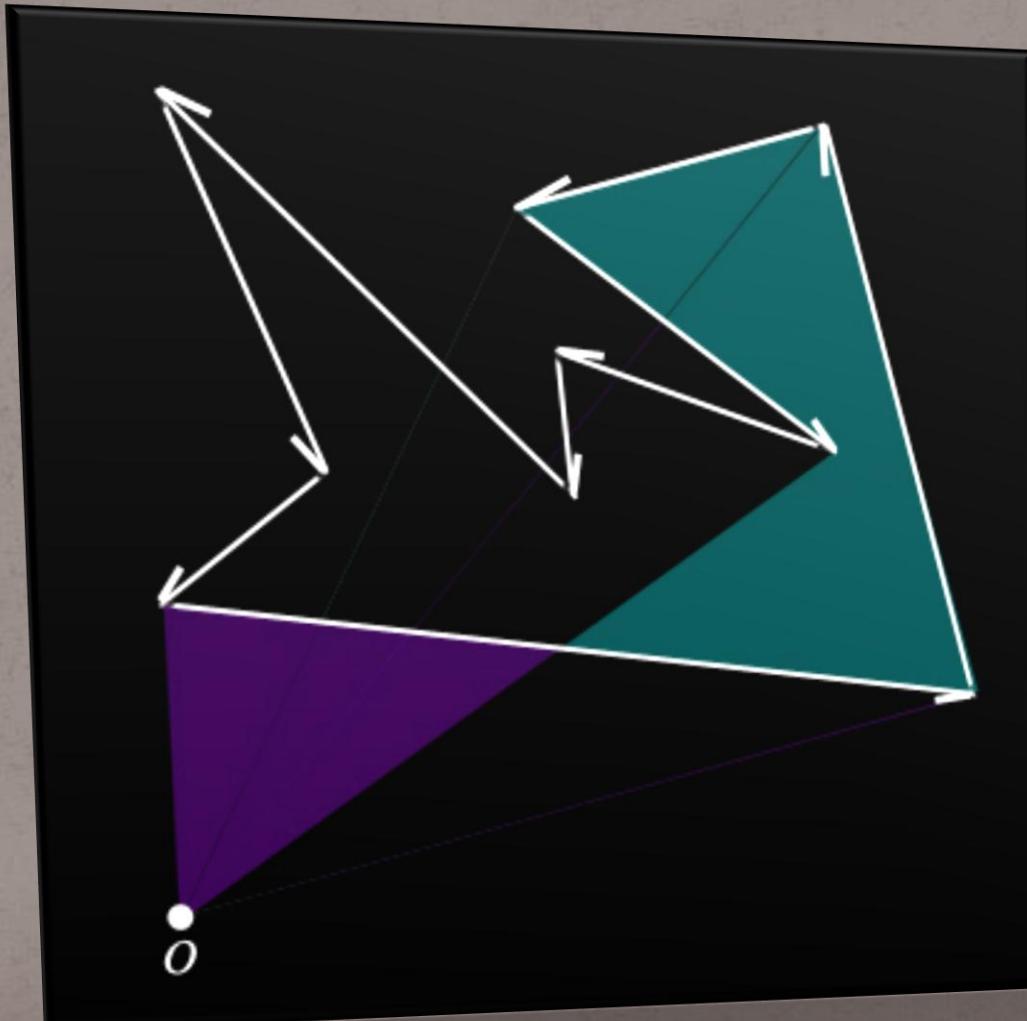
# 算法演示



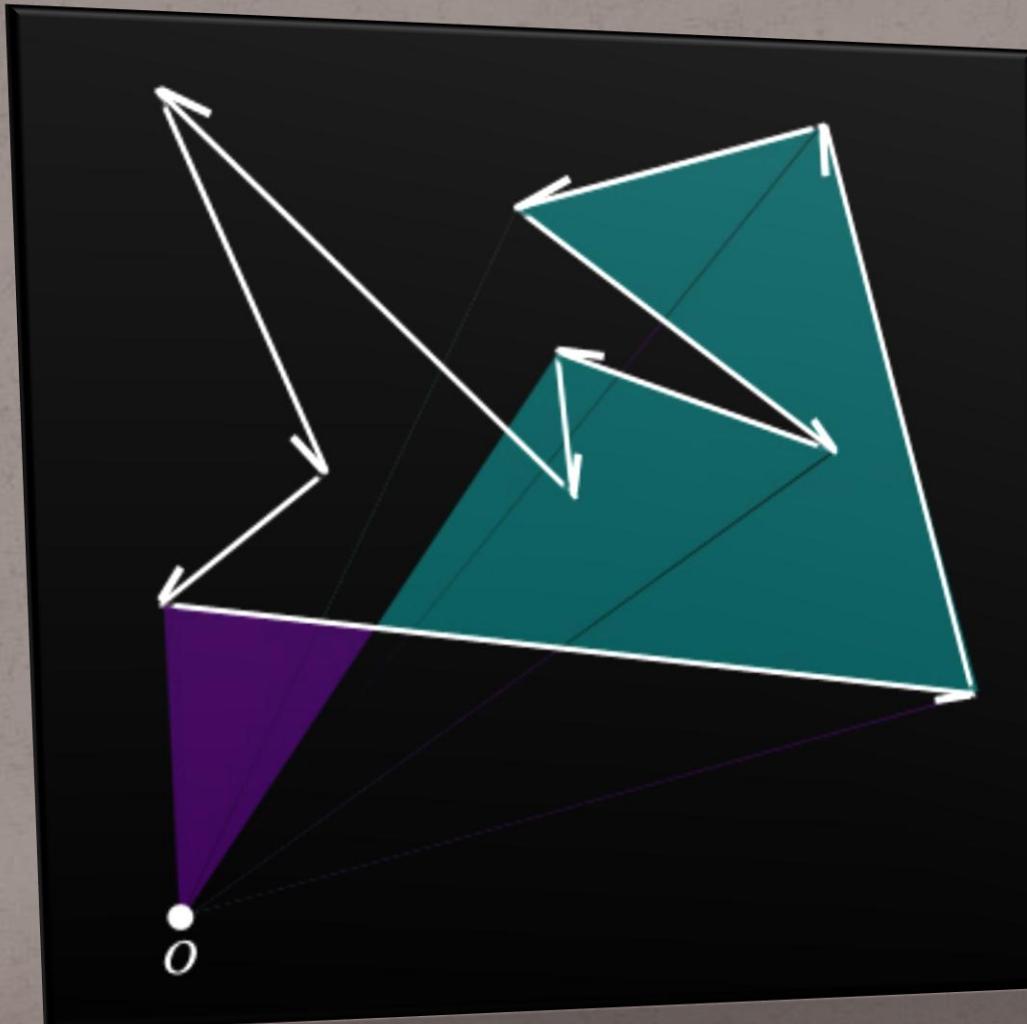
# 算法演示



# 算法演示



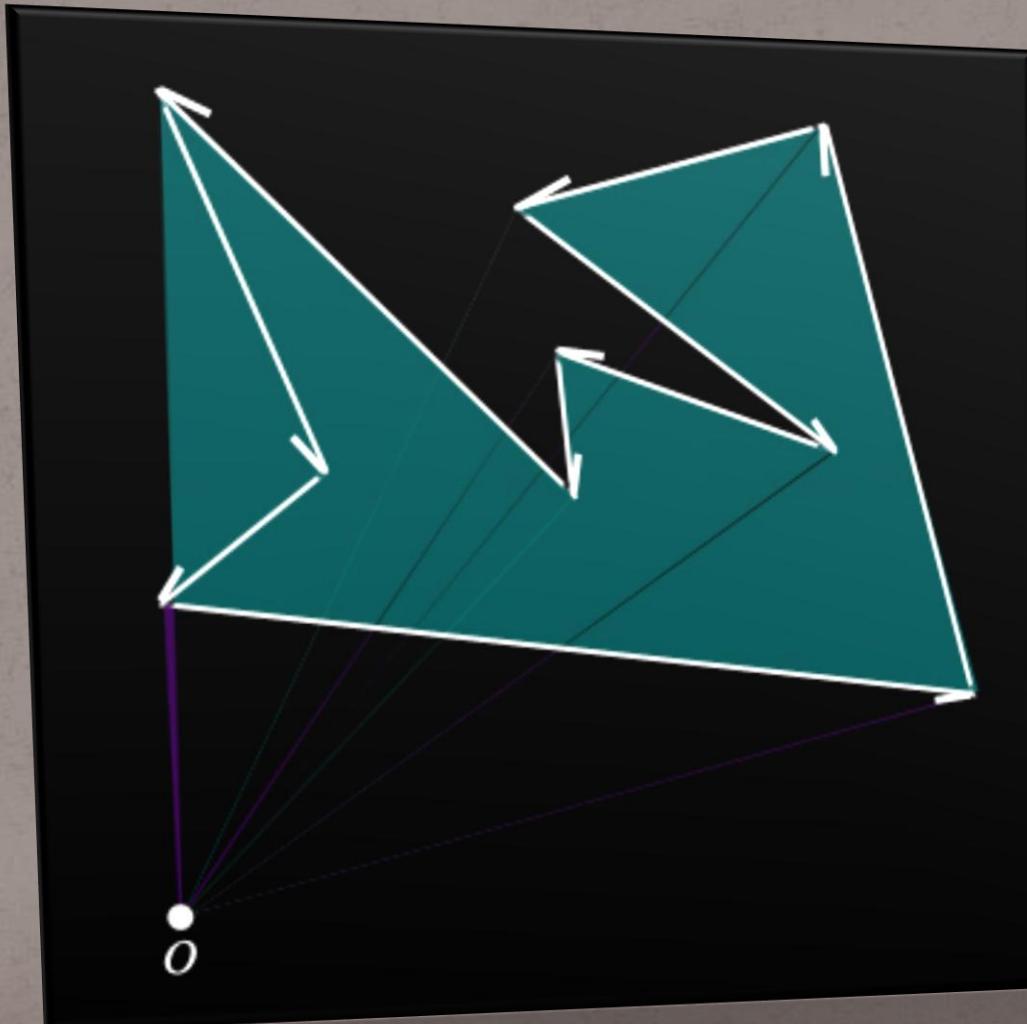
# 算法演示



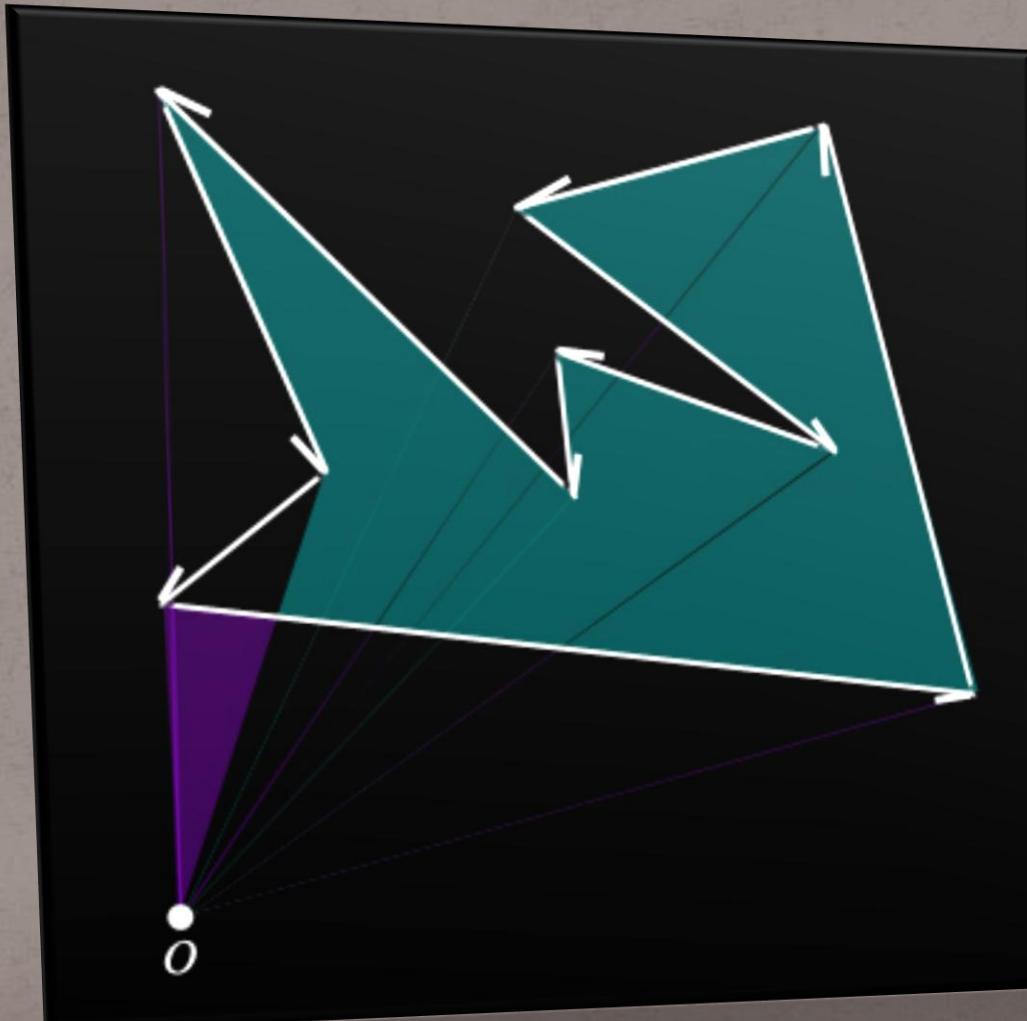
# 算法演示



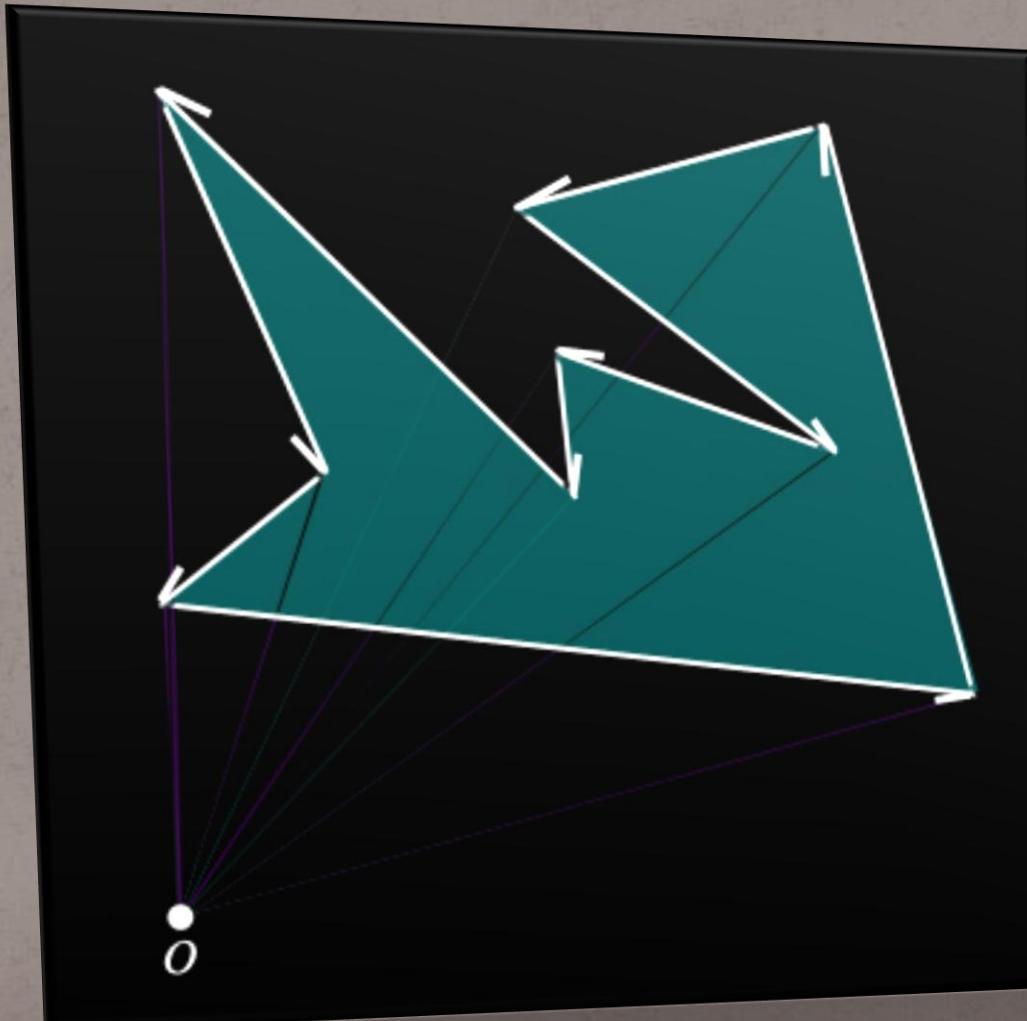
# 算法演示



# 算法演示



# 算法演示



# 点、线的表示

```
class CPoint {  
    double x, y;  
}
```

```
class CLine {  
    CPoint a, b;  
}
```

矢量和点的表示法相同，因此也可以：

```
#define CPoint CVector
```

给出了“点”的表示

# 常用常数与函数

```
double PI = acos(-1);
double INF = 1e20;
double EPS = 1e-6; // 精度不是越高越好
```

```
bool IsZero(double x) {
    return -EPS < x && x < EPS;
}
bool FLarger(double a, double b) {
    return a - b > EPS;
}
bool FLess(double a, double b) {
    return b - a > EPS;
}
```

# 点与矢量

- 从A点指向B点的矢量 $\overrightarrow{AB}$ 可用 $\mathbf{B} - \mathbf{A}$ 来表示
- 将A点沿矢量 $\mathbf{p}$ 的方向移动矢量 $\mathbf{p}$ 长度的距离到B，B可以用 $\mathbf{A} + \mathbf{p}$ 来表示

```
CVector operator -(CPoint b, CPoint a) {  
    return CVector(b.x - a.x, b.y - a.y);  
} // c = a - b;
```

```
CPoint operator +(CPoint a, CVector p) {  
    return CPoint(a.x + p.x, a.y + p.y);  
} // p = p + v; p是点, v是向量
```

# 点与点距离

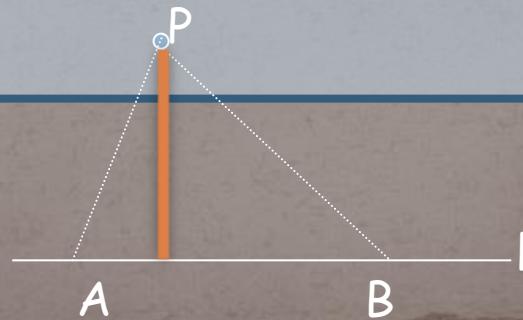
- 利用两点间矢量的模长

```
double dist(CPoint p, CPoint q) {  
    return length(p - q);  
} //从A点指向B点的矢量AB可用B-A来表示
```

# 点与线距离

- 利用叉积求面积，然后除以底即为高

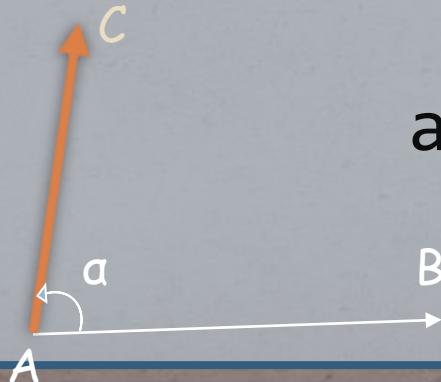
```
double dist(CPoint p, CLine l) {  
    return fabs((p - l.a) ^ (l.b - l.a))  
        / length(l.b - l.a);  
}
```



# 点绕点旋转(二维)

- 旋转矢量AB到AC
- 注：在xy平面上逆时针旋转 $\alpha$ 角（弧度制）

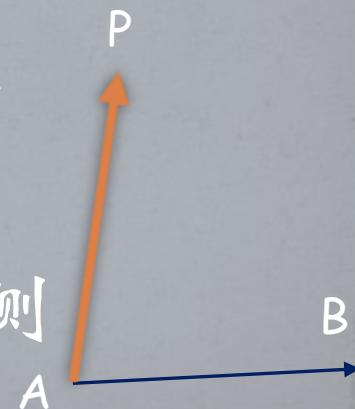
```
CPoint rotate(CPoint b, CPoint a,  
              double alpha) { // 返回点C坐标  
    CVector p = b - a;  
    return CPoint(a.x + (p.x * cos(alpha)  
                         - p.y * sin(alpha)),  
                  a.y + (p.x * sin(alpha)  
                         + p.y * cos(alpha));  
}
```



# 点在直线左侧还是右侧

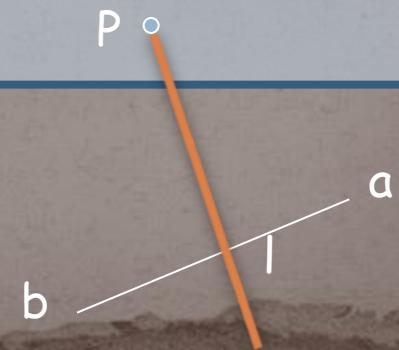
- 直线用两个点A, B表示，是有方向的，假设方向是A->B

```
int sideOfLine (Point p, Point a, Point b)
{ // 判断p在直线 a->b 的哪一侧
    double result = (b - a) ^ (p - a);
    if (IsZero(result))
        return 0; // p 在 a->b 上
    else if (result > 0)
        return 1; // p 在 a->b 左侧
    else
        return -1; // p 在 a->b 右侧
}
```



# 过点作线的垂线（二维）

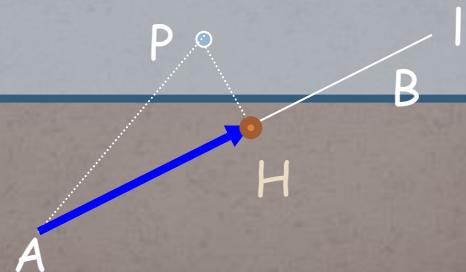
```
CLine Vertical(CPoint p, CLine l) {  
    return CLine(p,  
                p + (rotate(l.b, l.a, PI / 2) - l.a));  
}
```



# 点到线的垂足

- 利用点积求投影，进而求出垂足
- 应用：求对称点
- 注：在平面上也可作垂线，利用线与线交点（后面会提到）来求

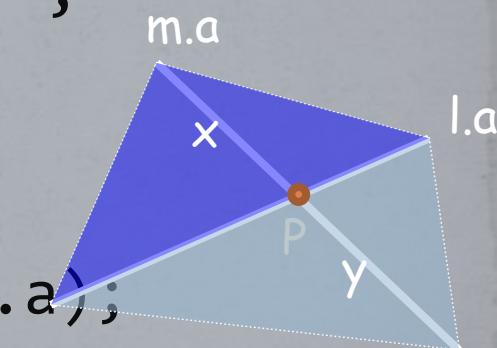
```
CPoint foot(CPoint p, CLine l) {  
    return l.a + project(p - l.a, l.b - l.a)  
        * unit(l.b - l.a);  
}
```



# 两条直线的交点

- 先判断是否有唯一解（不平行），再利用叉积求解

```
CPoint intersect(CLine l, CLine m, string msg) {  
    double x = area(m.a - l.a, l.b - l.a);  
    double y = area(l.b - l.a, m.b - l.a);  
    if (isZero(x + y)) {  
        if (isZero(dist(l, m))) msg = “重合”;  
        else msg = “平行”;  
        return null;  
    }  
    return m.a + x / (x + y) * (m.b - m.a);  
} //即便线段 l.a->l.b和 m.a->m.b没交点，也适用
```



# 判断线段规范相交

- 即判断是否任一条线段的两个端点都分属另一条线段的两侧。

```
double Cross(const Vector & v1, const Vector & v2)
{
    //叉积
    return v1.x * v2.y - v1.y * v2.x;
}

bool IsFormalCross(Point p1, Point p2, Point p3,
                    Point p4)
{//判断p1->p2和p3->p4是否规范相交
    return Cross(p2 - p1, p3 - p1) *
           Cross(p2 - p1, p4 - p1) < -eps   &&
           Cross(p4 - p3, p1 - p3) *
           Cross(p4 - p3, p2 - p3) < -eps;
}
```

# 两条线段相交的各种情况

```
struct Seg //线段
{
    Point a, b; //向量是 a->b ,即 b-a
    Seg(const Point & aa, const Point & bb):
        a(aa), b(bb) { }

    //直线两点式方程  $(y-y_1)/(y_2-y_1) = (x-x_1)/(x_2-x_1)$ 
    double getX(double y) { //给定y坐标, 求直线上的 x坐标
        return (y - a.y) / (b.y - a.y) * (b.x - a.x) + a.x;
    }

    double getY(double x) { //给定x坐标, 求直线上的y坐标
        return (x - a.x) / (b.x - a.x) * (b.y - a.y) + a.y;
    }
};
```

# 两条线段相交的各种情况

```
pair<int, Point> CrossPoint(Seg s1, Seg s2);
```

```
pair<int, Point> result = CrossPoint(s1, s2);
```

返回值 `result.first`:

- 0 规范相交,
- 1 端点重合, 但不平行, 不共线
- 2 一个端点在另一个内部 s1端点在 s2内部 (不平行, 不共线)
- 3 一个端点在另一个内部 s2端点在 s1内部 (不平行, 不共线)
- 4 无交点, 不平行, 不共线, 两直线交点是`result.second`
- 5 平行
- 6 共线且有公共点
- 7 共线且无公共点
- 8 s1,s2无交点, 但是s2所在直线和s1有交点, 即交点在s1上
- 9 s1,s2无交点, 但是s1所在直线和s2有交点, 即交点在s2上

有交点的情况下, 交点都是: `result.second`

```
pair<int, Point> CrossPoint(Seg s1, Seg s2) {
    Point p1 = s1.a;
    Point p2 = s1.b;
    Point p3 = s2.a;
    Point p4 = s2.b;

    double a1 = Area(p3 - p1, p4 - p1);
    double a2 = Area(p4 - p2, p3 - p2); // 这些顺序不能乱
    if (Cross(p2 - p1, p3 - p1) *
        Cross(p2 - p1, p4 - p1) < -EPS &&
        Cross(p4 - p3, p1 - p3) *
        Cross(p4 - p3, p2 - p3) < -EPS) { // 规范相交
        return make_pair(0, p1 + (p2 - p1) * (a1 / (a1 + a2)));
    }
    if (! (IsZero(Cross(p2 - p1, p3 - p4)))) {
        // 不平行, 不共线
        if (p1 == p3 || p1 == p4)
            // 端点重合且不平行, 不共线
            return make_pair(1, p1);
    }
}
```

```
if (p2 == p3 || p2 == p4)
    return make_pair(1, p2);
if (PointInSeg(p1, s2))
    return make_pair(2, p1);
if (PointInSeg(p2, s2))
    return make_pair(2, p2);
if (PointInSeg(p3, s1))
    return make_pair(3, p3);
if (PointInSeg(p4, s1))
    return make_pair(3, p4);
Point crossPoint =
    p1+(p2 - p1)*(a1/(a1+ a2));
if (PointInSeg(crossPoint, s1))
    return make_pair(8, crossPoint);
if (PointInSeg(crossPoint, s2))
    return make_pair(9, crossPoint);
return make_pair(4, crossPoint);
// 直线和线段也无交点，不平行，不共线，两直线交点是second
}
```

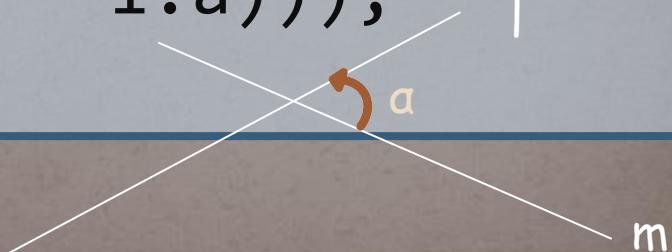
```
if (!IsZero(Distance(p1, s2)))
    return make_pair(5, Point(0, 0)); //平行
//下面都是共线，且有公共点
if (PointInSeg(p1, s2))
    return make_pair(6, p1);
if (PointInSeg(p2, s2))
    return make_pair(6, p2);
if (PointInSeg(p3, s1))
    return make_pair(6, p3);
if (PointInSeg(p4, s1))
    return make_pair(6, p4);
return make_pair(7, Point(0, 0)); //共线，且无公共点
}
```

```
bool PointInSeg(Point p, Seg L)
{
    double tmp = Cross(L.a - p, L.a - L.b);
    if (!IsZero(tmp))
        return false;
    if ( FLessEq(min(L.a.x, L.b.x),p.x) &&
        FLessEq(p.x ,max(L.a.x, L.b.x)) &&
        FLessEq(min(L.a.y, L.b.y), p.y) &&
        FLessEq(p.y , max(L.a.y, L.b.y)))
        return true;
    return false;
}
bool FLessEq(double a, double b)
{
    return b - a > EPS || IsZero(b-a);
}
```

# 线与线夹角

- 利用投影（也可以利用叉积或余弦定理）
- 应用：两直线的位置关系，射线夹角（注意方向即可）

```
double angle(CLine l, CLine m) {  
    return acos(fabs(  
        project(l.b - l.a, m.b - m.a)  
        / length(l.b - l.a)));  
}
```



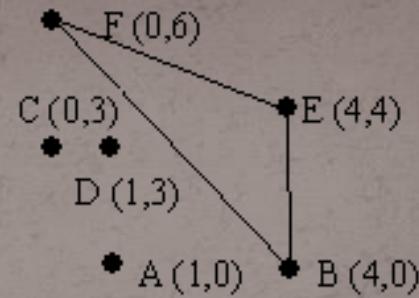
# 4th Point

- POJ 2624
- 已知平行四边形的两条邻边，求第四个点的坐标
- 矢量和



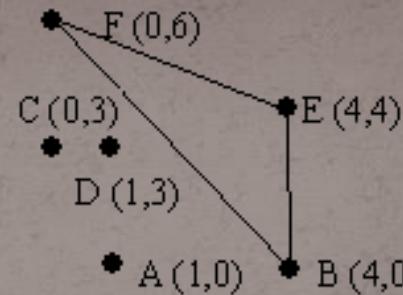
# Myacm Triangles

- POJ 1569
- 平面上有一些点（很少），求以这些点为顶点的三角形中，内部无其他点的面积最大的三角形是哪个



# My ACM Triangles

- POJ 1569
- 平面上有一些点（很少），求以这些点为顶点的三角形中，内部无其他点的面积最大的三角形是哪个
- 枚举三角形三个顶点 $a, b, c$ ，看 $(b-a) \wedge (c-b)$  符号得出是顺时针序还是逆时针序，然后用叉积判断其他点是否在三角形内（边为逆时针序的情况下，点同时在三条边的左边则为在三角形内。此法同样适用于判断点是否在凸多边形内）



# *Segments*

- POJ 3304
- 给定一些线段，问是否有直线 $L$ 存在，使得所有线段到 $L$ 的投影有至少一个公共点。

# *Segments*

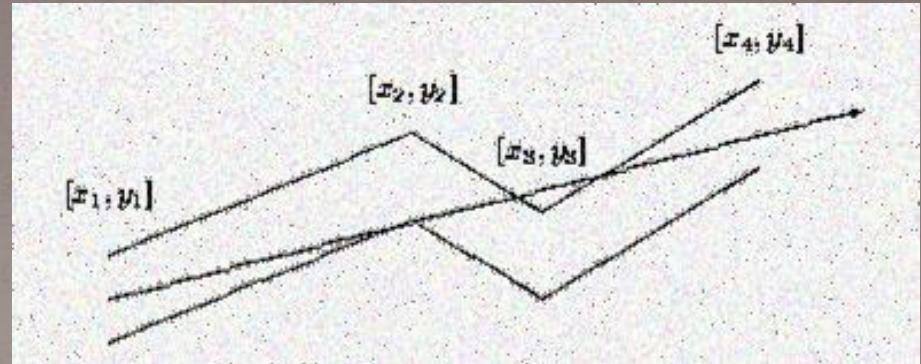
- POJ 3304
- 给定一些线段，问是否有直线 $L$ 存在，使得所有线段到 $L$ 的投影有至少一个公共点。
- 实际上就是问，是否存在一条直线 $K$ ，和所有这些线段都相交。 $L$ 是 $K$ 的垂线即可。

# *Segments*

- POJ 3304
- 给定一些线段，问是否有直线 $L$ 存在，使得所有线段到 $L$ 的投影有至少一个公共点。
- 实际上就是问，是否存在一条直线 $K$ ，和所有这些线段都相交。 $L$ 是 $K$ 的垂线即可。
- 若存在一条直线与所有线段相交，该直线必定经过这些线段的某两个端点（否则可以平移或转动使之靠上端点）。枚举任意两个端点构造直线并看它是否与每条线段相交即可。

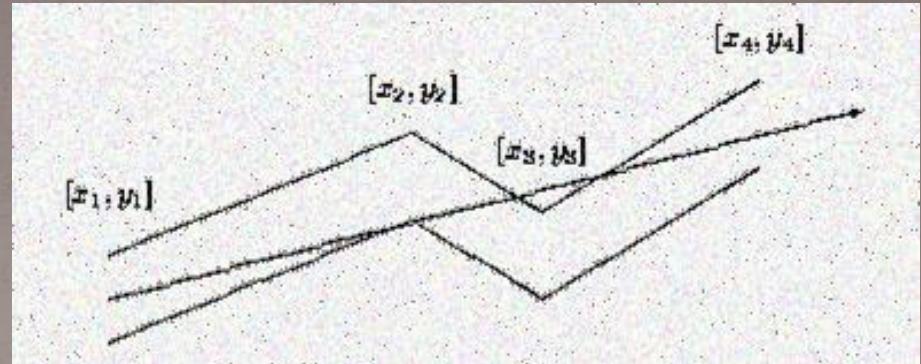
# Pipe

- POJ 1039
- 在平面上有一根由线段（至多20根）组成的折线管道，管道任意处上边界比下边界高1，求是否存在一束光能穿过管道

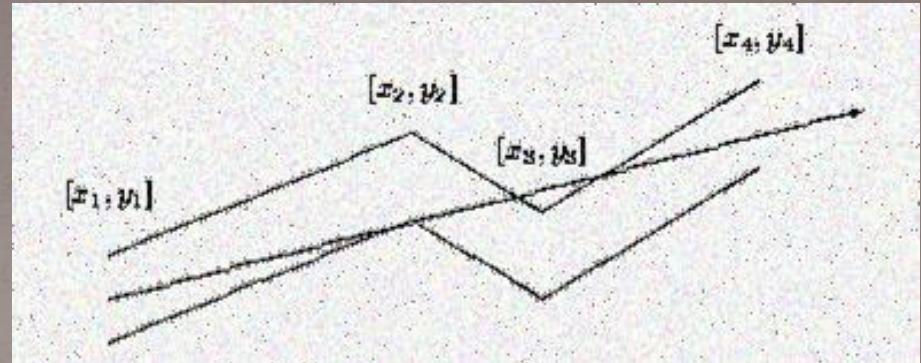


# Pipe

- POJ 1039
- 在平面上有一根由线段（至多20根）组成的折线管道，管道任意处上边界比下边界高1，求是否存在一束光能穿过管道
- 枚举一个上边界的顶点和一个下边界的顶点，组成一束光（符合要求的光线一定能通过转动或平移靠上上下边界各一个顶点），利用线与线的交点判断是否在管内



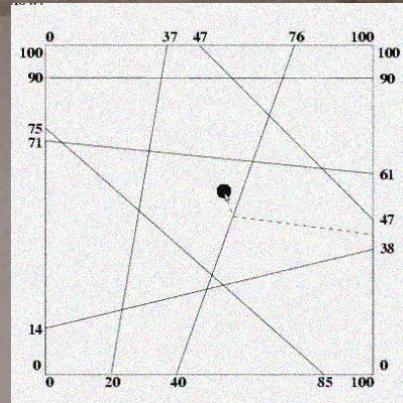
# Pipe



- 若在管内，必和每条由管子转折处的上拐点和下拐点构成的垂直线段相交（包括出入口）。

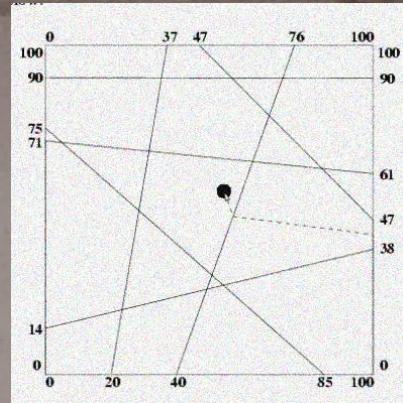
# Treasure Hunt

- POJ 1066
- 在正方形区域内有一些墙（至多30堵），每堵墙都是贯穿整个区域的且不会有超过两堵墙相交在同一点，求从正方形外走到形内指定点至少要穿越多少堵墙



# Treasure Hunt

- POJ 1066
- 在每堵墙指定点至少要有一个整点，内穿一穿越多堵墙（至多30堵），且形状为正方形的区域从少堵墙（至多30堵）到多堵墙（至多30堵）的区域求出多少枚点数。求出所有起始点和终点，将所有墙的交点按顺序排序。



# 进阶——多边形、半平面

---

从抽象到具体

# 问题列表

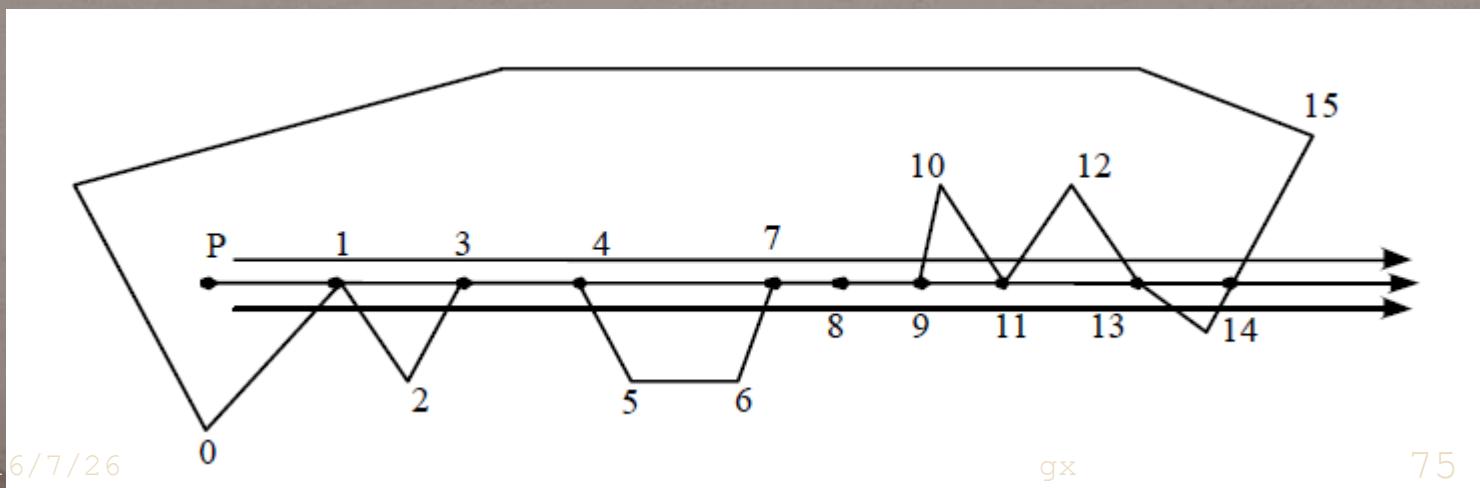
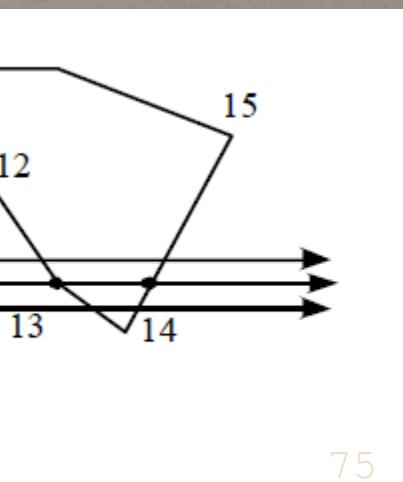
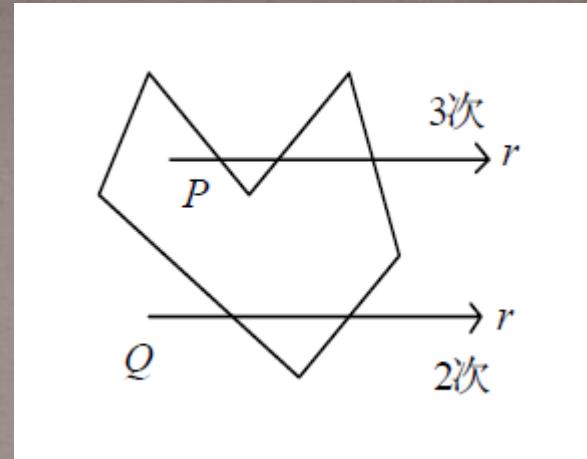
- 点与多边形的位置关系
- 凸包
- 半平面交

# 点与多边形的位置关系

- 点在多边形内
- 点在多边形上
- 点在多边形外

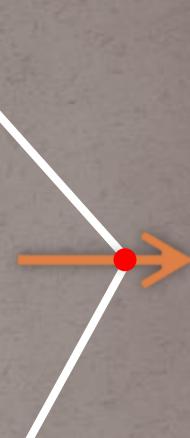
# 点在多边形内外

- 射线法
  - 交点奇数 / 偶数
  - 射线经过顶点：何种方式经过
  - 射线经过边：平移小距离
  - 点本身在边上：把特殊情况先预判掉！

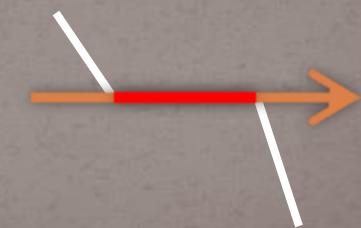
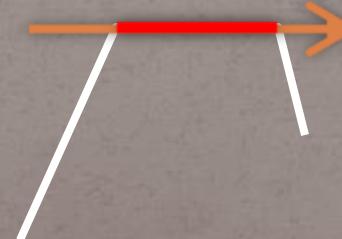
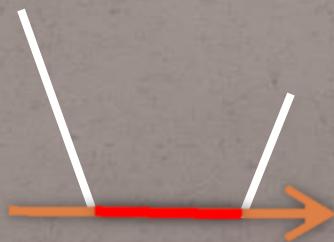


# 射线法

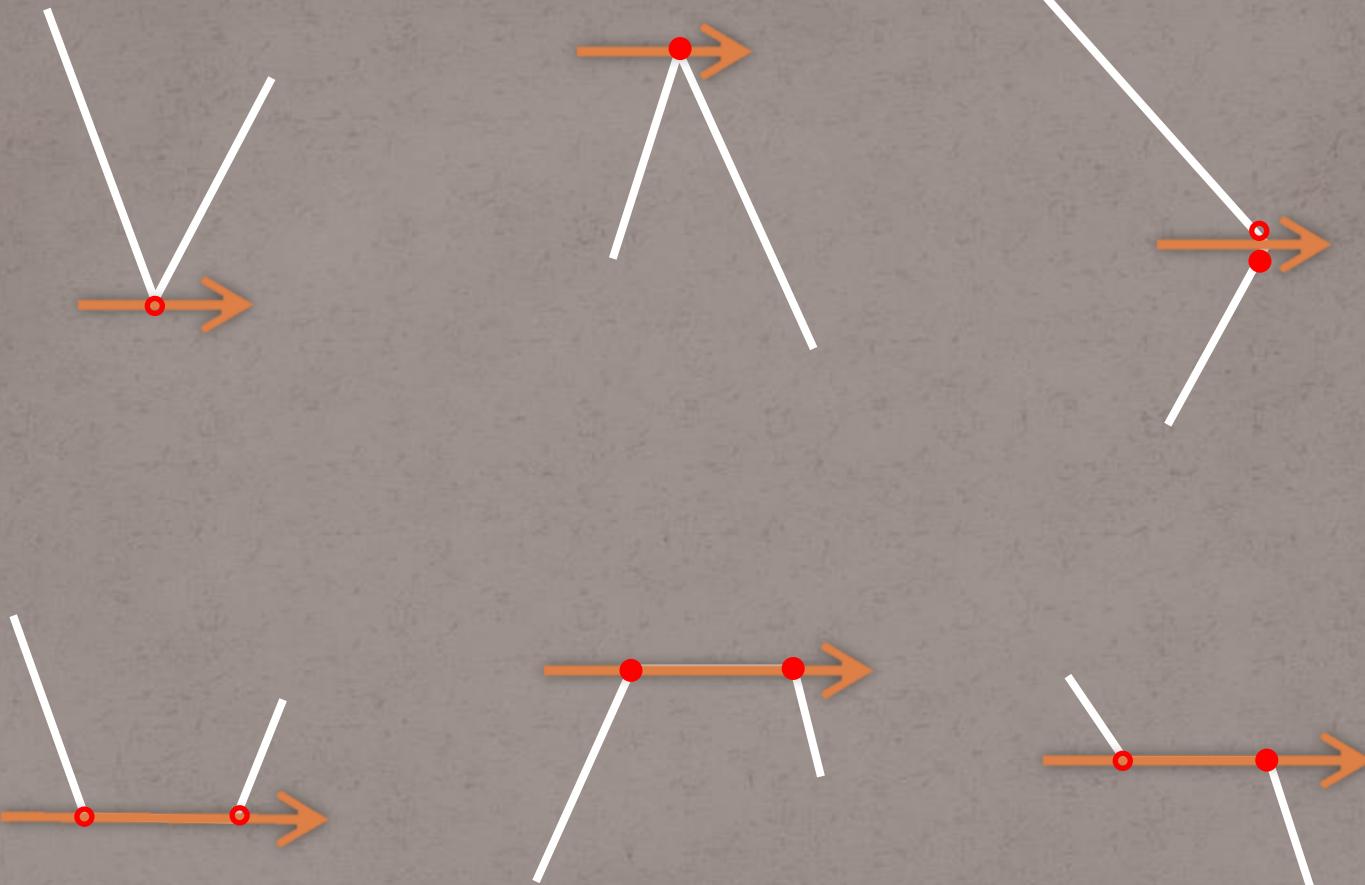
- 特殊情况
  - 与顶点相交



- 与边重合



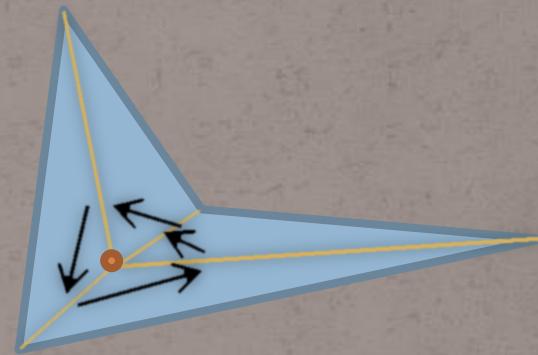
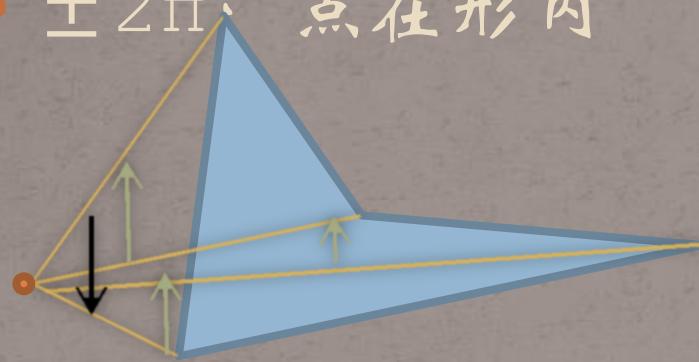
# 射线法



只要取的射线足够好（比如随机），根本不用考虑这些情况

# 转角法

- 沿多边形走一圈，累计绕点旋转了多少角度
  - 0：点在形外
  - $\pi$ ：点在形边
  - $\pm 2\pi$ ：点在形内



- 角度计算
  - $\theta = \cos^{-1} ((\mathbf{a} \cdot \mathbf{b}) / (|\mathbf{a}| |\mathbf{b}|))$

# 射线法与转角法

- 射线法
  - 运算速度快，精度高
  - 特殊情况较多
- 转角法
  - 几乎没有特殊情况
  - 需要反三角函数、开方等，精度低，速度较慢

# 凸包

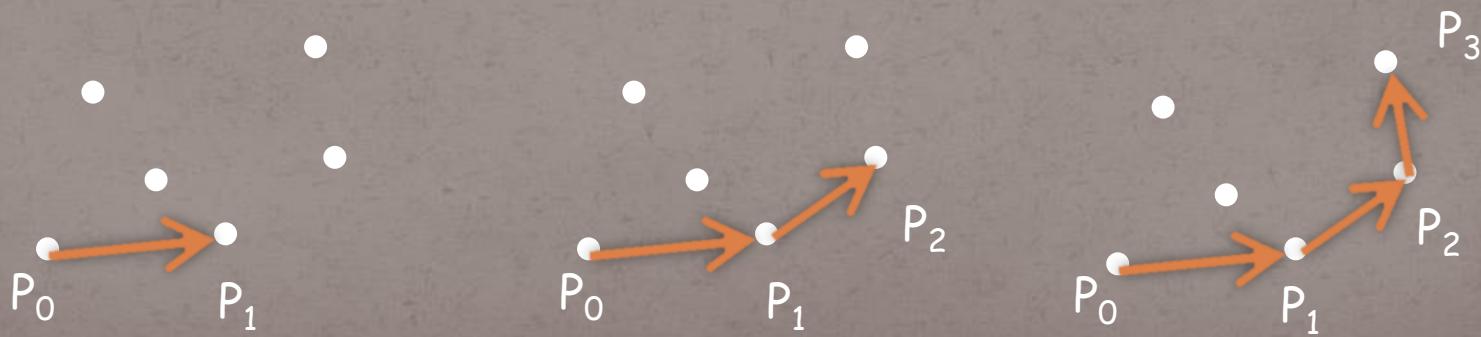
- 定义
  - 给定一个平面点集
  - 要求找到一个最小的凸多边形，满足
  - 点集中的所有点都在该凸多边形的内部或边上
- 性质
  - 任意两点的连线都在凸包内

# 凸包

直观来看，把点集  $S$  中所有点钉在一个木板上，用一个橡皮筋框起所有点来，一撒手，橡皮筋就表示出了  $S$  的凸包。

# 卷包裹法

- 从最左最低点  $P_0$  出发
- 找一点  $P_1$ ，使得  $P_0P_1$  与水平方向夹角最小
- 找一点  $P_2$ ，使得  $P_0P_2$  与  $P_0P_1$  夹角最小
- .....
- 最终， $P_0P_1P_2\dots P_{m-1}$  构成凸包

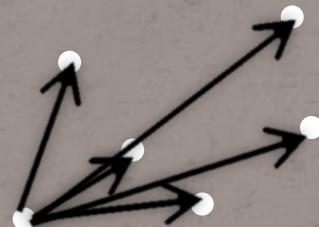


# 卷包裹法

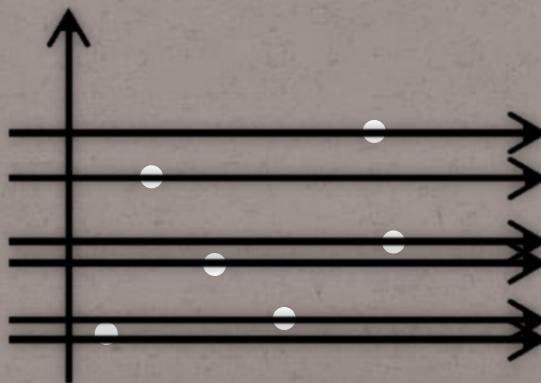
- 使用叉积便可找到夹角最小的矢量
- 时间复杂度 $\mathcal{O}(n^2)$
- 每一步得到的都是最终凸包上的一条边

# Graham 扫描法

- 将点排序
  - 极角序

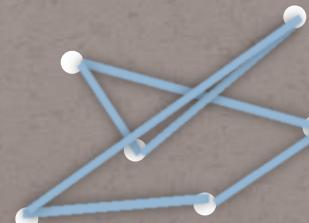


- 水平序



# Graham 扫描法

- 同样通过叉积判断点和边的位置关系
- 时间复杂度
  - 排序  $O(n \log n)$
  - 扫描  $O(n)$
  - 总计  $O(n \log n)$
- 不排序可能导致错误



# 极角序 Graham 扫描法

- 找出最左下的点（首先要最下，有多个最下则找最左），记为  $p[0]$
- 以  $p[0]$  为原点，对  $p[1] \dots p[n-1]$  进行按极角排序。极角相同的，距离  $p[0]$  近的算小
- 栈里面放入  $p[0], p[1], p[2]$

```
for(int i = 3; i < n; ++i) {
```

```
    while(true) {
```

考察栈顶元素  $k_2, k_2$  下方元素  $k_1$ ，以及  $p[i]$

考虑  $k_1 \rightarrow k_2 \rightarrow p[i]$

if( 在  $k_2$  这一点直走或者向右拐了)

将  $k_2$  出栈

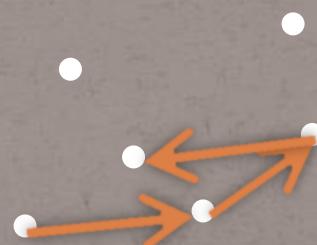
```
else break;
```

```
}
```

$p[i]$  入栈

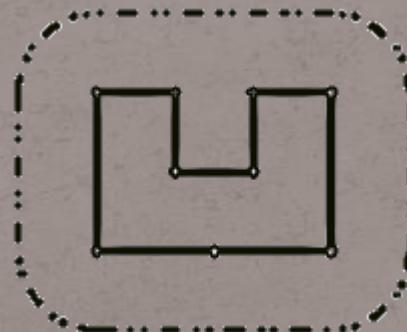
```
}
```

- 栈中的点就是凸包的所有顶点



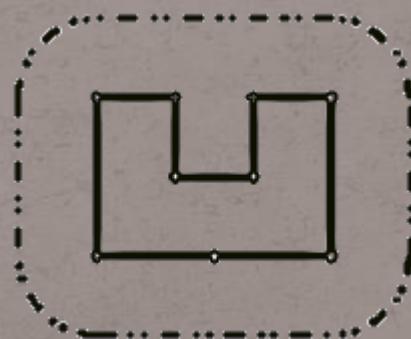
# Wall

- POJ 1113
- 给定 $n$ 个点，要求建尽量短的、包围所有点的围墙，且每个点到围墙的最短距离不得小于8英尺



# Wall

- POJ 1113
- 给定 $n$ 个点，要求建尽量短的、包围所有点的围墙，且每个点到围墙的最短距离不得小于8英尺
- 求凸包的周长，然后加上一个圆的周长即可



//POJ1113 Wall , 极角序扫描法求凸包 by Guo Wei

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <cstdio>
using namespace std;
#define EPS 1e-6
```

int Sign(double x) { // 判断 x 是大于0, 等于0还是小于0  
 return fabs(x)<EPS?0:x>0?1:-1;  
}

```
struct Point {  
    double x,y;  
    Point(double xx = 0,double yy = 0):x(xx),y(yy) {}  
    Point operator-(const Point & p) const {  
        return Point(x-p.x,y-p.y);  
    }  
    bool operator <(const Point & p) const {  
        if( y < p.y)  
            return true;  
        else if( y > p.y)  
            return false;  
        else  
            return x < p.x;  
    }  
};  
typedef Point Vector;
```

```
double Cross(const Vector & v1, const Vector & v2)
{ // 叉积
    return v1.x * v2.y - v2.x * v1.y;
}

double Distance(const Point & p1, const Point & p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) + (p1.y-
    p2.y)*(p1.y-p2.y));
}
```

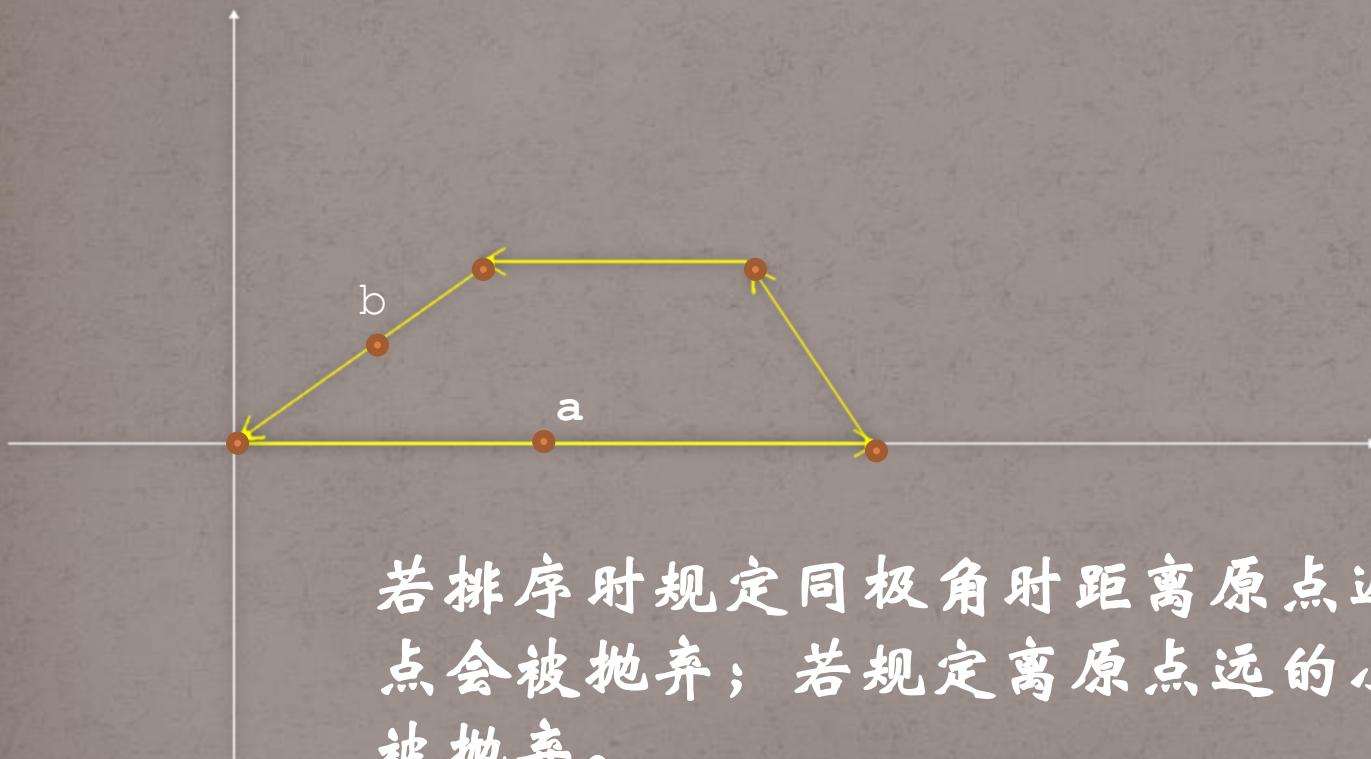
```
struct Comp { //用来定义极角排序规则的函数对象
    Point p0; //以p0为原点进行极角排序，极角相同的，离p0近算小
    Comp(const Point & p):p0(p.x,p.y) { }
    bool operator ()(const Point & p1,const Point & p2)
    const {
        int s = Sign(Cross(p1-p0,p2-p0));
        if( s > 0)
            return true;
        else if( s < 0)
            return false;
        else {
            if( Distance(p0,p1)<Distance(p0,p2))
                return true;
            else
                return false;
        }
    }
};
```

```
int Graham(vector<Point> & points,
           vector<Point> & stack) {
    //points是点集合
    if( points.size() < 3)
        return 0; //返回凸包顶点数
    stack.clear();
    //先按坐标排序，最左下的放到points[0]
    sort(points.begin(),points.end());
    //以points[0] 为原点进行极角排序
    sort(points.begin()+1,points.end(),Comp(points[0]));
    stack.push_back(points[0]);
    stack.push_back(points[1]);
    stack.push_back(points[2]);
```

```
for(int i = 3; i< points.size(); ++i) {
    while(true) {
        Point p2 = * (stack.end()-1);
        Point p1 = * (stack.end()-2);
        if( Sign(Cross(p2-p1,points[i]-p2) <= 0))
            //p2->points[i] 没有向左转，就让p2出栈
            stack.pop_back();
        else
            break;
    }
    stack.push_back(points[i]);
}
return stack.size();
}
```

# 极角序 Graham 扫描法的局限性

如果要保留凸包边上非顶点（只有右拐时才出栈），则极角序法不适用。



若排序时规定同极角时距离原点近的小，则b点会被抛弃；若规定离原点远的小，则a点会被抛弃。

# 水平序 Graham 扫描法

- 对顶点按  $x$  为第一关键字,  $y$  为第二关键字进行排序。
- 准备一个空栈, 并将前两个点压入栈。
- 对于每一个顶点  $A$ , 只要栈顶中还有至少两个顶点, 记栈顶为  $T$ , 栈中第二个为  $U$ 。若  $\overrightarrow{UT} \times \overrightarrow{TA} \leq 0$ , 则将  $T$  弹出。重复此过程。
- 直到上一步不再弹出顶点, 将  $A$  压入栈。扫描完一遍之后得到凸包的下凸壳。
- 将点集倒过来再进行一次, 得到凸包的上凸壳, 组合起来即可。

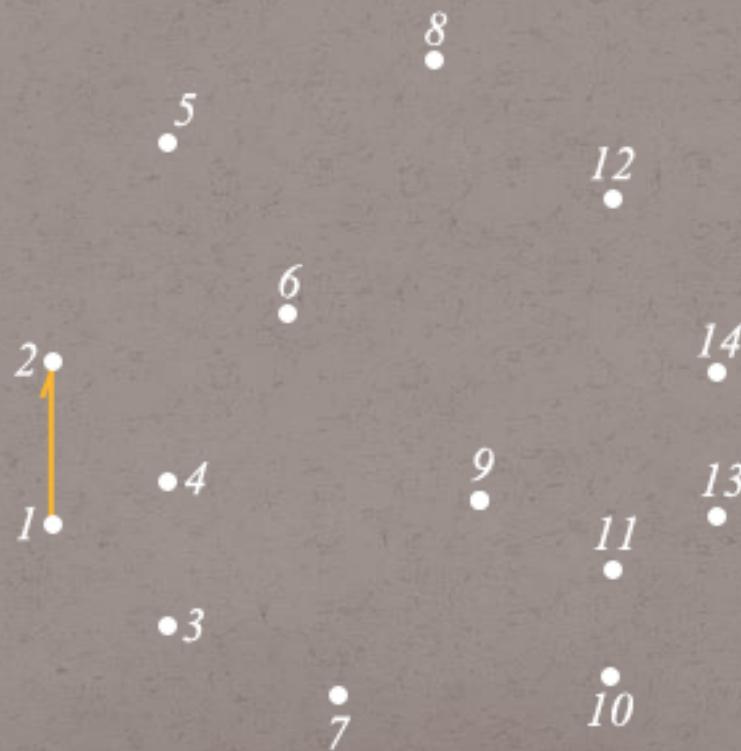
# 算法演示

首先将所有点排序。



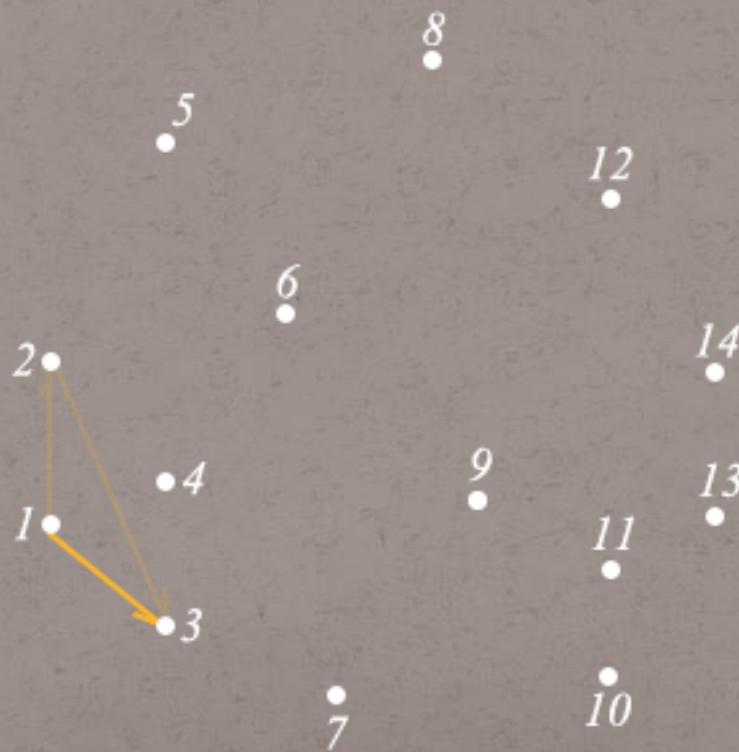
# 算法演示

将 1 和 2 压入栈。



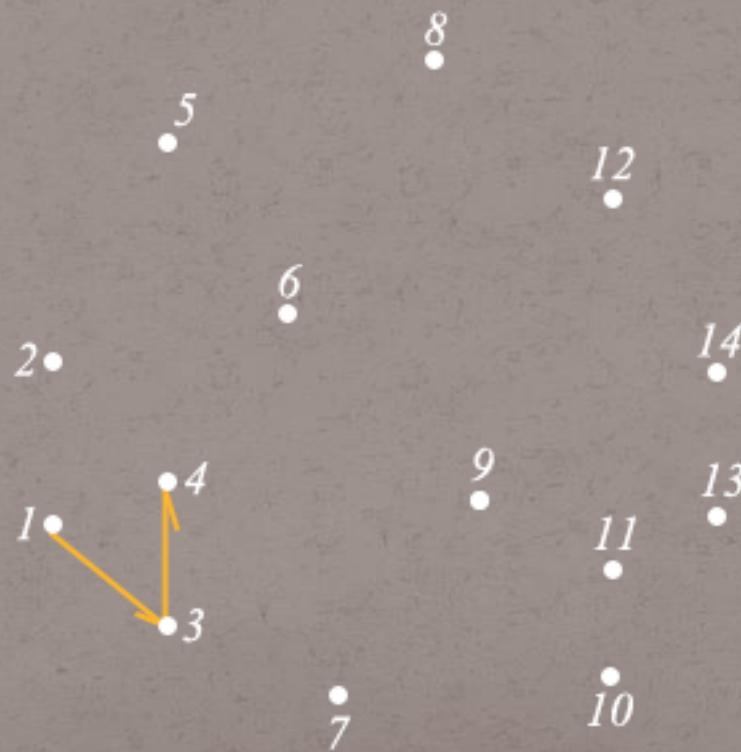
# 算法演示

2 被弹出栈，3 进栈。



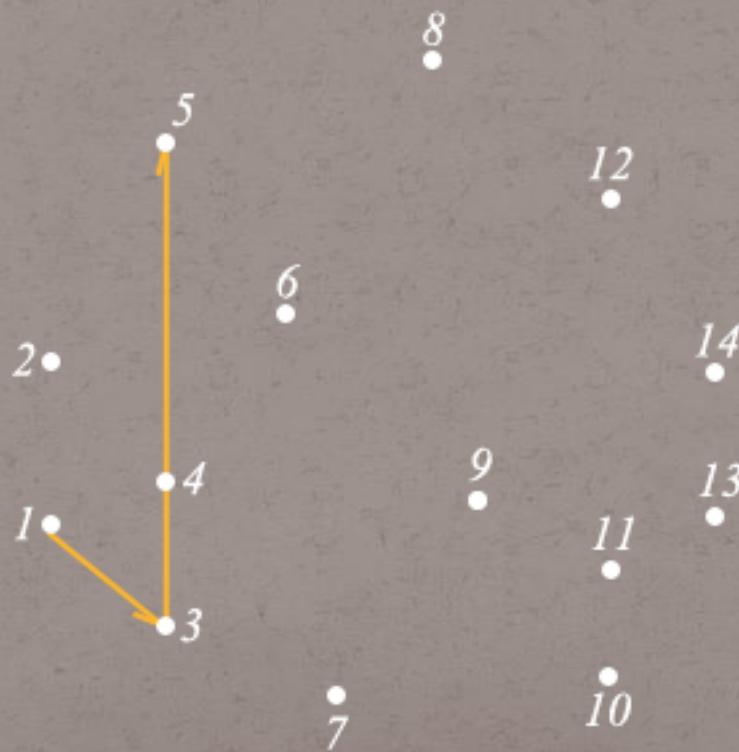
# 算法演示

没有点被弹出栈，4 进栈。



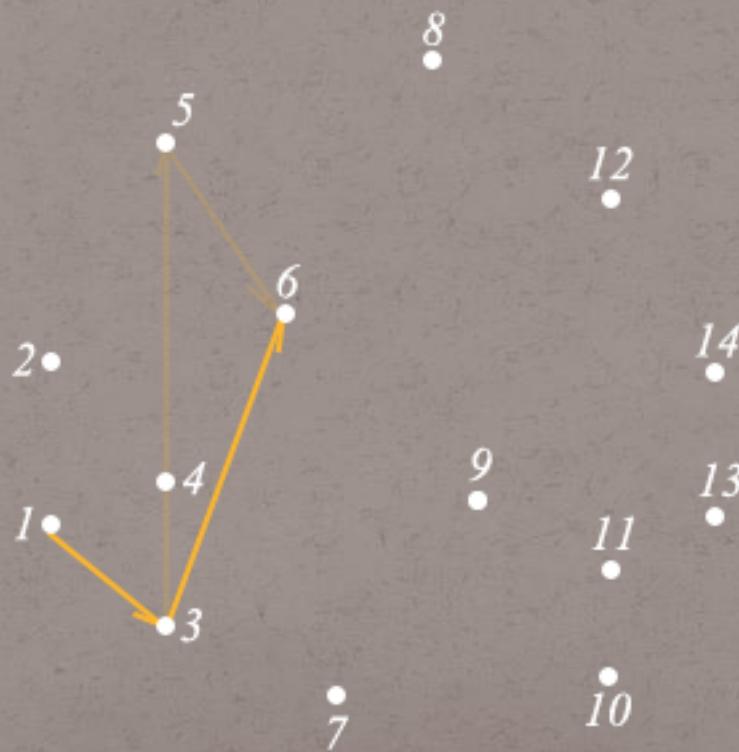
# 算法演示

4 被弹出栈，5 进栈。



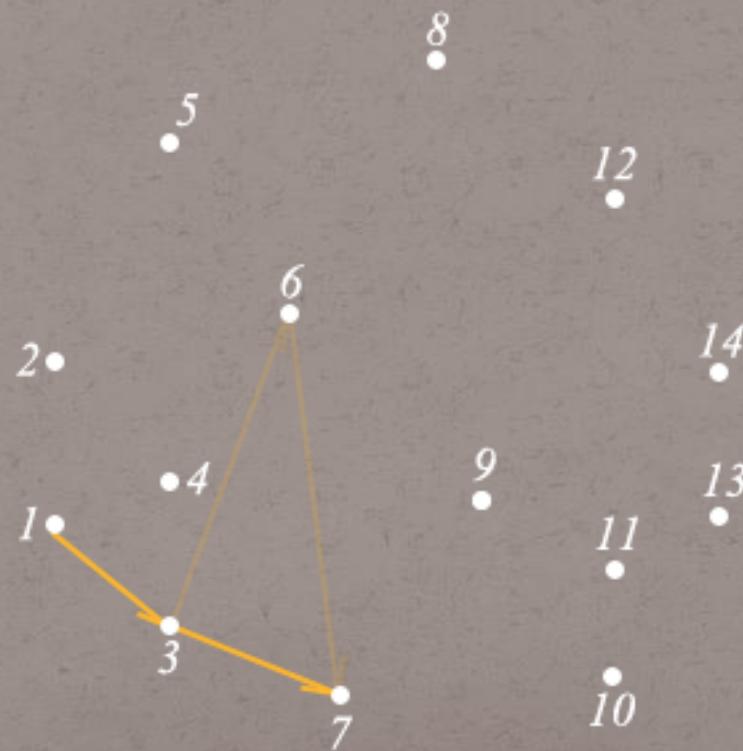
# 算法演示

5 被弹出栈，6 进栈。



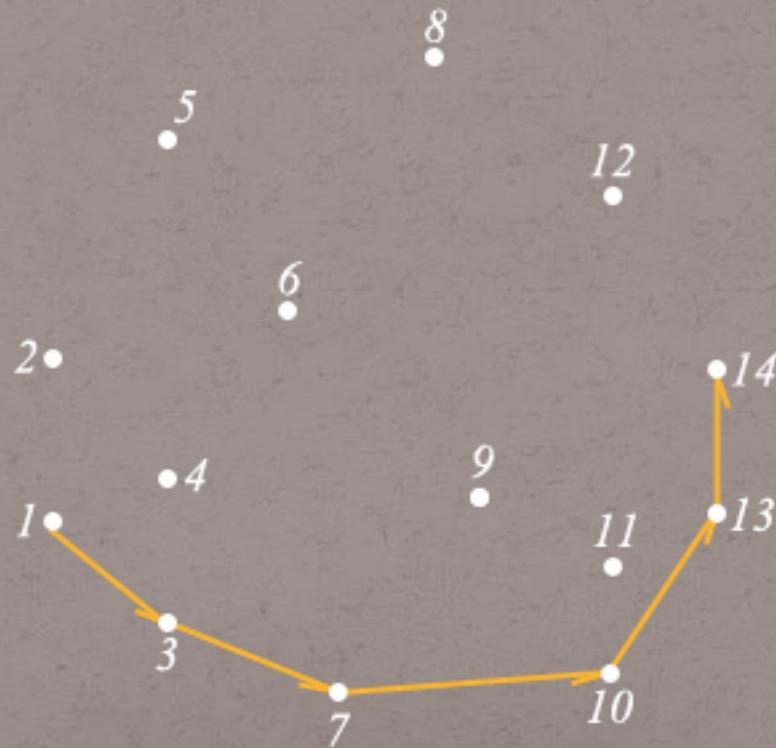
# 算法演示

6 被弹出栈，7 进栈。



# 算法演示

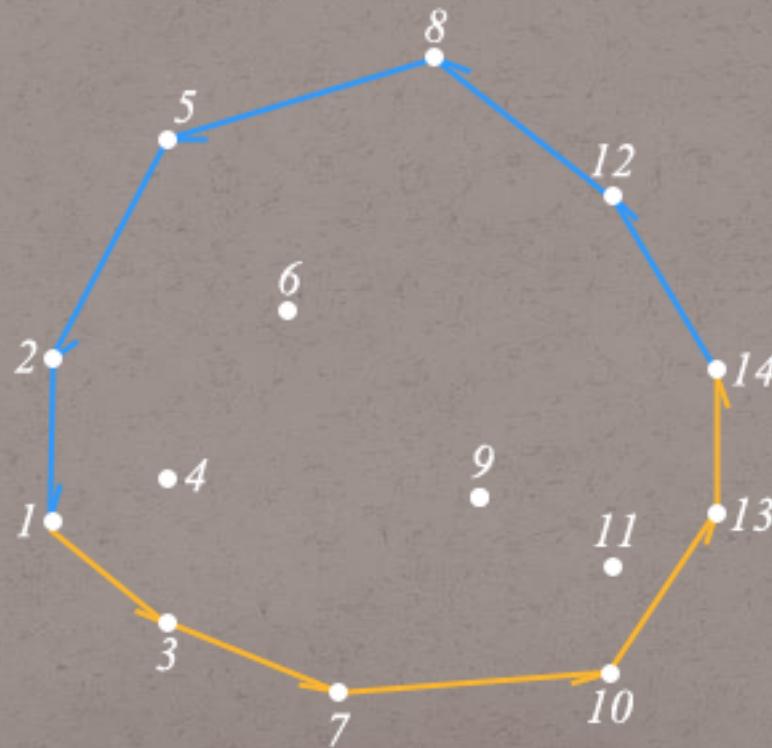
如此直到下凸壳被找到。



# 算法演示

倒序进行扫描找到上凸壳。

- 以y作为第一关键字排序，先求右半凸包，再求左半凸包也一样



```
//POJ1113 Wall , 水平序扫描法求凸包 by Guo Weiint
Graham(vector<Point> & points, vector<Point> &
stack) {
    //在凸包上，但不是顶点的点，没有抛弃，留在stack里面
    if( points.size() < 3)
        return 0;
    stack.clear();
    //先按坐标排序，最左下的放到points[0]
    sort(points.begin(),points.end());
    stack.push_back(points[0]);
    stack.push_back(points[1]);
    int n = points.size();
```

```
for(int i = 2; i< n; ++i) { //做右半凸包
    while(stack.size()>1) //定要这一条
        Point p2 = * (stack.end()-1);
        Point p1 = * (stack.end()-2);
        if( Sign(Cross(p2-p1,points[i]-p2) < 0)) {
            //p2->points[i] 向右转，才让p2出栈，这样能保留凸
            包边上的点
            stack.pop_back();
        }
    else
        break;
}
stack.push_back(points[i]);
}
```

```
int size = stack.size();
//此时栈顶定是points[n-1]
stack.push_back(points[n-2]);
for(int i = n-3; i >= 0;--i) { //做左半凸包
    while(stack.size() > size) {
        Point p2 = * (stack.end()-1);
        Point p1 = * (stack.end()-2);
        if( Sign(Cross(p2-p1,points[i]-p2) < 0))
            stack.pop_back();
        else
            break;
    }
    stack.push_back(points[i]);
}
stack.pop_back();
return stack.size();
}
```

# *Grandpa's Estate*

- POJ 1228
- 一个由钉子加绳子围成的凸多边形农场（原本所有钉子和绳子都在凸多边形的边上）。现在绳子和部分钉子缺失，问能否通过剩余的钉子确定原农场的样子。

# *Grandpa's Estate*

- POJ 1228
- 做凸包。当围成的凸包每条边上都有至少3个钉子（含端点），则可确定原农场所样子。

# Grandpa's Estate

- POJ 1228
- 做凸包。当围成的凸包每条边上都有至少3个钉子（含端点），则可确定原农场样子。
- 因为如果只有两个钉子，那么这两条边的性质就不同了。如果这两个钉子位于凸包的同一条边上，那么它们之间的部分就是凹的，而其他部分是凸的。因此，如果在一条边上只有两个钉子，那么这条边就是凹的，其他的边就是凸的。这样，我们就可以根据钉子的位置来判断每条边的性质，从而确定整个农场的形状。

# 半平面交

- 一条直线将平面分为两个半平面  
直线是有向的，不妨规定沿直线方向的左边  
是我们关心的直线对应的半平面（包含直线）
- 半平面交
  - 求被所有给定半平面包含的点的集合
- 性质
  - 半平面交的结果是一个凸区域

# 增量法

- 一开始先构造一个足够大的矩形，第一个半平面和它求交。然后每次拿求交的结果（一个凸多边形，由顶点集合表示）再和下一个半平面求交。
- 凸多边形和一个半平面（实际上是一条直线）如何求交：
  - 按逆时针序对每条边进行处理。
  - 如果边起点位于新平面上，则将其加入结果
  - 求边与直线的交点，若有交点，则将交点添加到结果
- 时间复杂度 $O(n^2)$
- 半平面求交有其他  $n \log n$  的算法

```
typedef vector<Point> Polygon;

int CutPolygon(const Polygon & src, Point a, Point b,
               Polygon & result)

{ //用直线a->b 切割src, 返回其左侧的多边形，放到result
  //src里的点什么顺序无所谓

  int n = src.size();

  result.clear();

  for (int i = 0; i < n; ++i) {
    Point c = src[i];
    Point d = src[(i + 1) % n];
    if (Sign(Cross(b - a, c - a)) >= 0) //叉积>=0
      result.push_back(c);
    pair<int, Point> r = CrossPoint(Seg(c, d),
                                      Seg(a, b));
    //c,d在 a->b上，或者 c->d重合于 a->b这两种情况都不要考虑，因为会在上面两行处理了
```

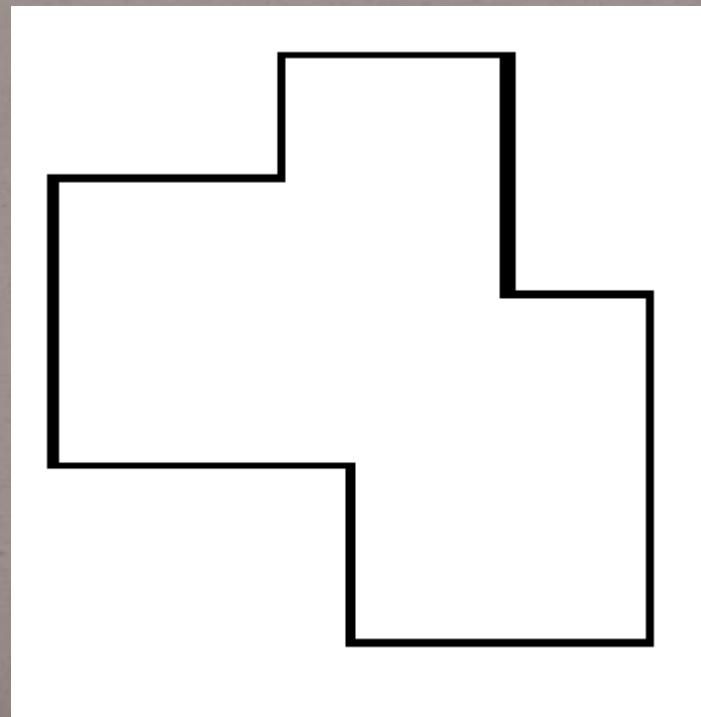
```
    if (r.first == 0 || r.first == 8 || r.first == 3)
//规范相交或虽非规范相交但是交点在c->d上
        result.push_back(r.second);
}
return result.size();
}
```

# *Video Surveillance*

- POJ 1474
- 给出一个简单多边形，其各边与坐标轴平行，询问其是否存在一点，可以“看到”所有的边。

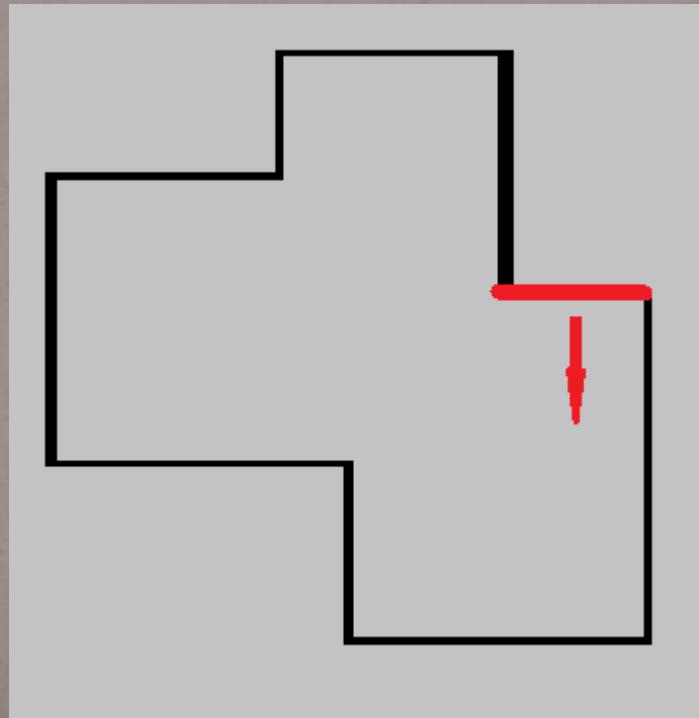
# *Video Surveillance*

- 开始的多边形是一个足够大的矩形，把原图包在内部
- 每次用一个半平面去切割当前多边形



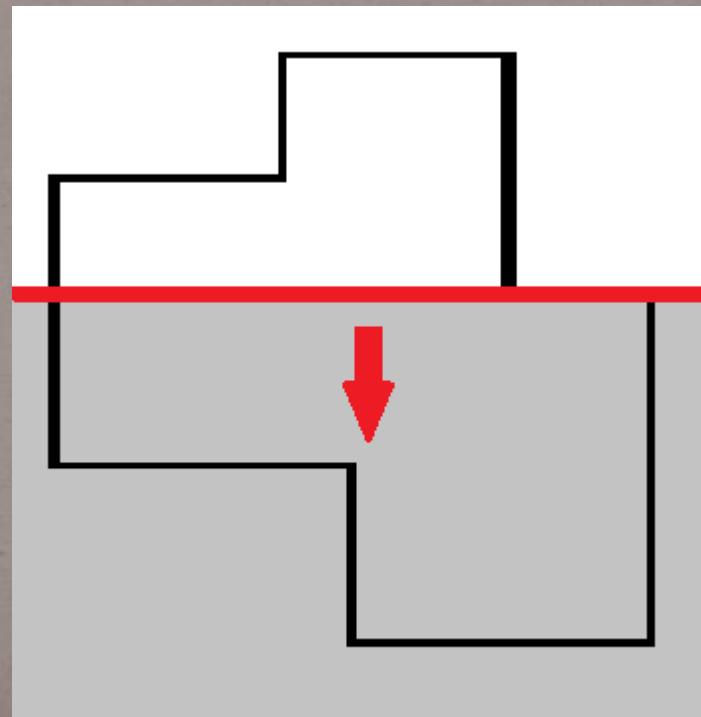
# *Video Surveillance*

- 开始的多边形是一个足够大的矩形，把原图包在内部
- 每次用一个半平面去切割当前多边形



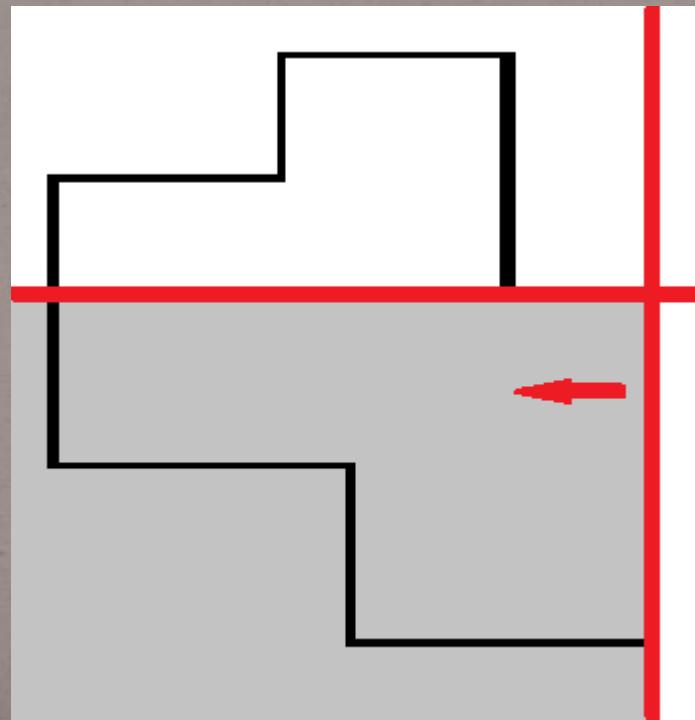
# *Video Surveillance*

- 开始的多边形是一个足够大的矩形，把原图包在内部
- 每次用一个半平面去切割当前多边形



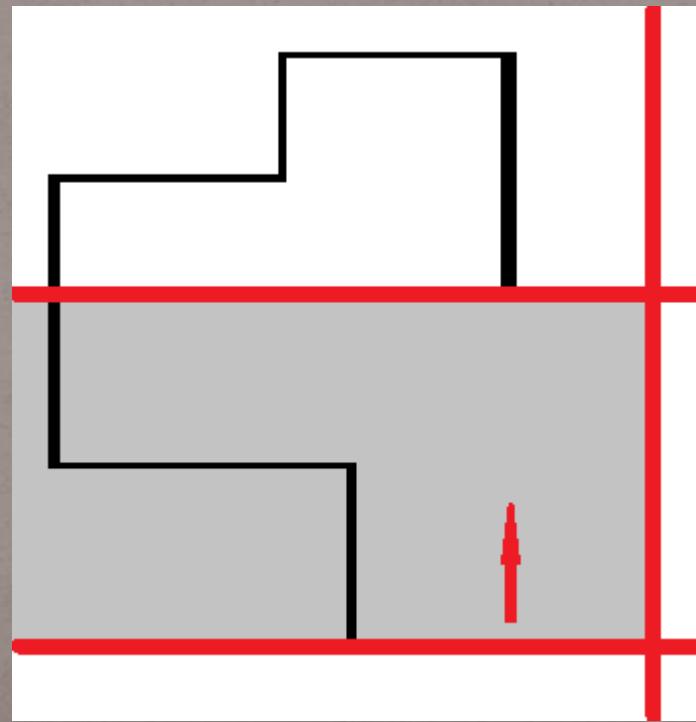
# *Video Surveillance*

- 开始的多边形是一个足够大的矩形，把原图包在内部
- 每次用一个半平面去切割当前多边形



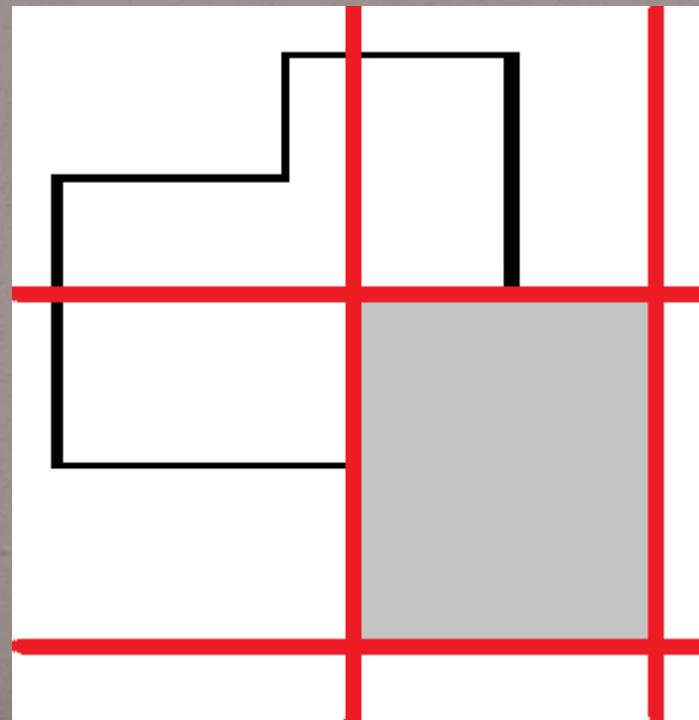
# *Video Surveillance*

- 开始的多边形是一个足够大的矩形，把原图包在内部
- 每次用一个半平面去切割当前多边形



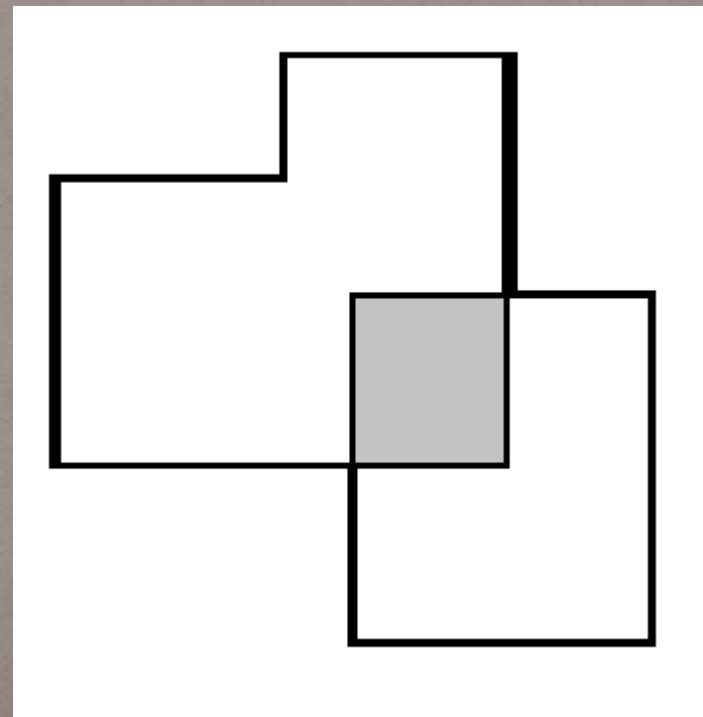
# *Video Surveillance*

- 开始的多边形是一个足够大的矩形，把原图包在内部
- 每次用一个半平面去切割当前多边形



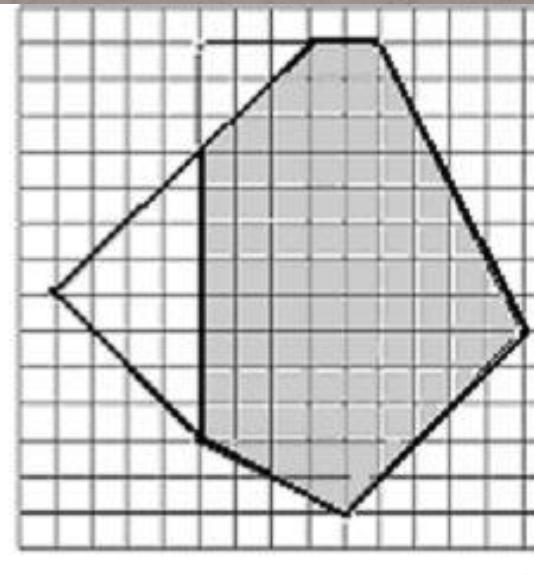
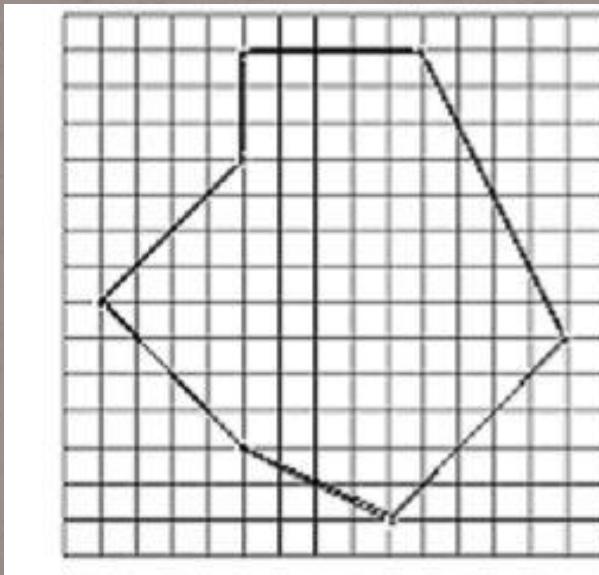
# *Video Surveillance*

- 开始的多边形是一个足够大的矩形，把原图包在内部
- 每次用一个半平面去切割当前多边形



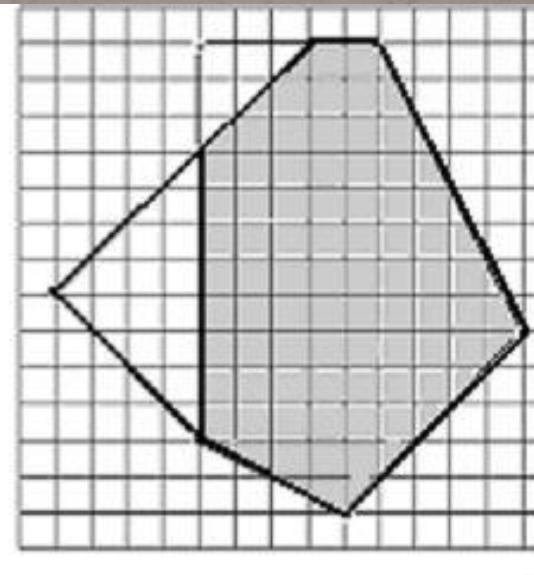
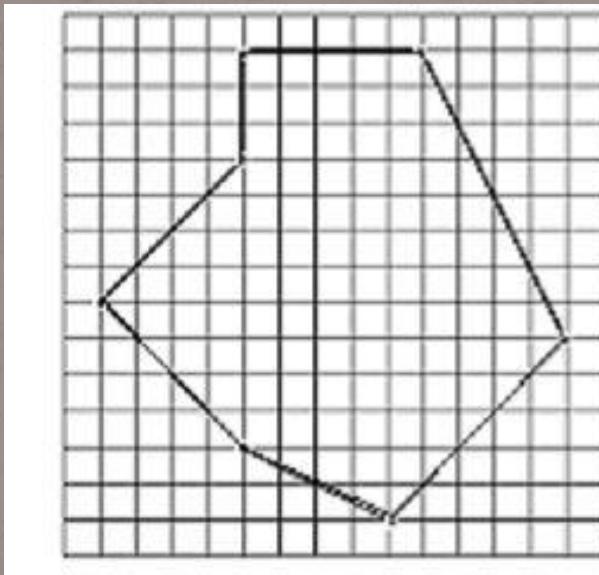
# Art Gallery

- POJ 1279
- 求一个凸多边形的核（能看到所有点的点集合）的面积



# Art Gallery

- 求所有边形成的半平面的交
- 要从输入样例判断给的点是顺时针还是逆时针序



# *Most Distant Point from the Sea*

- POJ 3525
- 求一个凸多边形内部点与边界的距离的最大值

# Most Distant Point from the Sea

- POJ 3525
- 求一个凸多边形内部点与边界的距离的最大值
- 二分答案 $L$ , 将每条边向内推进 $L$ , 然后求每条边的半平面的交, 若有解则继续增大距离; 否则减小距离; 直到交里恰好只有一个点

