

计算几何基础

1. 计算几何中的向量表示

1.1 点的表示：

```
Struct Point{  
    double x;  
    double y;  
}
```

1.2 向量的表示

```
Struct V{  
    Point start;  
    Point end1;  
}
```

1.3 向量的点积

```
double dotProduct(V v1, V v2 )  
{  
    V vt1, vt2;  
    vt1.start.x=0;  
    vt1.start.y=0;  
    vt1.end1.x=v1.end1.x-v1.start.x;  
    vt1.end1.y=v1.end1.y-v1.start.y;  
  
    vt2.start.x=0;  
    vt2.start.y=0;  
    vt2.end1.x=v2.end1.x-v2.start.x;  
    vt2.end1.y=v2.end1.y-v2.start.y;  
  
    return vt1.end1.x*vt2.end1.x+vt1.end1.y*vt2.end.y;  
}
```

1.4 向量的叉积

```
double crossProduct(V v1,V v2)//是 v1 叉乘 v2
{
    V vt1, vt2;
    vt1.start.x=0;
    vt1.start.y=0;
    vt1.end1.x=v1.end1.x-v1.start.x;
    vt1.end1.y=v1.end1.y-v1.start.y;

    vt2.start.x=0;
    vt2.start.y=0;
    vt2.end1.x=v2.end1.x-v2.start.x;
    vt2.end1.y=v2.end1.y-v2.start.y;

    return vt1.end1.x*vt2.end1.y-vt2.end.x*vt1.end1.y;
}
```

2. 点的定位

2.1 判断点是否再直线上

需要满足两个条件

1. 叉积为 0
2. 点 Q 的 x 坐标与 y 坐标均处于两个端点之间

```
bool onSegment (Point pi, Point pj, point Q )
{
    if( (Q.x-pi.x)*(pj.y-pi.y)==(pj.x-pi.x)*(Q.y-pi.y)&&min(pi.x,pj.x)<=Q.x&&Q.x<=max(pi.x,pj.x)&&
    min(pi.y,pj.y)<=Q.y&&Q.y<=max(pi.y,pj.y) )
        return true;
    return false;
}
```

因为线段可能是竖着的也可能是横着的，所以 x 和 y 坐标都要进行限制

2.2.判断点在三角形的内外

从 Q 点向 A,B,C 引三条直线，判断三个小三角形面积之和与大三角形面积的大小

```
struct triangle
```

```

{
    Point A,B,C;
};

bool inTriangle( triange t,Point p )
{
    V AB,AC,PA,PB,PC;
    AB.start=t.A;
    AB.end=t.B;
    PA.start=p;
    PA.end=t.A;
    .....

    double Sabc=fabs( crossProduct(AB,AC) );
    double Spab=fabs(crossProduct(PA,PB) );
    double Spac=fabs(crossProduct(PC,PA));
    double Spbc=fabs(crossProduct(PB,PC))
    If( Spab+Spac+Spbc==Sabc ) return true;
    return false;
}

```

2.3 判断点在多边形的内外

```

const double eps=1e-5;
inline double dabs(double a){ return a<0? -a:a; }
Point poly[maxn];
int n,m;
Bool insidepolygon( Point p )
{
    Int counter=0;
    Double xinters;
    Point p1,p2;
    for(int i=1;i<=n;i++)
    {
        P2=poly[i%n];
        If( onSegment(p1,p2,p) ) return true;
        If( p.y>min(p1.y,p2.y) ){
            if( p.y<=max(p1.y,p2.y) ){
                If(p.x<=max(p1.x,p2.x)){
                    if( p1.y!=p2.y ){
                        xinters=(p.y-p1.y)*(p2.x-p1.x)/(p2.y-p1.y)+p1.x;
                        If( p1.x==p2.x || p.x<=xinters ) couter++;
                    }
                }
            }
        }
    }
}

```

```

        }
    }
    p1=p2;
}

If( counter%2==0 )
return false;
return true;
}

```

2.4 判断两线段相交

```

Double multi (Point p1,Point p2,Point p0)
{
    Return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}

Int Aross(V v1,V v2)
{
    If(max(v1.start.x,v1.end1.x)>=min(v2.start,v2.end1.x)
    &&max(v2.start.x,v2.end1.x)>=min(v1.start.x,v1.end1.x)
    &&max(v1.start.y,v1.end1.y)>=min(v2.start.y,v2.end1.y)
    &&multi(v2.start,v1.end,v1.start)*multi(v1.end,v2.end,v1.start)>0
    &&multi(v1.start,v2.end,v2.start)*multi(v2.end1,v1.end1,v2.start )>0 )
    Return 1;
    Return 0;
}

```

但是也包含了端点在线段上的情况，如果要进一步判断端点是否在线上需要在加一步点在线上的判断。

问题：1.一组直线能把原先的一个平面分成多少个平面

$$f(n)=f(n-1)+tn+1;$$

$f(n-1)$:n-1 条直线可以把平面分成的平面个数。

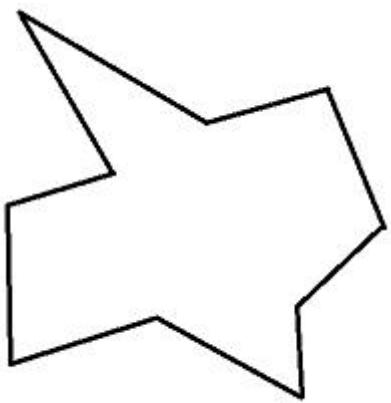
tn: 第 n 条直线加进来后与其他直线相交所得到的点

3. 半平面求交

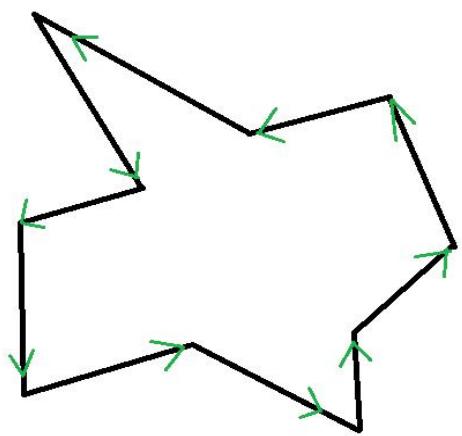
3.1 应用场所

用于求多边形内的一个区域可以看到多边形的各个角落
把一个圆放进多边形里，求这个圆的半径最大可能是多大

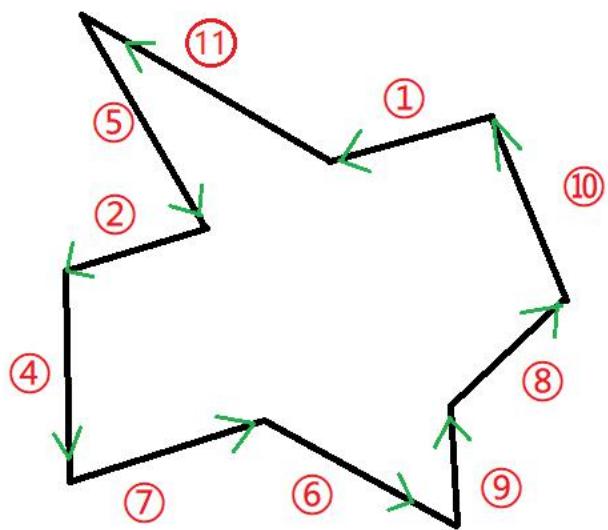
3.2 排序增量法：



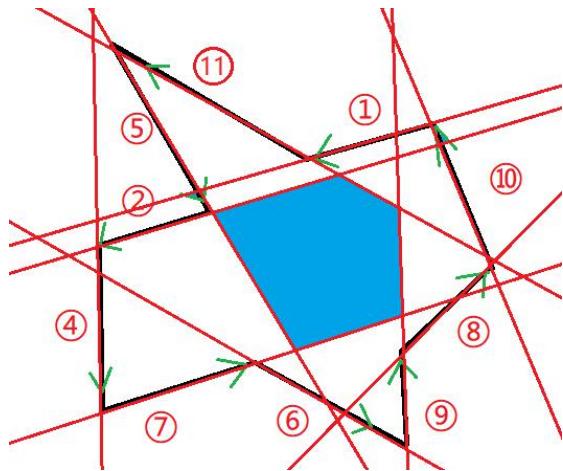
1.逆时针取环方向



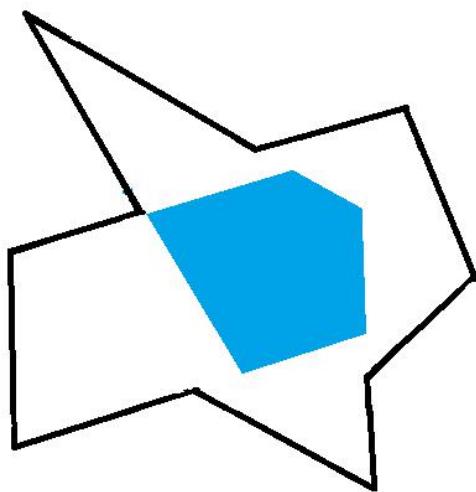
2.极角排序



3 按顺序遍历每条线段，取左边区域，删右边区域



4. 最后的区域



最后的到的区域是由几个直线围成的区域

$$L = \{2, 5, 7, 9, 11\}$$

如果题目还要求算出区域面积，则计算出几条直线的交点然后求一个凸包

例题：一个博物馆，多边形形状。要设置一个守卫，可以看到博物馆里的每一个角落。问是否存在这个点。

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;
const int maxn = 1e3;
const double EPS = 1e-5;
int T, n;
typedef struct Grid {
    double x, y;
    Grid(double a = 0, double b = 0) {x = a, y = b;}
} Point, Vector;
Vector operator - (Point a, Point b) {return Vector(b.x - a.x, b.y - a.y);}
double operator ^ (Vector a, Vector b) {return a.x * b.y - a.y * b.x;}//叉乘
struct Line {
    Point s, e;
    Line() {}
    Line(Point a, Point b) {s = a, e = b;}
};
Point p[maxn];
Line L[maxn], que[maxn];

//得到极角角度
double getAngle(Vector a) {
    return atan2(a.y, a.x);
}

//得到极角角度
double getAngle(Line a) {
    return atan2(a.e.y - a.s.y, a.e.x - a.s.x);
}

//排序：极角小的排前面，极角相同时，最左边的排在最后面，以便去重
bool cmp(Line a, Line b) {
    Vector va = a.e - a.s, vb = b.e - b.s;
    double A = getAngle(va), B = getAngle(vb);
    if (fabs(A - B) < EPS) return ((va) ^ (b.e - a.s)) >= 0;
    return A < B;
}
```

```

//得到两直线相交的交点
Point getIntersectPoint(Line a, Line b) {
    double a1 = a.s.y - a.e.y, b1 = a.e.x - a.s.x, c1 = a.s.x * a.e.y - a.e.x * a.s.y;
    double a2 = b.s.y - b.e.y, b2 = b.e.x - b.s.x, c2 = b.s.x * b.e.y - b.e.x * b.s.y;
    return Point((c1*b2-c2*b1)/(a2*b1-a1*b2), (a2*c1-a1*c2)/(a1*b2-a2*b1));
}

//判断 b,c 的交点是否在 a 的右边
bool onRight(Line a, Line b, Line c) {
    Point o = getIntersectPoint(b, c);
    if (((a.e - a.s) ^ (o - a.s)) < 0) return true;
    return false;
}

bool HalfPlaneIntersection() {
    sort(L, L + n, cmp); //排序
    int head = 0, tail = 0, cnt = 0; //模拟双端队列
    //去重， 极角相同时取最后一个。
    for (int i = 0; i < n - 1; i++) {
        if (fabs(getAngle(L[i]) - getAngle(L[i + 1])) < EPS) {
            continue;
        }
        L[cnt++] = L[i];
    }
    L[cnt++] = L[n - 1];

    for (int i = 0; i < cnt; i++) {
        //判断新加入直线产生的影响
        while (tail - head > 1 && onRight(L[i], que[tail - 1], que[tail - 2])) tail--;
        while (tail - head > 1 && onRight(L[i], que[head], que[head + 1])) head++;
        que[tail++] = L[i];
    }
    //最后判断最先加入的直线和最后的直线的影响
    while (tail - head > 1 && onRight(que[head], que[tail - 1], que[tail - 2])) tail--;
    while (tail - head > 1 && onRight(que[tail - 1], que[head], que[head + 1])) head++;
    if (tail - head < 3) return false;
    return true;
}

//判断输入点的顺序，如果面积 <0，说明输入的点为逆时针，否则为顺时针
bool judge() {
    double ans = 0;

```

```

for (int i = 1; i < n - 1; i++) {
    ans += ((p[i] - p[0]) ^ (p[i + 1] - p[0]));
}
return ans < 0;
}

int main()
{
    scanf("%d", &T);
    while (T--) {
        scanf("%d", &n);
        for (int i = n - 1; i >= 0; i--) {
            scanf("%lf %lf", &p[i].x, &p[i].y);
        }

        if (judge()) {//判断输入顺序，保证逆时针连边。
            for (int i = 0; i < n; i++) {
                L[i] = Line(p[(i + 1)%n], p[i]);
            }
        } else {
            for (int i = 0; i < n; i++) {
                L[i] = Line(p[i], p[(i + 1)%n]);
            }
        }

        if (HalfPlaneIntersection()) printf("YES\n");
        else printf("NO\n");
    }

    return 0;
}

```

4 解析几何

4.1 求三角形的面积

两条边的向量的叉积的二分之一就是三角形面积

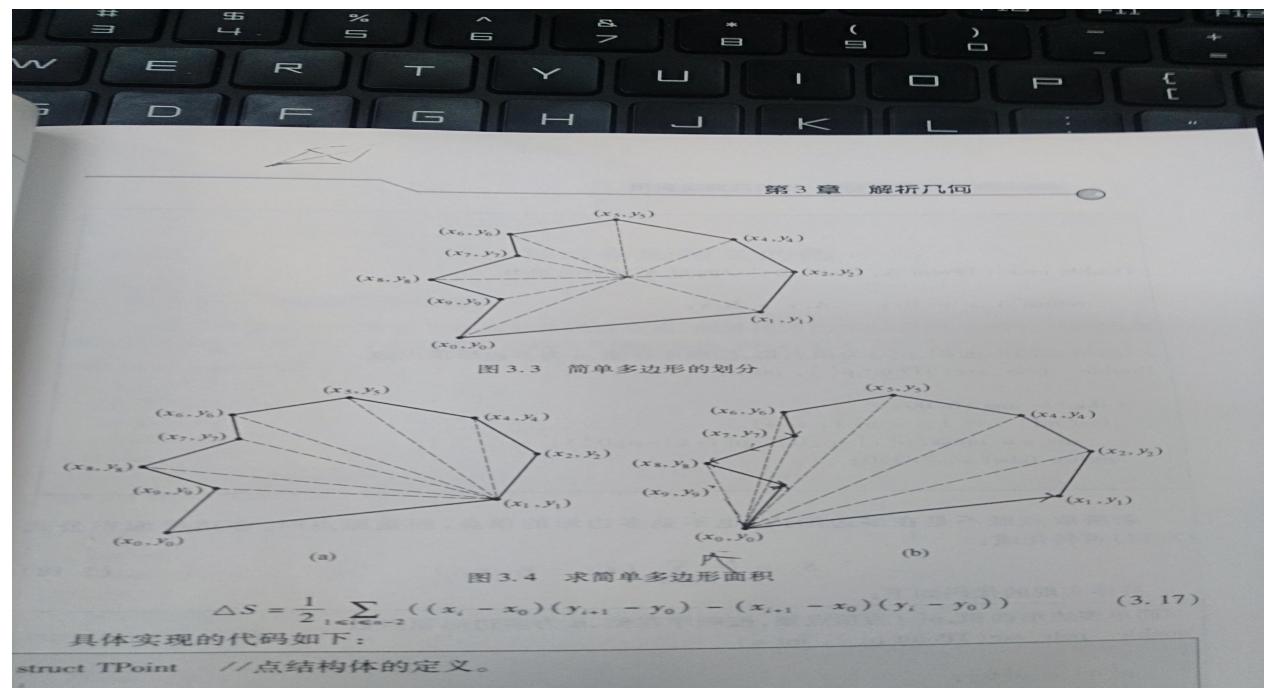
```

double triangleArea(Point A,Point B,Point C)
{
    return fabs( (B.x-A.x)*(C.y-A.y) - (C.x-A.x)*(B.y-A.y) )/2;
}

```

4.2 多边形面积的计算

固定一个点连接它与其余的点，用叉积求得此点和其余点构成的三角形面积，最后相加就得到了多边形的面积



$$\Delta S = \frac{1}{2} \sum_{i=0}^{n-2} ((x_i - x_0)(y_{i+1} - y_0) - (x_{i+1} - x_0)(y_i - y_0)) \quad (3.17)$$

具体实现的代码如下：

```
struct TPoint //点结构体的定义。
```

```
Struct Point{
    Double x,y;
    Point operator -(Point &b) const{
        Return Point (x-b.x,y-b.y);
    }
};

Double cross(Point A,Point B)
{
    Return A.x*B.y-A.y*B.x;
}

Double poly_area(Point p[],int n)
{
    Double ans=0.0;
    for(int i=1;i<n-1;i++)
        ans+=cross( p[i]-p[0],p[i+1]-p[0] );
    Return fabs(ans)/2.0;
}
```

4.3 三角形的外接圆

$S_{\triangle} = abc/4R$ (R 是三角形外接圆的半径)

圆的表示

```
Struct Circle{
    Double r;
    Point centre;
}
```

```
Struct Triangle{
    Point A,B,C;
}
```

```
Double distance( Point a, Point b )
{
    Return sqrt( (a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y) );
}
```

求外接圆

```
Circle circu{
    Circle tmp;
    Double a,b,c,c1,c2;
    a=distance(A,B );
    b=distance(A,C);
    c=distance(B,C);

    tmp.r=a*b*c/triangleArea(A,B,C)/4;
    Double x_a,y_a,x_b,y_b,x_c,y_c;
    x_a=A.x, y_a=A.y;
    .....
    c1=( x_a*x_a + y_a*y_a - x_b*x_b - y_b*y_b )/2;
    c2=( x_a*x_a + y_a*y_a - x_c*x_c - y_c*y_c )/2;
    tmp.centre.x=( c1*(y_a-y_c) - c2*( y_a-y_b ) )/(
        (x_a-x_b)*(y_a-y_c)-(x_a-x_c)*(y_a-y_b) );
    tmp.centre.y=( c1*(x_a-x_c) - c2*( x_a-x_b ) )/(
        (y_a-y_b)*(x_a-x_c)-(y_a-y_c)*(x_a-x_b) )

    Return tmp;
}
```

4.4 三角形的内切圆

三角形面积=($a+b+c$)* $r/2$; (r 是三角形的内切圆半径)

三角形内切圆的圆心是三角形内角平分线的交点。

三角形三条边的向量:

$\overrightarrow{BC}=a$, $\overrightarrow{CA}=b$, $\overrightarrow{AB}=c$;

内心坐标 M : (x_m, y_m)

则有, $a^*MA+b^*MB+c^*MC=0$;

$MA=(x_a-x_m, y_a-y_m)$;

$MB=(x_b-x_m, y_b-y_m)$;

$MC=(x_c-x_m, y_c-y_m)$;

4.5 对称

点的对称

若 A, B 关于点 C 对称, 那么 $(x_A+x_B)/2=x_C$, $(y_A+y_B)/2=y_C$

求 $p1$ 关于 $p2$ 的对称点

Point duicheng(Point p1,Point p2)

{

 Point p3;

 p3.x=2*p2.x-p1.x;

 p3.y=2*p2.y-p1.y;

 Return p3;

}

点关于直线的对称点

已知 $A(x_1, y_1)$, 直线 $l: ax+by+c=0$, 则 A' 的坐标为

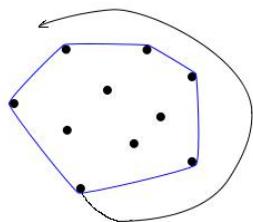
5 凸包问题

什么是凸多边形: 从多边形的任意一个边做一条直线, 多边形都在这条直线的一侧。

什么是凸包: 用一个面积最小的凸多边形把所有的点都包围起来

求凸包的方法

卷包裹法：



拿根线从最底下的点开始拿一根线开始逆时针绕，这根线只会经过最外边的点。这样绕出来的一个多边形就是凸包。类比墙上一堆钉子，要把这些钉子用一条线全部绕在里面，最节省材料的做法，肯定是用线把最外围一圈钉子给它绕一下。

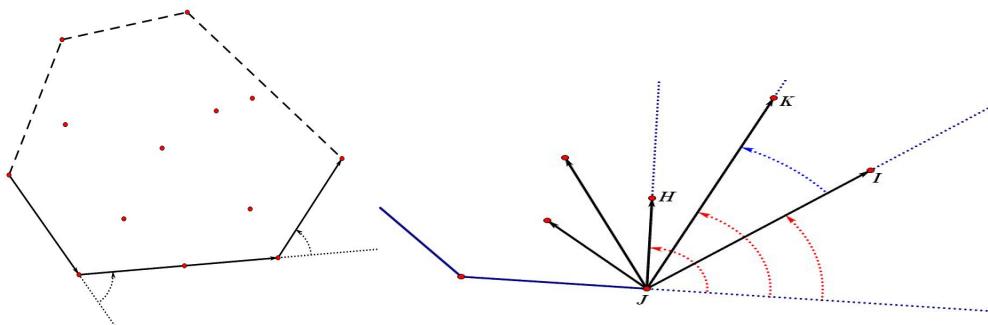
时间复杂度： $O(nh)$ h 是落在凸包边界上的点的个数

实际操作步骤：

1. 对所有的点进行排序，按照 y 坐标从小到大排，如果 y 坐标相同就选择 x 坐标最左边的。目的是找到最底下的点。

```
bool cmp(Point a,Point b)
{
    return (a.y<b.y) | (a.y==b.y&&a.x<b.x);
}
```

2. 先把排好序的第一个点和第二个点加入队列， sta ，然后从第三个点开始循环到最后一个点，判断每一次新加进去的点对队列里的点的影响，如果新加进去的点使得一些点变成了靠近里面的点，就删除这个点。并且把这个新的点加进队列。到最后队列里保存的都是最外围的点。



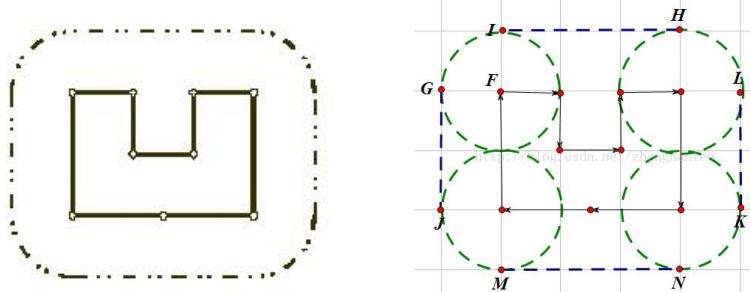
3. 如何判断哪些点更靠近外围，用叉积的方法，例如 JL 的向量转到 JK 是逆时针，则向量 JL 叉积 $JK > 0$ ，这样 L 点就在 K 点的右边，则 L 点比 J 点更靠近外围。

4. 从 1 循环到 n 做一遍叉积的判断只能判断出哪些点在最右围。这样的话最靠近左边界点由于在最左边，永远也加不进队列里，这样队列里保存的边界点是不完整的，所以为了把靠近左边的点也加进队列，从 n 到 1 还要循环一遍做一次叉积，把左边界点加进去。这样边

界就是完整的了。

5.最后求凸包的周长或者是面积都在这个保存了最外围点的队列上进行。

例题：国王要建一堵墙，围绕它的城堡，且城与墙之间的距离不小于一个整数 L 。输入城堡的各个节点的坐标和 L ，问你最小的墙壁周长。（[poj1113](#)）



其实外围的城墙如果不看四个顶点处的圆弧其实它的周长就等于组成城堡的点的凸包的周长。所以 $C = \text{凸包周长} + \text{四个圆弧周长}$

```
#include<iostream>
#include<cstdio>
#include<cmath>
#include<algorithm>
using namespace std;
const int maxn=1100;

int n,L,ans[maxn],cnt,sta[maxn],tail;
struct Point{
    int x,y;
}poly[maxn];

bool cmp(Point a,Point b)
{
    return (a.y<b.y||a.y==b.y&&a.x<b.x);
}

double distance0(Point a,Point b)
{
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}

bool CrossLeft(Point p1,Point p2,Point p3)
{
    return ((p3.x-p1.x)*(p2.y-p1.y)-(p2.x-p1.x)*(p3.y-p1.y)<0 );
}

void Jarvis()
```

```

{
    tail=cnt=0;
    sort(poly, poly+n, cmp);
    sta[tail++]=0, sta[tail++]=1;
    for(int i=2;i<n;i++)
    {
        while(tail>1&&!CrossLeft(poly[sta[tail-1]], poly[sta[tail-2]], poly[i]))
            tail--;
        sta[tail++]=i;
    }

    for(int i=0;i<tail;i++)
        ans[cnt++]=sta[i];

    tail=0;sta[tail++]=n-1;sta[tail++]=n-2;
    for(int i=n-3;i>=0;i--)
    {
        while(tail>1&&!CrossLeft(poly[sta[tail-1]], poly[sta[tail-2]], poly[i]))
            tail--;
        sta[tail++]=i;
    }

    for(int i=0;i<tail;i++)
        ans[cnt++]=sta[i];
}

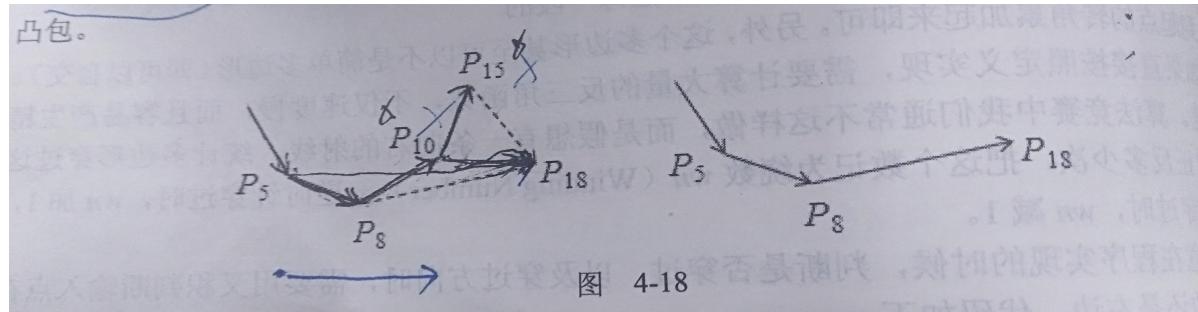
int main()
{
    scanf("%d%d", &n, &L);
    for(int i=0;i<n;i++)
    {
        scanf("%d%d", &poly[i].x, &poly[i].y);
    }
    Jarvis();
    double re=4*acos(0.0)*L;
    for(int i=0;i<cnt-1;i++)
    {
        re+=distance0(poly[ans[i]], poly[ans[i+1]]);
    }

    printf("%.0f\n", re);
    return 0;
}

```

Graham 算法：

时间复杂的 $O(n \log(n))$



先按照 x 坐标从小到大排序， x 坐标相同按照 y 坐标从小到大排序。先把第一个点和第二个点入栈。然后从第三个点到最后一个点遍历，每遍历一个新的点看它对栈中点的影响，如果他能够使得栈中的一些点变成靠近里面的点，则将栈中受到影响的点出栈，直到栈中不再有受到它的影响。把这个新点入栈。最后栈中只剩下凸包边界上的点。

实际操作步骤：

和上面的卷包裹法没什么太大区别。

例题：

给出一组包含原点在内的点，求出这组点的凸包的各个顶点，并且按照逆时针方向，从原点开始输出整个凸包的顶点。

输入：从第一行开始，每行两个整数 X_i, Y_i ，表示一个点的横坐标和纵坐标。原点不一定在第一行，输入保证三点不共线。

输出：从原点开始，按逆时针顺序，每行输出一个点格式为 (X_i, Y_i) 。

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

```
const int maxn=1010;
struct Point{
    double x,y;
}p[maxn];

int n,res[maxn],top;

bool cmp(Point a,Point b)
{
    return a.y<b.y||a.y==b.y&&a.x<b.x;
}
```

```

bool CrossLeft(Point sp,Point ep,Point op)
{
    return (sp.x-op.x)*(ep.y-op.y)>=(ep.x-op.x)*(sp.y-op.y);
}

void Graham()
{
    int i,len;
    top=1;
    sort(p,p+n,cmp);
    if(n==0) return ;res[0]=0;
    if(n==1) return ;res[1]=1;
    if(n==2) return ;res[2]=2;
    for(i=2;i<n;i++)
    {
        while(top&&CrossLeft(p[i],p[res[top]],p[res[top-1]])) top--;
        res[++top]=i;
    }

    len=top;
    res[++top]=n-2;
    for(i=n-3;i>=0;i--)
    {
        while(top!=len&&CrossLeft(p[i],p[res[top]],p[res[top-1]])) top--;
        res[++top]=i;
    }
}

int main()
{
    int s,i;
    while(scanf("%lf%lf",&p[n].x,&p[n].y)!=EOF) n++;
    Graham();

    for(s=0;s<top;s++)
        if(!p[res[s]].x&&!p[res[s]].y) break;

    for(i=s;i<top;i++)
        printf("%.1f%.1f\n",p[res[i]].x,p[res[i]].y);
    for(i=0;i<s;i++)
        printf("%.1f%.1f\n",p[res[i]].x,p[res[i]].y);
    return 0;
}

```

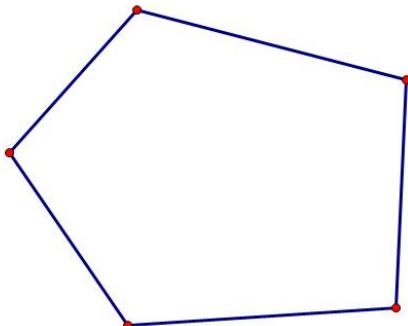
```
/*
0 0
70 -50
60 30
-30 -50
80 20
50 -60
90 -20
-30 -40
-10 -60
90 10
```

```
*/
```

旋转卡壳：用于求凸多边形的直径

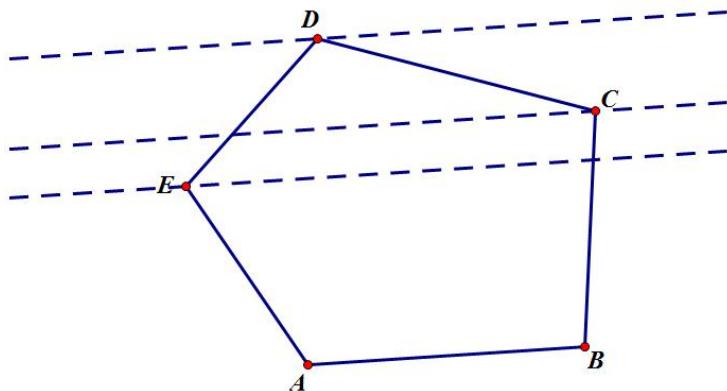
例题：这里有众多的点，现在要求你求出其中距离最远的两个点的距离。
很明显这两个点一定在这些点的凸包上，所以先求出凸包，再继续在凸包的边界上用旋转卡壳法求出最远的两个点的距离，即直径。

朴素做法枚举凸包上两两点，时间复杂度 $O(n^2)$



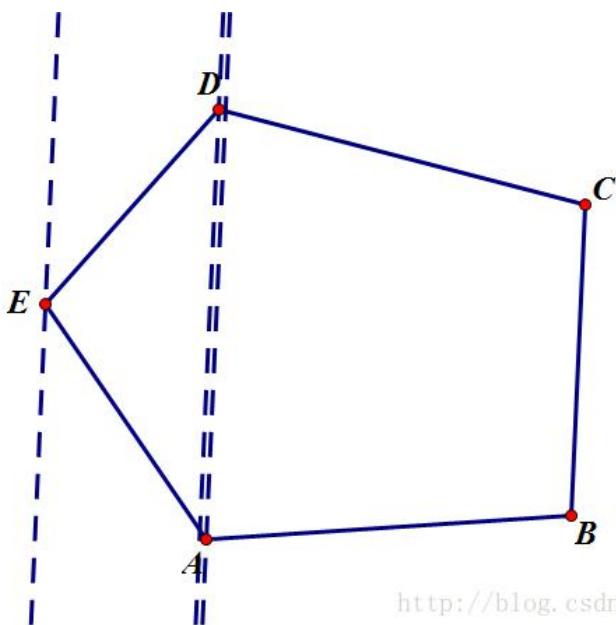
http://blog.csdn.net/qq_30974369

但是我们可以发现两个性质。**1：**某个点如果到某条边的距离是最远的，那么距离这个点最近的点一定是这条边两个端点之一的点。**2：**随着边逆时针旋转，到这条边的最近的点也在逆时针旋转。



a

http://blog.csdn.net/qq_30974369



http://blog.csdn.net/qq_30974369

所以我们去枚举每条边，然后再去枚举点，当找到某条边的最远点后，下一次计算下一条边最远的点时，这个点一定在上一条边的最远点再往逆时针方向转，这样我们就不用从头枚举每一点了，所以这样每一个点都会顶多被枚举一次。所以算法复杂度 $O(n)$

```
long long GetMax() //求出直径
{
    rg long long re=0;
    if(top==1) //仅有两个点
        return Dis(S[0], S[1]);
    S[++top]=S[0];//把第一个点放到最后
    int j=2;
    for(int i=0; i<top; ++i)//枚举边
    {

```

```

while(Cross(S[i], S[i+1], S[j])<Cross(S[i], S[i+1], S[j+1]))
    j=(j+1)%top;
    re=max(re, max(Dis(S[i], S[j]), Dis(S[i+1], S[j])));
}
return re;
}

```

三维凸包

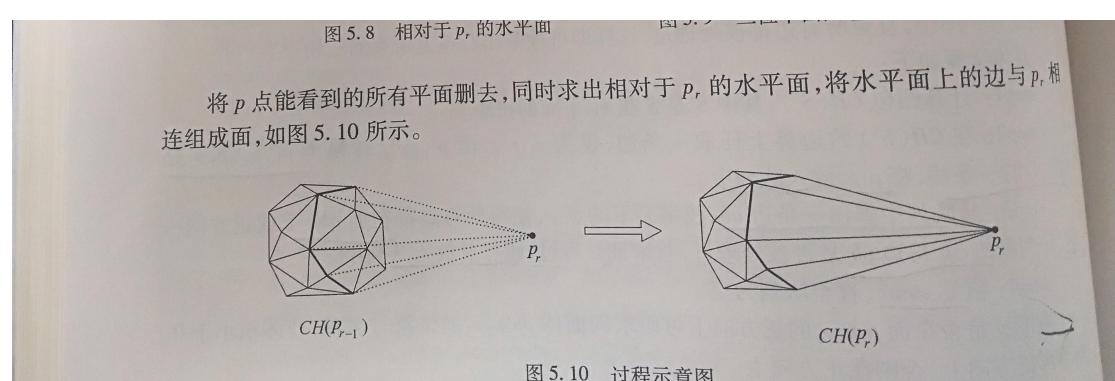
二维凸包是一个多变形，三维凸包是最小的凸面体。

点增量法 ($O(n^2)$)

先找到两个不同的点，在找到不共线的第三个点，在找到不共面的第四个点，这四个点组成了一个初始的四面体。接下来考虑剩下的点

1. 新点在当前凸包内部，忽略该点
 2. 新点在凸包外部，从这个点向原凸包看过去，删除原凸包上可以看到的平面，然后再把边界上的端点和新点相连接就是新的凸包了。
- 重复第一二步

判断新点 p 原凸包外面还是里面的方法，规定面的法线方向朝向凸包外部，若存在一点和某一平面所组成的四面体有向体积为正，则 p 点在凸包外部。



例题：输入一个整数 N ，接下来 N 行每行输入三个整数代表某个外星人的位置。
输出凸多面体表面积的最小值，保留三位小数。

```

#include<bits/stdc++.h>
using namespace std;
const int eps=1e-8;
const int N=510;
struct Point{
    double x,y,z;
    Point() {};
    Point(double _x, double _y, double _z):x(_x), y(_y), z(_z) {}
    Point operator -(const Point p){return Point(x-p.x, y-p.y, z-p.z);}
    Point operator *(const Point p){return Point(y*p.z-z*p.y, z*p.x-x*p.z, x*p.y-y*p.x);}//叉积
    double operator ^(const Point p){return x*p.x+y*p.y+z*p.z;}//点积
};

struct fac{
    int a,b,c;
    bool ok;
};

struct dhull{
    int n;//初始点个数
    Point ply[N];//初始点
    int trianglecnt;//凸包上的三角形个数
    fac tri[6*N];//凸包上的三角形面
    int vis[N][N];//点 i。 j 属于哪个面
    double dist(Point a){return sqrt(a.x*a.x+a.y*a.y+a.z*a.z);}//两点
    length
    double area(Point a,Point b,Point c){return dist((b-a)*(c-a));}
    double volume(Point a,Point b,Point c,Point d){return
    (b-a)*(c-a)^*(d-a);}//四面体有向体积乘以 6;
    double ptoplane(Point &p,fac &f)//正， 点在面同向
    {
        Point m=ply[f.b]-ply[f.a],n=ply[f.c]-ply[f.a],t=p-ply[f.a];
        return (m*n)^t;
    }
    void deal(int p,int a,int b)
    {
        int f=vis[a][b];
        fac add;
        if(tri[f].ok)
        {
            if((ptoplane(ply[p],tri[f]))>eps) dfs(p,f);
            else

```

```

    {
        add. a=b, add. b=a, add. c=p, add. ok=1;
        vis[p][b]=vis[a][p]=vis[b][a]=trianglecnt;
        tri[trianglecnt++]=add;
    }
}

void dfs(int p, int cnt)//维护凸包，如果点 p 在凸包外更新凸包
{
    tri[cnt].ok=0;
    deal(p, tri[cnt].b, tri[cnt].a);
    deal(p, tri[cnt].c, tri[cnt].b);
    deal(p, tri[cnt].a, tri[cnt].c);
}

void construct() //构建凸包
{
    int i, j;
    trianglecnt=0;
    if(n<4) return ;
    bool tmp=true;
    for(i=1;i<n;i++)//前两点不共线
    {
        if( (dist(ply[0]-ply[i]))>eps )
        {
            swap(ply[1], ply[i]);tmp=false;break;
        }
    }

    if(tmp) return;
    tmp=true;

    for(i=2;i<n;i++)//前三点不共线
    {
        if(dist((ply[0]-ply[1])*(ply[1]-ply[i]))>eps)
        {
            swap(ply[2], ply[i]);tmp=false;break;
        }
    }

    if(tmp) return;
    tmp=true;
}

```

```

for(i=3;i<n;i++)//前四点不共面
{
    if(fabs((ply[0]-ply[1])*(ply[1]-ply[2])^(ply[0]-ply[i]))>eps)
    {
        swap(ply[3],ply[i]);tmp=false;break;
    }
}

if(tmp) return;
fac add;

for(int i=0;i<4;i++)
{
    add.a=(i+1)%4,add.b=(i+2)%4,add.c=(i+3)%4,add.ok=1;
    if(ptoplane(ply[i],add)>0)) swap(add.b,add.c);

vis[add.a][add.b]=vis[add.b][add.c]=vis[add.c][add.a]=trianglecnt;

tri[trianglecnt++]=add;
}

for(i=4;i<n;i++)//构建更新凸包
{
    for(j=0;j<trianglecnt;j++)
    {
        if(tri[j].ok&&ptoplane(ply[i],tri[j])>eps)
        {
            dfs(i,j);break;
        }
    }
}

int cnt=trianglecnt;
trianglecnt=0;
for(i=0;i<cnt;i++)
{
    if(tri[i].ok)
    {
        tri[trianglecnt++]=tri[i];
    }
}
}

```

```
double area()//表面积
{
    double ret=0;
    for(int i=0;i<trianglecnt;i++)
    {
        ret+=area(ply[tri[i].a],ply[tri[i].b],ply[tri[i].c]);
    }
    return ret/2.0;
}

}hull;

int main()
{
    while(scanf("%d",&hull.n)!=EOF)
    {
        int i;
        for(i=0;i<hull.n;i++)
        {

scanf("%lf%lf%lf",&hull.ply[i].x,&hull.ply[i].y,&hull.ply[i].z);
        }

        hull.construct();
        printf("%.3f\n",hull.area());
    }

    return 0;
}
```