

ACM-ICPC 培训资料汇编

(7)

计算几何分册

(版本号 1.0.0)

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

序

2012 年 5 月，哈尔滨理工大学承办了 ACM-ICPC 黑龙江省第七届大学生程序设计竞赛。做为本次竞赛的主要组织者，我还是很在意本校学生是否能在此次竞赛中取得较好成绩，毕竟这也是学校的脸面。因此，当 2011 年 10 月确定学校承办本届竞赛后，我就给齐达拉图同学很大压力，希望他能认真训练参赛学生，严格要求参训队员。当然，齐达拉图同学半年多的工作还是很有成效，不仅带着黄李龙、姜喜鹏、程宪庆、卢俊达等队员开发了我校的 OJ 主站和竞赛现场版 OJ，还集体带出了几个比较像样的新队员，使得今年省赛我校取得了很好的成绩（当然，也承蒙哈工大和哈工程关照，没有派出全部大牛来参赛）。

在 2011 年 9 月之前，我对 ACM-ICPC 关心甚少。但是，我注意到我校队员学习、训练没有统一的资料，也没有按照竞赛所需知识体系全面系统培训新队员。2011-2012 年度的学生教练们做了一个较详细的培训计划，每周都会给 2011 级新队员上课，也会对老队员进行训练，辛辛苦苦忙活了一年——但是这些知识是根据他们个人所掌握情况来给新生讲解的，新生也是杂七杂八看些资料和做题。在培训的规范性上欠缺很多，当然这个责任不在学生教练。2011 年 9 月，我曾给老队员提出编写培训资料这个任务，一是老队员人数少，有的还要去百度等企业实习；二是老队员要开发、改造 OJ；三是培训新队员也很耗费精力，因此这项工作虽很重要，但却不是那时最迫切的事情，只好被搁置下来。

2012 年 8 月底，2012 级新生满怀梦想和憧憬来到学校，部分同学也被 ACM-ICPC 深深吸引。面对这个新群体的培训，如何提高效率和质量这个老问题又浮现出来。市面现在已经有了各种各样的 ACM-ICPC 培训教材，主要算法和解题思路都有了广泛深入的分析和讨论。同时，互联网博客、BBS 等中也隐藏着诸多大牛对某些算法的精彩论述和参赛感悟。我想，做一个资料汇编，采撷各家言论之精要，对新生学习应该会有较大帮助，至少一可以减少他们上网盲目搜索的时间，二可以给他们构造一个相对完整的知识体系。

感谢 ACM-ICPC 先辈们作出的杰出工作和贡献，使得我们这些后继者们可以站在巨人的肩膀上前行。

感谢校集训队各位队员的无私、真诚和抱负的崇高使命感、责任感，能够任劳任怨、以苦为乐的做好这件我校的开创性工作。

唐远新
2012 年 10 月

编写说明

本资料为哈尔滨理工大学 ACM-ICPC 集训队自编自用的内部资料，不作为商业销售目的，也不用于商业培训，因此请各参与学习的同学不要外传。

本分册大纲由黄李龙编写，内容由彭文文、曹振海、杨和禹等分别编写和校核。

本分册内容大部分采编自各 OJ、互联网和部分书籍。在此，对所有引用文献和试题的原作者表示诚挚的谢意！

由于时间仓促，本资料难免存在表述不当和错误之处，格式也不是很规范，请各位同学对发现的错误或不当之处向acm@hrbust.edu.cn邮箱反馈，以便尽快完善本文档。在此对各位同学的积极参与表示感谢！

哈尔滨理工大学在线评测系统（Hrbust-OJ）网址：<http://acm.hrbust.edu.cn>，欢迎各位同学积极参与AC。

国内部分知名 OJ：

杭州电子科技大学：<http://acm.hdu.edu.cn>

北京大学：<http://poj.org>

浙江大学：<http://acm.zju.edu.cn>

以下百度空间列出了比较全的国内外知名 OJ：

http://hi.baidu.com/leo_xxx/item/6719a5ffe25755713c198b50

哈尔滨理工大学 ACM-ICPC 集训队
2012 年 12 月

目 录

序.....	I
编写说明	II
第 8 章 计算几何	4
8.1 基本几何运算	4
8.1.1 基本原理	4
8.1.2 解题思路	25
8.1.3 模板代码	25
8.1.4 扩展变型	26
8.2 凸包	28
8.2.1 基本原理	28
8.2.2 求凸包的几种算法	28
8.2.3 解题思路	34
8.2.4 模板代码	34
8.2.5 经典题目	38
8.3 半平面交	40
8.3.1 基本原理	40
8.3.2 求半平面交的方法	40
8.3.3 解题思路	44
8.3.4 模板代码	45
8.3.5 经典题目	48
8.3.6 扩展变形	50
8.4 最近点对	50
8.4.1 基本原理	51
8.4.2 解题思路	52
8.4.3 模板代码	52
8.4.4 经典题目	54
8.5 最远点对	56
8.5.1 基本原理	56
8.5.2 解题思路	56
8.5.3 模板代码	56
8.5.4 经典题目	57
8.6 模拟退火(Simulated Annealing)	59
8.6.1 基本原理	59
8.6.2 解题思路	61
8.6.3 模板代码	61
8.6.4 经典题目	61
8.6.5 扩展变型	63

第8章 计算几何

8.1 基本几何运算

参考文献:

Vagaa 总结(齐达拉图, 哈尔滨理工大学)

《算法艺术与信息学竞赛》(刘汝佳、黄亮, 清华大学出版社)

《计算机图形学基础教程》(孙家广, 清华大学出版社) p48~49

《ACMICPC 程序设计系列—计算几何》(哈尔滨工业大学出版社)

<http://www.cnblogs.com/lxglbk/archive/2012/08/17/2644805.html>

<http://hi.baidu.com/aekdycoin/item/2d54f9c0fef55457ad00efd6>

编写: 彭文文

校核: 曹振海

8.1.1 基本原理

8.1.1.1 计算几何中的向量表示以及运算

1. 点的表示:

以二维平面为例, 点的表示如下, 包括横坐标 x 和纵坐标 y

```
struct point{
    double x,y;
}
```

向量的长度为

```
double distance(point p1,point p2){
    return (p2.x-p1.x)*(p2.x-p1.x)+(p2.y-p1.y)*(p2.y-p1.y) ;
}
```

三维的请读者自己思考。

2. 向量的概念:

既有大小又有方向的矢量叫做向量, 即有起点 A 和终点 B 的线段

```
struct line{
    point start;
    point end;
}
```

向量的大小叫做向量的模, 表示为 $|AB|$ 。

3. 向量的加减法:

两向量的 a 和 b 的和为一个向量, 记作 c , 即 $c=a+b$;

表示方法为 $P=(x1,y1), Q=(x2,y2)$, 则 $P+Q=(x1+x2,y1+y2)$

同样向量的减法看做是 $P+(-Q)$ 即可。

4. 向量的点积:

两向量的点积为一个标量 $a \cdot b$, 它的大小为 $a \cdot b = |a| |b| \cos \theta$, 其中 θ 是 $\langle a, b \rangle$

表示方法为 $P=(x1,y1), Q=(x2,y2)$, 其中 P, Q 为向量, 则 $P \cdot Q = x1 \times x2 + y1 \times y2$

```
double dot(point P, double Q){
    return P.x*Q.x+P.y*Q.y;
}
```

5. 向量的叉积:

两向量的叉积为一个矢量 $a \times b$, 设向量 $P(x1,y1), Q(x2,y2)$, 向量 a 和向量 b 的叉积还是

一个向量，长度为 $|PQ| = x_1y_2 - x_2y_1$ ；即为向量 P, Q 围成的四边形的面积。它的方向与向量 PQ 垂直，并且使 $(P, Q, P \times Q)$ 成右手系。

已知两个向量 P, Q 的叉积函数 cross:

```
double cross(point P, point Q) {
    return P.x*Q.y - P.y*Q.x;
}
```

已知三个点 p_0, p_1, p_2 的叉积函数 cross:

```
double cross(point p0, point p1, point p2) {
    return (p1.x - p0.x) * (p2.y - p0.y) - (p2.x - p0.x) * (p1.y - p0.y);
}
```

叉积的一个重要性质，判断两向量互相之间的顺逆时针关系。

若 $P \times Q > 0$ ，则 P 在 Q 的顺时针方向；

若 $P \times Q < 0$ ，则 P 在 Q 的逆时针方向；

若 $P \times Q = 0$ ，则 P 和 Q 共线，但可能同向也可能反向；

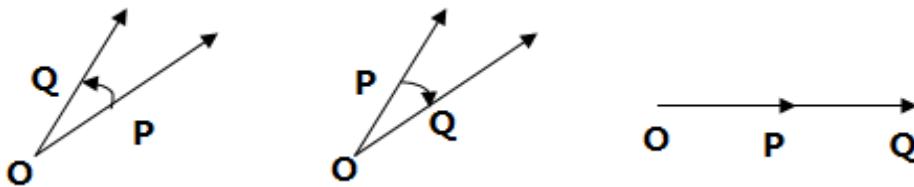


图 1.1.5: P 和 Q 的位置

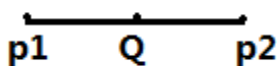
利用这一性质，可以将一个多边形的顶点按照顺时针或者逆时针方向排序，这就是**极角排序**。利用 sort 或者 qsort 函数，自己定义 cmp 函数即可，结合 poj1696, hrbustoj1318, hrbustoj1305, poj2280（经典）练习。

代码参考：

```
//逆时针极角排序比较函数(double)
bool cmp(const point &a, const point &b) {
    double x = atan2(a.y, a.x), y = atan2(b.y, b.x);
    return x < y;
}
//象限极角排序比较函数(int)，不失精度
int cross(point p1, point p2, point p0) {
    return (p1.x - p0.x) * (p2.y - p0.y) - (p2.x - p0.x) * (p1.y - p0.y);
}
sort(p+1, p+n, cmp);
```

8.1.1.2 点定位

1. 判断点是否在线段上



判断点 Q 在线段 P_1P_2 上的两条依据：

① $(Q - P_1) \times (P_2 - P_1) = 0$;

② Q 在以 P_1, P_2 为对角顶点的矩形内。

```
bool onSegment(point p1, point p2, point Q) {
    if( (Q.x - p1.x) * (p2.y - p1.y) == (p2.x - p1.x) * (Q.y - p1.y) &&
        min(p1.x, p2.x) <= Q.x && Q.x <= max(p1.x, p2.x) &&
```

```

min(p1.y, p2.y) <= Q.y && Q.y <= max(p1.y, p2.y) )
    return 1 ;
    else return 0 ;
}
    
```

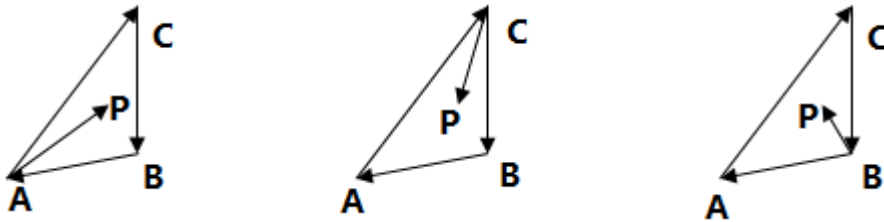
2. 判断点是否在三角形内

点 P 在三角形 ABC 内部常用的又两种方法，面积法和叉积法，面积法即为

$$S_{\triangle PAB} + S_{\triangle PAC} + S_{\triangle PBC} = S_{\triangle ABC}$$

其中三角形面积计算公式为 $S = 1/2 \times |\text{cross}(a, b)|$

叉积法

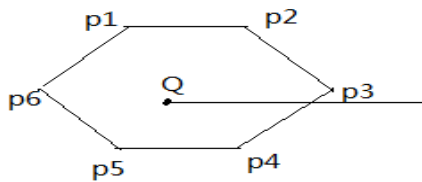


利用叉积的正负号判断，如图所示，AP 在向量 AC 的顺时针方向，CP 在向量 BC 的顺时针方向，BP 在向量 BA 的顺时针方向，利用这一性质推广，那么可以利用叉积的正负号来判断一个点是否在一个凸多边形内部。结合练习题 hrbustoj1142, 1291 练习。

3. 判断点是否在多边形内

若是凸多边形，那么可以利用上述方法（叉积判别法）直接判断。下面重点讨论满足凹多边形的情况。常用方法有四个方法，其中前三个的时间复杂度为 $O(N)$ ，第四个方法的时间复杂度为 $O(\log N)$ 。

方法一 射线法：



如图，设点 Q 及多边形 P1P2P3P4P5P6，判断点是否是多边形内，首先以点 Q 为端点，向任意方向做射线，由于多边形是有界的，所以射线一定会延伸到多边形外。若射线与多边形没有交点，则点在多边形外；若有一个交点，则点在多边形内；若有两个交点，则点在多边形外。依次类推，在通常情况下，当射线与多边形的交点数目是奇数时，Q 在多边形内部，是偶数时，在多边形外部。

但是，某些特殊情况要单独考虑，如图所示，图 a 射线与多边形的顶点相交，这时交点只能算一个；图 b 射线和多边形顶点的交点不应被计算；图 c 射线和多边形的边重合，这条边应该被忽略。

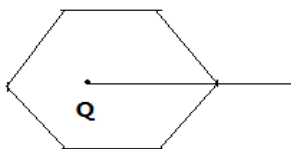


图 a

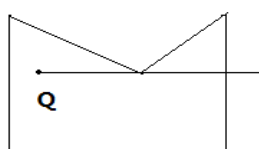


图 b

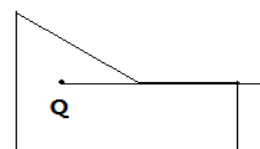


图 c

多边形的一条边重合，这条边应该被忽略。

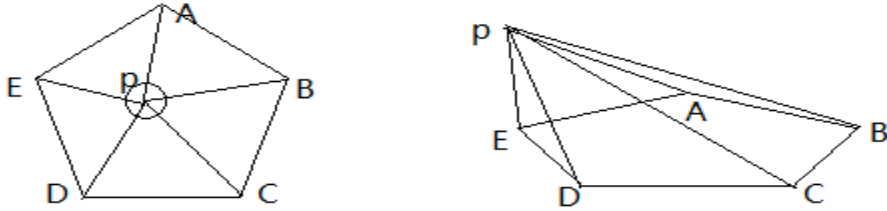
为了统一，射线可设定为水平向右，设点 Q' 的纵坐标与 Q 相同，Q' 的横坐标为一大整数，则可用 QQ' 代替射线。算法描述：

- ①对于多边形的水平边不考虑；
- ②对于多边形的顶点和射线相交的情况，如果该顶点是其所属的边上纵坐标较大的顶点，则计数，否则，忽略该点。
- ③对于 Q 在多边形边上的情况，直接可判断 Q 属于多边形。

该算法的时间复杂度为 $O(n)$ ，具体代码实现在模板代码中会给出

方法二 角度和的判断法：

对于多面多边形来说，连接多边形内点与多边形所有顶点所形成的所有角的角度和在要求精度范围内应该等于 360° ，如果大于或小于 360° ，则该点不在多边形中，如图所示：



部分代码如下：

```
double angle=0 ;
realPointList ::iterator iter1 = points.begin() ;
for(realPointList ::iterator iter2=(iter1 + 1) ;iter2 <
point.end() ;++iter1,++iter2) {
    double x1=(*iter1).x-p.x ;
    double y1=(*iter1).y-p.y ;
    double x2=(*iter2).x-p.x ;
    double y2=(*iter2).y-p.y ;
    angle += angle2D(x1,y1,x2,y2) ;
}
if(fabs(angle-span ::PI2)<0.01) return 1 ;
else return 0;
```

特别判断点在矩形和圆形的时候更简洁，这里不介绍了。

方法三 改进的弧长法（简单）

最大的优点是具有很高的精度，只需做乘法和减法，若针对整数坐标则完全没有精度问题。实现起来很简单，比转角法和射线法都要好写。

弧长法要求多边形是有向多边形，一般规定沿多边形的正向，边的左侧为多边形的内侧域。以被测点为圆心作单位圆，将全部有向边向单位圆作径向投影，并计算其中单位圆上弧长的代数和。若代数和为 0 ，则点在多边形外部；若代数和为 2π 则点在多边形内部；若代数和为 π ，则点在多边形上。

将坐标原

点平移到被测点 P ，这个新坐标系将平面划分为 4 个象限，对每个多边形顶点 P ，只考虑其所在的象限，然后按邻接顺序访问多边形的各个顶点 P ，分析 P 和 $P[i+1]$ ，有下列三种情况：

- (1) $P[i+1]$ 在 P 的下一象限。此时弧长和加 $\pi/2$ ；
- (2) $P[i+1]$ 在 P 的上一象限。此时弧长和减 $\pi/2$ ；
- (3) $P[i+1]$ 在 P_i 的相对象限。首先计算 $f=y[i+1]*x-x[i+1]*y$ （叉积），若 $f=0$ ，则点在多边形上；若 $f<0$ ，弧长和减 π ；若 $f>0$ ，弧长和加 π 。

最后对算出的代数和和上述的情况一样判断即可。

其实说的形象一点，像是把多边形看成一个不规则的气球，然后气球里面有个小球，气球泄气之后，就会完全包裹在这个小球上。看成二维的来看，就是这个气球的横切面，那么就是包裹在一个单位圆上，那么单位圆的周长是 2π 。

实现的时候还有两点要注意，第一个是若 P 的某个坐标为 0 时，一律当正号处理；第二点是若被测点和多边形的顶点重合时要特殊处理。

其实还存在一个问题。那就是当多边形的某条边在坐标轴上而且两个顶点分别在原点的两侧时会出错。如边 $(3, 0) - (-3, 0)$ ，按以上的处理，象限分别是第一和第二，这样会使代数和加 $\pi/2$ ，有可能导致最后结果是被测点在多边形外。

而实际上被测点是在多边形上（该边穿过该点）。

对于这点，我的处理办法是：每次算 P 和 $P[i+1]$ 时，就计算叉积和点积，判断该点是否在该边上，是则判断结束，否则继续上述过程。这样牺牲了时间，但保证了正确性。

具体实现的时候，由于只需知道当前点和上一点的象限位置，所以附加空间只需 $O(1)$ 。实现的时候可以把上述的“ $\pi/2$ ”改成 1，“ π ”改成 2，这样便可以完全使用整数进行计算。不必考虑顶点的顺序，逆时针和顺时针都可以处理，只是最后的代数和符号不同而已。整个算法编写起来非常容易。

可以结合 [z oj1081](#)，[hrbustoj1306](#) 结合练习。代码可参考：

```
struct node{
    double x,y;
}p[105];
int inpolygon(node t,int n) {
    int i,t1,t2,sum,f;
    for(i=0;i<=n;i++){
        p[i].x=t.x;
        p[i].y=t.y;
    }
    t1=p[0].x>=0?(p[0].y>=0?0:3):(p[0].y>=0?1:2);
    for(sum=0,i=1;i<=n;i++){
        if(!p[i].x&&!p[i].y)break;
        f=p[i].y*p[i-1].x-p[i].x*p[i-1].y;
        if(!f&&p[i-1].x*p[i].x<=0&&p[i-1].y*p[i].y<=0) break;
        t2=p[i].x>=0?(p[i].y>=0?0:3):(p[i].y>=0?1:2);
        if(t2==(t1+1)%4) sum+=1;
        else if(t2==(t1+3)%4) sum-=1;
        else if(t2==(t1+2)%4) {
            if(f>0) sum+=2;
            else sum-=2;
        }
        t1=t2;
    }
    if(i<=n||sum) return 1;
    return 0;
}
int main() {
    int n,i;
    node t;
    while(scanf("%d",&n)!=EOF) {
        if(n==0) break;
        scanf("%lf%lf",&t.x,&t.y);
        for(i=0;i<n;i++){
            scanf("%lf%lf",&p[i].x,&p[i].y);
```

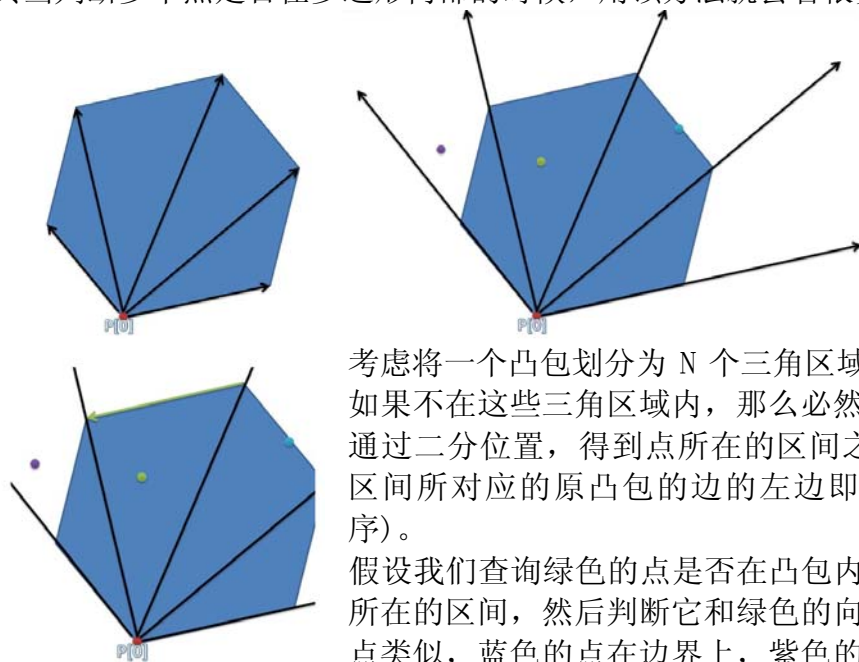
```

    }
    p[n]=p[0];
    if(inpolygon(t,n)) printf("Yes\n");
    else printf("No\n");
}
}

```

方法四 二分 $O(\log n)$:

尤其当判断多个点是否在多边形内部的时候, 用该方法就会省很多时间。



考虑将一个凸包划分为 N 个三角区域, 于是可知对于某个点, 如果不在这些三角区域内, 那么必然不在凸包内。否则, 可以通过二分位置, 得到点所在的区间之后只需要判断点 是否在区间所对应的原凸包的边的左边即可(逆时针给出凸包点顺序)。

假设我们查询绿色的点是否在凸包内, 我们首先二分得到了它所在的区间, 然后判断它和绿色的向量的关系, 蓝色和紫色的点类似, 蓝色的点在边界上, 紫色的点在边界右边。因此一个

查询在 $O(\log N)$ 内解决。

参考 hrbust1429 凸多边形, 代码参考:

```

struct node{
    long long x,y;
}a[100005],b[100005];
long long mul(node p1,node p2,node p3){
    return (p2.x-p1.x)*(p3.y-p1.y)-(p3.x-p1.x)*(p2.y-p1.y);
}
int main(){
    int n,m,i,low,high,mid,flag;
    while(scanf("%d",&n)!=EOF){
        for(i=0;i<n;i++)
            scanf("%lld%lld",&a[i].x,&a[i].y);
        scanf("%d",&m);
        for(i=0;i<m;i++)
            scanf("%lld%lld",&b[i].x,&b[i].y);
        flag=0;
        for(i=0;i<m;i++){
            if(mul(a[0],a[1],b[i])>0||mul(a[0],a[n-1],b[i])<=0){
                flag=1;
                goto loop;
            }
        }
    }
}

```

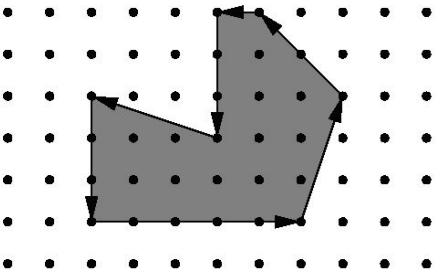
```

    }
    low=2; high=n-1;
    while(low<high){
        mid=(low+high)>>1;
        if(mul(a[0],a[mid],b[i])>0)
            high=mid;
        else low=mid+1;
    }
    if(mul(a[low],a[low-1],b[i])<=0){
        flag=1;
        goto loop;
    }
}
loop: if(flag)
    printf("NO\n");
    else printf("YES\n");
}
return 0;
}
    
```

8.1.1.3 网格和 pick 定理

1. 网格，pick 定理：

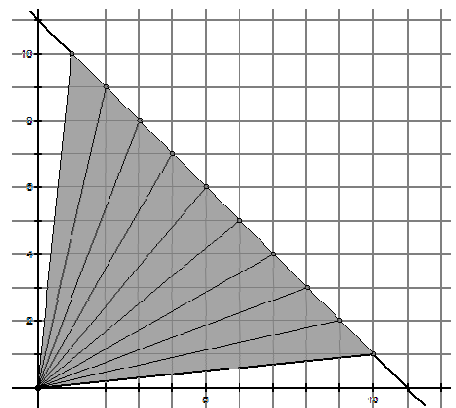
计算类似这样一个图形的面积、边上的格点数、内部格点数，解法：
 这里用到一个定理，叫 pick 定理
 $\text{面积} = \text{边上点数} / 2 - 1 + \text{内部点数}$
 然后求边上的点数直接用 $\text{gcd}(dx, dy)$ 就可以了。
 网格图是一个神奇的图，里面有很多诡异的结论。



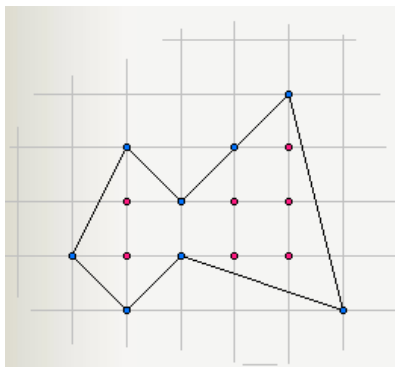
pick 定理

Pick 定理的几个出人意料的应用 (摘自 matrix67)

考虑直线 $x+y=n$ ，其中 n 是一个素数。这条直线将恰好通过第一象限里的 $n-1$ 个格点 (如上图，图中所示的是 $n=11$ 的情况)。将这 $n-1$ 个点分别和原点相连，于是得到了 $n-2$ 个灰色的三角形。仔细数数每个三角形内部的格点数，你会发现一个惊人的事实：每个三角形内部所含的格点数都是一样多。这是为什么呢？



Pick 定理是
 边形的顶点
 等于边界上
 减一。例



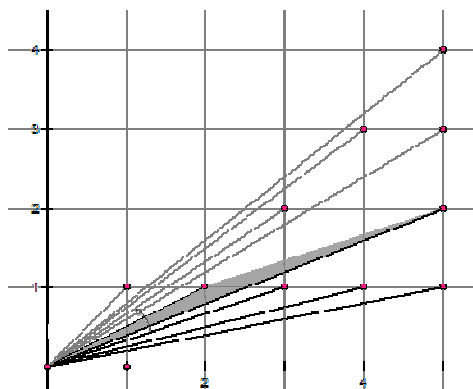
说，在一个平面直角坐标系内，如果一个多
 全都在格点上，那么这个图形的面积恰好就
 经过的格点数的一半加上内部所含格点数再
 如，上图多边形的边界上有 8 个格点，内部

含有 7 个格点，那么其面积就等于 $8/2+7-1=10$ 。我们曾经在这里看到过一个非常神奇非常诡异的证明。这个定理有一些非常巧妙的应用。在上面的问题里，所有三角形都是等底等高的，因此它们的面积都相等。另外，注意到 x 与 y 的和是一个素数，这表明 x 和 y 是互素的（否则 $x+y$ 可以提出一个公因数 d ，与和为素数矛盾），也就是说 (x, y) 和原点的连线不会经过其它格点。既然所有三角形的面积都相等，边界上的格点数也相等，由 Pick 定理，我们就能直接得出每个三角形内部的格点数也相等了。

另一个有趣的问题则是，一个 $n*n$ 的正方形最多可以覆盖多少个格点？把这个正方形中规中矩地放在直角坐标系上，显然能够覆盖 $(n+1)^2$ 个格点。貌似这已经是最多的了，不过如何证明呢？利用 Pick 定理，我们能够很快说明它的最优性。注意到由于任两个格点间最近也有一个单位的间距，再考虑到正方形的周长为 $4n$ ，因此该正方形的边界上最多有 $4n$ 个格点。把正方形边界上的格点数记作 B ，内部所含格点数记为 I ，于是它所能覆盖的总格点数等于 $I+B$ ，由于 $I+B = I+B/2-1 + B/2+1 \leq n^2 + 4n/2 + 1 = (n+1)^2$ ，结论立即得证。

一个东西最出神入化的运用还是见于那些与它八杆子打不着的地方。Farey 序列是指把在 0 到 1 之间的所有分母不超过 n 的分数从小到大排列起来所形成的数列，我们把它记作 F_n 。例如， F_5 就是

0/1, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 1/1



Farey 序列有一个神奇的性质：前一项的分母乘以一项的分子，一定比前一项的分子与后一项分母之积大 1。用 Pick 定理来证明这个结论异常简单。把分母不超过 n 的每一个 0 和 1 之间的分数都标在平面直角坐标系上，例如 0/1 就对应点 $(1, 0)$ ，1/5 就对应点 $(5, 1)$ 。考虑一根从原点出发的射线由 x 轴正方向逆时针慢慢转动到 y 轴正方向，这根射线依次扫过的标记点恰好就是一个 Farey 序列（因为 Farey 序列相当于是给每个标记点的斜率排序）。考虑这根射线扫过的两个相邻的标记点，它们与原点所组成的三角形面积一定为 $1/2$ ——由于分数都是最简分数，因此它们与原点的连线上没有格点；又因为这是射线扫过的两个相邻的标记点，因此三角形内部没有任何格点。另外注意到，由于三角形面积等于叉积的一半，因此两个点 (m, n) 和 (p, q) 与原点组成的三角形面积应该为 $(mq-np)/2$ 。于是，对于 Farey 序列的两个相邻分数 n/m 和 q/p ，我们有 $(mq-np)/2 = 1/2$ ，即 $mq-np=1$ 。

结合 poj1265, hrbustoj1508，代码可参考：

```
#define eps 1e-10
struct point{int x,y;}p[110];
int abs(int a){return a>=0?a:-a;}
int gcd(int a,int b){
    return b?gcd(b,a%b):a;
}
int grid_onedge(int n,point* p){
    int i,ret=0;
    for (i=0;i<n;i++){
        ret+=gcd(abs(p[i].x-p[(i+1)%n].x),abs(p[i].y-p[(i+1)%n].y));
    }
    return ret;
}
```

```

int grid_inside(int n, point* p) {
    int i, ret=0;
    for (i=0; i<n; i++)
        ret+=p[(i+1)%n].y*(p[i].x-p[(i+2)%n].x);
    return (abs(ret)-grid_onedge(n, p))/2+1;
}

int main() {
    int n, T, a, b;
    scanf("%d", &T);
    while(T--) {
        scanf("%d", &n);
        p[0].x=0; p[0].y=0;
        for(int i=1; i<n; i++) {
            scanf("%d%d", &a, &b);
            p[i].x=p[i-1].x+a;
            p[i].y=p[i-1].y+b;
        }
        scanf("%d%d", &a, &b);
        p[0].x=p[n-1].x+a;
        p[0].y=p[n-1].y+b;
        printf("%d\n", grid_inside(n, p));
    }
}
    
```

2. 面的重心:

这里矩形，三角形和圆形的重心不再介绍，简单介绍一下扇形和半圆的重心，会在 8.2 凸包中讲解多边形和多面体重心的求法。

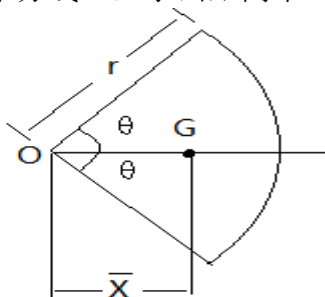
扇形的重心：一定在圆心角的角平分线上，求出距离中心点 O 即可：

$$\bar{X} = \frac{2r \sin \theta}{3 \theta} = \frac{2rb}{3s}$$

式中：

s: 表弧长

b: 表弦长



可结合 FZU1330 扇形的重心练习。

半圆的重心：半圆就是 θ 为 90° 的扇形，所以以上公式化简为

$$OG = (4r) / (3\pi)。同理解 1/4 圆的重心。$$

3. 根据经纬度求球面距离:

经纬坐标为 A(a1, b1), B(a2, b2), a1a2 为经度, b1b2 为纬度。纬度北正南负, 经度东正西负)

$$d = R \cdot \arccos[\cos b1 \cdot \cos b2 \cdot \cos(a1 - a2) + \sin b1 \cdot \sin b2]$$

4. 三角形的重心，外心，内心，垂心，费马点:

◆重心：三角形三条中线的交点叫做三角形重心。

定理：设三角形重心为 O, BC 边中点为 D, 则有 $AO = 2OD$ 。

◆外心：三角形三边的垂直平分线的交点，称为三角形外心。

外心到三顶点距离相等。

过三角形各顶点的圆叫做三角形的外接圆，外接圆的圆心即三角形外心，这个三角形叫做这个圆的内接三角形。

✧内心：三角形内心为三角形三条内角平分线的交点。

与三角形各边都相切的圆叫做三角形的内切圆，内切圆的圆心即是三角形内心，内心到三角形三边距离相等。这个三角形叫做圆的外切三角形。

✧垂心：三角形三边上的三条高线交于一点，称为三角形垂心。

锐角三角形的垂心在三角形内；直角三角形的垂心在直角的顶点；钝角三角形的垂心在三角形外。

✧费马点：在一个三角形中，到 3 个顶点距离之和最小的点。

计算方法：

(1) 若三角形 ABC 的 3 个内角均小于 120° ，那么 3 条距离连线正好平分费马点所在的周角。所以三角形的费马点也称为三角形的等角中心。

(2) 若三角形有一内角不小于 120° ，则此钝角的顶点就是距离和最小的点。

如何计算等角中心呢？

做任意一条边的外接等边三角形，得到另一点，将此点与此边在三角形中对应的点相连。如此再取另一边作同样的连线，相交点即费马点

```
#include <math.h>
struct point{double x,y;};
struct line{point a,b;};
double distance(point p1,point p2){
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
//已知两条直线求出交点
point intersection(line u,line v){
    point ret=u.a;
    double t=((u.a.x-v.a.x)*(v.a.y-v.b.y)-(u.a.y-v.a.y)*(v.a.x-v.b.x))
        /((u.a.x-u.b.x)*(v.a.y-v.b.y)-(u.a.y-u.b.y)*(v.a.x-v.b.x));
    ret.x+=(u.b.x-u.a.x)*t;
    ret.y+=(u.b.y-u.a.y)*t;
    return ret;
}
//外心
point circumcenter(point a,point b,point c){
    line u,v;
    u.a.x=(a.x+b.x)/2;
    u.a.y=(a.y+b.y)/2;
    u.b.x=u.a.x-a.y+b.y;
    u.b.y=u.a.y+a.x-b.x;
    v.a.x=(a.x+c.x)/2;
    v.a.y=(a.y+c.y)/2;
    v.b.x=v.a.x-a.y+c.y;
    v.b.y=v.a.y+a.x-c.x;
    return intersection(u,v);
}
```

//内心

```
point incenter(point a, point b, point c) {
    line u, v;
    double m, n;
    u. a=a;
    m=atan2(b. y-a. y, b. x-a. x);
    n=atan2(c. y-a. y, c. x-a. x);
    u. b. x=u. a. x+cos((m+n)/2);
    u. b. y=u. a. y+sin((m+n)/2);
    v. a=b;
    m=atan2(a. y-b. y, a. x-b. x);
    n=atan2(c. y-b. y, c. x-b. x);
    v. b. x=v. a. x+cos((m+n)/2);
    v. b. y=v. a. y+sin((m+n)/2);
    return intersection(u, v);
}
```

//垂心

```
point perpencenter(point a, point b, point c) {
    line u, v;
    u. a=c;
    u. b. x=u. a. x-a. y+b. y;
    u. b. y=u. a. y+a. x-b. x;
    v. a=b;
    v. b. x=v. a. x-a. y+c. y;
    v. b. y=v. a. y+a. x-c. x;
    return intersection(u, v);
}
```

//重心

//到三角形三顶点距离的平方和最小的点

//三角形内到三边距离之积最大的点

```
point barycenter(point a, point b, point c) {
    line u, v;
    u. a. x=(a. x+b. x)/2;
    u. a. y=(a. y+b. y)/2;
    u. b=c;
    v. a. x=(a. x+c. x)/2;
    v. a. y=(a. y+c. y)/2;
    v. b=b;
    return intersection(u, v);
}
```

//费马点（模拟退火）

//到三角形三顶点距离之和最小的点

/*模拟退火，POJ 2420 A Star not a Tree?实际上就是求平面内多边形的费马点问题，只不过这里要的不是点，而是最短距离。这里使用模拟退火的方法来逼近费马点。所谓费马点就是指满足这样条件的一个点：这个点到多边形所有定点的距离和最短。*/

```

point fermentpoint(point a, point b, point c) {
    point u, v;
    double step=fabs(a.x)+fabs(a.y)+fabs(b.x)+fabs(b.y)+fabs(c.x)+fabs(c.y);
    int i, j, k;
    u.x=(a.x+b.x+c.x)/3;
    u.y=(a.y+b.y+c.y)/3;
    while (step>1e-10)
        for (k=0;k<10;step/=2, k++)
            for (i=-1;i<=1;i++)
                for (j=-1;j<=1;j++) {
                    v.x=u.x+step*i;
                    v.y=u.y+step*j;
                }
    if
        (distance(u, a)+distance(u, b)+distance(u, c)>distance(v, a)+distance(v, b)+distance(v, c))
        u=v;
    return u;
}
    
```

8.1.1.4 解析几何

1. 交点的计算：求两线段，两直线，线段和直线的交点

①判断两线段是否相交：

我们分两步确定两条线段是否相交：

(1)快速排斥试验

设以线段 P_1P_2 为对角线的矩形为 R ， 设以线段 Q_1Q_2 为对角线的矩形为 T ， 如果 R 和 T 不相交， 显然两线段不会相交。

(2)跨立试验

如果两线段相交， 则两线段必然相互跨立对方。 若 P_1P_2 跨立 Q_1Q_2 ， 则矢量 $(P_1 - Q_1)$ 和 $(P_2 - Q_1)$ 位于矢量 $(Q_2 - Q_1)$ 的两侧， 即

$$(P_1 - Q_1) \times (Q_2 - Q_1) * (P_2 - Q_1) \times (Q_2 - Q_1) < 0。$$

上式可改写成

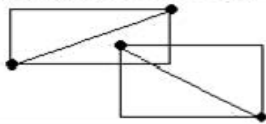
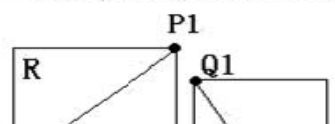
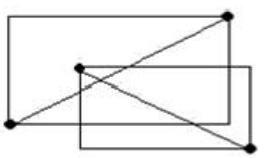

$$(P_1 - Q_1) \times (Q_2 - Q_1) * (Q_2 - Q_1) \times (P_2 - Q_1) > 0。$$

当 $(P_1 - Q_1) \times (Q_2 - Q_1) = 0$ 时， 说明 $(P_1 - Q_1)$ 和 $(Q_2 - Q_1)$ 共线， 但是因为已经通过快速排斥试验， 所以 P_1 一定在线段 Q_1Q_2 上； 同理， $(Q_2 - Q_1) \times (P_2 - Q_1) = 0$ 说明 P_2 一定在线段 Q_1Q_2 上。 所以判断 P_1P_2 跨立 Q_1Q_2 的依据是：

$$(P_1 - Q_1) \times (Q_2 - Q_1) * (Q_2 - Q_1) \times (P_2 - Q_1) \geq 0。$$

同理判断 Q_1Q_2 跨立 P_1P_2 的依据是：

$$(Q_1 - P_1) \times (P_2 - P_1) * (P_2 - P_1) \times (Q_2 - P_1) \geq 0。$$

	通过快速排斥实验	未通过快速排斥实验
未通过立实验		
通过立实验		

另外，“规范相交”指的是两条线段恰有唯一一个不是端点的公共点；而如果一条线段的一个端点在另一条线段上，或者两条线段部分重合，则视为“非规范相交”，以下代码是“非规范相交”。

结合 poj2653 练习，代码可参考：

```
const double eps=1e-10;
struct point{double x,y;};
struct Line{point begin,end;}line[100001];
double min(double a,double b){return a<b?a:b;}
double max(double a,double b){return a>b?a:b;}
bool inter(const Line & M,const Line & N) {
    point a=M.begin;  point b=M.end;
    point c=N.begin;  point d=N.end;
    if( min(a.x,b.x)>max(c.x,d.x) ||
        min(a.y,b.y)>max(c.y,d.y) ||
        min(c.x,d.x)>max(a.x,b.x) ||
        min(c.y,d.y)>max(a.y,b.y)) return 0;
    double h,i,j,k;
    h=(b.x-a.x)*(c.y-a.y)-(b.y-a.y)*(c.x-a.x);
    i=(b.x-a.x)*(d.y-a.y)-(b.y-a.y)*(d.x-a.x);
    j=(d.x-c.x)*(a.y-c.y)-(d.y-c.y)*(a.x-c.x);
    k=(d.x-c.x)*(b.y-c.y)-(d.y-c.y)*(b.x-c.x);
    return h*i<=eps&&j*k<=eps;
}
```

求解出线段交点的任务留给读者（参考《ACMICPC 程序设计系列—计算几何》p21-22）。

②判断两直线是否相交：

首先判断两条直线是否在同一条直线上，或者两条直线平行，否则一定相交，不平行且不在一条直线上则相交并且求出交点。

可结合 hrbustoj1104 练习，代码可参考：

```
#include<stdio.h>
int main() {
    int t;
    double x1,x2,x3,x4,y1,y2,y3,y4,x,y;
    scanf("%d",&t);
    while(t--){
        scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);
```

```

scanf("%lf%lf%lf%lf", &x3, &y3, &x4, &y4);
if((x2-x1)*(y4-y3)==(x4-x3)*(y2-y1)) {
    if((x3-x1)*(y4-y2)==(x4-x2)*(y3-y1) && (y4-y3)!=0)
        puts("LINE");//在一条直线的情况
    else
        puts("NONE");//平行的情况
} else {
    x=((y1*(x2-x1)-x1*(y2-y1))*(x4-x3)-(y3*(x4-x3)-x3*(y4-
y3))*(x2-x1))/((y4-y3)*(x2-x1)-(y2-y1)*(x4-x3));
    y=((y1*(x2-x1)-x1*(y2-y1))*(y4-y3)-(y3*(x4-x3)-x3*(y4-
y3))*(y2-y1))/((y4-y3)*(x2-x1)-(y2-y1)*(x4-x3));
    printf("POINT %.2f %.2f\n", x, y);
}
}
return 0;
}

```

③判断线段和直线的交点:

poj1039 判断直线和线段是否相交并求出交点

```
#define eps 1.0e-8
```

```

struct Point{
    double x, y;
} point[25];
int n;
double ans;
bool ok;
double intersect(Point a1, Point b1, Point a2, Point b2) {
    double x1=a1.x, x2=b1.x, x3=a2.x, x4=b2.x;
    double y1=a1.y, y2=b1.y, y3=a2.y, y4=b2.y;
    double x=(y3-y1+x1*(y2-y1)/(x2-x1)-x3*(y4-y3)/(x4-x3))/((y2-
y1)/(x2-x1)-(y4-y3)/(x4-x3));
    return x;
}
void work(Point a, Point b) {
    b.y-=1;
    for(int i=0; i<n; i++) {
        Point p, q1, q2;
        p.x=point[i].x;
        p.y=a.y-(b.y-a.y)/(b.x-a.x)*(a.x-p.x);
        if((p.y+eps<point[i].y&& p.y-eps>point[i].y-1) ||
            abs(p.y-point[i].y)<eps || abs(p.y-point[i].y+1)<eps)
            continue;
        if(i==0) return;
        if(p.y-eps>point[i].y)
            ans=max(ans, intersect(a, b, point[i-1], point[i]));
        else{

```

```

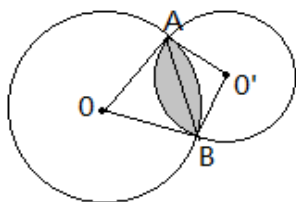
        q1=point[i-1];
        q1.y-=1;
        q2=point[i];
        q2.y-=1;
        ans=max(ans, intersect(a, b, q1, q2));
    }
    return;
}
ok=true;
}
int main() {
    while(scanf("%d", &n), n) {
        for(int i=0; i<n; i++)
            scanf("%lf%lf", &point[i].x, &point[i].y);
        ans=point[0].x;
        ok=false;
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                if(i!=j&&!ok)
                    work(point[i], point[j]);
        if(ok)
            printf("Through all the pipe.\n");
        else
            printf("%.2f\n", ans);
    }
    return 0;
}

```

2. 面积的计算：楔形，多边形等图形

①楔形面积，即如图楔形，即为

楔形面积=扇形 OAB - $\triangle OAB$ + 扇形 $O'AB$ - $\triangle O'AB$



求出 OO' 的长度 d ，又已知 OA 和 $O'A$ 的长度分别为 R 和 r ，则 $\cos \angle AOO' = (R^2 + d^2 - r^2) / (2 * a * b)$ ，则 $\triangle AOB$ 和扇形 AOB 都可以求出了。

代码参考：

```

double fusiform(double a, double c, double b) { // a, b 为半径，c 为  $OO'$  长度
    double angle = acos((a*a + b*b - c*c) / (2*a*b)) * 2;
    double s1 = a*a * PI * (angle / (2*PI)); // 扇形面积
    double s2 = a*a * sin(angle) / 2; // 三角形面积
    return s1 - s2;
}

```

可结合 hdu3264 亚洲区 09 年宁波题目练习（二分枚举+楔形面积）

②多边形面积，三角形面积和扇形面积在楔形面积里面已经用到了，不再介绍。任意多边形面积用到了叉积的几何性质，叉积的大小表示该两条边围成的平行四边形的面积，求三角形则除以 2，所以多边形面积公式为

$$S = 1/2 * \text{abs}(\sum (x_i y_{i+1} - x_{i+1} y_i))$$

代码参考：

```
double area(int num) {
    double area=0;
    for(int i=0;i<num;++i) {
        area+=(polygon[i].x*polygon[(i+1)%num].y)-
            (polygon[(i+1)%num].x*polygon[i].y);
    }
    return area;
}
```

可结合 hdu2036 练习。

3. 圆的问题：圆与圆，圆与直线，圆与三角形、矩形、多边形等

①圆与圆：

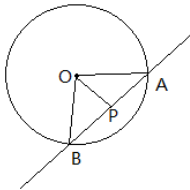
圆与圆的关系有相交，相离，相切三种情况。

两圆心的距离 d 和 $R+r$ 比较大小，判断位置关系，如果 $d \geq R+r$ 相离或相切，重叠面积为 0。否则按照 8.1.1.4 中讲的楔形面积来求解重叠面积。

②圆与直线：

圆与直线的关系有相交，相离，相切三种情况。

利用点到直线的公式求出圆心到直线的距离 d ，与 $R+r$ 的大小进行比较。



✧判直线和圆相交，包括相切；

✧判线段和圆相交，包括端点和相切；

✧判圆和圆相交，包括相切；

✧计算圆上到点 p 最近点，如 p 与圆心重合，返回 p 本身；

✧计算直线与圆的交点，保证直线与圆有交点；

✧求圆外一点 poi 对圆 o 的两个切点 result1 和 result2。

代码可参考如下：

```
#include <math.h>
#define eps 1e-8
struct point{double x,y};
double xmult(point p1,point p2,point p0) {
    return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}
double dist(point p1,point p2) {
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
double disptoline(point p,point l1,point l2) {
    return fabs(xmult(p,l1,l2))/dist(l1,l2);
}
```

```

}
point intersection(point u1,point u2,point v1,point v2){
    point ret=u1;
    double t=((u1.x-v1.x)*(v1.y-v2.y)-(u1.y-v1.y)*(v1.x-v2.x))
            /((u1.x-u2.x)*(v1.y-v2.y)-(u1.y-u2.y)*(v1.x-v2.x));
    ret.x+=(u2.x-u1.x)*t;
    ret.y+=(u2.y-u1.y)*t;
    return ret;
}
//判直线和圆相交,包括相切
int intersect_line_circle(point c,double r,point l1,point l2){
    return disptoline(c,l1,l2)<r+eps;
}
//判线段和圆相交,包括端点和相切
int intersect_seg_circle(point c,double r,point l1,point l2){
    double t1=dist(c,l1)-r,t2=dist(c,l2)-r;
    point t=c;
    if (t1<eps||t2<eps)
        return t1>-eps||t2>-eps;
    t.x+=l1.y-l2.y;
    t.y+=l2.x-l1.x;
    return xmult(l1,c,t)*xmult(l2,c,t)<eps&&disptoline(c,l1,l2)-r<eps;
}
//判圆和圆相交,包括相切
int intersect_circle_circle(point c1,double r1,point c2,double r2){
    return dist(c1,c2)<r1+r2+eps&&dist(c1,c2)>fabs(r1-r2)-eps;
}
//计算圆上到点 p 最近点,如 p 与圆心重合,返回 p 本身
point dot_to_circle(point c,double r,point p){
    point u,v;
    if (dist(p,c)<eps)
        return p;
    u.x=c.x+r*fabs(c.x-p.x)/dist(c,p);
    u.y=c.y+r*fabs(c.y-p.y)/dist(c,p)*((c.x-p.x)*(c.y-p.y)<0?-1:1);
    v.x=c.x-r*fabs(c.x-p.x)/dist(c,p);
    v.y=c.y-r*fabs(c.y-p.y)/dist(c,p)*((c.x-p.x)*(c.y-p.y)<0?-1:1);
    return dist(u,p)<dist(v,p)?u:v;
}
//计算直线与圆的交点,保证直线与圆有交点
//计算线段与圆的交点可用这个函数后判点是否在线段上
void intersection_line_circle(point c,double r,point l1,point l2,point&
p1,point& p2){
    point p=c;
    double t;
    p.x+=l1.y-l2.y;

```

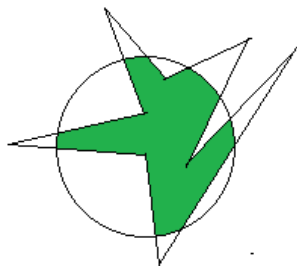
```

    p.y+=l2.x-l1.x;
    p=intersection(p,c,l1,l2);
    t=sqrt(r*r-dist(p,c)*dist(p,c))/dist(l1,l2);
    p1.x=p.x+(l2.x-l1.x)*t;
    p1.y=p.y+(l2.y-l1.y)*t;
    p2.x=p.x-(l2.x-l1.x)*t;
    p2.y=p.y-(l2.y-l1.y)*t;
}
//计算圆与圆的交点,保证圆与圆有交点,圆心不重合
void intersection_circle_circle(point c1,double r1,point c2,double r2,point&
p1,point& p2){
    point u,v;
    double t;
    t=(1+(r1*r1-r2*r2)/dist(c1,c2)/dist(c1,c2))/2;
    u.x=c1.x+(c2.x-c1.x)*t;
    u.y=c1.y+(c2.y-c1.y)*t;
    v.x=u.x+c1.y-c2.y;
    v.y=u.y-c1.x+c2.x;
    intersection_line_circle(c1,r1,u,v,p1,p2);
}
//将向量 p 逆时针旋转 angle 角度
point Rotate(point p,double angle){
    point res;
    res.x=p.x*cos(angle)-p.y*sin(angle);
    res.y=p.x*sin(angle)+p.y*cos(angle);
    return res;
}
//求圆外一点 poi 对圆 o 的两个切点 result1 和 result2
void TangentPoint_PC(point poi,point o,double r,point &result1,point &result2)
{
    double line=sqrt((poi.x-o.x)*(poi.x-o.x)+(poi.y-o.y)*(poi.y-o.y));
    double angle=acos(r/line);
    point unitvector,lin;
    lin.x=poi.x-o.x;
    lin.y=poi.y-o.y;
    unitvector.x=lin.x/sqrt(lin.x*lin.x+lin.y*lin.y)*r;
    unitvector.y=lin.y/sqrt(lin.x*lin.x+lin.y*lin.y)*r;
    result1=Rotate(unitvector,-angle);
    result2=Rotate(unitvector,angle);
    result1.x+=o.x;
    result1.y+=o.y;
    result2.x+=o.x;
    result2.y+=o.y;
    return;
}

```

③圆与三角形、矩形、多边形。

求解圆与矩形或多边形的公共面积，都拆成一个个三角形，利用有向面积来算的。这里重点介绍一个圆和一个任意多边形的公共面积的求法：

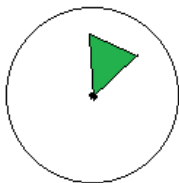


看上去很复杂，怎么办？其实想一想就知道，我们平日求多边形面积用的就是三角形剖分，所以说我们应该化繁为简，通过求出来各三角形与圆的交，从而求出总面积。而且，剖分用的原点正好在圆心，这是很方便的一件事情。

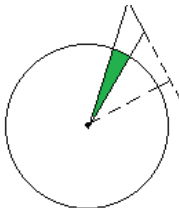
现在问题就转化成一个顶点在圆心的三角形了。但是这才是麻烦的开始，除去退化情况，我们应该至少考虑到以下五种情况：

一、三角形的两条边全部短于半径。

最方便了，直接求三角形面积？

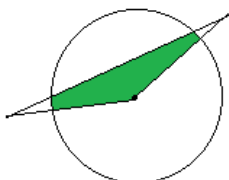


二、三角形的两条边全部长于半径，且另一条边与圆心的距离也长于半径。只需要求出扇形的面积即可。

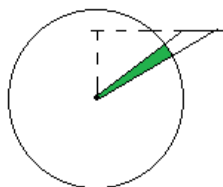


三、三角形的两条边全部长于半径，但另一条边与圆心的距离短于半径，并且垂足落在这条边上。

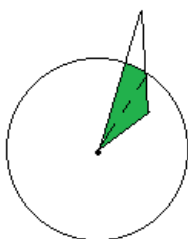
求出扇形的面积，再减去那个弓形的面积。也就是说再挖去一个扇形，补上一个三角形。



四、三角形的两条边全部长于半径，但另一条边与圆心的距离短于半径，且垂足未落在这条边上。



五、三角形的两条边一条长于半径，另外一条短于半径。先求出交点，再剖成扇形和三角形求解。



至于实现问题……可以用解析几何，但是据说精度有很大的问题；用叉积求交点以及用点积求夹角一类计算几何方法可以提高精度。

poj2986 一个圆与一个三角形的公共面积

zoj2675 一个圆与一个矩形的公共面积

poj3675, hdu4404(12 年亚洲区金华赛区网络预选赛)都是多边形与圆的公共面积，比赛的时候直接把 poj 的代码稍加修改就 A 了。

4. 对称和投影

①对称问题：

点关于点的对称：关于原点对称的点，即直接将 (x, y) 变换符号即可。若点 A 关于点 O 的对称点 B，那么 $(X_a + X_b)/2 = X_c$, $(Y_a + Y_b)/2 = Y_c$ 。

代码如下：

```
point symmetric(point p1, point p2) {
    point p;
    p.x = 2*p2.x - p1.x;
    p.y = 2*p2.y - p1.y;
    return p;
}
```

点关于直线的对称：求点关于直线的对称点，相当于求点关于直线上特定一点的对称点，即点关于垂足的对称点。若对称直线是 x 轴或 y 轴，则比较简单。下面我们讨论一下普通情况，直线 L: $ax + by + c = 0$ ($a \neq 0, b \neq 0$)。

点 A 的坐标为 (x_1, y_1) ，求点关于直线 L 的对称点 A'。设 A' (x, y) ，则点 $((x_1 + x)/2, (y_1 + y)/2)$ 在直线 L 上，且过点 A 与点 B 的直线与 L 垂直，即斜率乘积为 -1，可得公式：

$$a \frac{x + x_1}{2} + b \frac{y + y_1}{2} + c = 0$$

$$\frac{a(y - y_1)}{b(x - x_1)} = 1$$

由以上两个公式解的 A' 的坐标为

$$\left(\frac{(b^2 - a^2)x_1 - 2aby_1 - 2ac}{a^2 + b^2}, \frac{(a^2 - b^2)y_1 - 2abx_1 - 2bc}{a^2 + b^2} \right)$$

可结合 hrbustoj1471 练习。

②投影问题：

二维向量的投影问题常见有一个点光源投射到一个图形在地面或者墙上的阴影长度，例如 poj1375 问一个光源，被若干圆遮住，求地面阴影。则将光源抽象为一个点，求一个点和一个圆的两个切点之后再利用相似三角形求出在地面的阴影。求交点的过程再圆的问题里已经讲过。若不是圆形，而是多边形或者三角形等问题的时候，直接求即可。具体按照给的图形求。

可结合 hrbustoj1505, poj1375, poj3304 练习。

(poj3304 求是否存在一条直线，是所有线段到这条直线的投影至少有一个交点 \Leftrightarrow 是否存在一条直线与所有线段都相交。)

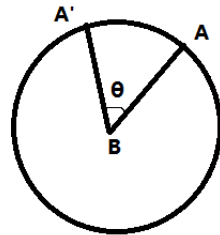
5. 平移和旋转

①向量的平移：

已知向量 $P(x_1, y_1)$ ，向左平移 3 个单位长度，即平移 $(-3, 0)$ ，那么变为 $P'(x_1 - 3, y_1)$ ，同理，向量的平移只需将原向量加上平移向量即可。

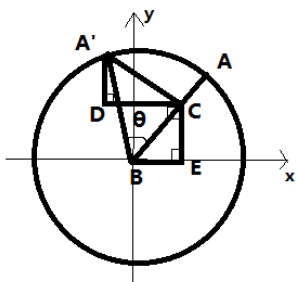
②向量的旋转:

首先我们先把问题简化一下, 我们先研究一个点绕另一个点旋转一定角度的问题。已知 A 点坐标 (x_1, y_1) , B 点坐标 (x_2, y_2) , 我们需要求得 A 点绕着 B 点旋转 θ 度后的位置。



如图: A' 就是 A 点绕 B 点旋转 θ 角度后得的点, 问题是我们要如何才能得到 A' 点的坐标 (向逆时针方向旋转角度正, 反之为负) 研究一个点绕另一个点旋转的问题, 我们可以先简化为一个点绕原点旋转的问题, 这样比较方便我们的研究。之后我们可以将结论推广到一般的形式上。

令 B 是原点, 我们先以 A 点向逆时针旋转为例, 我们过 A' 做 AB 的垂线, 交 AB 于 C, 过 C 做 x 轴的平行线交过 A' 做 x 轴的垂线于 D。过点 C 做 x 轴的垂线交 x 轴于点 E。



令 A 的坐标 (x, y) , A' 坐标 (x_1, y_1) , B 的坐标 $(0, 0)$ 。我们可以轻松的获取 AB 的长度, 而且显而易见 $A'B$ 长度等于 AB。假设我们已知 θ 角的大小那么我们可以很快求出 BC 和 $A'C$ 的长度。 $BC = AB \times \cos \theta$, $A'C = AB \times \sin \theta$ 。

因为 $\angle A'CB$ 和 $\angle DCE$ 为直角 (显然的结论), 则 $\angle A'CD + \angle DCB = \angle ECD + \angle DCB = 90$ 度。

则 $\angle A'CD = \angle ECD$, $\angle A'DC = \angle CEB = 90$ 度, 因此可以推断 $\triangle CA'D \sim \triangle CBE$ 。由此可以退出的结论有:

$$BC/BE = A'C/A'D \text{ 和 } BC/CE = A'C/CD$$

当然了 DC 和 $A'D$ 都是未知量, 需要我们求解, 但是我们却可以通过求出 C 点坐标和 E 点坐标间接获得 $A'C$ 和 CD 的长度。我们应该利用相似的知识求解 C 点坐标。

C 点横坐标等于: $((|AB| \times \cos \theta) / |AB|) \times x = x \times \cos \theta$

C 点纵坐标等于: $((|AB| \times \sin \theta) / |AB|) \times y = y \times \sin \theta$

则 CE 和 BE 的长度都可以确定。

我们可以通过 $\triangle CA'D \sim \triangle CBE$ 得出:

$$AD = x \times \sin \theta \quad DC = y \times \sin \theta$$

那么接下来很容易就可以得出:

$$x_1 = x \times \cos \theta - y \times \sin \theta \quad y_1 = y \times \cos \theta + x \times \sin \theta$$

则 A' 的坐标为 $(x \times \cos \theta - y \times \sin \theta, y \times \cos \theta + x \times \sin \theta)$

我们可以这样认为: 对于任意点 $A(x, y)$, A 非原点, 绕原点旋转 θ 角后点的坐标为: $(x \times \cos \theta - y \times \sin \theta, y \times \cos \theta + x \times \sin \theta)$

接下来我们对这个结论进行一下简单的推广, 对于任意两个不同的点 A 和 B (对于求点绕另一个点旋转后的坐标时, A B 重合显然没有太大意义), 求 A 点绕 B 点旋转 θ 角度后的坐标, 我们都可以将 B 点看做原点, 对 A 和 B 进行平移变换, 计算出的点坐标后, 在其横纵坐标上分别加上原 B 点的横纵坐标, 这个坐标就是 A' 的坐标。

推广结论: 对于任意两个不同点 A 和 B, A 绕 B 旋转 θ 角度后的坐标为:

$$(\Delta x \times \cos \theta - \Delta y \times \sin \theta + x_B, \Delta y \times \cos \theta + \Delta x \times \sin \theta + y_B)$$

注: x_B, y_B 为 B 点坐标。

结论的进一步推广: 对于任意非零向量 AB (零向量研究意义不大), 对于点 C 进行旋转, 我们只需求出点 A 和 B 对于点 C 旋转一定角度的坐标即可求出旋转后的向量 $A'B'$, 因为向量旋转后仍然是一条有向线段。同理, 对于任意二维平面上的多边形旋转也是如此。可结合 poj2194, hrbustoj1472 练习。

6. 最小圆覆盖, 最小球覆盖:

①最小圆覆盖:

给出平面上 n 个点, 求能覆盖这些点的最小的圆 (圆心和半径)。

常用的方法有: 暴力法, 点增量法, 三角形增量法, 模拟退火法等

暴力法 $O(n^3)$: 首先说一下, 三角形的最小覆盖圆, 可能是三角形的外接圆或者是两个顶点为直径、另外一个点在内部的圆。那么在求 n 个点的最小覆盖圆的时候, 先出两点间的最长距离。以这两点的距离为直径画一个圆, 如果包含了所有的点那么 就是所求解。如果不能包含就说明有三点及三点在这个圆上 根据三点确定一个圆 只要枚举出所有的三点 成立的圆 再比较所有成立的圆的半径最小的就是所求的点。在枚举过程中所耗费的时间是 $O(n^3)$ 。

点增量法:

Step1: 在点集中任取 3 个点 A, B, C 。

Step2: 作一个包含 A, B, C 三点的最小圆, 圆周可能通过这三点, 也可能只通过其中两点, 但包含第三个点。后一种情况, 圆周上的两点一定是位圆的一条直径的两端。

Step3: 在点集中找出距离 Step2 所建圆圆心最远的 D 点, 若 D 点已在圆内或圆周上, 则该圆即为所求的圆, 算法结束, 执行 Step4。

Step4: 在 A, B, C, D 中选择三个点, 使有它们生成的一个包含这四个点的圆为最小, 这三个点成为新的 A, B, C , 返回执行 Step2。若在 Step4 生成的圆的圆周只通过 A, B, C, D 中的两点, 则圆周上的两点取成新的 A 和 B , 从另外两点中任取一点作为新的 C 点。

三角形增量法:

Step1: 任意选择两点 P_i 和 P_j , 然后以 P_i 和 P_j 为端点, P_i 和 P_j 的中点为圆心, 构造一个圆, 如果这个圆包括所有的点, 那么它就是最小的圆, 中点也就是最小圆的圆心; 否则选择圆外的一点 P_k 。

Step2: 如果 P_i, P_j, P_k 这三个点形成的三角形是直角或者钝角三角形, 那么该直角或者钝角所对应的两点为 P_i, P_j 。然后再次重新构造一个以新 P_iP_j 为直径的圆, 重复 Step1 的步骤; 否则, 这三个点形成一个锐角三角形, 构造一个外接圆, 如果这个圆包含所有的点, 结束。否则进入 Step3;

Step3: 选择一些不在圆内的点 P_o , 设点 Q 为 $\{P_i, P_j, P_k\}$ 中离 P_o 最远的点, 连接并延长点 P_o 和点 Q , 将平面分成两个半平面, 设点 R 为 $\{P_i, P_j, P_k\}$ 中与 P_o 不在一个半平面中的点, 得到 P_o, Q, R 三点, 返回 Step1。

模拟退火法:

具体见 8.6 模拟退火

最小圆覆盖问题可结合 hdu3007, zoj1450 练习。

①最小球覆盖:

常用方法: 模拟退火 (具体见 8.6 模拟退火)

8.1.2 解题思路

计算几何的基础几何运算因为包含的内容很杂很多, 所以要想完全掌握这里面的所有知识点并不容易, 但是这样的题的题意和抽象一般都不会很难, 模型也比较容易建立, 剩下的很多就是直接贴模板代码了, 所以计算几何的题一定要有高度可靠的模板。

8.1.3 模板代码

因为计算几何基本运算的内容较多, 我们在这里罗列一些较常用, 以及较长的模板代码, 其他的代码在基本原理讲解的时候已经插进去了一部分, 不再重复讲述, 下面的代码排列顺序按照基本原理所给出的章节顺序进行讲解

```
//判断点是否在任意一个多边形内部, 采用射线法判断相交的点数, 具体讲解见基本原理部分
int max(int a,int b){return a>b?a:b;}
int min(int a,int b){return a<b?a:b;}
```

```

struct point{
    int x,y;
}q[100];
int mult(int x,int y,int x1,int y1){
    return x*y1-y*x1;
}
//下面函数判断点是否在线上，保证叉积为 0 并且满足坐标相对位置条件
int online(point a,point b,point c){
    if(mult(a.x-b.x,a.y-b.y,c.x-b.x,c.y-b.y)==0&&
    a.x>=min(b.x,c.x)&&a.x<=max(b.x,c.x)&&a.y>=min(b.y,c.y)&&a.y<=max(b.y,c.y))
    return 1;
    return 0;
}
int main(){
    int i,n,t,count,flag;
    point a,b,c,d;
    while(scanf("%d",&n),n){
        flag=count=0;
        scanf("%d%d",&a.x,&a.y);
        b.x=-100;    b.y=a.y;
        for(i=0;i<n;i++){
            scanf("%d%d",&q[i].x,&q[i].y);
            //开始枚举每一条边和射线是否有交点，或者点是否在线上
            for(i=0;i<n;i++){
                c.x=q[i].x;
                c.y=q[i].y;
                d.x=q[(i+1)%n].x; d.y=q[(i+1)%n].y;
                if(online(a,c,d))
                {flag=1; break;}
                if(c.y!=d.y&&a.y>min(c.y,d.y)&&a.y<=max(c.y,d.y)){
                    if(mult(a.x-d.x,a.y-d.y,c.x-d.x,c.y-d.y)*mult(b.x-d.x,b.y-d.y,c.x-d.x,c.y-d.y)<0)
                        count++;
                }
            }
        }
        if(count&1)flag=1;
        if(flag)printf("Yes\n");
        else printf("No\n");
    }
    return 0;
}
    
```

8.1.4 扩展变型

常用几何公式：

三角形：

1. 半周长 $P=(a+b+c)/2$
2. 面积 $S=aHa/2=absin(C)/2=sqrt(P(P-a)(P-b)(P-c))$
3. 中线 $Ma=sqrt(2(b^2+c^2)-a^2)/2=sqrt(b^2+c^2+2bccos(A))/2$
4. 角平分线 $Ta=sqrt(bc((b+c)^2-a^2))/(b+c)=2bccos(A/2)/(b+c)$
5. 高线 $Ha=bsin(C)=csin(B)=sqrt(b^2-((a^2+b^2-c^2)/(2a))^2)$
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

$$1. a^2 + b^2 + c^2 + d^2 = D1^2 + D2^2 + 4M^2$$

$$2. S = D1D2 \sin(A) / 2$$

(以下对圆的内接四边形)

$$3. ac + bd = D1D2$$

$$4. S = \sqrt{(P-a)(P-b)(P-c)(P-d)}, P \text{ 为半周长}$$

正 n 边形:

R 为外接圆半径, r 为内切圆半径

$$1. \text{中心角 } A = 2\pi/n$$

$$2. \text{内角 } C = (n-2)\pi/n$$

$$3. \text{边长 } a = 2\sqrt{R^2 - r^2} = 2R \sin(A/2) = 2r \tan(A/2)$$

$$4. \text{面积 } S = nar/2 = nr^2 \tan(A/2) = nR^2 \sin(A)/2 = na^2/(4 \tan(A/2))$$

圆:

$$1. \text{弧长 } l = rA$$

$$2. \text{弦长 } a = 2\sqrt{2hr - h^2} = 2r \sin(A/2)$$

$$3. \text{弓形高 } h = r - \sqrt{r^2 - a^2/4} = r(1 - \cos(A/2)) = a \tan(A/4)$$

$$4. \text{扇形面积 } S1 = r^2 A/2$$

$$5. \text{弓形面积 } S2 = (r^2 A - a(r-h))/2 = r^2(A - \sin(A))/2$$

棱柱:

$$1. \text{体积 } V = Ah, A \text{ 为底面积, } h \text{ 为高}$$

$$2. \text{侧面积 } S = lp, l \text{ 为棱长, } p \text{ 为直截面周长}$$

$$3. \text{全面积 } T = S + 2A$$

棱锥:

$$1. \text{体积 } V = Ah/3, A \text{ 为底面积, } h \text{ 为高}$$

(以下对正棱锥)

$$2. \text{侧面积 } S = lp/2, l \text{ 为斜高, } p \text{ 为底面周长}$$

$$3. \text{全面积 } T = S + A$$

棱台:

$$1. \text{体积 } V = (A1 + A2 + \sqrt{A1A2})h/3, A1, A2 \text{ 为上下底面积, } h \text{ 为高}$$

(以下为正棱台)

$$2. \text{侧面积 } S = (p1 + p2)l/2, p1, p2 \text{ 为上下底面周长, } l \text{ 为斜高}$$

$$3. \text{全面积 } T = S + A1 + A2$$

圆柱:

$$1. \text{侧面积 } S = 2\pi r h$$

$$2. \text{全面积 } T = 2\pi r(h + r)$$

$$3. \text{体积 } V = \pi r^2 h$$

圆锥:

$$1. \text{母线 } l = \sqrt{h^2 + r^2}$$

$$2. \text{侧面积 } S = \pi r l$$

$$3. \text{全面积 } T = \pi r(l + r)$$

$$4. \text{体积 } V = \pi r^2 h/3$$

圆台:

$$1. \text{母线 } l = \sqrt{h^2 + (r1 - r2)^2}$$

$$2. \text{侧面积 } S = \pi(r1 + r2)l$$

$$3. \text{全面积 } T = \pi r1 l + \pi r1^2 + \pi r2 l + \pi r2^2$$

4. 体积 $V = \pi(r_1^2 + r_2^2 + r_1 r_2)h/3$

球:

1. 全面积 $T = 4\pi r^2$

2. 体积 $V = 4\pi r^3/3$

球台:

1. 侧面积 $S = 2\pi r h$

2. 全面积 $T = \pi(2rh + r_1^2 + r_2^2)$

3. 体积 $V = \pi h(3(r_1^2 + r_2^2) + h^2)/6$

球扇形:

1. 全面积 $T = \pi r(2h + r_0)$, h 为球冠高, r_0 为球冠底面半径

2. 体积 $V = 2\pi r^2 h/3$

8.2 凸包

参考文献:

《算法导论》(机械工业出版社)

《算法艺术与信息学竞赛》(刘汝佳, 黄亮 清华大学出版社)

编写: 彭文文, 杨和禹

校核: 曹振海

8.2.1 基本原理

凸包的概念是在一个实属向量空间 V 中, 对于给定集合 X , 所有包含 X 的凸集的交集 S 被称为 X 的凸包, 通俗来说, 就是在这个空间内包含这些所有点的最小的凸多边形或凸多面体。

8.2.2 求凸包的几种算法

凸包算法的时间复杂度下界是 $O(n \log(n))$, 但是当凸包的定点数 h 也被考虑, Krikpatrick 和 Seidel 的剪枝搜索算法可以达到 $O(n \log(h))$ 。最常用的算法有 Graham 扫描法和 Jarvis 步进法。还有卷包裹法, 分治法, 增量法。我们在这里介绍 Graham 扫描法和 Jarvis 步进法。

1. Graham 扫描法

Graham 算法是在某种意义上来说求解二维静态凸包的一种最优的算法, 这种算法目前被广泛的应用于对各种以二维静态凸包为基础的 ACM 题目的求解。Graham 算法的时间复杂度大约是 $n \log n$, 因此在求解二维平面上几万个点构成的凸包时, 消耗的时间相对较少。

知识点: 极角排序、栈的使用和叉积的应用

算法描述:

这里描述的 Graham 算法是经过改进后的算法而不是原始算法, 因为改进之后的算法更易于对算法进行编码。

1) 已知有 n 个点的平面点集 $p(p[0] \sim p[n-1])$, 找到二维平面中最下最左的点, 即 y 坐标最小的点。若有多个 y 值最小的点, 取其中 x 值最小的点。

2) 以这个最下最左的点作为基准点(即 $p[0]$), 对二维平面上的点进行极角排序。

3) 将 $p[0]$ 、 $p[1]$ 、 $p[2]$ 三个点压入栈中(栈用 st 表示, top 表示栈顶指针的位置)。并将 $p[0]$ 的值赋给 $p[n]$ 。

4) 循环遍历平面点集 $p[3]$ 到 $p[n]$ 。对于每个 $p[i]$ ($3 \leq i \leq n$) 若存在 $p[i]$ 在向量 $st[top-1]st[top]$ 的顺时针方 (包括共线) 向且栈顶元素不多于 2 个时, 将栈顶元素出栈, 直到 $p[i]$ 在向量 $st[top-1]st[top]$ 的逆时针方向或栈中元素个数小于 3 时将 $p[i]$ 入栈。

5) 循环结束后, 栈 st 中存储的点正好就是凸包的所有顶点, 且这些顶点以逆时针的顺序存储在栈中 ($st[0] \sim st[top-1]$)。

注意: 由于第三步中, 将 $p[0]$ 的值赋给了 $p[n]$, 此时栈顶元素 $st[top]$ 和 $st[0]$ 相同, 因为最后入栈的点是 $p[n]$ 。

由于 Graham 算法是基于极角排序的, 对平面上所有点极角排序的时间复杂度是 $n \log n$, 而之后逐点扫描的过程的时间复杂度是 n , 因此整个 Graham 算法的时间复杂度接近 $n \log n$ 。

实现细节的注意事项:

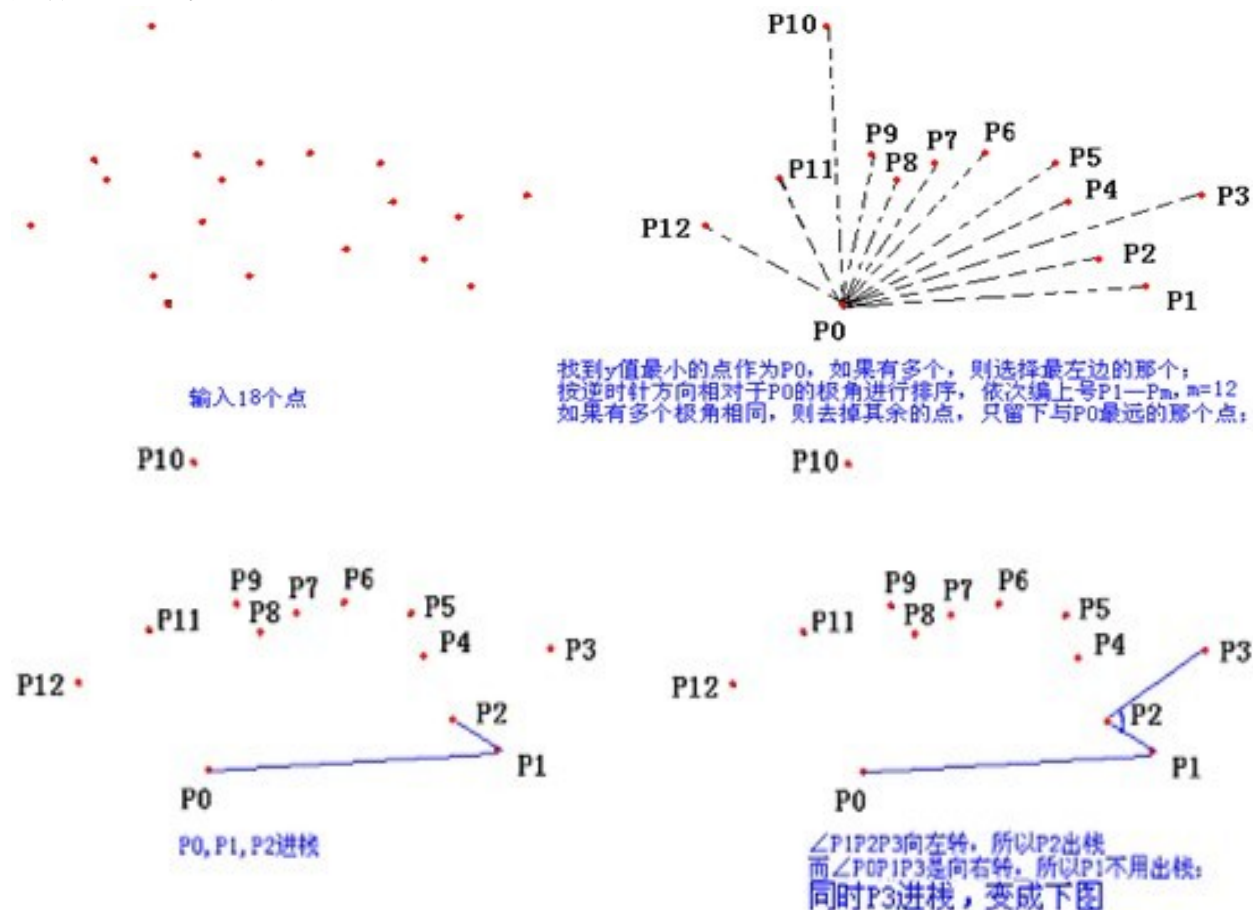
1) 极角大小问题:

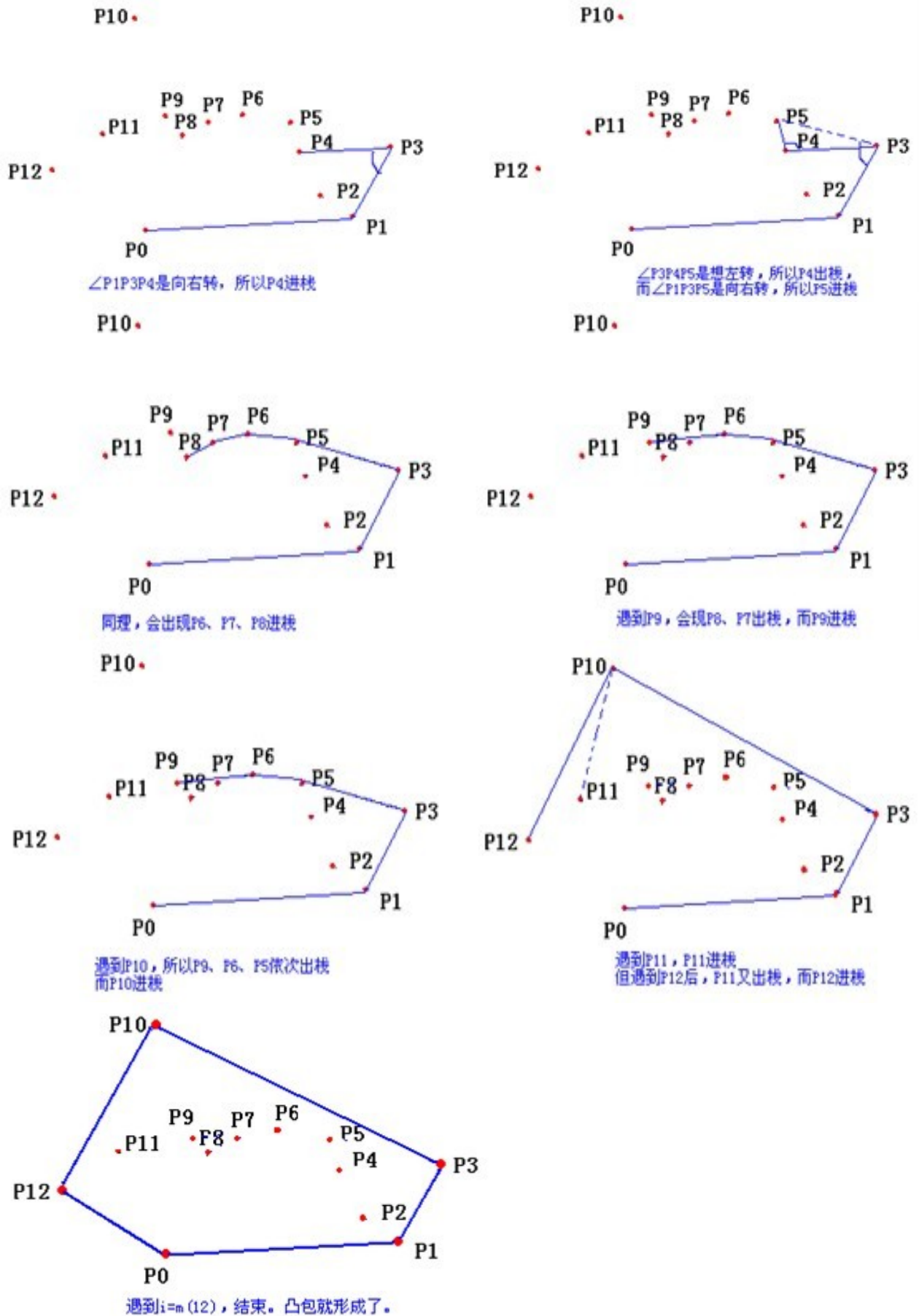
实际实现 Graham 算法的极角排序并不是真正的按照极角大小排序, 因为计算机在表示和计算浮点数时会有一定的误差。一般会利用叉积判断两个点的相对位置来实现极角排序的功能。假设以确定平面中最下最左的点 (基准点) P , 并已知平面上其它两个不同的点 A B 。若点 A 在向量 PB 的逆时针方向, 那么我们认为 A 的极角大于 B 的极角, 反之 A 的极角小于 B 的极角 (具体实现应借助叉积)。

2) 极角相同点的处理:

在 Graham 算法中, 经常会出现两个点极角相同的情况。对于具有相同极角的两个不同点 A B , 那么我们应该把 A B 两点的按照距离基准点距离的降序排列。而对于完全重合的两点, 可以暂不做处理。

具体可以参考以下图形:





补充: Graham 算法是一种从某种意义上来说是计算二维静态凸包的最优算法, 也是 ACMer 们必备的算法之一。但是 Graham 算法也有弊端, 就是无法像增量法那样扩展到三维凸包

的求解。而且如果我们想要了计算维动态凸包的每一个状态，那么 Graham 算法显然是不及增量算法更省时间。

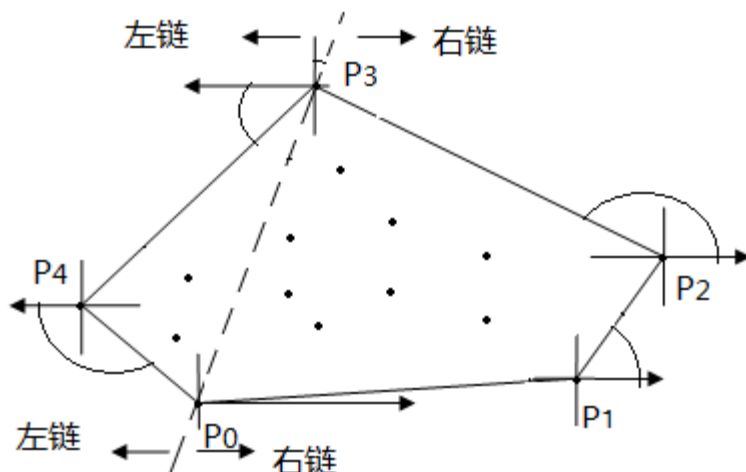
可结合 POJ1113, POJ3348 练习。

2. Jarvis 步进法

Jarvis 步进法运用了一种称为打包的技术来计算一个点集 Q 的凸包。算法的运行时间为 $O(nh)$ ，其中 h 为凸包 $CH(Q)$ 的顶点数。当 h 为 $O(\lg(n))$ ，Jarvis 步进法在渐进意义上比 Graham 算法的速度快一点。

从直观上看，可以把 Jarvis 步进法相像成在集合 Q 的外面紧紧的包了一层纸。开始时，把纸的末端粘在集合中最低的点上，即粘在与 Graham 算法开始时相同的点 p_0 上。该点为凸包的一个顶点。把纸拉向右边使其绷紧，然后再把纸拉高一些，知道碰到一个点。该点也必定为凸包中的一个顶点。使纸保持绷紧状态，用这种方法继续围绕顶点集合，直到回到原始点 p_0 。

更形式的说，Jarvis 步进法构造了 $CH(Q)$ 的顶点序列 $H=(P_0, P_1, \dots, P_{h-1})$ ，其中 P_0 为原始点。如图所示，下一个凸包顶点 P_1 具有相对与 P_0 的最小极角。（如果有数个这样的点，选择最远的那个点作为 P_1 。）类似地， P_2 具有相对于 P_1 的最小的极角，等等。当达到最高顶点，如 P_k （如果有数个这样的点，选择最远的那个点）时，我们构造好了 $CH(Q)$ 的右链了，为了构造其左链，从 P_k 开始选取相对于 P_k 具有最小极角的点作为 P_{k+1} ，这时的 x 轴是原 x 轴的反方向，如此继续，根据负 x 轴的极角逐渐形成左链，知道回到原始点 P_0 。



可以用围绕凸包的一次概念性扫出来实现 Jarvis 步进法，即无需分别构造左链和右链。在这样一种典型的实现方法中，要随时记录上一次选取的凸包的边的角度，并要求凸包边的角度序列严格递增（在 0 到 2π 弧度范围内）。分别构造左右链的优点是无需显式地计算角度。

可结合 POJ1113, POJ3348 练习。

3. 旋转卡壳

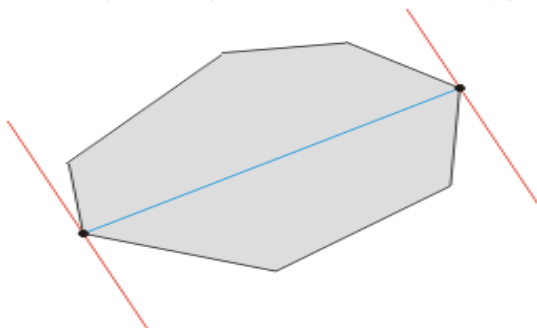
旋转卡壳可以用于求凸包的直径、宽度，两个不相交凸包间的最大距离和最小距离等。虽然算法的思想不难理解，但是实现起来真的很容易让人“卡壳”。

首先了解与旋转卡壳有关的概念：

1. 支撑线：如果一条直线 L 通过凸多边形 P 的一个顶点，且多边形在这条直线的一侧，则称 L 是 P 的支撑线。
2. 对踵点：如果过凸包的两个点可以画一对平行直线，是凸包上的所有点来夹在两条平行线之间或落在平行线上，那么这两个点称为一对对踵点。两条平行的支撑线所过的两点就是一对对踵点。可以证明，一个凸 n 边形的对踵点对最多只有 $3n/2$ （对上取

整) 对。

3. 凸多边形的直径：一个多边形两点间的距离的最大值为多边形的直径。如图所示：



旋转卡壳步骤：

Step1: 计算多边形 y 方向上的端点，称之为 $ymin$ 和 $ymax$ 。

Step2: 通过 $ymin$ 和 $ymax$ 构造两条水平切线，由于它们已经是一对对踵点，计算它们之间的距离并维护一个当前最大值。

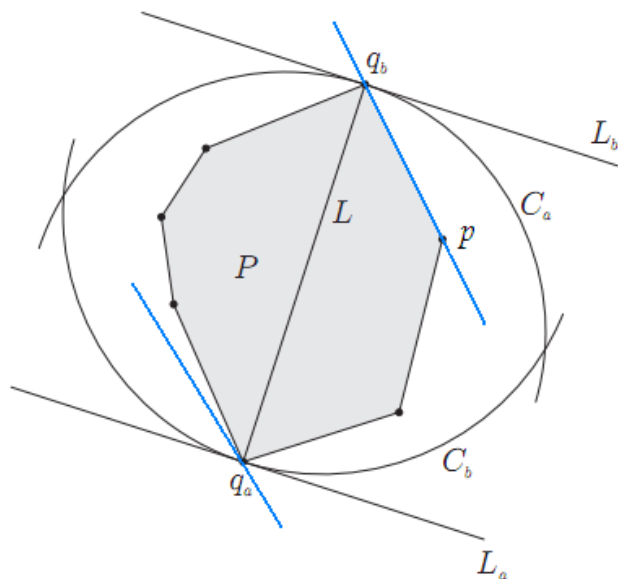
Step3: 同时旋转两条线知道其中一条与多边形的一条边重合。

Step4: 一个新的对踵点对此产生，计算新的距离，并和当前最大值比较，若大于当前最大值，则更新。

Step5: 重复 Step3 和 Step4 的过程直到再次产生对踵点对 ($ymin, ymax$)。

Step6: 输出确定最大直径的对踵点对。

可以看出，如果单对一个凸包进行旋转卡壳，其时间复杂度为 $O(n)$ ，然而，由于一般在此之前要计算凸包，其时间复杂度为 $O(n \log(n))$ ，因此，求一组点的最大距离的时间复杂度一般为 $O(n \log(n))$ 。



直接按照这个描述可以实现旋转卡壳算法，但是代码肯定相当冗长。逆向思考，如果 qa, qb 是凸包上最远两点，必然可以分别过 qa, qb 画出一对平行线。通过旋转这对平行线，我们可以让它和凸包上的一条边重合，如图中蓝色直线，可以注意到， qa 是凸包上离 p 和 qb 所在直线最远的点。于是我们的思路就是枚举凸包上的所有边，对每一条边找出凸包上离该边最远的顶点，计算这个顶点到该边两个端点的距离，并记录最大的值。直观上这是一个 $O(n^2)$ 的算法，和直接枚举任意两个顶点一样了。但是注意到当我们逆时针枚举边的时候，最远点的变化也是逆时针的，这样就可以不用从头计算最远点，而可以紧接着上一次的最近点继续计算。于是我们得到了 $O(n)$ 的算法。

//计算凸包直径，输入凸包 ch ，顶点个数为 n ，按逆时针排列，输出直径的平方

```

int rotating_calipers(Point *ch, int n) ... {
    int q=1, ans=0;
    ch[n]=ch[0];
    for(int p=0; p<n; p++) ... {
        while(cross(ch[p+1], ch[q+1], ch[p]) > cross(ch[p+1], ch[q], ch[p]))
            q=(q+1)%n;
        ans=max(ans, max(dist(ch[p], ch[q]), dist(ch[p+1], ch[q+1])));
    }
    return ans;
}

```

很难想象这个看起来那么麻烦的算法只有这么几行代码吧！其中 `cross` 函数是计算叉积，可以想成是计算三角形面积，因为凸包上距离一条边最远的点和这条边的两个端点构成的三角形面积是最大的。之所以既要更新 $(ch[p], ch[q])$ 又要更新 $(ch[p+1], ch[q+1])$ 是为了处理凸包上两条边平行的特殊情况。

可结合 POJ2187, POJ2079 练习。

4. 三维凸包的增量法

在亚洲区比赛中以及省赛东北赛中，三维凸包的题目屡见不鲜，所以在二维的基础上我们需要解决三维凸包了。解决三维凸包常见的有两种方法，卷包裹法，分治算法，以及 Z3-3 算法和增量法，但卷包裹法的时间复杂度是 $O(n^2)$ ，且代码实现麻烦，所以我们在这里主要介绍增量法，其期望时间复杂度为 $O(n \log(n))$ 。

初始时需要一个四面体。可以先找两个不同点 P_1, P_2 ，寻找和它们不共线的第三个点 P_3 ，再找不共面的第四个点 P_4 。如果找不到，则调用二维凸包算法。

接下来计算剩下点的随机排列。每次加一个点，有两种情况：

情况 1：新点在当前凸包内部，只需简单地忽略该点，如图 1 所示。

情况 2：新点在当前凸包外部，需要计算并的凸包，在这种情况下，首先需要计算原凸包相对于 P_r 的水平面，即 P_r 可以看到的封闭区域，如图 2 所示。

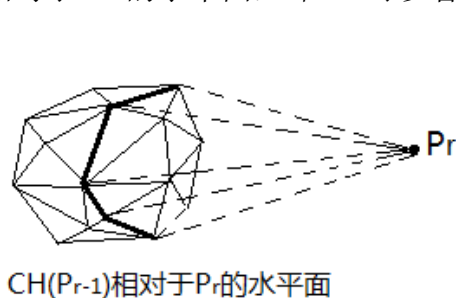


图 1

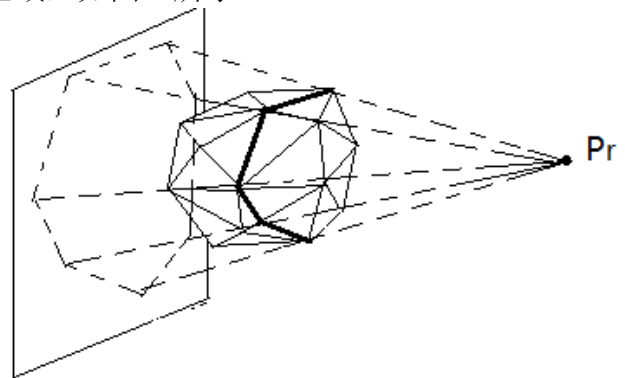


图 2

将 P 点能看到的所有平面删去，同时求出相对于 P_r 的水平面，将水平面上的边与 P_r 相连组成面，如图 3 所示。

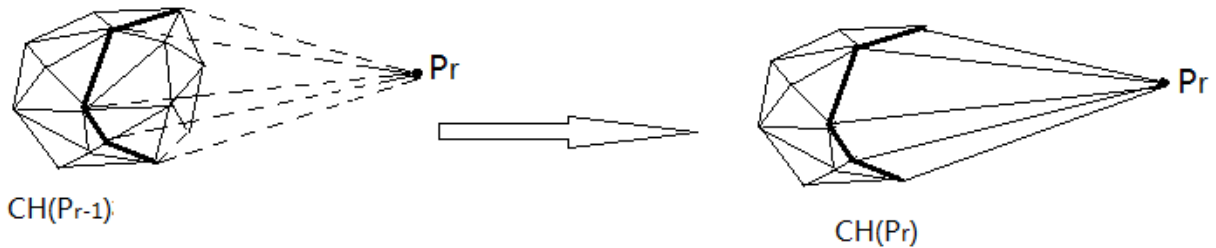


图 3

判断一个点 P 在凸包内还是凸包外，可利用有向体积的方法。在存储面时，保证面的法线方向朝向凸包外部，若存在某一平面和点 P 所组成的四面体的有向体积为正，则 P 点在凸包外部，并且此面可被 P 点看见。此时，只要将此面删去并用同样的方法判断与它相邻的其他平面是否可被 P 点看见，可以使用深度优先搜索。

此算法对于每个点需要求出它所能看到的所有平面，最简单的方法便是枚举所有当前凸包上的平面，一一判断，此时算法时间复杂度是 $O(n^2)$ 。此外还有更高效的方法来寻找 P 点的可见区域，算法的时间复杂度为 $O(n \log(n))$ ，但过于繁琐，不适合程序设计的应用。

8.2.3 解题思路

凸包问题的解题思路很多时候都比较明显，直接套用模板即可，特别是三维凸包只要能提取出模型基本都能利用模板解决。

8.2.4 模板代码

```
// Graham 扫描法
#include<stdio.h>
#include<math.h>
#include<algorithm>
using namespace std;
struct Point
{
    double x,y,len;
}Pt[20000],Stack[20000],Point_A;
double Cross(Point a,Point b,Point c)
{
    return (b.x-a.x)*(c.y-a.y)-(b.y-a.y)*(c.x-a.x);
}
double Dis(Point a,Point b)
{
    return sqrt(pow(a.x-b.x,2)+pow(a.y-b.y,2));
}
void FindPoint(int n)
{
    int i,tempNumber=0;
    Point tempPoint;
    Point_A=Pt[0];
    for(i=1;i<n;i++)
    {
        if(Pt[i].y<Point_A.y||Pt[i].y==Point_A.y&&Pt[i].x<Point_A.x)
        {
            tempNumber=i;
            Point_A=Pt[i];
        }
    }
    tempPoint=Pt[0];
    Pt[0]=Pt[tempNumber];
    Pt[tempNumber]=tempPoint;
```

```

    }
    bool Cmp(Point a,Point b)
    {
        double k=Cross(Point_A,a,b);
        if(k>0) return true;
        if(k<0) return false;
        a.len=Dis(Point_A,a);
        b.len=Dis(Point_A,b);
        return a.len>b.len;
    }
    void Graham(int n)
    {
        int i,top=2;
        Pt[n]=Pt[0];
        Stack[0]=Pt[0];
        Stack[1]=Pt[1];
        Stack[2]=Pt[2];
        for(i=3;i<=n;i++)
        {
            while(Cross(Stack[top-1],Stack[top],Pt[i])<=0&&top>1) top--;
            Stack[++top]=Pt[i];
        }
    }
    int main(void)
    {
        int i,Num;
        while(scanf("%d",&Num)!=EOF)
        {
            for(i=0;i<Num;i++) scanf("%lf%lf",&Pt[i].x,&Pt[i].y);
            FindPoint(Num);
            sort(Pt,Pt+Num,Cmp);
            Graham(Num);
        }
        return 0;
    }
    //三维凸包增量法
    #include<stdio.h>
    #include<string.h>
    #include<math.h>
    #include<algorithm>
    #include <iostream>
    using namespace std;
    #define PR 1e-9
    #define N 1100
    struct TPoint
    {
        double x,y,z;
        TPoint(){}
        TPoint(double _x,double _y,double _z):x(_x),y(_y),z(_z){}
        TPoint operator-(const TPoint p) {return TPoint(x-p.x,y-p.y,z-p.z);}
        TPoint operator*(const TPoint p) {return TPoint(y*p.z-z*p.y,z*p.x-x*p.z,x*p.y-
y*p.x);} //叉积
        double operator^(const TPoint p) {return x*p.x+y*p.y+z*p.z;} //点积
    };
    TPoint dd;
    struct fac//
    {
        int a,b,c;//凸包一个面上的三个点的编号
        bool ok;//该面是否是最终凸包中的面
    }

```

```

};
TPoint xmult(TPoint u,TPoint v)
{
    return TPoint(u.y*v.z-v.y*u.z,u.z*v.x-u.x*v.z,u.x*v.y-u.y*v.x);
}
double dmult(TPoint u,TPoint v)
{
    return u.x*v.x+u.y*v.y+u.z*v.z;
}
TPoint subt(TPoint u,TPoint v)
{
    return TPoint(u.x-v.x,u.y-v.y,u.z-v.z);
}
double vlen(TPoint u)
{
    return sqrt(u.x*u.x+u.y*u.y+u.z*u.z);
}
TPoint pvec(TPoint a,TPoint b,TPoint c)
{
    return xmult(subt(a,b),subt(b,c));
}
double Dis(TPoint a,TPoint b,TPoint c,TPoint d)
{
    return fabs(dmult(pvec(a,b,c),subt(d,a)))/vlen(pvec(a,b,c));
}
struct T3dhull
{
    int n;//初始点数
    TPoint ply[N];//初始点
    int trianglecnt;//凸包上三角形数
    fac tri[N];//凸包三角形
    int vis[N][N];//点 i 到点 j 是属于哪个面
    double dist(TPoint a){return sqrt(a.x*a.x+a.y*a.y+a.z*a.z);}//两点长度
    double area(TPoint a,TPoint b,TPoint c){return dist((b-a)*(c-a));}//三角形面积*2
    double volume(TPoint a,TPoint b,TPoint c,TPoint d){return (b-a)*(c-a)^(d-a);}//四面体有向体积*6
    double ptoplane(TPoint &p,fac &f)//正: 点在面同向
    {
        TPoint 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];//与当前面(cnt)共边(ab)的那个面
        fac add;
        if(tri[f].ok)
        {
            if((ptoplane(ply[p],tri[f]))>PR) dfs(p,f);//如果 p 点能看到该面 f, 则继续深度探索 f 的 3 条边, 以便更新新的凸包面
            else//否则因为 p 点只看到 cnt 面, 看不到 f 面, 则 p 点和 a、b 点组成一个三角形。
            {
                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; //当前面需要删除,因为它在更大的凸包里面
 //下面把边反过来(先 b,后 a),以便在 deal()中判断与当前面(cnt)共边(ab)的那个面。即判断与当头面(cnt)相邻的 3 个面(它们与当前面的共边是反向的,如下图中(1)的法线朝外(即逆时针)的面 130 和 312,它们共边 13,但一个方向是 13,另一个方向是 31)

```

        deal(p,tri[cnt].b,tri[cnt].a);
        deal(p,tri[cnt].c,tri[cnt].b);
        deal(p,tri[cnt].a,tri[cnt].c);
    }
    bool same(int s,int e) { //判断两个面是否为同一面
        TPoint a=ply[tri[s].a],b=ply[tri[s].b],c=ply[tri[s].c];
        return fabs(volume(a,b,c,ply[tri[e].a]))<PR
            &&fabs(volume(a,b,c,ply[tri[e].b]))<PR
            &&fabs(volume(a,b,c,ply[tri[e].c]))<PR;
    }
    void construct() { //构建凸包
        int i,j;
        trianglecnt=0;
        if(n<4) return ;
        bool tmp=true;
        for(i=1;i<n;i++){ //前两点不共点
            if((dist(ply[0]-ply[i]))>PR) {
                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])))>PR) {
                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]))>PR) {
                swap(ply[3],ply[i]); tmp=false; break;
            }
        }
        if(tmp) return ;
        fac add;
        for(i=0;i<4;i++){ //构建初始四面体(4 个点为 ply[0],ply[1],ply[2],ply[3])
            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); //保证逆时针,即法向量朝外,这样
            新点才可看到。
            vis[add.a][add.b]=vis[add.b][add.c]=vis[add.c][add.a]=trianglecnt; // 逆向的
            有向边保存
            tri[trianglecnt++]=add;
        }
        for(i=4;i<n;i++){ //构建更新凸包
            for(j=0;j<trianglecnt;j++){ //对每个点判断是否在当前 3 维凸包内或外(i 表示当前点,j 表
            示当前面)
                if(tri[j].ok&&(ptoplane(ply[i],tri[j]))>PR) { //对当前凸包面进行判断,看是否
                点能否看到这个面
                    dfs(i,j); break; //点能看到当前面,更新凸包的面(递归,可能不止更新一个面)。当
                前点更新完成后 break 跳出循环
                }
            }
        }
    }

```

```

int cnt=trianglecnt;//这些面中有一些 tri[i].ok=0, 它们属于开始建立但后来因为在更大凸包
内故需删除的, 所以下面几行代码的作用是只保存最外层的凸包
trianglecnt=0;
for(i=0;i<cnt;i++){
    if(tri[i].ok)
        tri[trianglecnt++]=tri[i];
}
}
double res(){
    double _min=1e300;
    for(int i=0;i<trianglecnt;i++){
        double now=Dis(ply[tri[i].a],ply[tri[i].b],ply[tri[i].c],dd);
        if(_min>now) _min=now;
    }
    return _min;
}
}hull;
int main(){
    double now,_min;
    while(scanf("%d",&hull.n)!=EOF) {
        if(hull.n==0) break;
        int i,j,q;

        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();
        scanf("%d",&q);
        for(j=0;j<q;j++){
            scanf("%lf%lf%lf",&dd.x,&dd.y,&dd.z);
            printf("%.4lf\n",hull.res());
        }
    }
    return 0;
}

```

8.2.5 经典题目

1. 题目出处/来源

POJ-1113Wall

2. 题目描述

给出一些点, 要求修建一个围墙, 并且围墙离点的距离不小于 L , 求围墙的最小长度

3. 分析

这道题首先要找出所有点构成的凸包的周长, 外层修建的围墙是在这个周长的基础上往外拓展, 然后在每一个转弯的地方用圆弧代替, 这样整圈下来圆弧就构成了一个整圆, 所以整个的长度就是凸包的长度加上一个整圆的长度, 圆的半径是 L

4. 代码

```

//POJ1113.cpp
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define N 1005
#define pi 3.141592653
struct point
{
    int x,y;
};
int n;

```

```

int dis(point p1,point p2)//距离的平方
{
    return (p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y);
}
point p[N],res[N];
int mul(point p1,point p2,point p3)//叉积, 这里和上面的叉积函数名有些出入, 但是不影响
{
    return (p2.x-p1.x)*(p3.y-p1.y)-(p2.y-p1.y)*(p3.x-p1.x);
}
int cmp(const void *a,const void *b)//极角排序, 要注意快速排序的排序函数
{
    point c=(point *)a;
    point d=(point *)b;
    int t=mul(p[0],c,d);
    if(t)
        return -t;
    return dis(p[0],d)-dis(p[0],c);
}
int graham()//扫描法求凸包部分
{
    int i,j,k,top=2;
    j=0;
    for(i=1;i<n;i++)
    {
        if(p[i].y<p[j].y||(p[i].y==p[j].y&& p[i].x<p[j].x))
            j=i;
    }
    if(j)
    {
        point temp=p[j];
        p[j]=p[0];
        p[0]=temp;
    }
    qsort(p+1,n-1,sizeof(p[0]),cmp);
    //for(i=0;i<n;i++)
    //printf("%d %d\n",p[i].x,p[i].y);
    res[0]=p[0];
    res[1]=p[1];
    res[2]=p[2];
    for(i=3;i<n;i++)
    {
        while(top>1&&mul(res[top-1],res[top],p[i])<=0)
            --top;
        res[++top]=p[i];
    }
    return top;
}
int main()
{
    int i,j,k;
    int l;
    while(scanf("%d%d",&n,&l)!=EOF)
    {
        for(i=0;i<n;i++)
            scanf("%d%d",&p[i].x,&p[i].y);
        int t=graham();
        res[t+1]=res[0];
        double ans=0;
        for(i=0;i<=t;i++)

```



```

    ans+=sqrt((double)dis(res[i],res[i+1]));
    ans+=2*pi*1;//加上圆的周长
    printf("%.1f\n",ans);
}
return 0;
}

```

8.3 半平面交

参考文献:

《算法艺术与信息学竞赛》(刘汝佳、黄亮,清华大学出版社)

《计算机图形学基础教程》(孙家广,清华大学出版社)

《ACMICPC 程序设计系列—计算几何》(哈尔滨工业大学出版社)

扩展阅读: $O(n\log n)$ 半平面交算法

编写: 彭文文

校核: 曹振海

8.3.1 基本原理

1. 什么是半平面? 顾名思义, 半平面就是指平面的一半, 我们知道, 一条直线可以将平面分为两个部分, 那么这两个部分就叫做两个半平面。
2. 半平面怎么表示呢? 二维坐标系下, 直线可以表示为 $ax + by + c = 0$, 那么两个半平面则可以表示为 $ax + by + c \geq 0$ 和 $ax + by + c < 0$, 这就是半平面的表示方法。
3. 半平面的交是什么? 其实就是一个方程组, 让你画出满足若干个式子的坐标系上的区域 (类似于线性规划的可行域), 方程组就是由类似于上面的这些不等式组成的。
4. 半平面交可以干什么? 半平面交虽然说是半平面的问题, 但它其实就是关于直线的问题。一个一个的半平面其实就是一个一个有方向的直线而已。

8.3.2 求半平面交的方法

最常见的 n 个半平面的交有三种解法:

1. 增量法:

假设已经得到前 $n-1$ 个半平面的交, 对于第 n 个半平面, 只需用它来“切割”前 $n-1$ 个半平面交出的多边形。“切割”的过程需要遍历整个多边形, 时间复杂度为 $O(n)$, 则算法的时间复杂度为 $O(n^2)$ 。

2. 分治法:

将 n 个半平面分成两部分, 分别求完交后再将两部分的交合并求交集。整个算法的效率依赖于能够高效地取两个凸多边形的交。利用多边形的交可以在 $O(n)$ 时间内求取的技术, 这个算法的时间复杂度是 $O(n\log(n))$ 。

3. 排序增量法:

step1. 将所有半平面按极角排序, 对于极角相同的, 选择性的保留一个。 $O(n\log n)$

step2. 使用一个双端队列(deque), 加入最开始 2 个半平面。

step3. 每次考虑一个新的半平面:

a. while deque 顶端的两个半平面的交点在当前半平面外:删除 deque 顶端的半平面

b. while deque 底部的两个半平面的交点在当前半平面外:删除 deque 底部的半平面

c. 将新半平面加入 deque 顶端

step4. 删除两端多余的半平面。

具体方法是:

a. while deque 顶端的两个半平面的交点在底部半平面外:删除 deque 顶端的半平面
 b. while deque 底部的两个半平面的交点在顶端半平面外:删除 deque 底部的半平面
 重复 a, b 直到不能删除为止。

step5: 计算出 deque 顶端和底部的交点即可。

这个算法描述的非常清晰。当初写的时候有两个地方想的不太明白: step 1 如何选择性的保留一个。step3 如何判断交点在半平面外。

其实这两个问题都可以用叉积来解决。首先根据给定的两点顺序规定好极角序。假定两点 o_1o_2 的输入方向是顺时针, 那

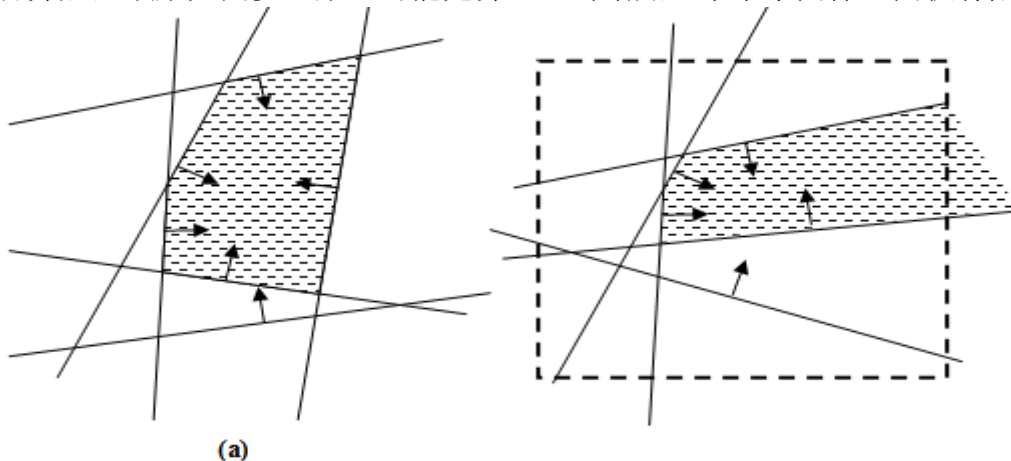
么另一点 P 是否在其平面内只要判断 o_1P 这个向量是否在 o_1o_2 这个向量的右手边即可。对于相同角度的两个半平面

(a_1a_2, b_1b_2), 可以看 a_1b_1 这个向量是否在 a_1a_2 这个向量的右手边, 每次都要选择更靠近右手边的那个半平面。

半平面的交是当今学术界热烈讨论的问题之一, 本文将介绍一个全新的 $O(n \log n)$ 半平面交算法, 强调它在实际运用中的价值, 并且在某种程度上将复杂度下降至 $O(n)$ 线性。最重要的是, 我将介绍的算法非常便于实现。

§1 什么是半平面交. §2 凸多边形交预备知识. §3 简要介绍旧 D&C 算法. §4 揭开我的新算法 S&I 神秘面纱. §5 总结和实际运用。

众所周知, 直线常用 $ax+by=c$ 表示, 类似地半平面以 $ax+by \leq (\geq) c$ 为定义。给定 n 个形如 $a_i x + b_i y \leq c_i$ 的半平面, 找到所有满足它们的点所组成的点集合并后区域形如凸多边形, 可能无界。此时增加 4 个半平面保证面积有限



每个半平面最多形成相交区域的一条边, 因此相交区域不超过 n 条边。

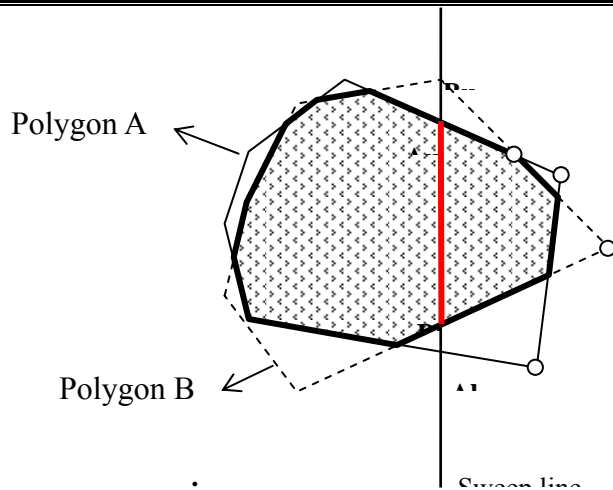
注意相交后的区域, 有可能是一个直线、射线、线段或者点, 当然也可能是空集。

求两个凸多边形 A 和 B 的交 (一个新凸多边形)。我们描绘一个平面扫描法。

主要思想: 以两凸多边形边的交点为分界点, 将边分为内、外两种。内边互相连接, 成为所求多边形。

假设有一个垂直的扫描线, 从左向右扫描。我们称被扫描线扫描到的 x 坐标叫做 x 事件。

任何时刻, 扫描线和两个多边形最多 4 个交点



Au、Bu 中靠下的，和 Al、Bl 中靠上的，组成了当前多边形的内部区域。

Obviously, the sweep line may not go through all the x-event with rational coordinates. Call the edges where Au, Al, Bu and Bl are: e1, e2, e3 and e4 respectively. The next x-event should be chosen among four endpoints of e1, e2, e3 and e4, and four potential intersections: $e1 \cap e3$, $e1 \cap e4$, $e2 \cap e3$ and $e2 \cap e4$. 当然，我们不能扫描所有有理数！称 Au, Al, Bu, Bl 所在的边叫做 e1, e2, e3, e4，下一个 x 事件将在这四条边的端点，以及两两交点中选出。

The above operation could be implemented with $O(n)$ running time, since there are $O(n)$ x-events, and the maintenance of Au, Al, Bu and Bl takes only $O(1)$.

分：将 n 个半平面分成两个 $n/2$ 的集合。

治：对两子集合递归求解半平面交。

合：将前一步算出来的两个交(凸多边形)利用第 2 章的 CPI 求解。

总时间复杂度可以用递归分析法。

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = O(n \log n)$$

My New Solution:

Sort-and-Incremental Algorithm (abbr. S&I)

Definition of h-plane's polar angle:

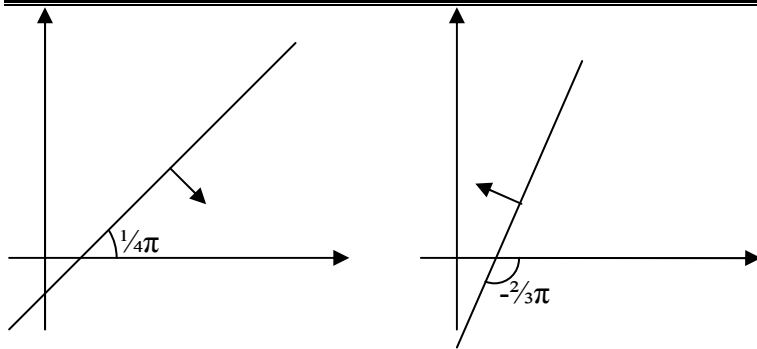
for the h-plane like $x-y \geq \text{constant}$, we define its polar angle to $\frac{1}{4}\pi$.

for the h-plane like $x+y \leq \text{constant}$, we define its polar angle to $\frac{3}{4}\pi$.

for the h-plane like $x+y \geq \text{constant}$, we define its polar angle to $-\frac{1}{4}\pi$.

for the h-plane like $x-y \leq \text{constant}$, we define its polar angle to $-\frac{3}{4}\pi$.

半平面的极角定义：比如 $x-y \geq \text{常数}$ 的半平面，定义它的极角为 $\frac{1}{4}\pi$ 。

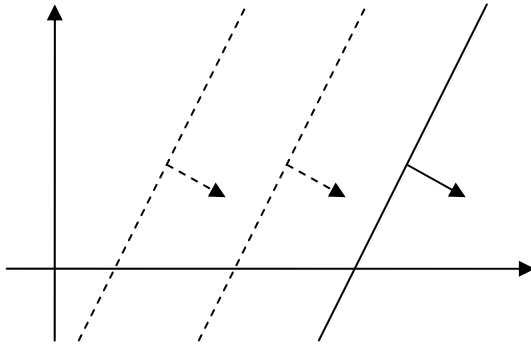


Definition of h-plane's constant:

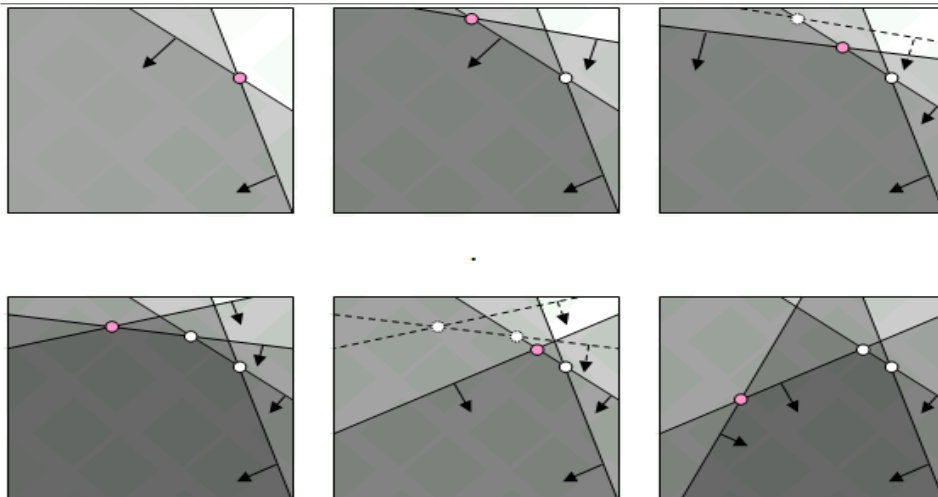
for the h-plane like $ax+by \leq c$, we say its constant is c .

My new Sort-and-Incremental Algorithm seems lengthy since I am going to introduce it in details:

Step 1: 将半平面分成两部分, 一部分极角范围 $(-\frac{1}{2}\pi, \frac{1}{2}\pi]$, 另一部分范围 $(-\pi, -\frac{1}{2}\pi] \cup (\frac{1}{2}\pi, \pi]$



Step 2: 考虑 $(-\frac{1}{2}\pi, \frac{1}{2}\pi]$ 的半平面 (另一个集合类似地做 Step3/4), 将他们极角排序。对极角相同的半平面, 根据常数项保留其中之一。

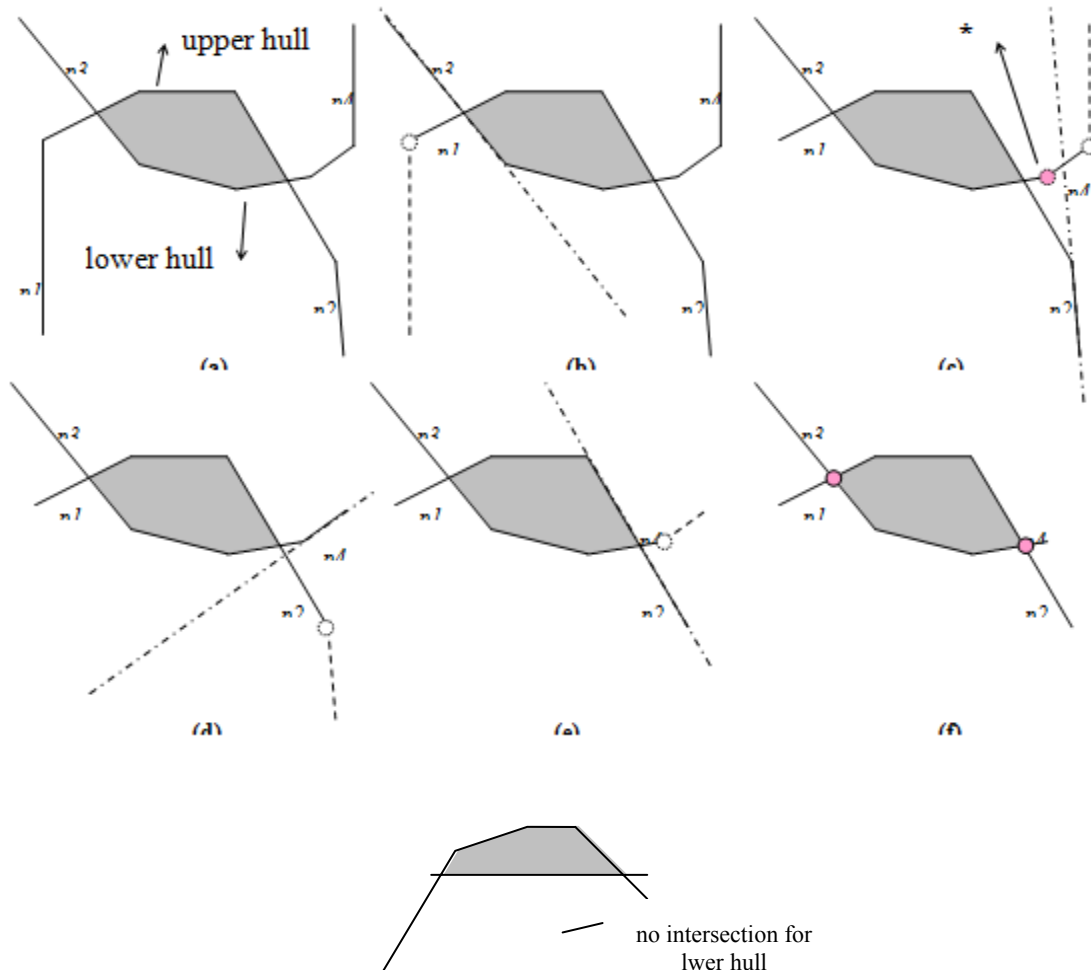


Step 3: 从排序后极角最小两个半平面开始, 求出它们的交点并且将他们押入栈。

每次按照极角从小到大顺序增加一个半平面, 算出它与栈顶半平面的交点。如果当前的交点在栈顶两个半平面交点的右边, 出栈 (pop)。前问我们说到出栈, 出栈只需要一次么? Nie! 我们要继续交点检查, 如果还在右边我们要继续出栈, 直到当前交点在栈顶交点的左边。

Step 4: 相邻半平面的交点组成半个凸多边形。我们有两个点集, $(-\frac{1}{2}\pi, \frac{1}{2}\pi]$ 给出上半个, $(-\pi, -\frac{1}{2}\pi] \cup (\frac{1}{2}\pi, \pi]$ 给出下半个

初始时候, 四个指针 $p1, p2, p3$ and $p4$ 指向上/下凸壳的最左最右边。 $p1, p3$ 向右走, $p2, p4$ 向左走。任意时刻, 如果最左边的交点不满足 $p1/p3$ 所在半平面的限制, 我们相信这个交点需要删除。 $p1$ 或 $p3$ 走向它右边的相邻边。类似地我们处理最右边的交点。重复运作直到不再有更新出现——迭代。



除了 Step2 中的排序以外, S&I 算法的每一步都是线性的。通常我们用快速排序实现 Step2, 总的时间复杂度为 $O(n \log n)$, 隐蔽其中的常数因子很小

S&I 算法似乎和 D&C 算法时间复杂度相同, 但是它有着压倒性的优势。新的 S&I 算法代码容易编写, 相对于 D&C 大大简单化, C++ 程序语言实现 S&I 算法仅需 3KB 不到。

S&I 算法复杂度中的系数, 远小于 D&C, 因为我们不再需要 $O(n \log n)$ 次交点运算。通常意义上来讲, S&I 程序比 D&C 快五倍。

如果给定半平面均在 $(-\frac{1}{2}\pi, \frac{1}{2}\pi]$ (或任意一个跨度为 π 的区间), S&I 算法可被显著缩短, C++ 程序只需要约二十行。USAIC0 比赛中就出现了这样一题。

本算法瓶颈是排序, 这里的排序不是比较排序, 因此可以将快速排序替换成基数排序, 降低程序渐进时间复杂度到线性。

8.3.3 解题思路

半平面交有一个重要的应用就是多边形的核:

多边形的核:

半平面交的一个重要应用就是求多边形的核。多边形的核又是什么? 它是平面简单多边

形的核是该多边形内部的一个点集，该点集中任意一点与多边形边界上一点的连线都处于这个多边形内部。就是一个在一个房子里面放一个摄像头，能将所有的地方监视到的放摄像头的地点的集合即为多边形的核。经常会遇到让你判定一个多边形是否有核的问题。

半平面交的最终奥义就是求出一个满足条件的凸多边形，而解决这个问题的前提就是直线切割多边形，让直线不断的去切割当前多边形，然后得到新的多边形，然后继续。。最终得到一个符合条件的多边形，如果最终的多边形顶点数目为 0，那么就说明半平面交无解（半平面交的解可以是一个凸多边形，一条直线，一个点，我们用顶点来记录这些解）。

关于直线切割多边形，流程是这样的：

对凸多边形（指的是当前多边形）的每一个顶点，如果这个顶点在直线的指定的一侧（暨在该半平面上），那么就将该顶点直接加入到新的多边形中，否则，看与该点相邻的多边形上的两个点（判断线段是否和直线相交），如果和直线相交，则把交点加入到新的多边形中。这样，就可以得到一个新的凸多边形。关于最初的多边形，我们可以设一个四方形区域，比如说 $(-\infty, -\infty)$, $(-\infty, \infty)$, (∞, ∞) , $(\infty, -\infty)$ ，然后开始切割。

8.3.4 模板代码

半平面交 $O(n^2)$ 增量法

```
Point points[MAXN], p[MAXN], q[MAXN];
int n;
double r;
int cCnt, curCnt;
inline void getline(Point x, Point y, double &a, double &b, double &c){
    a = y.y - x.y;
    b = x.x - y.x;
    c = y.x * x.y - x.x * y.y;
}
inline void initial(){
    for(int i = 1; i <= n; i++) p[i] = points[i];
    p[n+1] = p[1];
    p[0] = p[n];
    cCnt = n;
}
inline Point intersect(Point x, Point y, double a, double b, double c){
    double u = fabs(a * x.x + b * x.y + c);
    double v = fabs(a * y.x + b * y.y + c);
    return Point( (x.x * v + y.x * u) / (u + v) , (x.y * v + y.y * u) / (u + v) );
}
inline void cut(double a, double b, double c){
    curCnt = 0;
    for(int i = 1; i <= cCnt; ++i){
        if(a*p[i].x + b*p[i].y + c >= EPS) q[++curCnt] = p[i];
        else {
            if(a*p[i-1].x + b*p[i-1].y + c > EPS){
                q[++curCnt] = intersect(p[i], p[i-1], a, b, c);
            }
            if(a*p[i+1].x + b*p[i+1].y + c > EPS){
                q[++curCnt] = intersect(p[i], p[i+1], a, b, c);
            }
        }
    }
    for(int i = 1; i <= curCnt; ++i) p[i] = q[i];
    p[curCnt+1] = q[1]; p[0] = p[curCnt];
    cCnt = curCnt;
}
inline void solve(){
```

```

//注意：默认点是顺时针，如果题目不是顺时针，规整化方向
initial();
for(int i = 1; i <= n; ++i){
    double a,b,c;
    getline(points[i],points[i+1],a,b,c);
    cut(a,b,c);
}
/*
如果要向内推进 r，用该部分代替上个函数
for(int i = 1; i <= n; ++i){
    Point ta, tb, tt;
    tt.x = points[i+1].y - points[i].y;
    tt.y = points[i].x - points[i+1].x;
    double k = r / sqrt(tt.x * tt.x + tt.y * tt.y);
    tt.x = tt.x * k;
    tt.y = tt.y * k;
    ta.x = points[i].x + tt.x;
    ta.y = points[i].y + tt.y;
    tb.x = points[i+1].x + tt.x;
    tb.y = points[i+1].y + tt.y;
    double a,b,c;
    getline(ta,tb,a,b,c);
    cut(a,b,c);
}
*/
//多边形核的面积
double area = 0;
for(int i = 1; i <= curCnt; ++i)
    area += p[i].x * p[i + 1].y - p[i + 1].x * p[i].y;
area = fabs(area / 2.0);
//此时 cCnt 为最终切割得到的多边形的顶点数，p 为存放顶点的数组
}
inline void GuiZhengHua(){
    //规整化方向，逆时针变顺时针，顺时针变逆时针
    for(int i = 1; i < (n+1)/2; i ++){
        swap(points[i], points[n-i]); //头文件加 iostream
    }
}
inline void init(){
    for(int i = 1; i <= n; ++i) points[i].input();
    points[n+1] = points[1];
}
}
    
```

半平面交 $O(n \log(n))$ 模板

题意：给出很多个半平面，这里每个半平面由线段组成，都是指向线段方向的左边表示有
 $(x_1 - x) * (y_2 - y) - (x_2 - x) * (y_1 - y) \geq 0$ (≥ 0 表示左边， ≤ 0 表示右边)
 要你求个半平面的核，就是所有半平面所围成的面积

算法： $O(n \log n)$ 的半平面交算法，统计出得到的多边形的点，然后利用叉积公式求出面积

```

#include<cstdio>
#include<vector>
#include<cmath>
#include<algorithm>
using namespace std;
const double eps=1e-10,big=10000.0;
const int maxn = 20010;
struct point{ double x,y; };
struct polygon { //存放最后半平面交中相邻边的交点，就是一个多边形的所有点
    int n;
    point p[maxn];
};
    
```

```

struct line { //半平面, 这里是线段
    point a,b;
};
double at2[maxn];
int ord[maxn],dq[maxn+1],lnum;
int n;
polygon pg;
line ls[maxn]; //半平面集合
inline int sig(double k) { //判是不是等于 0, 返回-1, 0, 1, 分别是小于, 等于, 大于
    return (k < -eps)? -1: k > eps;
}
//叉积>0 代表在左边, <0 代表在右边, =0 代表共线
//e 是否在 o->s 的左边 onleft(sig(multi))>=0
inline double multi(point o, point s, point e)
{ //构造向量, 然后返回叉积
    return (s.x-o.x)*(e.y-o.y)-(e.x-o.x)*(s.y-o.y);
}
//直线求交点
point isIntersected(point s1, point e1, point s2, point e2) {
    double dot1,dot2;
    point pp;
    dot1 = multi(s2,e1,s1); dot2 = multi(e1,e2,s1);
    pp.x = (s2.x * dot2 + e2.x * dot1) / (dot2 + dot1);
    pp.y = (s2.y * dot2 + e2.y * dot1) / (dot2 + dot1);
    return pp;
}
//象限排序
inline bool cmp(int u,int v) {
    if(sig(at2[u]-at2[v])==0)
        return sig(multi(ls[v].a,ls[v].b,ls[u].b))>=0;
    return at2[u]<at2[v];
}
//判断半平面的交点在当前半平面外
bool judgein(int x,int y,int z){
    point pnt = isIntersected(ls[x].a, ls[x].b, ls[y].a, ls[y].b); //求交点
    return sig(multi(ls[z].a,ls[z].b,pnt)) < 0; //判断交点位置, 如果在右面, 返回 true, 如果要排除三点共线, 改成<=
}
//半平面交
void HalfPlaneIntersection(polygon &pg) { //预处理
    int n = lnum, tmpn, i;
    /* 对于 atan2(y,x)
    结果为正表示从 x 轴逆时针旋转的角度, 结果为负表示从 x 轴顺时针旋转的角度。
    atan2(a, b) 与 atan(a/b)稍有不同, atan2(a,b)的取值范围介于 -pi 到 pi 之间 (不包括 -pi),
    而 atan(a/b)的取值范围介于-pi/2 到 pi/2 之间 (不包括±pi)*/
    for(i = 0 ; i < n ; i ++ )
    { //atan2(y,x)求出每条线段对应坐标系的角度
        at2[i] = atan2( ls[i].b.y - ls[i].a.y, ls[i].b.x - ls[i].a.x);
        ord[i] = i;
    }
    sort(ord, ord + n, cmp);
    for (i = 1, tmpn = 1; i < n; i++) //处理重线的情况
        if( sig(at2[ord[i-1]] - at2[ord[i]]) != 0 ) ord[tmpn++] = ord[i];
    n = tmpn;
    //圈地
    int bot = 1, top = bot + 1; //双端栈, bot 为栈底, top 为栈顶
    dq[bot] = ord[0]; dq[top] = ord[1]; //先压两根线进栈
    for(i = 2 ; i < n ; i ++ ) {
        //bot < top 表示要保证栈里至少有 2 条线段, 如果剩下 1 条, 就不继续退栈
    }
}
    
```



```

//judgein, 判断如果栈中两条线的交点如果在当前插入线的右边, 就退栈
while( bot < top && judgein(dq[top-1] , dq[top] , ord[i]) ) top--;
//对栈顶要同样的操作
while( bot < top && judgein(dq[bot+1] , dq[bot] , ord[i]) ) bot++;
dq[++top] = ord[i];
}
//最后还要处理一下栈里面存在的栈顶的线在栈底交点末尾位置, 或者栈顶在栈尾两条线的右边
while( bot < top && judgein(dq[top-1] , dq[top] , dq[bot]) ) top--;
while( bot < top && judgein(dq[bot+1] , dq[bot] , dq[top]) ) bot++;
//最后一条线是重合的
dq[--bot] = dq[top];
//求多边形
pg.n = 0;
for(i = bot + 1; i <= top ; i++) //求相邻两条线的交点
    pg.p[pg.n++] = isIntersected(ls[dq[i-1]].a, ls[dq[i-1]].b,
ls[dq[i]].a,ls[dq[i]].b);
}
inline void add(double a,double b,double c,double d)
{ //添加线段
    ls[lnum].a.x = a; ls[lnum].a.y = b;
    ls[lnum].b.x = c; ls[lnum].b.y = d;
    lnum++;
}
int main() {
    int n,i;
    scanf("%d",&n);
    double a,b,c,d;
    for(i = 0 ; i < n ; i++) {
        //输入代表一条向量(x = (c - a),y = (d - b));
        scanf("%lf%lf%lf%lf",&a,&b,&c,&d);
        add(a,b,c,d);
    }
    //下面是构造一个大矩形边界
    add(0,0,big,0); //down
    add(big,0,big,big); //right
    add(big,big,0,big); //up
    add(0,big,0,0); //left
    HalfPlaneIntersection(pg); //求半平面交
    double area = 0;
    n = pg.n;
    //最后多边形的各个点保存在 pg 里面
    for(i = 0 ; i < n ; i++)
        area += pg.p[i].x * pg.p[(i+1)%n].y - pg.p[(i+1)%n].x * pg.p[i].y; //x1 * y2 -
x2 * y1 用叉积求多边形面积
    area=fabs(area)/2.0; //所有面积应该是三角形面积之和, 而叉积求出来的是四边形的面积和, 所
以要除 2
    printf("%.1f\n",area);
    return 0;
}

```

8.3.5 经典题目

1. 题目出处/来源

P0J-3335 Rotating Scoreboard

2. 题目描述

有一个多边形的区域, 现在要在区域内的某个位置放一个摄像头, 使得摄像头能够照

到区域内的所有的位置，问这样的位置是否存在。

3. 分析

这道题是一道很直接的求多边形的核的问题，只要求出这个多边形是否有核存在即可

4. 代码

//POJ3335.cpp

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<cmath>
#define eps 1e-8
#define N 105
using namespace std;
struct point
{
    double x,y;
};
point p[N];
point s[N];
point q[N];
int n,size,si;
void init()
{
    int i;
    for(i=1;i<=n;i++)
        p[i]=s[i];
    p[n+1]=p[1];
    p[0]=p[n];
    size=n;
}
int sig(double k)
{
    return (k<-eps)?-1:(k>eps);
}
void getl(point p1,point p2,double &a,double &b,double &c)//得到直线方程
{
    a=p2.y-p1.y;
    b=p1.x-p2.x;
    c=p2.x*p1.y-p2.y*p1.x;
}
point intersect(point x,point y,double a,double b,double c)//求交点
{
    double u=fabs(a*x.x+b*x.y+c);
    double v=fabs(a*y.x+b*y.y+c);
    point temp;
    temp.x=(x.x*v+y.x*u)/(u+v);
    temp.y=(x.y*v+y.y*u)/(u+v);
    return temp;
}
void cut(double a,double b,double c)//对多边形的每一条边进行切割
{
    int i;
    si=0;
    for(i=1;i<=size;i++)//枚举在暂时确定的多边形内，有多少点不在直线的右侧
    {
        if(sig(a*p[i].x+b*p[i].y+c)>=0)
            q[++si]=p[i];
        else
            {

```

```

        if(sig(a*p[i-1].x+b*p[i-1].y+c)>0)
            q[++si]=intersect(p[i],p[i-1],a,b,c);
        if(sig(a*p[i+1].x+b*p[i+1].y+c)>0)
            q[++si]=intersect(p[i],p[i+1],a,b,c);
    }
}
for(i=1;i<=si;i++)//更新整个多边形的核
p[i]=q[i];
p[0]=p[si];
p[si+1]=p[1];
size=si;
//for(i=1;i<=si;i++)
//printf("%lf %lf\n",p[i].x,p[i].y);
//printf("%lf %lf %lf %d\n",a,b,c,si);
}
bool solve()
{
    int i;
    double a,b,c;
    for(i=1;i<=n;i++)
    {
        getl(s[i],s[i+1],a,b,c);
        cut(a,b,c);
    }
    //printf("%d\n",size);
    if(size)//不存在核
        return true;
    return false;
}
int main()
{
    int t;
    scanf("%d",&t);
    int i;
    while(t--)
    {
        scanf("%d",&n);
        for(i=1;i<=n;i++)
            scanf("%lf%lf",&s[i].x,&s[i].y);
        s[n+1]=s[1];
        s[0]=s[n];
        init();
        if(solve())
            printf("YES\n");
        else
            printf("NO\n");
    }
    return 0;
}

```

8.3.6 扩展变形

可以结合 POJ3335, POJ1474, POJ1279, POJ3525, POJ3384, POJ1755, POJ2540(半平面交求线性规划可行区域的面积), 2451($O(n \log(n))$ 算法)。

8.4 最近点对

参考文献:

《算法导论》

[分治——二维空间求最近点对](#)[最近点对问题](#)

扩展阅读:

[编程之美之寻找最近点对](#)[最近点对问题](#)

编写: 曹振海

校核: 彭文文

8.4.1 基本原理

最近点对问题的提法是: 给定平面上 n 个点, 找其中的一对点, 使得在 n 个点组成的所有点对中, 该点对间的距离最小。

最近点对问题的基本原理就是分治的思想, 另外还要用到抽屉原理, 在这里我们只介绍一维和二维的最近点对问题, 因为包括三维在内的最近点对问题的解决办法都是很类似的。

一维问题:

严格的讲, 最接近点对可能多于 1 对, 为简单起见, 只找其中的 1 对作为问题的解。简单的说, 只要将每一点与其它 $n-1$ 个点的距离算出, 找出达到最小距离的 2 点即可。但这样效率太低, 故想到分治法来解决这个问题。也就是说, 将所给的平面上 n 个点的集合 S 分成 2 个子集 S_1 和 S_2 , 每个子集中约有 $n/2$ 个点。然后在每个子集中递归的求其最接近的点对。这里, 关键问题是如何实现分治法中的合并步骤, 即由 S_1 和 S_2 的最接近点对, 如何求得原集合 S 中的最接近点对。如果组成 S 的最接近点对的 2 个点都在 S_1 中或都在 S_2 中, 则问题很容易解决, 但如果这 2 个点分别在 S_1 和 S_2 中, 问题就不那么简单了。下面的基本算法中, 将对其作具体分析。

假设用 x 轴上某个点 m 将 S 划分为 2 个集合 S_1 和 S_2 , 使得 $S_1 = \{x \in S \mid x \leq m\}$; $S_2 = \{x \in S \mid x > m\}$ 。因此, 对于所有 $p \in S_1$ 和 $q \in S_2$ 有 $p < q$ 。

递归的在 S_1 和 S_2 上找出其最接近点对 $\{p_1, p_2\}$ 和 $\{q_1, q_2\}$, 并设 $d = \min\{|p_1 - p_2|, |q_1 - q_2|\}$ 。由此易知, S 中的最接近点对或者是 $\{p_1, p_2\}$, 或者是 $\{q_1, q_2\}$, 或者是某个 $\{p_3, q_3\}$, 其中 $p_3 \in S_1$ 且 $q_3 \in S_2$ 。如下图所示:

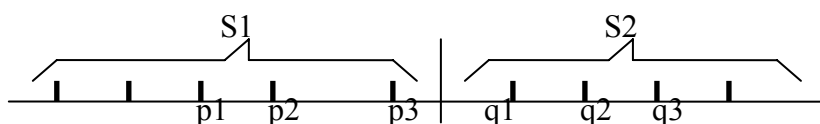


图 1 一维情形的分治法

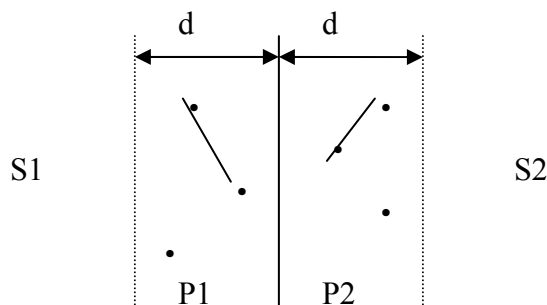
注意到, 如果 S 的最接近点对是 $\{p_3, q_3\}$, 即 $|p_3 - q_3| < d$, 则 p_3 和 q_3 两者与 m 的距离不超过 d , 即 $|p_3 - m| < d$, $|q_3 - m| < d$ 。也就是说, $p_3 \in (m-d, m]$, $q_3 \in (m, m+d]$ 。由于每个长度为 d 的半闭区间至多包含 S_1 中的一个点, 并且 m 是 S_1 和 S_2 的分割点, 因此 $(m-d, m]$ 中至少包含一个 S 中的点。同理, $(m, m+d]$ 中也至少包含一个 S 中的点。由上图知, 若 $(m-d, m]$ 中有 S 的点, 则此点就是 S_1 中最大点。同理, 若 $(m, m+d]$ 中有 S 的点, 则此点就是 S_2 中最小点。因此, 用线性时间就可以找到区间 $(m-d, m]$ 和 $(m, m+d]$ 中所有点, 即 p_3 和 q_3 。从而用线性时间就可以将 S_1 的解和 S_2 的解合并成为 S 的解。其中, 为使 S_1 和 S_2 中有个数大致相等的点, 选取 S 中点坐标的中位数来作分割点 m 。

二维问题:

搞懂一维问题, 二维问题就会好懂很多, 用到的基本原理都是一样的, 只是这里抽屉原理的应用会体现的更明显一些。

S 中的点为平面上的点，它们都有两个坐标值 x 和 y 。为了将平面上点集 S 线形分割为大小大致相等的两个子集 S_1 和 S_2 ，选取一垂直线 $l: x=m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数。由此将 S 分割为 $S_1=\{p \in S | x(p) \leq m\}$ 和 $S_2=\{p \in S | x(p) > m\}$ 。从而使 S_1 和 S_2 分别位于直线 l 的左侧和右侧，且 $S=S_1 \cup S_2$ 。由于 m 是 S 中各点 x 坐标值的中位数，因此 S_1 和 S_2 中的点数大致相等。

递归的在 S_1 和 S_2 上解最接近点对问题，分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2 。现设 $d=\min\{d_1, d_2\}$ 。若 S 的最接近点对 (p, q) 之间的距离小于 d ，则 p 和 q 必分属于 S_1 和 S_2 。不妨设 $p \in S_1, q \in S_2$ 。 p 和 q 距离直线 l 的距离均小于 d 。因此，若用 P_1 和 P_2 分别表示直线 l 的左边和右边的宽为 d 的两个垂直长条。则 $p \in P_1, q \in P_2$ ，如下图所示。



在二维条件下， P_1 中所有点与 P_2 中所有点构成的点对均为最接近点对的候选者。由 d 的意义可知， P_2 中任何两个 S 中的点的距离都不小于 d 。由此可推出矩形 R 中最多只有 6 个 S 中的点。事实上，我们可以将矩形 R 的长为 $2d$ 的边 3 等分，将它的长为 d 的边 2 等分，由此导出 6 个 $(d/2) \times (2d/3)$ 的矩形。若矩形 R 中有多于 6 个 S 中的点，则由鸽舍原理易知至少有一个 $d \times 2d$ 的小矩形中有 2 个以上 S 中的点。设 u, v 是这样 2 个点，它们位于同一小矩形中，因此 $d(u, v) \leq 5\delta/6 < \delta$ 。这与 d 的意义相矛盾。也就是说矩形 R 中最多只有 6 个 S 中的点。

由此稀疏性质，对于 P_1 中任一点 p ， P_2 中最多只有 6 个点与它构成最接近点对的候选者。因此，在分治法的合并步骤中，最多只需要检查 $6 \cdot n/2 = 3 \cdot n$ 个候选者。但并不确切知道要检查哪 6 个点。为解决这问题，可以将 p 和 P_2 中所有 S_2 的点投影到垂直线 l 上。由于能与 p 点一起构成最接近点对候选者的 S_2 中的点一定在 $d \times 2d$ 的矩形中，所以它们在直线 l 上的投影点距 p 在 l 上投影点的距离小于 d 。由上述分析可知，这种投影点最多有 6 个。因此，若将 P_1 和 P_2 中所有 S 中点按其 y 坐标排好序，则对 P_1 中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选者。对 P_1 中每一点最多只要检查 P_2 中排好序的相继 6 个点。

8.4.2 解题思路

最近点对问题的题意和思路一般都比较简单，只要能看懂题意直接套用模板就可以了。

8.4.3 模板代码

```
//一维最近点对.cpp
int max(int s[],int b,int e)//数组的最大值
{
    int m1=s[b],i;
    for(i=b+1;i<=e;i++)
    {
        if(s[i]>m1)
            m1=s[i];
    }
    return m1;
}
```

```

    }
    int min(int s[],int b,int e)//数组最小值
    {
        int m1=s[b],i;
        for(i=b+1;i<=e;i++)
        {
            if(s[i]<m1)
                m1=s[i];
        }
        return m1;
    }
    int val[N];
    int mindis(int s[],int n)//求最近点对函数
    {
        int temp=100000;
        if(n<2)
            return temp;
        int m1=max(s,0,n-1);
        int m2=min(s,0,n-1);
        int m=(m1+m2)/2;//取中位数
        int i,j,k;
        int s1[N],s2[N];
        j=k=0;
        for(i=0;i<n;i++)
        {
            if(s[i]<=m)
                s1[j++]=s[i];
            else
                s2[k++]=s[i];
        }
        int d1=mindis(s1,j);
        int d2=mindis(s2,k);
        int p=max(s1,0,j-1);
        int q=min(s2,0,k-1);
        int dis=d1<d2?d1:d2;
        dis=dis<(q-p)?dis:(q-p);
        return dis;
    }
    //二维最近点对，下面给出的是已经按照 x,y 升序排好序的点对的算法总时间复杂度 nlogn
    基本几何运算：
    向量加减法，叉积，向量积，判断点在线段或直线的某一侧，点在线段或直线上
    判断线段相交。判断点是否在凸多边形中，判断点是否在任意多边形中，
    判断线段是否在多边形中。计算线段或直线的交点
    struct point
    {
        double x,y;
    };
    point p[N];
    point tmp1[N];
    point tmp2[N];
    double dis(point p1,point p2)//两点间距离
    {
        return sqrt(((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y)));
    }
    double mindis(int l,int r)//对应下标 l-r 上的最近点距离
    {
        double d=inf;
        if(l==r)//一个点时返回无穷大
            return d;
    }

```

```

    if(l+1==r)
        return dis(p[l],p[r]);
    int mid=(l+r)>>1; //分割线选择排序好的中间下标的点
    double d1=mindis(l,mid); //分割点左右两边的最近点距离
    double d2=mindis(mid+1,r);
    d=min(d1,d2);
    int i,j,k;
    int cnt1,cnt2;
    cnt1=cnt2=0;
    for(i=mid;i>=l;i--) //对分割线左侧的点进行扫描
    {
        if(p[mid].x-p[i].x<d)
            tmp1[cnt1++]=p[i];
    }
    for(i=mid+1;i<=r;i++)
    {
        if(p[i].x-p[mid].x<d)
            tmp2[cnt2++]=p[i];
    }
    for(i=0;i<cnt1;i++) //判断左侧的点是否有更优的解
    for(j=0;j<cnt2;j++)
    {
        d1=dis(tmp1[i],tmp2[j]);
        if(d1<d)
            d=d1;
    }
    return d;
}
}

```

8.4.4 经典题目

1. 题目出处/来源

POJ-3714Raid

2. 题目描述

联军要攻击敌人的能量供给系统，这个能量供给系统由 N 个发电站组成，攻击任何一个发电站都能导致整个系统的瘫痪，现在派出去 N 个士兵去攻击这个系统，给出 N 个士兵和 N 个电站的坐标位置，要求找出所有士兵中距离电站最近的士兵到最近电站的距离

3. 分析

根据题意就能看出这是最远点对的问题，但是这里有一个问题就是并不是找所有点之间的最近点对，这些点是有分别的，我们在可以给这些点做不同的标记，相同类型的点之间的距离设置为无穷大，就可以找出不同类型的点之间的最近点对了。

4. 代码

```

//POJ3714.cpp
#include<iostream>
#include<cstring>
#include<cstdio>
#include<cstdlib>
#include<cmath>
#include<algorithm>
#define N 200005
#define min(a,b) ((a)<(b)?(a):(b))
#define inf 1e50//无穷大定义
using namespace std;
struct point{
    double x,y;
    int flag;
};

```

```

point p[N];
point tmp1[N];
point tmp2[N];
bool cmp(point a,point b) {
    if(a.x!=b.x)
        return a.x<b.x;
    return a.y<b.y;
}
double dis(point p1,point p2) {
    if(p1.flag==p2.flag)//同一类型的点距离为无穷大
        return inf;
    return sqrt(((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y)));
}
double mindis(int l,int r) {
    double d=inf;
    if(l==r)
        return d;
    if(l+1==r)
        return dis(p[l],p[r]);
    int mid=(l+r)>>1;
    double d1=mindis(l,mid);//分治过程
    double d2=mindis(mid+1,r);
    d=min(d1,d2);
    int i,j,k;
    int cnt1,cnt2;
    cnt1=cnt2=0;
    for(i=mid;i>=l;i--){
        if(p[mid].x-p[i].x<d)
            tmp1[cnt1++]=p[i];
    }
    for(i=mid+1;i<=r;i++){
        if(p[i].x-p[mid].x<d)
            tmp2[cnt2++]=p[i];
    }
    for(i=0;i<cnt1;i++)
        for(j=0;j<cnt2;j++){
            d1=dis(tmp1[i],tmp2[j]);
            if(d1<d)
                d=d1;
        }
    return d;
}
int main(){
    int n,i,j,k;
    int T;
    scanf("%d",&T);
    while(T--){
        scanf("%d",&n);
        for(i=0;i<n;i++){
            scanf("%lf%lf",&p[i].x,&p[i].y);
            p[i].flag=0;//对点的标记
        }
        for(i=n;i<2*n;i++){
            scanf("%lf%lf",&p[i].x,&p[i].y);
            p[i].flag=1;
        }
        n*=2;
        sort(p,p+n,cmp);//先对点进行一遍排序，这样可以方便处理
        printf("%.3lf\n",mindis(0,n-1));
    }
}

```



```

    }
    return 0;
}

```

5. 思考与扩展：可以借鉴北大培训教材中做法。

8.5 最远点对

参考文献：

《算法艺术与信息学竞赛》（刘汝佳、黄亮，清华大学出版社）

《计算机图形学基础教程》（孙家广，清华大学出版社）

《ACMICPC 程序设计系列—计算几何》（哈尔滨工业大学出版社）

编写：彭文文

校核：曹振海

8.5.1 基本原理

求一个平面上的一些点中的最远点对，那么最常见的就是求出这些点的凸包，然后利用旋转卡壳求出最远点对。具体步骤详见 1.2 节凸包的旋转卡壳。这里我们结合两道题讲解 poj2187 最远点对问题：这个问题规模比较大，体现出了旋转卡壳的优势。凸包上的点依次与对应边产生的距离成单峰函数，我们首先固定一条边，然后找到第一个 k ，使得 (i, j, k) 的叉积 $< (i, j, k+1)$ 的叉积，这时更新最大值，然后枚举下一个边，这个复杂度是 $O(n)$ 的，因为我们逆时针枚举边的时候， k 也是逆时针变化的。

8.5.2 解题思路

对于最远点对问题一般都比较直接，明确问题之后直接用旋转卡壳算法去解决即可

8.5.3 模板代码

```

//用旋转卡壳求最远点对
#include<iostream>
#include<algorithm>
using namespace std;
struct point{
    int x , y;
}p[50005];
int top , stack[50005];    // 凸包的点存在于 stack[]中
inline double dis(const point &a , const point &b){
    return (a.x - b.x)*(a.x - b.x)+(a.y - b.y)*(a.y - b.y);
}
inline int max(int a , int b){
    return a > b ? a : b;
}
inline int xmult(const point &p1 , const point &p2 , const point &p0){
    //计算叉乘--线段旋转方向和对应的四边形的面积--返回(p1-p0)*(p2-p0)叉积
    //if 叉积为正--p0p1 在 p0p2 的顺时针方向; if(x==0)共线
    return (p1.x-p0.x)*(p2.y-p0.y) - (p1.y-p0.y)*(p2.x-p0.x);
}
int cmp(const void * a , const void * b){ //逆时针排序 返回正数要交换
    struct point *p1 = (struct point *)a;
    struct point *p2 = (struct point *)b;
    int ans = xmult(*p1 , *p2 , p[0]); //向量叉乘
    if(ans < 0)    //p0p1 线段在 p0p2 线段的上方，需要交换
        return 1;
    else if(ans == 0 && ( (*p1).x >= (*p2).x))    //斜率相等时，距离近的点在先
        return 1;
}

```

```

else
    return -1;
}
void graham(int n){ //形成凸包
    qsort(p+1 , n-1 , sizeof(point) , cmp);
    int i;
    stack[0] = 0 , stack[1] = 1;
    top = 1;
    for(i = 2 ; i < n ; ++i)
    {
        while(top > 0 && xmult( p[stack[top]] , p[i] , p[stack[top-1]]) <= 0)
            top--; //顺时针方向--删除栈顶元素
        stack[++top] = i; //新元素入栈
    }
    int temp = top;
    for(i = n-2 ; i >= 0 ; --i)
    {
        while(top > temp && xmult(p[stack[top]] , p[i] , p[stack[top-1]]) <= 0)
            top--;
        stack[++top] = i; //新元素入栈
    }
}
int rotating_calipers(){ //卡壳
    int i , q=1;
    int ans = 0;
    stack[top]=0;
    for(i = 0 ; i < top ; i++){
        while( xmult( p[stack[i+1]] , p[stack[q+1]] , p[stack[i]] ) >
            xmult( p[stack[i+1]] , p[stack[q]] , p[stack[i]] ) )
            q = (q+1)%(top);
        ans = max(ans , max( dis(p[stack[i]] , p[stack[q]]) , dis(p[stack[i+1]] ,
            p[stack[q+1]])));
    }
    return ans;
}
int main(){
    int i , n , leftdown;
    while(scanf("%d",&n) != EOF){
        leftdown = 0;
        for(i = 0 ; i < n ; ++i){
            scanf("%d %d",&p[i].x,&p[i].y);
            if(p[i].y < p[leftdown].y || (p[i].y == p[leftdown].y && p[i].x <
                p[leftdown].x)) //找到最左下角的点
                leftdown = i;
        }
        swap(p[0] , p[leftdown]);
        graham(n);
        printf("%d\n",rotating_calipers());
    }
    return 0;
}

```

8.5.4 经典题目

1. 题目出处/来源

POJ 3384 Feng Shui

2. 题目描述

半平面交+最远点对，用两个圆覆盖一个多边形，问最多能覆盖多边形的面积

3. 分析:

用半平面交将多边形的每条边一起向“内”推进 R , 得到新的多边形, 然后求多边形的最远两点

4. 代码:

```
#include <cstdio>
#include <cstring>
#include <cmath>
#include <iostream>
using namespace std;
#define EPS 1e-10
const int MAXN = 300;
struct Point{
    double x,y;
    Point(){}
    Point(double _x,double _y):x(_x),y(_y){}
    void input(){
        scanf("%lf%lf",&x,&y);
    }
};
Point points[MAXN],p[MAXN],q[MAXN];
int n;
double r;
int cCnt,curCnt;
void getline(Point x,Point y,double &a,double &b,double &c){
    a = y.y - x.y;
    b = x.x - y.x;
    c = y.x * x.y - x.x * y.y;
}
inline void initial(){
    for(int i = 1; i <= n; ++i)p[i] = points[i];
    p[n+1] = p[1];
    p[0] = p[n];
    cCnt = n;
}
inline Point intersect(Point x,Point y,double a,double b,double c){
    double u = fabs(a * x.x + b * x.y + c);
    double v = fabs(a * y.x + b * y.y + c);
    return Point( (x.x * v + y.x * u) / (u + v) , (x.y * v + y.y * u) / (u + v) );
}
inline void cut(double a,double b ,double c){
    curCnt = 0;
    for(int i = 1; i <= cCnt; ++i){
        if(a*p[i].x + b*p[i].y + c >= 0)q[++curCnt] = p[i];
        else {
            if(a*p[i-1].x + b*p[i-1].y + c > 0){
                q[++curCnt] = intersect(p[i],p[i-1],a,b,c);
            }
            if(a*p[i+1].x + b*p[i+1].y + c > 0){
                q[++curCnt] = intersect(p[i],p[i+1],a,b,c);
            }
        }
    }
    for(int i = 1; i <= curCnt; ++i)p[i] = q[i];
    p[curCnt+1] = q[1];p[0] = p[curCnt];
    cCnt = curCnt;
}
inline double pdis(Point a,Point b){
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}
```

```

}
int main(){
    scanf("%d%lf",&n,&r);
    for(int i = 1; i <= n; ++i)points[i].input();
    points[n+1] = points[1];
    initial();
    for(int i = 1; i <= n; ++i){
        Point ta, tb, tt;
        tt.x = points[i+1].y - points[i].y;
        tt.y = points[i].x - points[i+1].x;
        double k = r / sqrt(tt.x * tt.x + tt.y * tt.y);
        tt.x = tt.x * k;
        tt.y = tt.y * k;
        ta.x = points[i].x + tt.x;
        ta.y = points[i].y + tt.y;
        tb.x = points[i+1].x + tt.x;
        tb.y = points[i+1].y + tt.y;
        double a,b,c;
        getline(ta,tb,a,b,c);
        cut(a,b,c);
    }
    int ansx = 0,ansy = 0;
    double res = 0;
    for(int i = 1; i <= cCnt; ++i){
        for(int j = i + 1; j <= cCnt; ++j){
            double tmp = pdis(p[i],p[j]);
            if(tmp > res){
                res = tmp;
                ansx = i;
                ansy = j;
            }
        }
    }
    printf("%.4lf %.4lf %.4lf %.4lf/n",p[ansx].x,p[ansx].y,p[ansy].x,p[ansy].y);
    return 0;
}

```

8.6 模拟退火(Simulated Annealing)

参考文献:

模拟退火-百度百科

大白话解析模拟退火算法:<http://www.cnblogs.com/heaad/archive/2010/12/20/1911614.html>

扩展阅读:

启发式算法-百度百科

数学中国网-模拟退火算法

编写: 曹振海

校核: 彭文文

8.6.1 基本原理

“模拟退火”算法是源于对热力学中退火过程的模拟,在某一给定初温下,通过缓慢下降温度参数,使算法能够在多项式时间内给出一个近似最优解。

一. 爬山算法 (Hill Climbing)

介绍模拟退火前,先介绍爬山算法。爬山算法是一种简单的贪心搜索算法,该算法每次从当前解的临近解空间中选择一个最优解作为当前解,直到达到一个局部最优解。

爬山算法实现很简单,其主要缺点是会陷入局部最优解,而不一定能搜索到全局最优解。如图 1 所示:假设 C 点为当前解,爬山算法搜索到 A 点这个局部最优解就会停止搜

索，因为在 A 点无论向那个方向小幅度移动都不能得到更优的解。

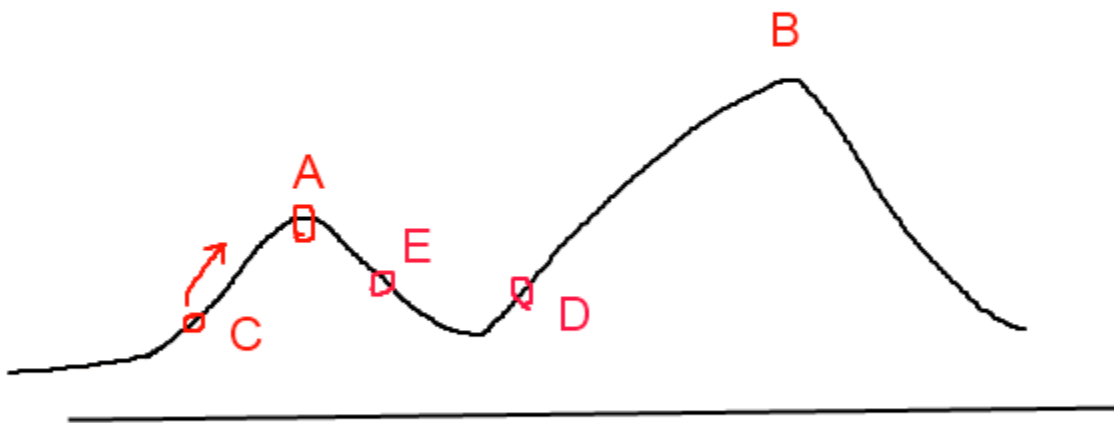


图 1

二. 模拟退火(SA, Simulated Annealing)思想

爬山法是完完全全的贪心法，每次都鼠目寸光的选择一个当前最优解，因此只能搜索到局部的最优值。模拟退火其实也是一种贪心算法，但是它的搜索过程引入了随机因素。模拟退火算法以一定的概率来接受一个比当前解要差的解，因此有可能会跳出这个局部的最优解，达到全局的最优解。以图 1 为例，模拟退火算法在搜索到局部最优解 A 后，会以一定的概率接受到 E 的移动。也许经过几次这样的不是局部最优的移动后会到达 D 点，于是就跳出了局部最大值 A。

模拟退火算法描述：

若 $J(Y(i+1)) \geq J(Y(i))$ (即移动后得到更优解)，则总是接受该移动

若 $J(Y(i+1)) < J(Y(i))$ (即移动后的解比当前解要差)，则以一定的概率接受移动，而且这个概率随着时间推移逐渐降低（逐渐降低才能趋向稳定）

这里的“一定的概率”的计算参考了金属冶炼的退火过程，这也是模拟退火算法名称的由来。

根据热力学的原理，在温度为 T 时，出现能量差为 dE 的降温的概率为 $P(dE)$ ，表示为：

$$P(dE) = \exp(dE/(kT))$$

其中 k 是一个常数， \exp 表示自然指数，且 $dE < 0$ 。这条公式说白了就是：温度越高，出现一次能量差为 dE 的降温的概率就越大；温度越低，则出现降温的概率就越小。又由于 dE 总是小于 0（否则就不叫退火了），因此 $dE/kT < 0$ ，所以 $P(dE)$ 的函数取值范围是 $(0,1)$ 。

随着温度 T 的降低， $P(dE)$ 会逐渐降低。

我们将一次向较差解的移动看做一次温度跳变过程，我们以概率 $P(dE)$ 来接受这样的移动。

从上面的描述中我们可以看到，模拟退火算法最难实现的地方就是一个概率性接受当前并不是最优的解，这在程序当中实现起来是很难的，我们解决这个问题的办法是用解替换一个解，初始解集随机的分布在所要求的范围内，这样就能从多个方向向各自范围内的最优解移动，理想情况下是可以找到一个接近最优解的解的，但是并不能保证能找到最优解。这也是启发式算法都有的特点，程序运行不稳定(因为要随机产生一些数据)而且效率比较低，一直要达到精度才能结束。但是对于一些精度要求不是很高的题，用模拟退火解决起来会变得很简单。对于模拟退火算法看描述是很难看懂的，看代码，很快就能理解了。

8.6.2 解题思路

模拟退火算法一般都要求计算最优解，而且会有一定的范围，当我们看到一道计算几何题要在一定范围的平面中找一个最优点的时候，模拟退火都是一个不错的选择，首先产生初始解集，确定初始步长（第一次产生新解集所能走的最大步长），确定步长的减小速率，确定减小的精度（一般取题中所给的精度的 $1/100$ ）

8.6.3 模板代码

```

模拟退火是一种算法思想，下面给出其一般情况下的伪代码
/*
 * J(y): 在状态 y 时的评价函数值
 * Y(i): 表示当前状态
 * Y(i+1): 表示新的状态
 * r: 用于控制降温的快慢
 * T: 系统的温度，系统初始应该要处于一个高温的状态
 * T_min : 温度的下限，若温度 T 达到 T_min, 则停止搜索, T_min 一般在题中表示精度要求的 1/100
 */
while( T > T_min )
{
    dE = J( Y(i+1) ) - J( Y(i) );

    if( dE >= 0 ) //表达移动后得到更优解，则总是接受移动
        Y(i+1) = Y(i); //接受从 Y(i)到 Y(i+1)的移动
    else
    {
        // 函数 exp( dE/T )的取值范围是(0,1) , dE/T 越大, 则 exp( dE/T )也
        if( exp( dE/T ) > random( 0 , 1 ) )
            Y(i+1) = Y(i); //接受从 Y(i)到 Y(i+1)的移动
    }
    T = r * T; //降温退火 , 0<r<1 。 r 越大, 降温越慢; r 越小, 降温越快
}
/*
 * 若 r 过大, 则搜索到全局最优解的可能会较高, 但搜索的过程也就较长。若 r
 * 过小, 则搜索的过程会很快, 但最终可能会达到一个局部最优值
 */
i++;
}

```

8.6.4 经典题目

1. 题目出处/来源

POJ-1379 Run Away

2. 题目描述

在一个 $X*Y$ 的平面内有一些陷阱，要求在平面内找出来一个安全点，满足这点距离所有的陷阱的最短距离最大，要求输出点的坐标，精确到一位小数

3. 分析

这道题有一个很直接的算法，但是算法很麻烦也很长，我们看这道题要求的精度很低，所以可以用模拟退火来做，把解集定义为 20 个，每次用一个随机产生 20 个新解，逐渐减小步长，步长的减小速率可以定义为 0.9 初始步长为 X, Y 的最大值

4. 代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<cstdio>
#include<cmath>
#include<cstring>
#include<string>
#include<ctime>
#include<cstdlib>
#define inf 1e50
using namespace std;
const int NUM=20;//解集数量定义为 20 个
const int RAD=1000;//产生的随机数精度定义为要求精度的 1/100
struct point
{
    double x,y,val;
    point(){}
    point(double _x,double _y)
    {
        x=_x;
        y=_y;
    }
};
point p[10001],May[NUM],e1,e2;//May 存储解集
int n;
double X,Y;
double dis(point a,point b)
{
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}
double judge(point t)//评价函数，得到点 t 的评价值 val
{
    double len=inf;
    for(int i=0;i<n;i++)
        len=min(len,dis(t,p[i]));
    return len;
}
double Rand(){return rand()%(RAD+1)/(1.0*RAD);}//随机产生 0-1 的浮点数
point Rand_point(point a,point b)//在 a,b 框定的四边形内随机生成点
{
    double xx=a.x+(b.x-a.x)*Rand();
    double yy=a.y+(b.y-a.y)*Rand();
    point tmp=point(xx,yy);
    tmp.val=judge(tmp);
    return tmp;
}
void solve(double D)
{

```

```

May[0]=point(0,0);
May[1]=point(X,Y);
May[2]=point(0,Y);
May[3]=point(X,0);
//4个顶点的可能行较大,所以特殊构造
for(int i=4;i<NUM;i++)
May[i]=Rand_point(May[0],May[1]); //步骤 2
while(D>0.01) //步骤 3 当精度达到要求的 1/100 时结束
{
    for(int i=0;i<NUM;i++)
        for(int j=0;j<NUM;j++)
        {
            point tmp=Rand_point(point(max(0.0,May[i].x-D),max(0.0,May[i].y-
D)),point(min(X,May[i].x+D),min(Y,May[i].y+D)));
            if(tmp.val>May[i].val)
            {
                May[i]=tmp;
            }
        }
    D*=0.9; //步长定义为 0.9
}
point ans;
ans.val=0;
for(int i=0;i<NUM;i++)
if(May[i].val>ans.val)
ans=May[i];
printf("The safest point is (%.1f, %.1f).\n",ans.x,ans.y);
}
int main()
{
    srand(time(0));
    e2=point(0,0);
    int Case;
    int i;
    scanf("%d",&Case);
    while(Case--)
    {
        scanf("%lf%lf%d",&X,&Y,&n);
        for(i=0;i<n;i++)
        {
            scanf("%lf%lf",&p[i].x,&p[i].y);
        }
        solve(max(Y,X)); //初始步长定义为 X,Y 的最大值
    }
    return 0;
}

```

5. 思考与扩展: 可以借鉴北大培训教材中做法。

8.6.5 扩展变型

GJK 算法 (米可夫斯基和, 碰撞检测)

计算几何+二分

矩形并

计算几何+simpson 积分