Chapter 1

计算几何

1.1 简单

1.1.1 定义

```
//判断浮点数的符号x
int sgn(double x)
{
    if(fabs(x) < eps) return 0;
    else if(x < 0) return -1;
    else return 1;
}

struct Point { 点的定义:
    double x, y;

    Point() {}
    Point(double _x, double _y) : x(_x), y(_y) {}
    bool operator == (const Point& rhs) const {
        return sgn(x - rhs.x) == 0 && sgn(y - rhs.y) == 0;
    }
    bool operator < (const Point& rhs) const {
        return sgn(x - rhs.x) == 0 ? sgn(y - rhs.y) < 0 : x < rhs.x;
    }
    Point operator - (const Point& rhs) const {
        return Point(x - rhs.x, y - rhs.y);
    }
    Point operator + (const Point& rhs) const {
        return Point(x + rhs.x, y + rhs.y);
    }
    Point operator * (const double d) const {
        return Point(x * d, y * d);
    }
    Point operator / (const double d) const {
        return Point(x / d, y / d);
    }
    double dot(const Point& rhs) const { //点积
        return x * rhs.x + y * rhs.y;
    }
    double cross(const Point& rhs) const { //叉积
        return x * rhs.y - y * rhs.x;
    }
    double dis(const Point& rhs) const { //两点距离
        return hypot(x - rhs.x, y - rhs.y);
    }
    double len() const { //长度
        return hypot(x, y);
    }
    //计算 pa 和 pb 的夹角，即从这个点看 a,b 所成的夹角返回弧度值且是正值,,
```

```
    double rad(const Point& a, const Point& b) const {
        Point p = *this;
        return fabs(atan2(fabs((a - p).cross(b - p)), (a - p).dot(b - p)));
    }
    Point to_vector(double r) const { //返回长度为 r 的向量
        double l = len();
        if(!sgn(l)) return *this;
        r /= l;
        return Point(x * r, y * r);
    }
    Point rotleft(){ //逆时针旋转 90 度
        return Point(-y, x);
    }
    Point rotright(){ //顺时针旋转 90 度
        return Point(y, -x);
    }
    Point rotate(const Point& rhs, const double angle){ //绕着 rhs 点逆时针旋转 angle 度
        Point v = (*this) - rhs;
        double c = cos(angle), s = sin(angle);
        return Point(rhs.x + v.x * c - v.y * s, rhs.y + v.x * s + v.y * c);
    }
};
```

```
struct Line{ 线段的定义:
    Point s, e;

    Line() {}
    Line(const Point& _s, const Point& _e) : s(_s), e(_e) {}
    bool operator == (const Line& rhs) const {
        return s == rhs.s && e == rhs.e;
    }
    //根据一个点和倾斜角 angle 确定直线,0<= angle < pi
    Line(const Point& rhs, const double angle) const {
        s = p;
        if(sgn(angle - pi / 2) == 0){ //竖直直线
            e = (s + Point(0, 1));
        }else {
            e = (s + Point(1, tan(angle)));
        }
    }
    double length() const{ //线段长度
        return s.dis(e);
    }
    double angle(){ //返回直线的倾斜角0<= angle < pi
        double k = atan2(e.y - s.y, e.x - s.x);
        if(sgn(k) < 0) k += pi;
        if(sgn(k - pi) == 0) k -= pi;
        return k;
    }
    //点和直线的关系: 1 在左侧, 2 在右侧, 3 在直线上
    int relation(const Point& rhs) const{
        int c = sgn((rhs - s).cross(e - s));
        if(c < 0) return 1;
        else if(c > 0) return 2;
        else return 3;
    }
    bool point_on_seg(const Point& rhs) const { //点在线段上的判断
        return sgn((rhs - s).cross(e - s)) == 0 && sgn((rhs - s).dot(rhs - e)) <= 0;
    }
    bool parallel(const Line v) const { //判断直线平行
        return sgn((e - s).cross(v.e - v.s)) == 0;
    }
    double point_to_line_dis(const Point& rhs) const{ //点到直线的距离
        return fabs((rhs - s).cross(e - s)) / length();
    }
```

```
    double point_to_seg_dis(const Point& rhs) const { //点到线段的距离
        if(sgn((rhs - s).dot(e - s)) < 0 || sgn((rhs - e).dot(s - e)) < 0){
            return min(rhs.dis(s), rhs.dis(e));
        }else return point_to_line_dis(rhs);
    }
};
```

1.1.2 顺时针输出所有顶点

将平面所有点都用上，构成一个多边形，顺时针 (或逆时针) 输出点的顺序。

多边形有可能是凸的也有可能是凹的。先找到最左下角的点 (x 值优先), 然后对其余点以最左下角为基点极角排序。除去最左侧的一系列和 *point*[0] 共线的点，其余的点的顺序即是逆时针的点的顺序。

```
const int MAX_N=2010;
const double eps=1e-10;

int T,n;
int ans[MAX_N];

Point point[MAX_N];

inline bool cmp(Point a,Point b)
{//按照最左下角极角排序
    double res = (a - point[0]).cross(b - point[0]);
    if(res != 0.0) return res > 0;//res > 0 说明 b 在左侧，此时 a 的极角较小
    else return a.dis(point[0]) < b.dis(point[0]); //共线时按照距离从小到大排序
}

inline void solve()
{
    //找到最左下角顶点优先, x 最小，其次 y 最小
    int k=0;
    for(int i=0;i<n;i++){
        if(point[i].x<point[k].x||(point[i].x==point[k].x&&point[i].y<point[k].y)){
            k=i;
        }
    }
    swap(point[0],point[k]);
    sort(point+1,point+n,cmp); //排序从下标 1 开始，因为 point[0] 就是起点
    int end=n-2;
    //找到最后的顶点 end 使得, point[end],point[end+1] 和 point[0] 不共线
    while(end>0){
        double res=(point[end]-point[0]).cross(point[end+1]-point[0]);
        if(res!=0.0) break;
        end--;
    }
    //从 point[end+1] 到 point[n-1] 都是和 point[0] 共线的点
    //即 point[end+1]..point[n-1],point[0] 都在一条直线上
    //因为极角排序时是按照到 point[0] 距离从小到大排序,
    //所以这些点的逆时针顶点应该是按照距离从大到小考虑
    //point[end+1] 实际上应该是第 n-1 个点, point[n-1] 实际上应该时第 end+1 个点
    for(int i=end+1;i<n;i++){
        ans[i]=point[n+end-i].index;
    }
    for(int i=0;i<=end;i++){
        ans[i]=point[i].index;
    }
    for(int i=0;i<n;i++){
        printf("%d%c",ans[i],i==n-1?'\n':' ');
    }
}

int main()
{
```

```
    scanf("%d",&T);
    while(T--){
        scanf("%d",&n);
        for(int i=0;i<n;i++){
            scanf("%lf%lf",&point[i].x,&point[i].y);
            point[i].index=i;
        }
        solve();
    }
    return 0;
}
```

1.1.3 求半径 R 圆覆盖最多点数及由圆上两点和半径求圆心

方法 1：
最优的情况一定是有两个点在圆弧上。先枚举两个点，计算两点在圆弧上的单位圆（一般会有两个）。但是可以统一取一个方向的（也就是 AB 取一个然后 BA 取另外一个）. 然后枚举所有点, 计算在这个单位圆内的点的个数。这样做的时间复杂度是 $O(n^3)$. 、
方法 2：
对每个点以 R 为半径画圆，对 N 个圆两两求交。这一步 $O(N^2)$。问题转化为求被覆盖次数最多的弧。因为如果最优圆覆盖 A 点那么最优圆一定在以 A 点为圆心的圆弧上。那么圆弧倍覆盖多少次也就意味着以这条圆弧为上任意一点为圆心花园能覆盖多少点。对每一个圆，求其上的每段弧重叠次数。
假如 A 圆与 B 圆相交。A 上 $[PI/3, PI/2]$ 的区间被 B 覆盖 (PI 为圆周率)。那么对于 A 圆，我们在 $PI/3$ 处做一个 +1 标记，在 $PI/2$ 处做一个 -1 标记。
对于 $[PI*5/3, PI*7/3]$ 这样横跨 0 点的区间只要在 0 点处拆成两段即可。将一个圆上的所有标记排序，从头开始扫描。初始 $total = 0$，碰到 +1 标记给 $total++$，碰到 -1 标记 $total--$。扫描过程中 $total$ 的最大值就是圆上被覆盖最多的弧。求所有圆的 $total$ 的最大值就是答案。极角排序需要 $2*n*\log n$，总复杂度 $O(N^2 * \log N)$

```
const int MAX_N=310;
const double eps=1e-6;
const double R=1.0;//定义覆盖圆半径为R

int T,n;

struct Point{
    double x,y;
}point[MAX_N];

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

inline Point GetCenter(Point a,Point b)
{//获取 a,b 两点在圆周上的单元圆圆心单位圆圆心有两个,
    struct Point mid,res;
    mid.x=(a.x+b.x)/2,mid.y=(a.y+b.y)/2; //mid 是 a,b 中点坐标
    double angle=atan2(b.y-a.y,b.x-a.x);//angle 是直线 ab 的倾斜角
    double tmp=dis(a,b)/2;//tmp 是线段 ab 长度的一半
    double d=sqrt(1.0-tmp*tmp);//d 是 ab 中点到圆心的距离
    res.x=mid.x-d*sin(angle);//res 是直线 ab 左边的那个圆心
    res.y=mid.y+d*cos(angle);
    //下面的 res 是直线 ab 右边的那个圆心
    //res.x=mid.x+d*sin(angle);
    //res.y=mid.y-d*cos(angle);
    return res;
}

int main()
{
    scanf("%d",&T);
    while(T--){
```

```
        scanf("%d",&n);
        for(int i=0;i<n;i++){
            scanf("%lf%lf",&point[i].x,&point[i].y);
        }
        int ans=1;//初始化至少能覆盖一个点!!!
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                if(i==j||dis(point[i],point[j])-2*R>eps) continue;
                int cnt=0;
                struct Point center=GetCenter(point[i],point[j]);
                for(int k=0;k<n;k++){
                    if(dis(point[k],center)-R<=eps) cnt++;
                }
                ans=max(ans,cnt);
            }
        }
        printf("%d\n",ans);
    }
    return 0;
}
```

```
const int MAX_N=310;
const double PI=acos(-1.0);
const double R=1.0;//定义覆盖圆半径为R

int T,n,total;

struct Point{
    double x,y;
}point[MAX_N];

struct Angle{
    double data;
    int is;
    bool operator < (const Angle& rhs) const {
        return data<rhs.data;
    }
}angle[MAX_N*2];

inline double Dis(Point a,Point b)//计算线段 ab 的长度
{
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}

inline double God(Point a,Point b)//计算向量 ab 的极角
{
    double res=atan(fabs((b.y-a.y)/(b.x-a.x)));
    if(b.y<a.y){
        if(b.x<a.x) res+=PI;
        else res=2*PI-res;
    }else {
        if(b.x<a.x) res=PI-res;
    }
    return res;
}

void solve()
{
    int res=1;
    for(int i=0;i<n;i++){
        total=0;
        for(int j=0;j<n;j++){
            if(i==j) continue;
            double dist=Dis(point[i],point[j]);
            if(dist>2*R) continue;
```

```
            double base=God(point[i],point[j]);
            double extra=acos(dist/2.0); //计算差角
            angle[total].data=base+extra;
            angle[total++].is=-1;
            angle[total].data=base-extra;
            angle[total++].is=1;
        }
        if(total<=res) continue;
        sort(angle,angle+total);
        int tmp=1;
        for(int j=0;j<total;j++){
            tmp+=angle[j].is;
            res=max(res,tmp);
        }
    }
    printf("%d\n",res);
}

int main()
{
    scanf("%d",&T);
    while(T--){
        scanf("%d",&n);
        for(int i=0;i<n;i++){
            scanf("%lf%lf",&point[i].x,&point[i].y);
        }
        solve();
    }
    return 0;
}
```

1.1.4 计算两圆公共部分面积

处理两圆相交情况
分析交点及两圆圆心组成的四边形和交点与两圆圆心组成的两个扇形
假设 $c1$ 圆扇形角为 $2*\alpha$ ，那么四边形的面积 s 等于：$0.5*dis*c1.r*sin(\alpha)*2 = dis*c1.r*sin(\alpha)$;
记 $c1, c2$ 扇形面积分别为 $s1, s2$. 那么 $s1 = (2*\alpha/(2*PI))*PI*c1.r*c1.r = \alpha*c1.r*c1.r$;
而且 α 可以用余弦定理计算出来:$cos(\alpha) = (d^2 + c1.r^2 - c2.r^2)/(2*d*c1.r)$;
同理也可以计算出 $s2$, 所以相交部分面积为:$s1 + s2 - s$.

```
const double eps=1e-8;
const double PI=acos(-1.0);

struct Circle{
    double x,y,r;
};

double GetDis(Circle c1,Circle c2)//计算圆心距
{
    return sqrt((c1.x-c2.x)*(c1.x-c2.x)+(c1.y-c2.y)*(c1.y-c2.y));
}

double CalcArea(Circle c1,Circle c2)
{
    double dis=GetDis(c1,c2);
    if(dis-(c1.r+c2.r)>=eps){ //两圆相离
        return 0;
    }else if(c1.r>=c2.r+dis){  //c1 包含c2
        return PI*c2.r*c2.r;
    }else if(c2.r>=c1.r+dis){ //c2 包含c1
        return PI*c1.r*c1.r;
    }
    double angle1=acos((dis*dis+c1.r*c1.r-c2.r*c2.r)/(2*dis*c1.r));
    double s1=angle1*c1.r*c1.r;     //c1 圆扇形面积
```

```
25
26      double s=dis*c1.r*sin(angle1);   //四边形面积
27
28      double angle2=acos((dis*dis+c2.r*c2.r-c1.r*c1.r)/(2*dis*c2.r)); //c2 圆扇形面积
29      double s2=angle2*c2.r*c2.r;
30
31      return s1+s2-s;
32  }
33
34  int main()
35  {
36      Circle a,b;
37      while(cin>>a.x>>a.y>>a.r>>b.x>>b.y>>b.r){
38          double ans=CalcArea(a,b);
39          printf("%.3lf\n",ans);
40      }
41      return 0;
42  }
```

1.1.5 判断两线段是否相交

已知 n 条木棍的起点和终点坐标，问第 i 条木棍和第 j 条木棍是否相连？
当两条木棍之间有公共点时，就认为他们时相连的。通过相连的木棍间接的连在一起的两根木棍也认为时相连的。

木棍就是二维平面上的线段，只要能判断线段是否相交，那么建图后可以通过 Floyd 算法或者并查集进行连接性判断。如何判断两条线段是否相交呢？首先会想到计算两条直线的交点，然后判断交点是否在线段上。那么两条直线的交点如何求得呢？虽然可以把直线表示成方程，通过建立方程组求解。但在几何问题中，运用向量的内积和外积进行计算是非常方便的。对于二维向量 $p1 = (x1, y1)$ 和 $p2 = (x2, y2)$, 我们定义内积 $p1 \cdot p2 = x1*x2+y1*y2$，外积 $p1*p2 = x1*y2-x2*y1$. 要判断点 q 是否在线段 $p1-p2$ 上，只要先利用外积根据是否有 $(p1-q)X(p2-q) == 0$ 来判断点 q 是否在直线 $p1-p2$ 上，再利用内积根据是否有 $(p1-q)\cdot(p2-q) <= 0$ 来判断点 q 是否落在 $p1-p2$ 之间。而要求两直线的交点，通过变量 t 将直线 $p1-p2$ 上的点表示为 $p1+t(p2-p1)$，交点又在直线 $q1-q2$ 上，所以有: $(q2-q1)X(p1+(p2-p1)-q1) == 0$ 于是可以利用下式求得 t 的值:

$$p1+(p2-p1)*((q2-q1)X(q1-p1)/(q2-q1)X(p2-p1))$$

但是使用这个方法还要注意边界情况。也就是平行的线段也可能有公共点。这时我们可以选择通过检查端点是否在另一条线段上来判断。

```
1   double EPS=1e-10;
2   const int MAX_N=20;
3
4   int n;
5   bool connected[MAX_N][MAX_N];
6   int pre[MAX_N];
7
8   //考虑误差的加法运算
9   double add(double a,double b)
10  {
11      if(abs(a+b)<EPS*(abs(a)+abs(b))) return 0;
12      return a+b;
13  }
14
15  struct Point st[MAX_N],ed[MAX_N];//st[i],ed[i] 分别是第 i 条线段的起点和终点
16
17  //判断点 q 是否在线段 p1-p2 上
18  bool on_seg(Point p1,Point p2,Point q)
19  {
20      return (p1-q).cross(p2-q)==0&&(p1-q).dot(p2-q)<=0;
21  }
22
23  //计算直线 p1-p2 与直线 q1-q2 的交点
24  Point intersection(Point p1,Point p2,Point q1,Point q2)
```

```
{
    return p1+(p2-p1)*((q2-q1).cross(q1-p1)/(q2-q1).cross(p2-p1));
}

//Floyd_Warshall 算法判断任意两条木棍是否相连
void Floyd_Warshall()
{
    for(int k=0;k<n;k++){
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                connected[i][j] |= connected[i][k]&&connected[k][j];
            }
        }
    }
}

int find(int x)
{
    return pre[x]==x?x:pre[x]=find(pre[x]);
}

//并查集判断任意两条木棍是否相连
void UnionFindSet()
{
    for(int i=0;i<n;i++) pre[i]=i;
    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            if(connected[i][j]&&find(i)!=find(j)){
                pre[j]=i;
            }
        }
    }
}

void solve()
{
    for(int i=0;i<n;i++){
        connected[i][i]=true;
        for(int j=0;j<i;j++){//判断木棍 i 和 j 是否有公共点
            if((st[i]-ed[i]).cross(st[j]-ed[j])==0){//木棍平行时
                connected[i][j]=connected[j][i]=on_seg(st[i],ed[i],st[j])
                                                || on_seg(st[i],ed[i],ed[j])
                                                || on_seg(st[j],ed[j],st[i])
                                                || on_seg(st[j],ed[j],ed[i]);
            } else {// 不平行时
                Point inter=intersection(st[i],ed[i],st[j],ed[j]);
                connected[i][j]=connected[j][i] =
                    on_seg(st[i],ed[i],inter) && on_seg(st[j],ed[j],inter);
            }
        }
    }
    Floyd_Warshall();
    UnionFindSet();
}

int main()
{
    while(~scanf("%d",&n)&&n){
        for(int i=0;i<n;i++){
            scanf("%lf%lf%lf%lf",&st[i].x,&st[i].y,&ed[i].x,&ed[i].y);
        }
        solve();
        int a,b;
        while(~scanf("%d%d",&a,&b)&&(a||b)){
            //if(connected[a-1][b-1]) printf("CONNECTED\n");
```

```
            if(find(a-1)==find(b-1)) printf("CONNECTED\n");
            else printf("NOT CONNECTED\n");
        }
    }
    return 0;
}
```

1.1.6 判断点和多边形的关系

判断多边形和点的关系: 3 点在顶点上,2 在边上,1 在内部,0 在外部

```
Point vertex[MAX_N]; //用来存储多边形的顶点
Lint line[MAX_N]; //用来存储多边形的边
int relation_with_point(Point q, int n)
{
    for(int i = 0; i < n; i++){
        if(q == vertex[i]) return 3;
    }
    vertex[n] = vertex[0];
    for(int i = 0; i < n; ++i){
        line[i] = Line(vertex[i], vertex[i + 1]);
    }
    for(int i = 0; i < n; i++){
        if(line[i].point_on_seg(q)) return 2;
    }
    int cnt = 0;
    for(int i = 0; i < n; i++){
        int j = i + 1;
        int k = sgn((q - vertex[j]).cross(vertex[i] - vertex[j]));
        int u = sgn(vertex[i].y - q.y);
        int v = sgn(vertex[j].y - q.y);
        if(k > 0 && u < 0 && v >= 0) cnt++;
        if(k < 0 && v < 0 && u >= 0) cnt--;
    }
    return cnt != 0;
}

//点 p 在多边形顶点、边上、内部返回 true ，否则返回false
bool in_polygon(Point p)
{
  int i, j, c = 0;
  double testx = p.x, testy = p.y;
  for (i = 0, j = n - 1; i < n; j = i++) {
    if ( ((vertex[i].y > testy) != (vertex[j].y > testy)) &&
     (testx < (vertex[j].x - verte1[i].x) *
     (testy - vertex[i].y) / (vertex[j].y - vertex[i].y) + vertex[i].x) )
       c = !c;
  }
  return c;
}
```

1.1.7 计算多边形的最小宽度

多边形的最小宽度是指以旋转多边形以某个方式垂直下落，通过长度为 L 的狭缝，狭缝长度的最小值即是多边形的最小宽度。例如长宽为 4，5 的矩形的最小宽度为 4.
先将多边形求凸包，枚举这个凸包的每一条边，求出所有顶点距离这条边的距离的最大值，这就是这条边的多对应的高度，在所有边的高度中取最小值就是最小宽度 L.

```
inline bool cmp_x(const Point a, const Point b)
{
    if(a.x == b.x) return a.y < b.y;
    return a.x < b.x;
}
```

```
6
7  //求凸包
8  inline int Andrew()
9  {
10     sort(point,point + m, cmp_x);
11     int k = 0;
12     for(int i = 0; i < m; i++){
13         while(k > 1 && (vertex[k - 1] - vertex[k - 2]).cross
14         (point[i] - vertex[k - 1]) <= 0){
15             k --;
16         }
17         vertex[k++] = point[i];
18     }
19     int m = k;
20     for(int i = n - 2; i >= 0; i--){
21         while(k > m && (vertex[k -1] - vertex[k - 2]).cross
22         (point[i] - vertex[k - 1]) <= 0){
23             k --;
24         }
25         vertex[k++] = point[i];
26     }
27     if(k > 1) k--;
28     return k;
29 }
30
31 //计算 c 到 a—b 的距离
32 inline double GetHight(Point a, Point b, Point c)
33 {
34     double d1 = a.dis(b);
35     double d2 = (b - a).cross(c - a);
36     return fabs(d2 / d1);
37 }
38
39 //计算凸包边相对顶点的最远距离在所有距离中取最小值,
40 inline double GetMInWidth(int k)
41 { //k 是凸包顶点数目
42     vertex[k] = vertex[0];
43     double res = 1e6;
44     for(int i = 0; i < k; i++){
45         double tmp = -1.0;
46         for(int j = 0; j < k; j++){
47             tmp = max(tmp, GetHight(vertex[i], vertex[i + 1], vertex[j]));
48         }
49         res = min(res, tmp);
50     }
51     return res;
52 }
```

1.1.8 计算多边形的最长内直径

多边形的最长内直径是指多边形顶点间的最大距离，并且这个最大距离的顶点连线不超出多边形内部。如果多边形是一个凸包的话，那很好办，直接暴力扫一遍顶点间距离然后取最大即可。但是多边形可能是凹多边形啊。首先最长内直径的一个端点一定是多边形的顶点，另一端点可能在某条边上（这时一定存在多边形顶点在这条内直径上）或者另一端点也是多边形的顶点，所以可以枚举多边形的顶点 a 和 b，计算直线 ab 和多边形的所有交点 (步骤 1)，得到的所有交点都是在一条直线上的，但是将所有交点按照距离最左下角交点距离远近排序，相邻两个交点构成了一条小线段，并不是每条小线段都是在多边形的内部的，这时只需要判断线段的中点是否在多边形的内部或者是多边形边上的点即可，将所有在多边形内部的小线段打上标记，那么，连续最长的在多边形内部的小线段就构成了枚举顶点 ab 时得到的最大内直径。枚举所有顶点对，取最大即可。

步骤 1 的实现：考虑直线 ab 和多边形所有边的交点。如果边和直线 ab 平行，那么这条边的两个端点都要取，如果边不和直线 ab 平行，计算边所在直线和 ab 的交点，判断交点是否在边上即可。这样取完交点后需要去重。多边形的顶点需预先逆时针/顺时针排序。

```
//vertex1 是多边形的顶点， point1 是枚举顶点直线和多边形边的所有交点
int line_flag[MAX_N];

struct Line{
    Point st, ed;

    Line () {}
    Line (Point _st, Point _ed) : st(_st), ed(_ed) {}
}line[MAX_N], line1[MAX_N];

//判断点 q 是否在线段 p1—p2 上
inline bool on_seg(Point p1, Point p2, Point q)
{
    return (p1 - q).cross(p2 - q) == 0 && (p1 - q).dot(p2 - q) <= 0;
}

//判断 p 是否在直线 a 上
bool on_straight_line(Line a, Point p)
{
    Point st = a.st, ed = a.ed;
    if(p == st || p == ed) return true;
    if((p.x - st.x) * (ed.y - st.y) == (p.y - st.y) *(ed.x - st.x)) return true;
    else return false;
}

//获得直线 a 和直线 b 的交点
inline Point GetInter(Line a, Line b)
{
    Point p1, p2, q1, q2;
    p1 = a.st, p2 = a.ed;
    q1 = b.st, q2 = b.ed;
    return p1 + (p2 - p1) *((q2 - q1).cross(q1 - p1) / (q2 - q1).cross(p2 - p1));
}

//判断直线 a 和 b 是否平行
inline bool Parallel(Line a, Line b)
{
    Point p1, p2, q1, q2;
    p1 = a.st, p2 = a.ed;
    q1 = b.st, q2 = b.ed;
    if((p2.y - p1.y) * (q2.x - q1.x) == (p2.x - p1.x) * (q2.y - q1.y)) return true;
    else return false;
}

//获得直线 a 和多边形的所有交点
inline int GetPoint(Line a)
{
    int total = 0;
    for(int i = 0; i < n; i++){
        if(Parallel(a, line[i])) {
            if(on_straight_line(a, line[i].st)){
                point1[total++] = line[i].st;
                point1[total++] = line[i].ed;
            }
        }else {
            Point inter = GetInter(a, line[i]);
            if(on_seg(line[i].st, line[i].ed, inter)){
                point1[total++] = inter;
            }
        }
    }
    return total;
}

Point zuoxiajiao;
```

```
inline Point Getzuoxiajiao(int total)
{
    Point ans = point1[0];
    for(int i = 1; i < total; i++){
        if(point1[i].x < ans.x ||
        (point1[i].x == ans.x && point1[i].y < ans.y)){
            ans = point1[i];
        }
    }
    return ans;
}

//将所有交点按照距离最左下角点的距离从小到大排序
inline bool cmp_zuoxiajiao(const Point a, const Point b)
{
    return a.dis(zuoxiajiao) < b.dis(zuoxiajiao);
}

inline bool in_polygon(Point p)
{ //判断点 p 是否在多边形内部
  int i, j, c = 0;
  double testx = p.x, testy = p.y;
  for (i = 0, j = n - 1; i < n; j = i++) {
    if ( ((vertex1[i].y > testy) != (vertex1[j].y > testy)) &&
     (testx < (vertex1[j].x - vertex1[i].x) *
     (testy - vertex1[i].y) / (vertex1[j].y - vertex1[i].y) + vertex1[i].x) )
       c = !c;
  }
  return c;
}

inline double GetDiameter() //获得多边形的最长内直径
{
    double res = -1.0;
    for(int i = 0; i < n; i++){
        for(int j = i + 1; j < n; j++){
            int total = GetPoint(Line(vertex1[i], vertex1[j]));
            zuoxiajiao = Getzuoxiajiao(total);
            sort(point1, point1 + total, cmp_zuoxiajiao);
            int line_cnt = unique(point1, point1 + total) - point1 - 1; //去重
            for(int k = 0; k < line_cnt; k++){
                line1[k] = Line(point1[k], point1[k + 1]);
            }
            memset(line_flag, 0, sizeof(line_flag));
            for(int k = 0; k < line_cnt; k++){
                Point mid;
                mid.x = (line1[k].st.x + line1[k].ed.x) / 2;
                mid.y = (line1[k].st.y + line1[k].ed.y) / 2;
                if(in_polygon(mid)){
                    line_flag[k] = 1;
                }
            }
            double tmp = -1.0;
            int have = 0;
            Point prest, nowed;
            for(int k = 0; k < line_cnt; k++){
                if(line_flag[k]){
                    if(have == 0){
                        have = 1;
                        prest = line1[k].st;
                    }
                    nowed = line1[k].ed;
                }else {
                    if(have == 1){
                        tmp = max(tmp, nowed.dis(prest));
```

```
131                             have = 0;
132                         }
133                     }
134                 }
135                 if(have == 1){
136                     tmp = max(tmp, nowed.dis(prest));
137                 }
138                 res = max(res, tmp);
139             }
140         }
141         return res;
142 }
```

1.2 凸包

1.2.1 求凸包顶点:Graham 扫描法

向量的叉积:$A*B=|A|*|B|*sin\alpha$,α 是向量 A 和向量 B 之间的夹角
向量的点积 $A\cdot B=|A|*|B|*cos\alpha$
向量 A 和 B 的叉积小于 0，说明向量 B 在向量 A 右侧

```
const int MAX_N=1010;
const double INF=1e90;
const double eps=1e-10;
const double PI=acos(-1.0);

int T,n,cases=0;

Point point[MAX_N],vertex[MAX_N];

inline bool cmp_x(const Point a,const Point b)
{//优先升序，其次升序排序xy
    if(a.x==b.x) return a.y<b.y;
    return a.x<b.x;
}

//Graham 扫描法
inline int Andrew()
{
    sort(point,point+n,cmp_x);
    int k=0;
    //构造下凸包
    for(int i=0;i<n;i++){
        while(k>1&&(vertex[k-1]-vertex[k-2]).cross(point[i]-vertex[k-1])<=0){
            k--;
        }
        //因为存凸包的顶点是从 0 开始的，而最左下的点一定是凸包顶点，所以k>1
        //因为如果 k=1 仍然继续执行的话，k-2 就小于 0 了
        //当然如果凸包顶点是从 1 开始存储的话，那么这里就应该是k>2
        vertex[k++]=point[i];
    }
    //构造上凸包
    int m=k; //m 是下凸包顶点数目，且最后一个顶点是point[n-1]
    for(int i=n-2;i>=0;i--){      //注意是从 point[n-2] 开始，避免重复point[n-1]
        while(k>m&&(vertex[k-1]-vertex[k-2]).cross(point[i]-vertex[k-1])<=0){
            k--;
        }
        vertex[k++]=point[i];
    }
    if(k>1) k--; //point[0] 重复
    return k;
}
```

1.2.2 判断稳定凸包

给出一个凸包，判断凸包是否唯一确定 (稳定凸包)，即判断凸包每条边上是否存在至少三个点

```
inline bool JudgeStableConvexHull()
{
    if(n<6) {
        return false;
    }
    int total=Andrew();
    vertex[total]=vertex[0];
    /*将边上的点也存进凸包顶点里，在判断的时候只需要判断凸包顶点中是否一定存在相邻向量叉积为
     0 即可上面
     Andrew() 函数中需要将
```

```
    (vertex[k-1]-vertex[k-2]).cross(point[i]-vertex[k-1])<=0改为
     (vertex[k-1]-vertex[k-2]).cross(point[i]-vertex[k-1])<0

    for(int i=1;i<total;i++){
        if((vertex[i-1]-vertex[i]).cross(vertexi+1]-vertex[i])!=0
        &&(vertex[i]-vertex[i+1]).cross(vertex[i+2]-vertex[i+1])!=0){
            return false;
        }
    }
    return true;
    */

    for(int i=0;i<total;i++){ //遍历凸包边
        Point a=vertex[i],b=vertex[i+1];//该边端点是 a 和 b
        int cnt=0;
        for(int j=0;j<n;j++){ //检查该边上点的个数
            if((point[j]-a).cross(b-point[j])==0){// 在这条边上的点
                cnt++;
                if(cnt>=3) break;
            }
        }
        if(cnt<3){ //少于三个点
            return false;
        }
    }
    return true;
}
```

1.2.3 凸包直径

旋转卡壳求凸包直径 (最远点距离)

```
inline double GetMostFarDistance()
{
    int total=Andrew();
    if(total==2){ //处理凸包退化的情况只有两个顶点,
        return vertex[0].dis(vertex[1]);
    }

    int i=0,j=0;//在某个方向上的对踵点
    //求出 x 轴方向上的对踵点对
    for(int k=0;k<total;k++){
        if(!cmp_x(vertex[i],vertex[k])) i=k;
        if(cmp_x(vertex[j],vertex[k])) j=k;
    }
    double ans=0;
    int si=i,sj=j;
    while(i!=sj||j!=si){ //将方向逐步旋转 180 度
        ans=max(ans,vertex[i].dis(vertex[j]));
        //判断先转到边 i-(i+1) 的法线方向还是边 j-(j+1) 的法线方向
        int nexti=(i+1)%total,nextj=(j+1)%total;
        if((vertex[nexti]-vertex[i]).cross(vertex[nextj]-vertex[j])<0){
            i=nexti; //先转到边 i-(i+1) 的法线方向
        }else {
            j=nextj; //先转到边 j-(j+1) 的法线方向
        }
    }
    return ans;
}
```

1.2.4 凸包周长及面积

计算凸包面积

将凸包看成一个个以凸包最左下顶点为顶点的凸包边为对边的三角形。那么依次扫个条边，计算三角形面积累加即可。已知三角形三条边计算三角形面积，可用海伦 – 秦九韶公式

```
inline double GetConvexHullArea()
{
    int total=Andrew();
    if(total<=2){
        return 0.0;
    }
    double area=0;
    for(int i=2;i<total;i++){
        double a=vertex[i-1].dis(vertex[0]);
        double b=vertex[i].dis(vertex[0]);
        double c=vertex[i].dis(vertex[i-1]);
        double p=(a+b+c)/2;
        area+=sqrt(p*(p-a)*(p-b)*(p-c));
    }
    return area;
}
```

计算凸包周长

```
inline double GetConvexHullPerimeter()
{
    int total = Andrew();
    vertex[total] = vertex[0];
    double ans = 0;
    for(int i = 0; i < total; i++){
        ans+=vertex[i].dis(vertex[i+1]);
    }
    return ans;
}

//计算顶点为a,b,的三角形面积c
inline double GetArea(Point a,Point b,Point c)
{
    double len1=a.dis(b);
    double len2=a.dis(c);
    double len3=b.dis(c);
    double p=(len1+len2+len3)/2;
    return sqrt(p*(p-len1)*(p-len2)*(p-len3));
}
```

1.2.5 凸包最大三角形面积

旋转卡壳求凸包最大三角形面积，时间复杂度 $O(total^2), total$ 是凸包顶点数

```
inline double GetBiggestTriangleAreaInConvexHull()
{
    int total=Andrew();
    vertex[total]=vertex[0];
    double ans=0.0;
    for(int i=0;i<total;i++){
        int j=(i+1)%total;
        int k=(j+1)%total;
        while(j!=i&&k!=i){
            ans=max(ans,fabs((vertex[j]-vertex[i]).cross(vertex[k]-vertex[i])/2.0));
            //已知三角形三顶点，利用叉积计算面积或者海伦秦九韶公式-
            //ans=max(ans,GetArea(vertex[i],vertex[j],vertex[k]));
            while(k!=i&&(vertex[j]-vertex[i]).cross((vertex[k+1]-vertex[k]))<0){
                k=(k+1)%total;
            }
```

```
            j=(j+1)%total;
        }
    }
    return ans;
}
```

1.2.6 凸包最大四边形面积

UESTC 371

给 $n \leq 1000$ 个点，从中选出不超过 4 个点，使得组成的多边形面积最大，输出最大面积。

求完凸包后，枚举对角线顶点，对另外两个顶点旋转卡壳。时间复杂度：$O(total^2)$

```
void solve()
{ // P[]: 初始点， Q[]: 凸包顶点
    int total = Andrew();
    if (total <= 2) {
        printf("0\n");
        return ;
    } else if (total == 3) {
        printf("%.01f\n", fabs((Q[1] - Q[0]).cross(Q[2] - Q[0])));
        return;
    }
    double ans;
    int first = 1;
    for (int i = 0; i < total; ++i) {
        int st = (i + 1) % total, ed = (i + 3) % total;
        for (int j = i + 2; j < total - 1; ++j) {
            Line line = Line(Q[i], Q[j]);
            while (st != j && sgn(line.point_to_line_dis(Q[st]) -
        line.point_to_line_dis(Q[(st + 1) % total])) <= 0) st = (st + 1) % total;
            while (ed != i && sgn(line.point_to_line_dis(Q[ed]) -
        line.point_to_line_dis(Q[(ed + 1) % total]) <= 0)) ed = (ed + 1) % total;
            double tmp1 = fabs((Q[i] - Q[st]).cross(Q[j] - Q[st]));
            double tmp2 = fabs((Q[i] - Q[ed]).cross(Q[j] - Q[ed]));
            if (first) ans = tmp1 + tmp2, first = 0;
            else ans = max(ans, tmp1 + tmp2);
        }
    }
    printf("%.01f\n", ans);
}
```

1.2.7 最小覆盖矩形面积

时间复杂度：$O(total), total$ 是顶点数

```
inline double MinimalRetangleCover()
{
    int total=Andrew();
    if(total<=2){
        return 0.0;
    }
    double ans=INF;
    vertex[total]=vertex[0];
    int r=1,p=1,q;
    for(int i=0;i<total;i++){
        double edge=vertex[i].dis(vertex[i+1]);
        double tmp1,tmp2;
        while(1){ //卡出离边 vertex[i]--vertex[i+1] 最远的点
            tmp1=(vertex[i+1]-vertex[i]).cross(vertex[r+1]-vertex[i]); //叉积
            tmp2=(vertex[i+1]-vertex[i]).cross(vertex[r]-vertex[i]);
```

```
            if(tmp2>tmp1) break;
            r=(r+1)%total;
        }
        double height = tmp2 / edge;

        while(1){ //卡出在 vertex[i]--vertex[i+1] 方向上正向最远的点
            tmp1=(vertex[i+1]-vertex[i]).dot(vertex[p+1]-vertex[i]); //点积
            tmp2=(vertex[i+1]-vertex[i]).dot(vertex[p]-vertex[i]);
            if(tmp2>tmp1) break;
            p=(p+1)%total;
        }
        double len1 = tmp2 / edge;
        //len1 是从 vertex[i] 出发沿 vertex[i]--vertex[i+1] 正方向能达到的最大距离

        if(i==0) q = p;
        while(1){ //卡出在 vertex[i]--vertex[i+1] 方向上负方向最远的点
            tmp1=(vertex[i+1]-vertex[i]).dot(vertex[q+1]-vertex[i]); //点积
            tmp2=(vertex[i+1]-vertex[i]).dot(vertex[q]-vertex[i]);
            if(tmp2<tmp1) break; //和上面的不一样
            q=(q+1)%total;
        }
        double len2=tmp2/edge;
        //len2 是从 vertex[i] 出发沿 vertex[i]--vertex[i+1] 负方向能达到的最大距离

        double len=len1-len2;
        ans=min(ans,len*height);
    }

    /* O(total^2) 算法
    for(int i=0;i<total;i++){ 遍历每一条边//
        Point A,B;
        A=vertex[i],B=vertex[(i+1)%total];
        double AB=A.dis(B);
        double leftA,rightA,leftB,rightB,height;
        leftA=rightA=leftB=rightB=height=0.0;
        for(int j=0;j<total;j++){ // 遍历凸包每一个顶点
            Point C=vertex[j];
            double d=(B-A).cross(C-A)/AB;
            height=max(height,d); // 更新高

            double tmp=(B-A).dot(C-A)/AB; // 更新 A 点左右最大距离
            if(tmp>0) rightA=max(rightA,tmp);
            else leftA=max(leftA,-tmp);

            tmp=(A-B).dot(C-B)/AB; // 更新 B 点左右最大距离
            if(tmp<0) rightB=max(rightB,-tmp);
            else leftB=max(leftB,tmp);
        //  printf("i=%d j=%d rihgtA=%.2f rightB=%.2f\n",i,j,rightA,rightB);
        }
        double len1=max(leftA,leftB-AB); // 左端最大超出
        double len2=max(rightB,rightA-AB); // 右端最大超出
        double len=len1+len2+AB; //len 即是长
        //printf("len1=%.2f len2=%.2f len=%.2f height=%.2f\n",len1,len2,len,height);
        ans=min(ans,len*height); // 更新最小外接矩形
    }
    */
    return ans;
}
```

1.2.8 凸包最小外接平行四边形面积

一定有两条边是凸包的边

```
double height[MAX_N]; //用于存储凸包每条边上最远顶点到这条边的距离
```

```
double edge[MAX_N]; //存储边长度

inline double MinimalParallelogramCover()
{
    int total=Andrew();
    if(total<=2){
        return 0.0;
    }
    vertex[total]=vertex[0];
    //求每条边最远顶点距离，即是这条边上对应的高
    for(int i=0;i<total;i++){
        height[i]=-1.0;
        edge[i]=vertex[i].dis(vertex[i+1]);
        for(int j=0;j<total;j++){
            double tmp=((vertex[i+1]-vertex[i]).cross(vertex[j]-vertex[i]));
            //edge; 顶点 vertex[j] 到边 i--(i+1) 的距离
            height[i]=max(height[i],fabs(tmp/edge[i]));
        }
    }
    double ans=INF;
    //枚举两条边计算以这两条边为平行四边形边的平行四边形
    for(int i=0;i<total;i++){
        Point tmp1=vertex[i+1]-vertex[i];
        for(int j=0;j<total;j++){
            Point tmp2=vertex[j+1]-vertex[j];
            double angle=(tmp1.cross(tmp2))/edge[i]/edge[j]; //计算向量夹角的正弦值
            if(fabs(angle)<eps) continue;
            ans=min(ans,height[i]*height[j]/fabs(angle));
            //已知平行四边形两组对边上的高及一个内角的正弦值求平行四边形的面积
        }
    }
    return ans;
}

int main()
{
    scanf("%d",&T);
    while(T--){
        scanf("%d",&n);
        for(int i=0;i<n;i++){
            scanf("%lf%lf",&point[i].x,&point[i].y);
        }
        .........
    }
    return 0;
}
```

1.2.9 两相离凸包最短距离

工具：

- 点到线段最短距离
- 线段上点最短距离
- 将所有点顺时针排序

```
const int MAX_N=10010;
const double INF=1e90;
const double eps=1e-10;

int n,m;

Point P[MAX_N],Q[MAX_N],center;
```

```
8
9  inline bool cmp(Point a,Point b)
10 {// a 点在 b 点顺时针方向返回 true, 否则返回 false
11     double res=(a-center).cross(b-center);
12     if(res<eps) return true;
13     if(res>eps) return false;
14     //向量共线，用距离判断
15     double d1=a.dis(center);
16     double d2=b.dis(center);
17     return d1>d2;
18 }
19
20 inline void Clockwise(Point point[],int num)
21 {//将所有点按照顺时针排序
22     //计算所有点重心
23     double x=0,y=0;
24     for(int i=0;i<num;i++){
25         x+=point[i].x;
26         y+=point[i].y;
27     }
28     center.x=x/num;
29     center.y=y/num;
30     sort(point,point+num,cmp);
31 }
32
33 inline double Get_Dis_Point(Point A,Point B,Point C)
34 {//计算 C 点到线段 AB 的最短距离
35     //AC.dot(AB)=|AC|*|AB|*cos(alpha),alpha 是角 BAC 的大小，点积值小于 0
36     // 说明 alpha>(PI所以/2), A 离 C 更近
37     if((C-A).dot(B-A)<-eps) return A.dis(C);
38     if((C-B).dot(A-B)<-eps) return B.dis(C);
39     //考虑 C 到 AB 的垂足落在线段 AB 上的情况
40     //AB.cross(AC)=|AB|*|AC|*sin(alpha其中), alpha 是角 BAC 的大小
41     return fabs((B-A).cross(C-A)/A.dis(B));
42 }
43
44 inline double Get_Dis_Segment(Point A,Point B,Point C,Point D)
45 {//求出线段 AB 和线段 CD 之间的最短距离只需计算端点最短距离,
46     double res=min(Get_Dis_Point(A,B,C),Get_Dis_Point(A,B,D));
47     res=min(res,Get_Dis_Point(C,D,A));
48     res=min(res,Get_Dis_Point(C,D,B));
49     return res;
50 }
51
52 inline double Get_Min_Dis(Point point1[],Point point2[],int num1,int num2)
53 {//获取两相离凸包的最短距离, point1[],point2[] 和 num1,num2 分别是两凸包顶点数组及顶点数量
54     int ymin=0,ymax=0;
55     // 获取一个凸包的 y 值最小点
56     for(int i=0;i<num1;i++){
57         if(point1[i].y<point1[ymin].y){
58             ymin=i;
59         }
60     }
61     // 获取另一个凸包的 y 值最大点
62     for(int i=0;i<num2;i++){
63         if(point2[i].y>point2[ymax].y){
64             ymax=i;
65         }
66     }
67     point1[num1]=point1[0], point2[num2]=point2[0];
68     double tmp,ans=point1[ymin].dis(point2[ymax]);
69     for(int i=0;i<num1;i++){
70         while(1){
71             tmp=(point2[ymax+1]-point1[ymin+1]).cross(point1[ymin]-point1[ymin+1])-
72                 (point2[ymax]-point1[ymin+1]).cross(point1[ymin]-point1[ymin+1]);
```

```
73              if(tmp>eps) break;
74              ymax=(ymax+1)%num2;
75          }
76          if(tmp<-eps){   // 只旋转到一条边上，计算点到线段距离
77              ans=min(ans,Get_Dis_Point(point1[ymin],point1[ymin+1],point2[ymax]));
78          }else { // 同时旋转到两条边上
79              ans=min(ans,Get_Dis_Segment(point1[ymin],point1[ymin+1],
80                                  point2[ymax],point2[ymax+1]));
81          }
82          ymin=(ymin+1)%num1;
83      }
84      return ans;
85
86  }
87
88  int main()
89  {
90      while(~scanf("%d%d",&n,&m)&&(n||m)){
91          for(int i=0;i<n;i++){
92              scanf("%lf%lf",&P[i].x,&P[i].y);
93          }
94          for(int j=0;j<m;j++){
95              scanf("%lf%lf",&Q[j].x,&Q[j].y);
96          }
97          Clockwise(P,n); // 将顶点顺时针排序
98          Clockwise(Q,m);
99          double ans=min(Get_Min_Dis(P,Q,n,m),Get_Min_Dis(Q,P,m,n));
100         printf("%f\n",ans);
101     }
102     return 0;
103 }
```

1.3 三维几何

1.3.1 任意四面体的外接球半径公式

若四面体 ABCD 中，$\angle CAD = \alpha$，$\angle CBD = \beta$，二面角 $A - CD - B$ 的大小为 θ，$CD = 2m$，则其外接球半径：

$$R = \frac{m}{\sin\alpha \sin\beta \sin\theta} \cdot \sqrt{1 - (\cos\alpha\cos\beta + \sin\alpha\sin\beta\cos\theta)^2}$$

1.3.2 定义

```
struct Point { // 三维点定义
    double x, y, z;

    Point() {}
    Point(double _x, double _y, double _z): x(_x), y(_y), z(_z) {}

    void input() {
        scanf("%lf%lf%lf", &x, &y, &z);
    }
    void output() {
        printf("%.2lf %.2lf %.2lf\n", x, y, z);
    }
    bool operator == (const Point& rhs) const {
        return sgn(x - rhs.x) == 0 && sgn(y - rhs.y) == 0 && sgn(z - rhs.z) == 0;
    }
    double len() {
        return sqrt(x * x + y * y + z * z);
    }
    double len2() {
        return x * x + y * y + z * z;
    }
    double dis(const Point& rhs) const {
        return sqrt((x - rhs.x) * (x - rhs.x) +
                (y - rhs.y) * (y - rhs.y) + (z - rhs.z) * (z - rhs.z));
    }
    Point operator - (const Point& rhs) const {
        return Point(x - rhs.x, y - rhs.y, z - rhs.z);
    }
    Point operator + (const Point& rhs) const {
        return Point(x + rhs.x, y + rhs.y, z + rhs.z);
    }
    Point operator * (const double& d) const {
        return Point(x * d, y * d, z * d);
    }
    Point operator / (const double& d) const {
        return Point(x / d, y / d, z /d);
    }
    double dot(const Point& rhs) const { // 点乘
        return x * rhs.x + y * rhs.y + z * rhs.z;
    }
    Point cross(const Point& rhs) const { // 叉乘
        return Point(y * rhs.z - z * rhs.y,
            z * rhs.x - x * rhs.z, x * rhs.y - y * rhs.x);
    }
    double area(const Point& rhs1, const Point& rhs2) const { // 空间三点三角形面积
        Point cur = *this;
        Point ret = (rhs1 - cur).cross(rhs2 - cur);
        return ret.len() / 2.0;
    }
    double point_to_plane(const Point& rhs1, const Point& rhs2,
        const Point& rhs3) const { // 点到平面距离
        Point cur = *this;
        Point ret = (rhs2 - rhs1).cross(rhs3 - rhs1);
```

```
        return ret.dot(cur - rhs1) / ret.len();
    }
    double volume(const Point& rhs1, const Point& rhs2, const Point& rhs3) const{
        // 四面体体积
        Point cur = *this;
        double s = rhs1.area(rhs2, rhs3);
        double h = cur.point_to_plane(rhs1, rhs2, rhs3);
        return fabs(s * h / 3.0);
    }
};
```

```
struct Line { // 三维线段、直线定义
    Point st, ed;

    Line() {}
    Line(Point _st, Point _ed): st(_st), ed(_ed) {}

    void input() {
        st.input();
        ed.input();
    }
    bool operator == (const Line& rhs) const {
        return st == rhs.st && ed == rhs.ed;
    }
    double length() {
        return st.dis(ed);
    }
    double point_to_line(const Point& rhs) const { // 点到直线距离
        return ((ed - st).cross(rhs - st)).len() / st.dis(ed);
    }
    double point_to_seg(const Point& rhs) const { // 点到线段距离
        if (sgn((rhs - st).dot(ed - st)) < 0 || sgn((rhs - ed).dot(st - ed)) < 0) {
            return min(rhs.dis(st), rhs.dis(ed));
        }
        return point_to_line(rhs);
    }
    Point projection(const Point& rhs) const { // 点到直线的投影
        return st + ( ((ed - st) * ((ed - st).dot(rhs - st))) / ((ed - st).len2()));
    }
    bool point_on_line(const Point& rhs) const { // 判断点是否在直线上
        return sgn( ((st - rhs).cross(ed - rhs)).len() ) == 0 &&
               sgn( (st - rhs).dot(ed - rhs) ) == 0;
    }
};
```

```
struct Plane {
    Point a, b, c, o; // 平面上的三个点以及法向量

    Plane() {}
    Plane (Point _a, Point _b, Point _c) {
        a = _a;
        b = _b;
        c = _c;
        o = pvec();
    }
    Point pvec() { // 法向量
        return (b - a).cross(c - a);
    }
    bool point_on_plane(const Point& rhs) const { // 点在平面上的判断
        return sgn((rhs - a).dot(o)) == 0;
    }
    double angleplane(const Plane& rhs) const { // 两平面夹角
        return acos(o.dot(rhs.o)) / (o.len() * rhs.o.len());
    }
};
```

1.3.3 给定四个点，求四面体内切球球心和半径

[HDU 5733] 给定四个点，求四面体内切球球心和半径。

设四面体 $A_1A_2A_3A_4$ 的顶点 $A_i(i=1,2,3,4)$ 所对的侧面 面积为 S_i，顶点 A_i 坐标为 (x_i,y_i,z_i)，四面体表面积之和为 sum，则四面体内心 I 的坐标 (x,y,z) 为：

$$x=\frac{\sum_{i=1}^{i=4}s_i*x_i}{sum}$$

$$y=\frac{\sum_{i=1}^{i=4}s_i*y_i}{sum}$$

$$z=\frac{\sum_{i=1}^{i=4}s_i*z_i}{sum}$$

```
int main()
{
    Point p[5];
    double sum, s[5];
    while (~scanf("%lf%lf%lf", &p[0].x, &p[0].y, &p[0].z)) {
        p[1].input(); p[2].input(); p[3].input();
        Plane plane = Plane(p[0], p[1], p[2]);
        if (plane.point_on_plane(p[3])) { // 四点共面
            printf("O O O O\n");
            continue;
        }
        sum = 0;
        s[0] = p[1].area(p[2], p[3]);
        s[1] = p[0].area(p[2], p[3]);
        s[2] = p[0].area(p[1], p[3]);
        s[3] = p[0].area(p[1], p[2]);
        for (int i = 0; i < 4; ++i) sum += s[i];
        double V = p[0].volume(p[1], p[2], p[3]);
        double r = V * 3.0 / sum; // r * 四个面的总面积 / 3 = 体积
        double ansx = 0, ansy = 0, ansz = 0;
        for (int i = 0; i < 4; ++i) {
            ansx += s[i] * p[i].x;
            ansy += s[i] * p[i].y;
            ansz += s[i] * p[i].z;
        }
        ansx /= sum, ansy /= sum, ansz /= sum;
        printf("%.4lf %.4lf %.4lf %.4lf\n", ansx, ansy, ansz, r);
    }
    return 0;
}
```

1.3.4 给定四面体的六条边长求四面体体积

[POJ 2208] 给定四面体的六条边长求四面体体积

设六条边长依次为 l,m,n,p,q,r，欧拉四面体体积公式：

$$V=\frac{1}{36}*\begin{vmatrix} p^2 & \frac{p^2+q^2-n^2}{2} & \frac{p^2+r^2-m^2}{2} \\ \frac{p^2+q^2-n^2}{2} & q^2 & \frac{q^2+r^2-l^2}{2} \\ \frac{p^2+r^2-m^2}{2} & \frac{q^2+r^2-l^2}{2} & r^2 \end{vmatrix}$$

```
int main()
{
    double l, n, m, p, q, r;
    while (~scanf("%lf%lf%lf%lf%lf%lf", &p, &q, &r, &n, &m, &l)) {
        double tmp1 = (p * p + q * q - n * n) / 2.0;
```

```
6           double tmp2 = (p * p + r * r - m * m) / 2.0;
7           double tmp3 = (q * q + r * r - l * l) / 2.0;
8           double ans1 = p * p * (q * q * r * r - tmp3 * tmp3);
9           double ans2 = tmp1 * (tmp1 * r * r - tmp2 * tmp3);
10          double ans3 = tmp2 * (tmp1 * tmp3 - q * q * tmp2);
11          double ans = sqrt(ans1 - ans2 + ans3) / 6.0;
12          printf("%.4lf\n", ans);
13      }
14      return 0;
15  }
```