Table of Contents

前言

这是总的算法手册的前言。

计算几何

计算几何在ACM/ICPC竞赛的题目中属于较容易的内容。但计算几何往往代码量多，有时需要按照多种情况进行讨论，还有考虑到复杂的浮点数精度问题，所以常常容易卡题。所以计算几何对平时模板的积累就非常重要。

- 浮点数相关的陷阱
- 向量
- 线段
- 三角形
- 多边形
- 凸包
- 半平面
- 圆
- 三维计算几何

浮点数相关的陷阱

误差修正

简述

因为被计算机表示浮点数的方式所限制，CPU在进行浮点数计算时会出现误差。如执行`0.1 + 0.2 == 0.3`结果往往为`false`，在四则运算中，加减法对精度的影响较小，而乘法对精度的影响更大，除法的对精度的影响最大。所以，在设计算法时，为了提高最终结果的精度，要尽量减少计算的数量，尤其是乘法和除法的数量。

浮点数与浮点数之间不能直接比较，要引入一个`eps`常量。`eps`是epsilon（ε）的简写，在数学中往往代表任意小的量。在对浮点数进行大小比较时，如果他们的差的绝对值小于这个量，那么我们就认为他们是相等的，从而避免了浮点数精度误差对浮点数比较的影响。eps在大部分题目时取`1e-8`就够了，但要根据题目实际的内容进行调整。

模板代码

```
// sgn返回x经过eps处理的符号，负数返回-1，正数返回1，x的绝对值如果足够小，就返回0。
const double eps = 1e-8;
int sgn(double x) { return x < -eps ? -1 : x > eps ? 1 : 0; }
```

整型比较	等价的浮点数比较
`a == b`	`sgn(a - b) == 0`
`a > b`	`sgn(a - b) > 0`
`a >= b`	`sgn(a - b) >= 0`
`a < b`	`sgn(a - b) < 0`
`a <= b`	`sgn(a - b) <= 0`
`a != b`	`sgn(a - b) != 0`

输入输出

用`scanf`输入浮点数时，`double`的占位符是`%lf`，但是浮点数`double`在`printf`系列函数中的标准占位符是`%f`而不是`%lf`，使用时最好使用前者，因为虽然后者在大部分的计算机和编译器中能得到正确结果，但在有些情况下会出错（比如在POJ上）。

开方

当提供给C语言中的标准库函数`double sqrt (double x)`的`x`为负值时，`sqrt`会返回`nan`，输出时

会显示成`nan`或`-1.#IND00`（根据系统的不同）。在进行计算几何编程时，经常有对接近零的数进行开方的情况，如果输入的数是一个极小的负数，那么`sqrt`会返回`nan`这个错误的结果，导致输出错误。解决的方法就是将`sqrt`包装一下，在每次开方前进行判断。

示例代码

```
double my_sqrt(double x) { return sgn(x) == 0 ? 0.0 : sqrt(x); }
```

负零

大部分的标程的输出是不会输出负零的，如下面这段程序：

```
int main() {
    printf("%.2f\n", -0.0000000001);
    return 0;
}
```

会输出`-0.00`。有时这样的结果是错误的，所以在没有Special Judge的题目要求四舍五入时，不要忘记对负零进行特殊判断。

向量

简介

向量，又称矢量，是既有大小又有方向的量，向量的长度即向量的大小称为向量的模。在计算几何中，从A指向B的向量记作AB。n维向量可以用n个实数来表示。向量的基本运算包括加减法、数乘、点积、叉积和混合积。使用向量这一个基本的数据结构，我们可以用向量表示点和更复杂的各种图形。

注意事项

我们一般用一个二维向量来表示点。注意，在有些计算几何相关的题目中，坐标是可以利用整形储存的。在做这样的题目时，坐标一定要用整形变量储存，否则精度上容易出错。具体的将点的坐标用整形变量储存可以需要使用一些技巧，比如计算中计算平方或将坐标扩大二倍等方式。

```
// Pt是Point的缩写
struct Pt {
    double x, y;
    Pt() { }
    Pt(double x, double y) : x(x), y(y) { }
};

double norm(Pt p) { return sqrt(p.x*p.x + p.y*p.y); }
void print(Pt p) { printf("(%f, %f)", p.x, p.y); }
```

基本计算

加减法

$a \pm b = (a_x \pm b_x, a_y \pm b_y)$

向量的加减法遵从平行四边形法则和三角形法则。

示例代码

```
Pt operator - (Pt a, Pt b) { return Pt(a.x - b.x, a.y - b.y); }
Pt operator + (Pt a, Pt b) { return Pt(a.x + b.x, a.y + b.y); }
```

长度

向量$a = (a_x, a_y)$的长度是$\sqrt{a_x^2 + a_y^2}$。

示例代码

```
double len(Pt p) { return sqrt(sqr(p.x)+sqr(p.y)); }
```

数乘

$ab = (ab_x, ab_y)$。

向量的数乘是一个向量和实数的运算。a如果是零，那么结果是一个零向量，如果a是一个负数，那么结果向量会改变方向。

示例代码

```
Pt operator * (double A, Pt p) { return Pt(p.x*A, p.y*A); }
Pt operator * (Pt p, double A) { return Pt(p.x*A, p.y*A); }
```

点积

又称内积。

$a \cdot b = a_x b_x + a_y b_y = |a||b|\cos\theta$，其中$\theta$是$a$与$b$的夹角。

应用

点积可以用来计算两向量的夹角。

$\cos\beta = \frac{a \cdot b}{|a||b|}$

示例代码

```
double dot(Pt a, Pt b) { return a.x * b.x + a.y * b.y; }
```

叉积

叉积又称外积。叉积运算得到的是一个向量，它的大小是a和b所构成的平行四边形的面积，方向与a和b所在平面垂直，a、b与$a \times b$成右手系。

设两向量$a = (a_x, a_y)$与$b = (b_x, b_y)$，它们在二维平面上的的叉积为：

$a \times b = a_x b_y - a_y b_x$

示例代码

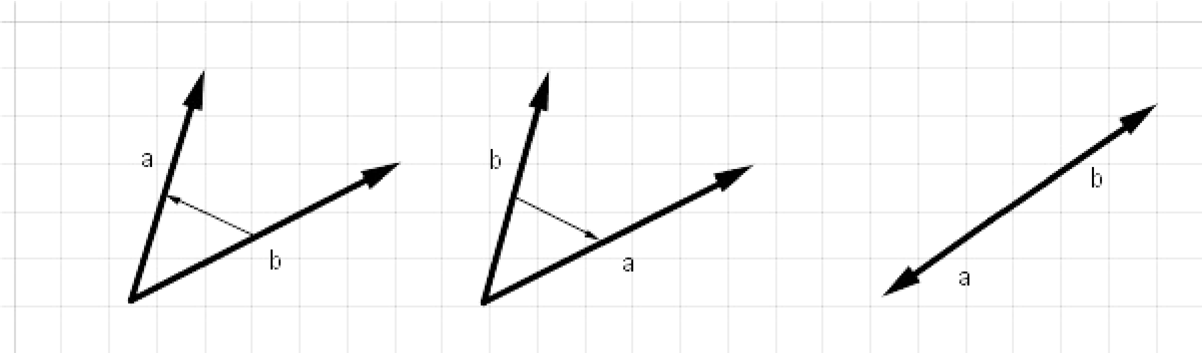
```
double det(Pt a, Pt b) { return a.x * b.y - a.y * b.x; }
```

性质与应用

叉积拥有两个重要的性质——面积与方向。

两向量叉积得到新向量的长度为这两个所构成的平行四边形的面积，利用这个性质我们可以求三角形的面积。

两向量叉积能反映出两向量方向的信息。如果$a \times b$的符号为正，那么b在a的逆时针方向；如果符号为负，那么b在a的顺时针方向；如果结果为零的话，那么a与b共线。



计算结果	b与a的方向
$\mid b \times a \mid > 0$	a在b的逆时针方向
$\mid b \times a \mid = 0$	a与b共线
$\mid b \times a \mid < 0$	a在b的顺时针方向

模板代码

```
Pt operator - (Pt a, Pt b) { return Pt(a.x - b.x, a.y - b.y); }
Pt operator + (Pt a, Pt b) { return Pt(a.x + b.x, a.y + b.y); }
Pt operator * (double A, Pt p) { return Pt(p.x*A, p.y*A); }
Pt operator * (Pt p, double A) { return Pt(p.x*A, p.y*A); }
Pt operator / (Pt p, double A) { return Pt(p.x/A, p.y/A); }
```

线段

直线与线段的表示方法

我们可以用一条线段的两个端点来表示一条线段。直线的表示有两种方式，一种方式是使用二元一次方程$y = kx + b$来表示，另一种是用直线上任意一条长度不为零的线段来表示。由于使用方程表示接近垂直于某坐标轴的直线时容易产生精度误差，所以我们通常使用直线上的某条线段来表示直线。

```
struct Sg {
    Pt s, t;
    Sg() { }
    Sg(Pt s, Pt t) : s(s), t(t) { }
    Sg(double a, double b, double c, double d) : s(a, b), t(c, d) { }
};
```

点在线段上的判断

判断点C在线段AB上的两条依据：

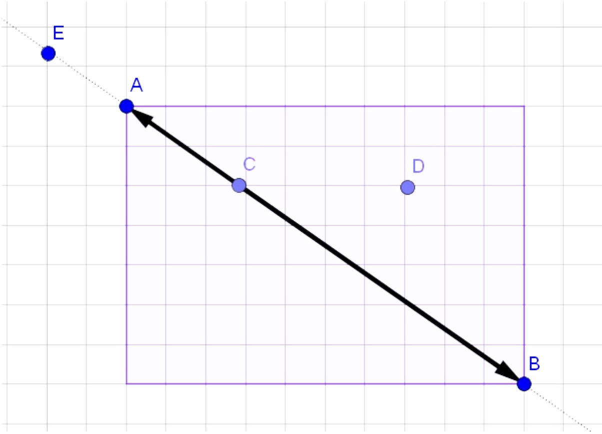
1. $CA \cdot CB = 0$。
2. C在以AB为对角顶点的矩形内。

示例代码

```
bool PtOnSegment(Pt s, Pt t, Pt a) {
    return !det(a-s, a-t) && min(s.x, t.x) <= a.x && a.x <= max(s.x, t.x) &&
        min(s.y, t.y) <= a.y && a.y <= max(s.y, t.y);
}
```

另一种方法

判断点C在AB为对角线定点的矩形内较麻烦，可以直接判断$CA \cdot CB$的符号来判断C在直线AB上是否在AB之间。



示例代码

```
bool PtOnSegment(Pt p, Pt a, Pt b) {
    return !sgn(det(p-a, b-a)) && sgn(dot(p-a, p-b)) <= 0;
}
```

把上例代码中的 `<=` 改成 `==` 就能实现不含线段端点的点在线段上的判断。

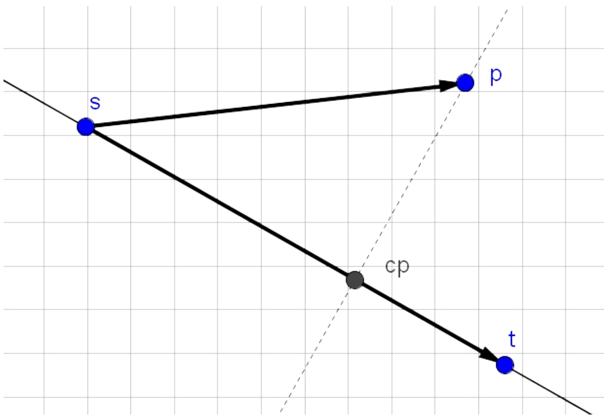
点在直线上的判断

点在直线上的判断很简单只要把点在线段上的判断的步骤2去掉即可。

示例代码

```
bool PtOnLine(Pt p, Pt s, Pt t) {
    return !sgn(det(p-a, b-a));
}
```

求点到直线的投影



示例代码

```
Pt PtLineProj(Pt s, Pt t, Pt p) {
    double r = dot(p-s, t-s) / (t - s).norm();
    return s + (t - s) * r;
}
```

判断直线关系

直线有相交和平行两种关系，靠叉乘能简单判断。

```
bool parallel(Pt a, Pt b, Pt s, Pt t) {
    return !sgn(det(a-b, s-t));
}
```

判断线段关系

线段有相交和不相交两种关系，通常按照以下步骤判断。

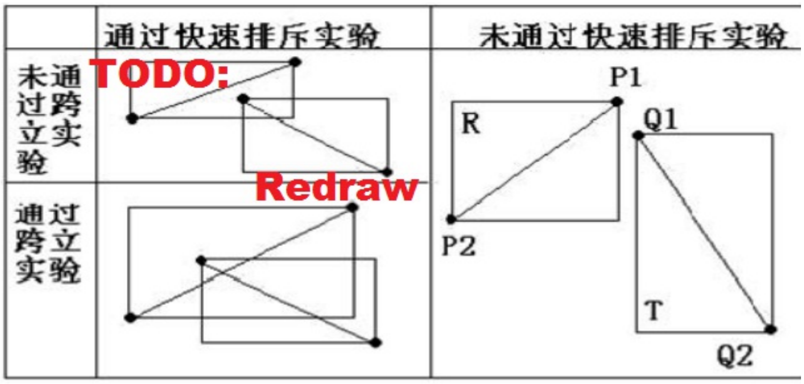
1. 快速排斥试验
2. 跨立试验

快速排斥试验

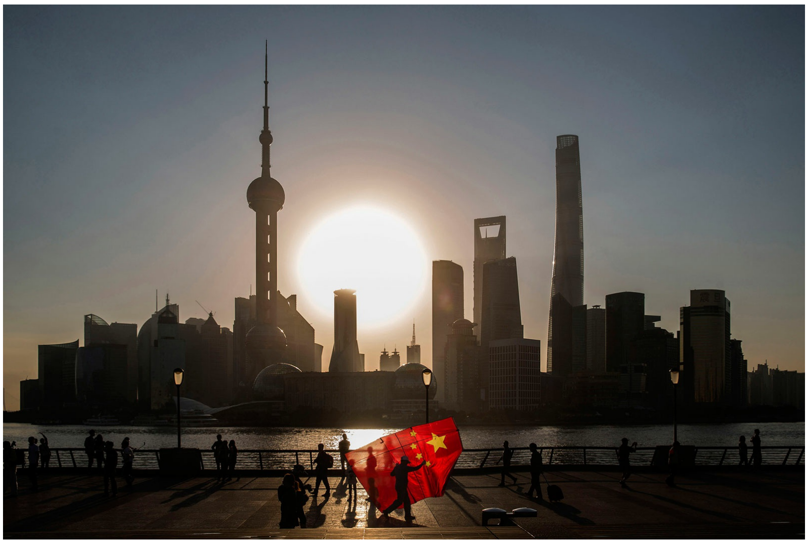
设以线段 P1P2 为对角线的矩形为 R， 设以线段 Q1Q2 为对角线的矩形为 T，如果 R 和 T 不相交，显然两线段不会相交。

跨立试验

如果两线段相交，则两线段必然相互跨立对方。若 P1P2 跨立 Q1Q2 ，则矢量 (P1 - Q1) 和(P2 - Q1)位于矢量(Q2 - Q1) 的两侧，即 (P1 - Q1) × (Q2 - Q1) *(P2 - Q1) × (Q2 - Q1) < 0*。 上式可改写成 *(P1 - Q1) × (Q2 - Q1)* (Q2 - Q1) × (P2 - Q1) > 0。 当 (P1 - Q1) × (Q2 - Q1) = 0 时，说明 (P1 - Q1) 和 (Q2 - Q1)共线，但 是因为已经通过快速排斥试验，所以 P1 一定在线段 Q1Q2 上；同理，(Q2 - Q1) ×(P2 - Q1) = 0 说明 P2 一定在线段 Q1Q2 上。所以判断 P1P2 跨立 Q1Q2 的依据是： (P1 - Q1) × (Q2 - Q1) *(Q2 - Q1) × (P2 - Q1) >= 0*。 同理判断 *Q1Q2* 跨立 *P1P2* 的依据是： *(Q1 - P1) × (P2 - P1)* (P2 - P1) × (Q2 - P1) >= 0。



求点到线段的距离



求线段ab到点p最短距离的方法为：

根据点p到的投影点的位置进行判断的方法：

1. 判断线段pa和ab所成的夹角，如果是钝角，那么|pa|是点到线段的最短距离。
2. 判断线段pb和ab所成的夹角，如果是钝角，那么|pb|是点到线段的最短距离。
3. 线段pa和线段pb与ab所成的夹角都不为钝角，那么点p到线段ab的距离是点p到直线ab的距离，这个距离可以用面积法直接算出来。

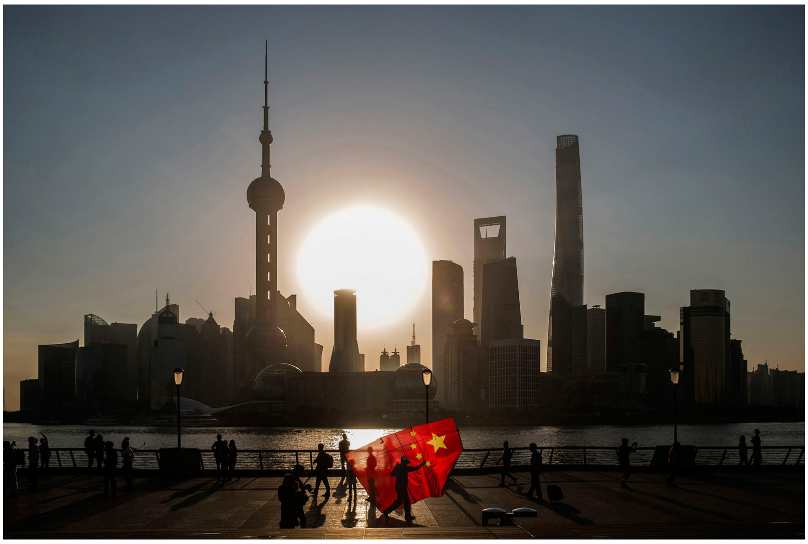
示例代码

```
double PtSegmentDist(Pt a, Pt b, Pt p) {
    if (sgn(dot(p-a, b-a)) <= 0) return (p-a).norm();
    if (sgn(dot(p-b, a-b)) <= 0) return (p-b).norm();
    return fabs(det(a-p, b-p)) / (a-b).norm();
}
```

三角形

三角形的面积

三角形的面积可以由叉积直接求出。



$S_{\triangle ABC} = \mid \frac{1}{2} AB \times AC \mid$

判断点在三角形内

_1

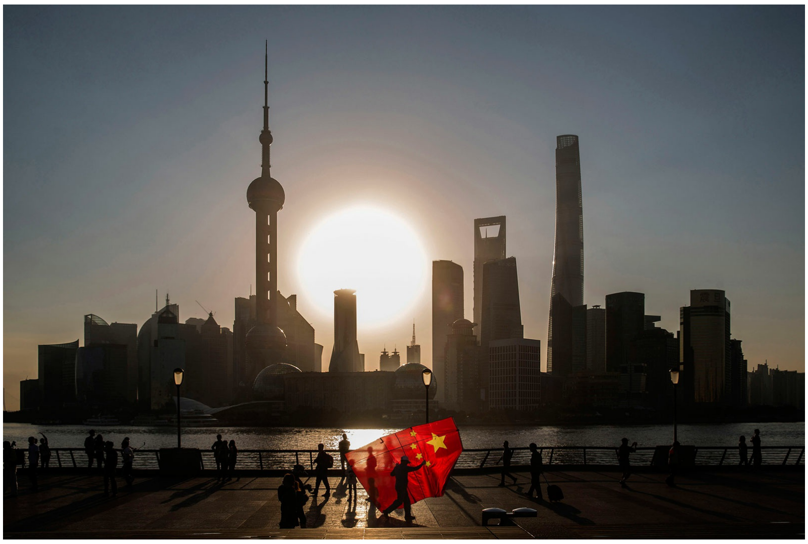
判断点P在三角形 ABC 内部常用的叉两种方法，面积法和叉积法。

面积法

$S\triangle PAB + S\triangle PAC + S\triangle PBC = S\triangle ABC$

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BC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。



三角形的重心

三角形三条中线的交点叫做三角形重心。

性质

设三角形重心为O，BC边中点为D，则有$AO = 2OD$。

三角形的外心

三角形三边的垂直平分线的交点，称为三角形外心。

性质

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。

三角形的内心

三角形内心为三角形三条内角平分线的交点。

性质

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。

三角形的垂心

三角形三边上的三条高线交于一点，称为三角形垂心。

性质

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

费马点是在一个三角形中，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度，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度，则此钝角的顶点就是距离和最小的点。

等角中心的计算方法

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）
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)+distanc
                           e(v,c))
                              u=v;
                  }
      return u;
  }
```

多边形

点在多边形内

1. 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(logN)。 方法一 射线法： 如图，设点 Q 及多边形 P1P2P3P4P5P6，判断点是否在多 边形内，首先以点 Q 为端点，向任意方法做射线，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 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 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 的整数，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，都则，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。

方法四 二分 O(logn)： 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(logN)内解决。

多边形的面积

```
double polygon_area(const Polygon &p) {
    double ans = 0.0;
    int n = p.size();
    for (int i = 0; i < n; ++i)
        ans += det(p[i], p[nxt(i)]);
    return ans / 2.0;
}
```

多边形的重心

```
Pt polygon_mass_center(const Polygon &p) {
    Pt ans = Pt(0, 0);
    double area = polygon_area(p);
    if (sgn(area) == 0) return ans;
    int n = p.size();
```

```
    for (int i = 0; i < n; ++i)
        ans = ans + (p[i]+p[nxt(i)]) * det(p[i], p[nxt(i)]);
    return ans / area / 6.0;
}
```

Pick定理

```
int polygon_border_point_cnt(const Polygon &p) {
    int ans = 0;
    int n = p.size();
    for (int i = 0; i < n; ++i)
        ans += gcd(Abs(int(p[next(i)].x-p[i].x)), Abs(int(p[next(i)].y-p[i].y)));
    return ans;
}

int polygon_inside_point_cnt(const Polygon &p) {
    return int(polygon_area(p)) + 1 - polygon_border_point_cnt(p) / 2;
}
```

凸包

点的有序化

凸包算法多要先对点进行排序。点排序的主要方法有两种——极角排序和水平排序。

极角排序

极角排序一般选择一个点做极点，然后以这个点为中心建立极坐标，将输入的点按照极角从小到大排序，如果两个点的极角相同，那么将距离极点较远的点排在前面。

水平排序

水平排序将所有点按照y坐标从小到大排列，y坐标相同的则按照x坐标从小到大排序。选取排序后最前面的A点和最后面的B点，将AB右边的点按照次序取出，再将左侧的点按照次序逆序取出后连起来就是最终的结果。

比较

虽然水平排序比较复杂，但水平排序因为不涉及三角函数操作，精度较高，在条件相同时，最好选择水平排序。

凸包求法

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 nlogn，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。

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]~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<=i<=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]~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nlogn，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 nlogn。

实现细节的注意事项

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）。

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。

模板代码

```
typedef vector<Pt> Convex;

// 排序比较函数，水平序
bool comp_less(Pt a, Pt b) {
    return sgn(a.x-b.x) < 0 || (sgn(a.x-b.x) == 0 && sgn(a.y-b.y) < 0);
}

// 返回a中点计算出的凸包，结果存在res中
void convex_hull(Convex &res, vector<Pt> a) {
    res.resize(2 * a.size() + 5);
    sort(a.begin(), a.end(), comp_less);
    a.erase(unique(a.begin(), a.end()), a.end());
    int m = 0;
    for (int i = 0; i < int(a.size()); ++i) {
        while (m>1 && sgn(det(res[m-1] - res[m-2], a[i] - res[m-2])) <= 0)
            --m;
        res[m++] = a[i];
    }
    int k = m;
    for (int i = int(a.size()) - 2; i >= 0; --i) {
        while (m>k && sgn(det(res[m-1] - res[m-2], a[i] - res[m-2])) <= 0)
            --m;
        res[m++] = a[i];
    }
    res.resize(m);
    if (a.size() > 1) res.resize(m-1);
}
```

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 p0 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 p0。 更形式的说，Jarvis 步进法构造了 CH(Q)的顶点序列 H=(P0,P1,…,Ph-1)，其中 P0 为 原始点。如图所示，下一个凸包顶点 P1 具有相对与 P0 的最小极角。（如果有数个这样的 点，选择最远的那个点作为 P1。）类似地，P2 具有相对于 P1 的最小的极角，等等。当达 到最高顶点，如 Pk（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 Pk 开始选取相对于 Pk 具有最小极角的点作为 Pk+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 P0。

旋转卡壳

模板代码

```
// 计算凸包a的直径
double convex_diameter(const Convex &a, int &first, int &second) {
    int n = a.size();
    double ans = 0.0;
    first = second = 0;
    if (n == 1) return ans;
    for (int i = 0, j = 1; i < n; ++i) {
        while (sgn(det(a[nxt(i)]-a[i], a[j]-a[i]) - det(a[nxt(i)]-a[i], a[nxt(j)]-a[i]))
            j = nxt(j);
        double d = max((a[i]-a[j]).norm(), (a[nxt(i)]-a[nxt(j)]).norm());
        if (d > ans) ans=d, first=i, second=j;
    }
    return ans;
}
```

半平面

简介

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 >= 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。

半平面交求法

我们用一个向量$(x_1,y_1) \rightarrow (x_x,y_x)$的左侧来描述一个半平面。首先将半平面按照极角排序，极角相同的则只保留最左侧的一个。然后用一个双端队列维护这些半平面：按照顺序插入，在插入半平面p_i之前判断双端队列尾部的两个半平面的交点是否在半平面p_i内，如果不是则删除最后一个半平面；判断双端队列尾部的两个半平面交是否在半平面p_i内，如果不是则删除第一个半平面。插入完毕之后再处理一下双端队列两端多余的半平面，最后求出尾端和顶端的两个半平面的交点即可。

模板代码

```
// 计算半平面交
void halfplane_intersect(vector<HP> &v, Convex &output) {
    sort(v.begin(), v.end(), cmp_HP);
    deque<HP> q;
    deque<Pt> ans;
    q.push_back(v[0]);
    int n = v.size();
    for (int i = 1; i < n; ++i) {
        if (sgn(arg(v[i].t-v[i].s) - arg(v[i-1].t-v[i-1].s)) == 0)
            continue;
        while (ans.size() > 0 && !satisfy(ans.back(), v[i])) {
            ans.pop_back();
            q.pop_back();
        }
        while (ans.size() > 0 && !satisfy(ans.front(), v[i])) {
            ans.pop_front();
            q.pop_front();
        }
        ans.push_back(crosspoint(q.back(), v[i]));
        q.push_back(v[i]);
    }
    while (ans.size() > 0 && !satisfy(ans.back(), q.front())) {
```

```
        ans.pop_back();
        q.pop_back();
    }
    while (ans.size() > 0 && !satisfy(ans.front(), q.back())) {
        ans.pop_front();
        q.pop_front();
    }
    ans.push_back(crosspoint(q.back(), q.front()));
    output = vector<Pt>(ans.begin(), ans.end());
}
```

凸多边形交

```
// 凸多边形交
void convex_intersection(const Convex &v1, const Convex &v2, Convex &out) {
    vector<HP> h;
    for (int i = 0, n = v1.size(); i < n; ++i)
        h.push_back(HP(v1[i], v1[nxt(i)]));
    for (int i = 0, n = v2.size(); i < n; ++i)
        h.push_back(HP(v2[i], v2[nxt(i)]));
    halfplane_intersect(h, out);
}
```

圆

圆与线求交

将线段AB写成参数方程P=A+t(B-A)，带入圆的方程，得到一个一元二次方程。解出t就可以求得线段所在直线与圆的交点。如果0<=t<=1则说明点在线段上。

```
void circle_cross_line(Pt a, Pt b, Pt o, double r, Pt ret[], int &num) {
    double ox = o.x, oy = o.y, ax = a.x, ay = a.y, bx = b.x, by = b.y;
    double dx = bx-ax, dy = by-ay;
    double A = dx*dx + dy*dy;
    double B = 2*dx*(ax-ox) + 2*dy*(ay-oy);
    double C = sqr(ax-ox) + sqr(ay-oy) - sqr(r);
    double delta = B*B - 4*A*C;
    num = 0;
    if (sgn(delta) >= 0) {
        double t1 = (-B - Sqrt(delta)) / (2*A);
        double t2 = (-B + Sqrt(delta)) / (2*A);
        if (sgn(t1-1) <= 0 && sgn(t1) >= 0)
            ret[num++] = Pt(ax + t1*dx, ay + t1*dy);
        if (sgn(t2-1) <= 0 && sgn(t2) >= 0)
            ret[num++] = Pt(ax + t2*dx, ay + t2*dy);
    }
}
```

圆与圆求交

```
// 计算圆a和圆b的交点，注意要先判断两圆相交
void circle_circle_cross(Pt ap, double ar, Pt bp, double br, Pt p[]) {
    double d = (ap - bp).norm();
    double cost = (ar*ar + d*d - br*br) / (2*ar*d);
    double sint = sqrt(1.0 - cost*cost);
    Pt v = (bp - ap) / (bp - ap).norm() * ar;
    p[0] = ap + rotate(v, cost, -sint);
    p[1] = ap + rotate(v, cost, sint);
}
```

圆与多边形交

圆的面积并

三维计算几何

三维凸包

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <vector>
#include <iomanip>
#include <iostream>
#include <cmath>
using namespace std;

/* Macros */
/****************************************************************************/
#define nxt(i) ((i+1)%n)
#define nxt2(i, x) ((i+1)%((x).size()))
#define prv(i) ((i+(x).size()-1)%n)
#define prv2(i, x) ((i+(x).size()-1)%((x).size()))
#define sz(x) (int((x).size()))
#define setpre(x) do{cout<<setprecision(x)<<setiosflags(ios::fixed);}while(0)

/* Real number tools */
/****************************************************************************/
const double PI = acos(-1.0);
const double eps = 1e-8;
double mysqrt(double x) {
    return x <= 0.0 ? 0.0 : sqrt(x);
}
double sq(double x) {
    return x*x;
}
int sgn(double x) {
    return x < -eps ? -1 : x > eps ? 1 : 0;
}

/* 3d Point */
/****************************************************************************/
struct Pt3 {
    double x, y, z;
    Pt3() { }
    Pt3(double x, double y, double z) : x(x), y(y), z(z) { }
};
typedef const Pt3 cPt3;
typedef cPt3 & cPt3r;

Pt3 operator + (cPt3r a, cPt3r b) { return Pt3(a.x+b.x, a.y+b.y, a.z+b.z); }
Pt3 operator - (cPt3r a, cPt3r b) { return Pt3(a.x-b.x, a.y-b.y, a.z-b.z); }
Pt3 operator * (cPt3r a, double A) { return Pt3(a.x*A, a.y*A, a.z*A); }
Pt3 operator * (double A, cPt3r a) { return Pt3(a.x*A, a.y*A, a.z*A); }
Pt3 operator / (cPt3r a, double A) { return Pt3(a.x/A, a.y/A, a.z/A); }
bool operator == (cPt3r a, cPt3r b) {
```

```
    return !sgn(a.x-b.x) && !sgn(a.y-b.y) && !sgn(a.z-b.z);
}
istream& operator >> (istream& sm, Pt3 &pt) {
    sm >> pt.x >> pt.y >> pt.z; return sm;
}
ostream & operator << (ostream& sm, cPt3r pt) {
    sm << "(" << pt.x << ", " << pt.y << ", " << pt.z << ")"; return sm;
}
double len(cPt3r p) { return mysqrt(sq(p.x) + sq(p.y) + sq(p.z)); }
double dist(cPt3r a, cPt3r b) { return len(a-b); }
Pt3 unit(cPt3r p) { return p / len(p); }
Pt3 det(cPt3r a, cPt3r b) {
    return Pt3(a.y*b.z-a.z*b.y, a.z*b.x-a.x*b.z, a.x*b.y-a.y*b.x);
}
double dot(cPt3r a, cPt3r b) {
    return a.x*b.x + a.y*b.y + a.z*b.z;
}
double mix(cPt3r a, cPt3r b, cPt3r c) {
    return dot(a, det(b, c));
}
/* 3d Line & Segment */
/*****************************************************************************/
struct Ln3 {
    Pt3 a, b;
    Ln3() { }
    Ln3(cPt3r a, cPt3r b) : a(a), b(b) { }
};
typedef const Ln3 cLn3;
typedef cLn3 & cLn3r;

bool ptonln(cPt3r a, cPt3r b, cPt3r c) {
    return sgn(len(det(a-b, b-c))) <= 0;
}

/* 3d Plane */
/*****************************************************************************/
struct Pl {
    Pt3 a, b, c;
    Pl() { }
    Pl(cPt3r a, cPt3r b, cPt3r c) : a(a), b(b), c(c) { }
};
typedef const Pl cPl;
typedef cPl & cPlr;

Pt3 nvec(cPlr pl) {
    return det(pl.a-pl.b, pl.b-pl.c);
}

/* Solution  */
/*****************************************************************************/
bool cmp(cPt3r a, cPt3r b) {
    if (sgn(a.x-b.x)) return sgn(a.x-b.x) < 0;
    if (sgn(a.y-b.y)) return sgn(a.y-b.y) < 0;
    if (sgn(a.z-b.z)) return sgn(a.z-b.z) < 0;
    return false;
}
```

```cpp
struct Face {
    int a, b, c;
    Face() { }
    Face(int a, int b, int c) : a(a), b(b), c(c) { }
};

void convex3d(vector<Pt3> &p, vector<Pl> &out) {
    sort(p.begin(), p.end(), cmp);
    p.erase(unique(p.begin(), p.end()), p.end());
    random_shuffle(p.begin(), p.end());
    vector<Face> face;
    for (int i = 2; i < sz(p); ++i) {
        if (ptonln(p[0], p[1], p[i])) continue;
        swap(p[i], p[2]);
        for (int j = i + 1; j < sz(p); ++j)
            if (sgn(mix(p[1]-p[0], p[2]-p[1], p[j]-p[0])) != 0) {
                swap(p[j], p[3]);
                face.push_back(Face(0, 1, 2));
                face.push_back(Face(0, 2, 1));
                goto found;
            }
    }
found:
    vector<vector<int> > mark(sz(p), vector<int>(sz(p), 0));
    for (int v = 3; v < sz(p); ++v) {
        vector<Face> tmp;
        for (int i = 0; i < sz(face); ++i) {
            int a = face[i].a, b = face[i].b, c = face[i].c;
            if (sgn(mix(p[a]-p[v], p[b]-p[v], p[c]-p[v])) < 0) {
                mark[a][b] = mark[b][a] = v;
                mark[b][c] = mark[c][b] = v;
                mark[c][a] = mark[a][c] = v;
            }else{
                tmp.push_back(face[i]);
            }
        }
        face = tmp;
        for (int i = 0; i < sz(tmp); ++i) {
            int a = face[i].a, b = face[i].b, c = face[i].c;
            if (mark[a][b] == v) face.push_back(Face(b, a, v));
            if (mark[b][c] == v) face.push_back(Face(c, b, v));
            if (mark[c][a] == v) face.push_back(Face(a, c, v));
        }
    }
    out.clear();
    for (int i = 0; i < sz(face); ++i)
        out.push_back(Pl(p[face[i].a], p[face[i].b], p[face[i].c]));
}

vector<Pt3> p;
vector<Pl> out;

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        Pt3 pt;
```

```
        cin >> pt;
        p.push_back(pt);
    }
    convex3d(p, out);
    double area = 0.0;
    for (int i = 0; i < sz(out); ++i)
        area += len(det(out[i].a-out[i].b, out[i].b-out[i].c));
    setpre(3);
    cout << area / 2.0 << "\n";
    return 0;
}
```