计算几何

精度

计算几何题目中常常不可避免的需要使用浮点数，而浮点数运算会带来精度误差，因此我们在比较浮点数时不能像整数那样直接比较，需要引入一个极小值 eps：

$$a = b \Leftrightarrow |a - b| < eps$$
$$a > b \Leftrightarrow a - b > eps$$
$$a < b \Leftrightarrow a - b < -eps$$

一、基础概念

1.点

一般我们在解决平面几何问题时，都是在平面直角坐标系上进行思考计算，因此我们的点（二维）都用它在直角坐标系上的 $P(x, y)$ 坐标来表示。

2.线段

我们用线段的两个端点来表示这条线段，如线段 OA 线段 PQ 等。有时我们也会使用点+向量的形式来表示。

线段 AB 上的点 C 满足：$\forall C \in AB, \exists p \in [0,1], C = pA + (1-p)B$

3.直线

虽然我们可以用直线方程 $ax + by + c = 0$ 或 $y = kx + b$ 来表示一条直线，但是在竞赛中往往使用几何方法比使用代数方法更为简便，因此直线与线段一样，常使用直线上两个不同点（或是点+向量）来表示。当然，直线方程在一些时候也是有用武之地的。

4.多边形

若干条线段首尾顺次相连所形成的平面图形。

通常按逆时针或顺时针的顺序存储多边形上所有的点。有时也会按顺序存边以及起点。

5.圆

通常我们存储圆心坐标以及圆的半径长，计算相关问题时可使用几何方法或是用圆方程来列方程求解。

6.向量

几何向量是线性空间中有**大小**与**方向**的量。

向量的表示

向量可以形象化地表示为**带箭头的线段**。箭头所指代表向量的方向，线段的长度代表向量的大小。如果给定了向量的起点 A 与终点 B，则向量可以表示为 $\overrightarrow{AB}$，也可以用字母 a,b,u,v 等表示（书写时需要在字母顶加上小箭头）。

在直角坐标系上，我们也能用数对的形式把向量表示出来。

若起点 $A=(x_1,y_1)$，终点 $B=(x_2,y_2)$，则 $\overrightarrow{AB}=(x_2-x_1,y_2-y_1)$.

设 $\overrightarrow{OA}=a$，则有向线段 OA 的长度叫做向量 a 的模（长度），记作 $|a|$，若它的坐标表示为 (x,y)，则 $|a|=\sqrt{x^2+y^2}$ 。

一般我们认为向量的要素只有大小与方向，所以对于一条有向线段，我们把它进行平移，将起点移到原点处，这样终点的坐标也就是这个向量的坐标了，方便我们处理。这里也能看出，许多向量它们的坐标表示是相同的。

对于坐标 (x,y) 它能用来表示点也能用来表示向量。

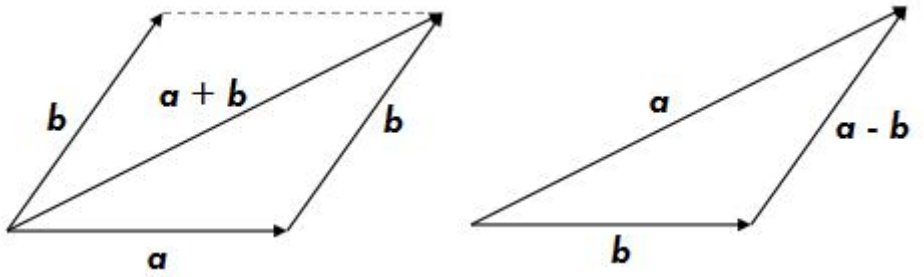
向量的运算

为了方便，我们可以定义向量和点的运算：

$$\text{点}-\text{点}=\text{向量}$$
$$\text{点}+\text{向量}=\text{点}$$
$$\text{点}-\text{向量}=\text{点}$$

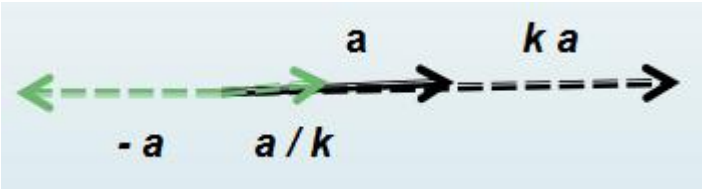
向量的加减法可以用坐标直观表示出来：

加法：$\begin{aligned} &a=(x_1,y_1),b=(x_2,y_2)\\ &a+b=(x_1+x_2,y_1+y_2)\end{aligned}$，减法：$\begin{aligned} &a=(x_1,y_1),b=(x_2,y_2)\\ &a-b=(x_1-x_2,y_1-y_2)\end{aligned}$



向量的乘除法代表向量的**伸长或缩短**，乘或除以一个负数还能使得向量的方向反向。

$$\begin{aligned} &a=(x,y)\\ &ka=(kx,ky)\\ &a/k=(x/k,y/k)\end{aligned}$$

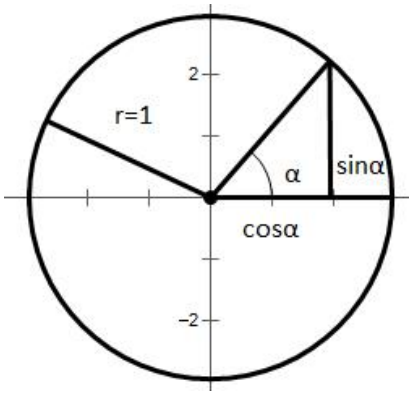


向量的夹角：两个非 0 向量a,b，在空间任取一点O作$\overrightarrow{OA}=a,\overrightarrow{OB}=b$，则$\angle AOB$叫做向量$a$与$b$的夹角，记作$<a,b>$。

弧度：等于半径长的圆弧所对的圆心角叫做 1 弧度的角，用符号 rad 表示，读作弧度。

弧度制与角度制换算：$1\deg=\frac{\pi}{180}rad\quad 1rad=\frac{180}{\pi}\deg$.

利用单位圆判断$\sin,\cos$的符号：



正弦定理：对于任意三角形 ABC，a,b,c 分别为 $\angle A,\angle B,\angle C$ 的对边，R 为三角形 ABC 外接圆半径，则有 $\frac{a}{\sin A}=\frac{b}{\sin B}=\frac{c}{\sin C}=2R$。

余弦定理：对于任意三角形 ABC，a,b,c 分别为 $\angle A,\angle B,\angle C$ 的对边，则有：

$$a^2=b^2+c^2-2bc\cos A$$
$$b^2=a^2+c^2-2ac\cos B$$
$$c^2=a^2+b^2-2ab\cos C$$

向量的点积（内积，数量积）：

$a=(x_1,y_1),b=(x_2,y_2)$ 的点积用 $a\cdot b$ 表示，$a\cdot b=x_1x_2+y_1y_2$。它的结果是一个**标量**（只有大小没有方向）。它的几何意义是向量 a 在向量 b 方向上的投影与向量 b 的模的乘积，即 $a\cdot b=|a|\times|b|\times\cos<a,b>$。

由它的几何意义可知，$a\cdot b=0\Leftrightarrow a\perp b$。利用点积的计算式子我们也能方便的算出两个向量的夹角的余弦值：$\cos<a,b>=\frac{a\cdot b}{|a||b|}$，进而可以知道夹角的大小。同样我们也能从计算式子中得出一个与向量 $a=(x,y)$ 垂直的向量为 $(y,-x)$。

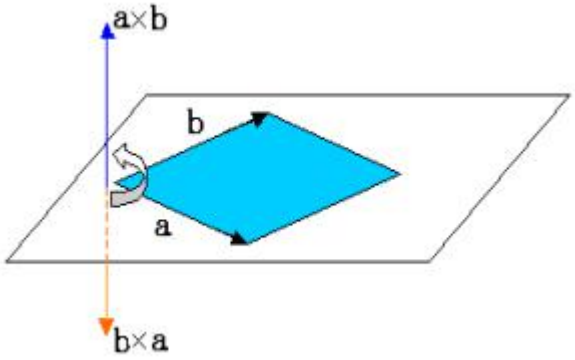
这些定义与性质也可以拓展到 n 维空间中，即 $a\cdot b=\sum_{i=1}^{n}a_ib_i$。

向量的叉积（外积，向量积）：

三维叉积：两个向量 $a=(x_1,y_1,z_1),b=(x_2,y_2,z_2)$ 的叉积的结果是一个**向量** c，记作

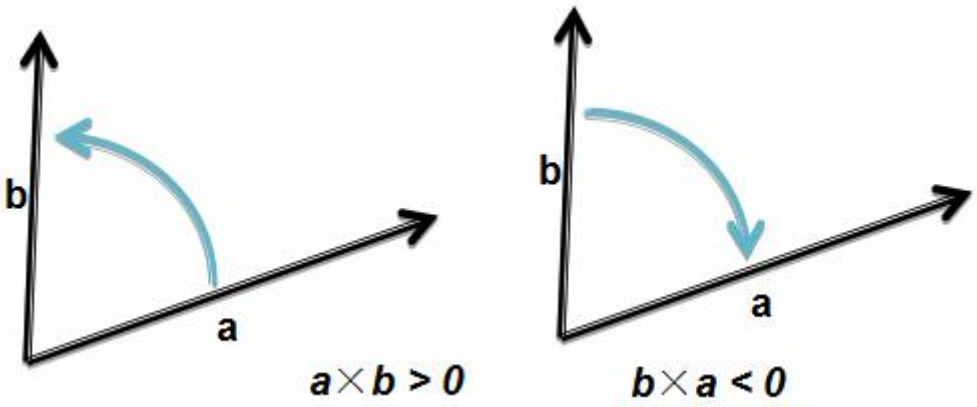
$c=a\times b$。$c=\begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}$，其中$i,j,k$分别是三个轴上的单位向量。

根据叉积的计算式子，c的模长等于以a,b为两条邻边所作成的平行四边形的面积，c的方向遵循右手定则（右手除大拇指外四指与a平行，方向与a一致，大拇指与a垂直，然后四指转过一个小于π的角到达b，此时大拇指的方向就是c的方向）。



二维叉积：若$a=(x_1,y_1),b=(x_2,y_2)$，则我们可以将它们看做是z轴为0的两个三维向量，根据定义我们可知叉积结果为$(0,0,x_1y_2-x_2y_1)$。因此我们定义二维叉积为$a\times b=x_1y_2-x_2y_1$，它的几何意义是，叉积结果大小等于以a,b为两条邻边所作成的平行四边形的**有向面积**，即$a\times b=|a|\times|b|\times\sin<a,b>$。

根据叉积结果，我们能判断a,b的方向。



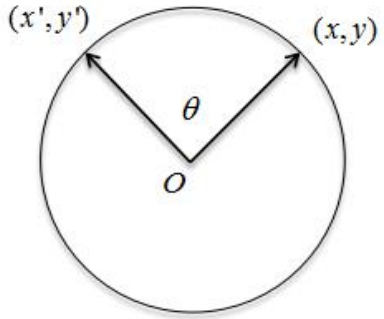
并且我们也可以推出$a\times b=-b\times a$　$a\times b=0\Leftrightarrow a,b$共线。

向量的旋转：向量 $a=(x,y)$（沿起点）逆时针旋转 θ（弧度），得到的向量 b 的坐标为 $b=(x\cos\theta-y\sin\theta,x\sin\theta+y\cos\theta)$。我们可以借助两个复数相乘，积的模等于两复数模的积，积的辐角等于两复数辐角的和这个性质，将旋转看做是两个复数相乘，即 $(x+yi)$ 与 $(\cos\theta+i\sin\theta)$ 相乘，再利用复数乘法式子 $(a+bi)\times(c+di)=(ac-bd)+(ad+bc)i$ 得出上面向量旋转的结果。（复数中的 i 代表 $\sqrt{-1}$）。

也可直接用和角公式计算：

向量(x,y)逆时针旋转θ度

$$x=\cos A*L$$
$$y=\sin A*L$$
$$\cos(A+\theta)=\cos A\cos\theta-\sin A\sin\theta$$
$$\sin(A+\theta)=\sin A\cos\theta+\cos A\sin\theta$$
$$x'=L\cos(A+\theta)=x\cos\theta-y\sin\theta$$
$$y'=L\sin(A+\theta)=y\cos\theta+x\sin\theta$$



二、基础问题

1. $a=(x_1,y_1),b=(x_2,y_2)$ 两点距离：$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$

2. 三角形 ABC 面积：

(1) 叉积：$2S_{\Delta ABC}=\left|\overrightarrow{AB}\times\overrightarrow{AC}\right|=\left|\overrightarrow{BA}\times\overrightarrow{BC}\right|=\left|\overrightarrow{CA}\times\overrightarrow{CB}\right|$

(2) 海伦公式：$p=\dfrac{a+b+c}{2},S=\sqrt{p(p-a)(p-b)(p-c)}$，$a,b,c$ 为三边长度。

3. 判断点 P 是否在线段 AB 上：判断是否 $|PA|+|PB|=|AB|$

4. 判断点 P 是否在直线 AB 上（三点共线）：判断是否 $\overrightarrow{PA}\times\overrightarrow{PB}=0$

5. 判断连续两条线段 AB, BC 的拐向：叉积判断，如 $\overrightarrow{AB} \times \overrightarrow{AC} > 0 \Leftrightarrow$ 逆时针拐弯

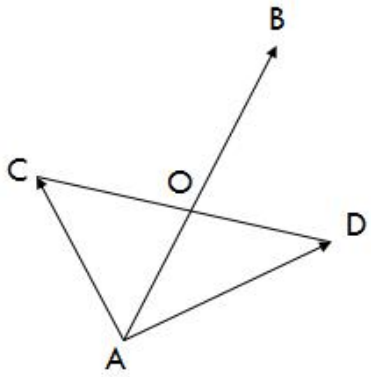
6. 求 $\angle AOB$ 的种类（锐角、直角、钝角）：$\overrightarrow{OA} \cdot \overrightarrow{OB} = 0 \Leftrightarrow$ 直角，$\overrightarrow{OA} \cdot \overrightarrow{OB} > 0 \Leftrightarrow$ 锐角，$\overrightarrow{OA} \cdot \overrightarrow{OB} < 0 \Leftrightarrow$ 钝角

7. 求点 P 到线段 AB 的最短距离：

(1) 若 $\angle BAP, \angle ABP$ 有一个是钝角，则最短距离为 $\min\left(\left|PA\right|, \left|PB\right|\right)$

(2) 否则最短距离为三角形 PAB 底边 AB 的高：$\dfrac{\left|\overrightarrow{PA} \times \overrightarrow{PB}\right|}{\left|\overrightarrow{AB}\right|}$

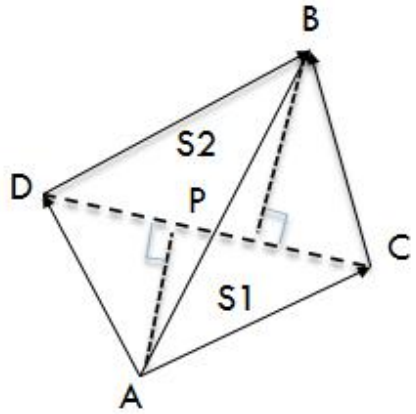
8. 判断两线段 AB, CD 是否相交：判断 A, B 是否在 CD 两边，以及 C, D 是否在 AB 两边，还要注意共线的特殊情况：若 AB，CD 不共线，则判断：$\left(\overrightarrow{CD} \times \overrightarrow{CA}\right) \times \left(\overrightarrow{CD} \times \overrightarrow{CB}\right) \le 0$ 且 $\left(\overrightarrow{AB} \times \overrightarrow{AC}\right) \times \left(\overrightarrow{AB} \times \overrightarrow{AD}\right) \le 0$；否则判断 AB 是否有至少一个点在线段 CD 上。



9. 求两直线（线段）交点（假设有交点且两线不共线）：

(1) 代数方法：求出两条直线的方程式，列方程组求解。

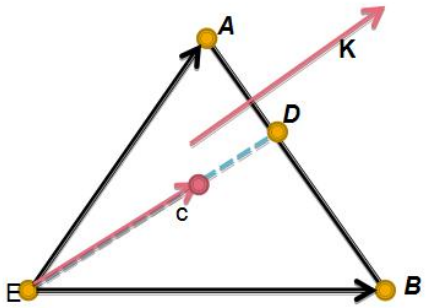
(2) 几何方法：$S1 = \overrightarrow{AC} \times \overrightarrow{AD}, S2 = \overrightarrow{BD} \times \overrightarrow{BC}, P = A + \dfrac{\overrightarrow{AB} \times S1}{S1 + S2}$



它在其它情况下也是对的，比如某条线段的两个点在另一条线段的同一边等。这种情况方便记忆与计算。求直线交点时只需任取直线上两点当做 A,B,C,D 即可。

10. 求点 E 到直线 AB 的垂足 D：

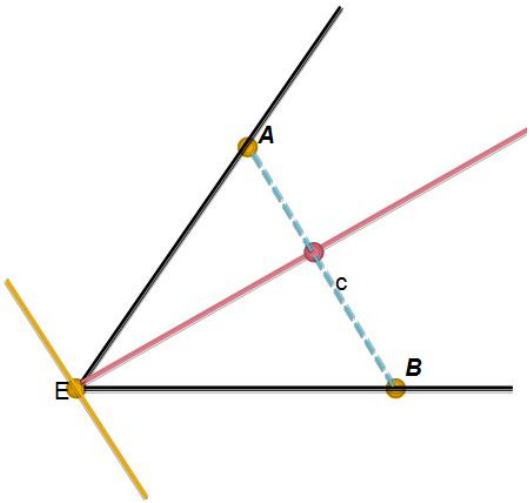
(1) $ED \perp AB$，因此将 $\overrightarrow{AB}$ 旋转 $\frac{\pi}{2}$，将问题变为直线求交。



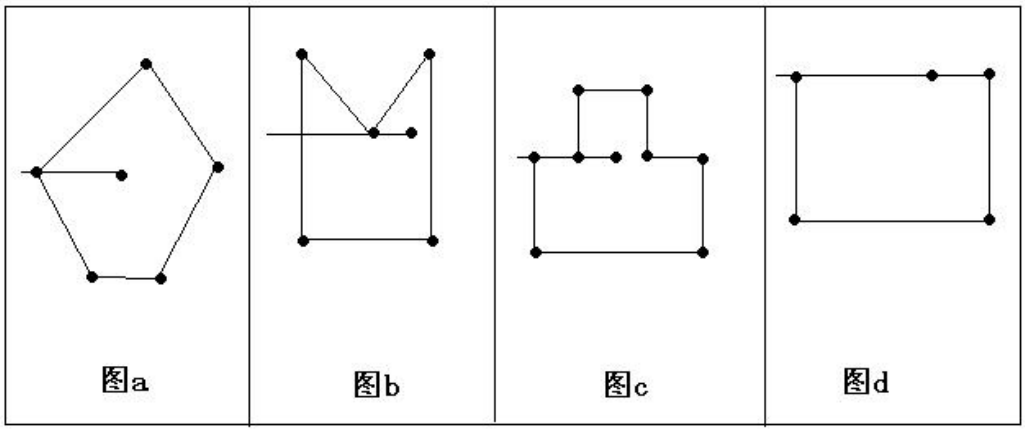
(2)
$$r = \frac{\overrightarrow{AE} \cdot \overrightarrow{AB}}{\overrightarrow{AB} \cdot \overrightarrow{AB}}$$
$$D = A + \overrightarrow{AB} \times r$$

11. 求 $\angle E$ 的角平分线：

做两点 A,B 使得 $EA = EB$，则此时将 $\overrightarrow{AB}$ 旋转 $\frac{\pi}{2}$ 即可。



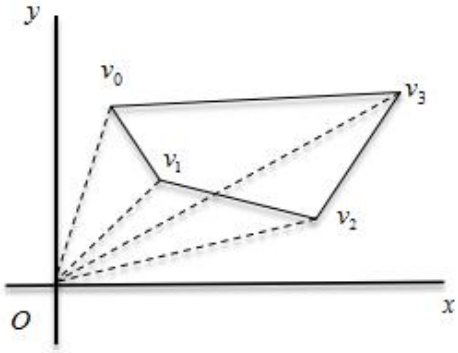
12. 求一个点 P 是否在多边形 Q 内部：射线法，过 P 做一条平行于 x 轴的射线 L，若与多边形有奇数个交点，则点在多边形内部，这可以转化为求 L 与 Q 的每一条边是否有交点，但要注意一些特殊情况，比如交点时多边形的顶点，这时我们需要规定交在边的下端点统计进答案或是交在边的上端点统计进答案（也就是保证一个点要么都被统计要么都不被统计）。



13. 多边形面积：利用叉积的几何意义（有向面积）求解。

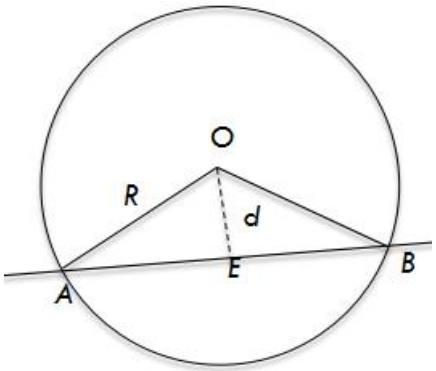
$$v_0, v_1, \ldots, v_{n-1}$$

$$S = \frac{1}{2}\left|\sum_{i=0}^{n-1} v_i \times v_{(i+1) \bmod n}\right|$$

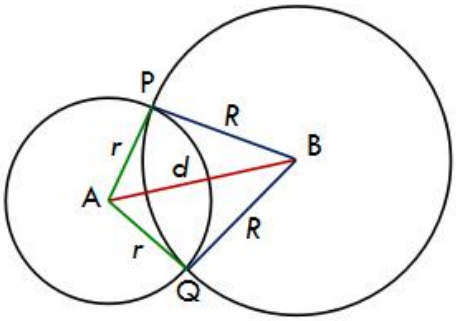


这个公式对无论什么形状的多边形都成立，它把图形分割为三角形，按这个方法分出的三角形中，有的面积是负的，即统计的是有向面积的和，按上述公式，矢量的端点沿着多边形转一圈，多边形内的区域被矢量扫过奇数次，面积被记入总和，形外的区域被扫过偶数次，两个相反方向的矢量积相互抵消，于是就得到了多边形的面积。

14. 圆与直线求交点：求圆心到直线的距离 d，根据半径 R 以及 d 求 $\angle AOE$，然后再以 O 为中心顺时针、逆时针分别旋转 $\overrightarrow{OE}$，再利用缩放得出 $\overrightarrow{OA}, \overrightarrow{OB}$，进而求出 A, B



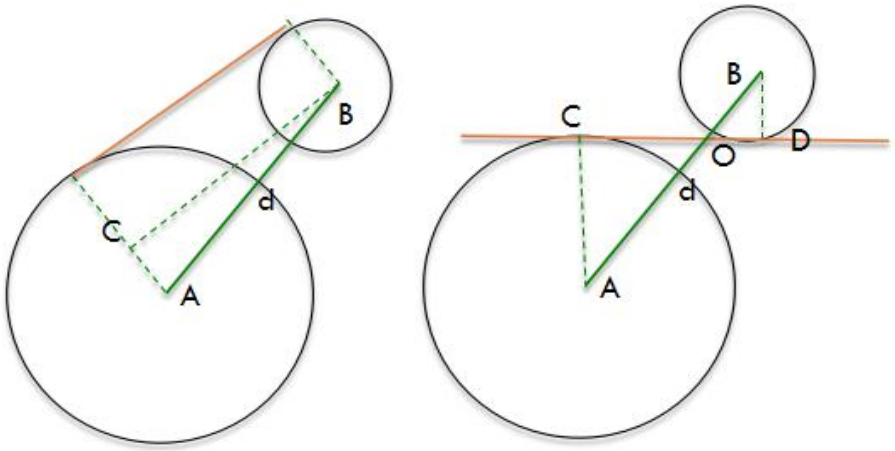
15. 圆与圆求交点：已知大小半径 R, r，并且可以利用圆心坐标求出圆心距 d，利用圆心距与半径的关系可以先判断两圆关系，若有交点则我们可以利用余弦定理求出 $\angle BAP$，再以 A 为中心旋转、缩放 $\overrightarrow{AB}$ 得到 $\overrightarrow{AP}, \overrightarrow{AQ}$，进而求出 P, Q。



16. 求两圆公切线：考虑求出两个切点，分两种情况讨论：

(1) AC 长度为两圆半径差，$\angle ACB$ 是直角，进而可知 $\angle BAC$ 大小，将 $\overrightarrow{AB}$ 旋转至 $\overrightarrow{AC}$ 再进行缩放得到切点坐标。另一个切点同理可求。（左图）

(2)由两圆半径比例可得 AO 长度，$\angle ACO$ 是直角且 AC 是半径，因此可求 $\angle OAC$,将 $\overrightarrow{AO}$ 旋转缩放得到 $\overrightarrow{AC}$，求出切点坐标。另一个切点同理可求。（右图）



例题

Points Within (ZOJ 1081)

题目大意：

给定一个点数为 n 的多边形，点按照顺序给出，再给出 m 个点，询问每个点是否在多边形内。 n <= 100

分析：

射线法。

代码：

```
#include <cstdio>
#include <algorithm>

const int N = 103;
int n, m;

struct point {
  int x, y;
  point() {}
  point(int _x, int _y) :
    x(_x), y(_y) {}
  friend inline point operator - (const point &lhs, const point &rhs) {
    return point(rhs.x - lhs.x, rhs.y - lhs.y);
  }
  friend inline int operator * (const point &lhs, const point &rhs) {
    return lhs.x * rhs.y - lhs.y * rhs.x;
  }
  friend inline int dot(const point &lhs, const point &rhs) {
    return lhs.x * rhs.x + lhs.y * rhs.y;
  }
} q;

// 判断点是否在线段上
inline bool check(const point &u, const point &v, const point &p) {
  int det = (u - p) * (v - p);
  if (det != 0) return false;
  int Dot = dot(u - p, v - p);
  return Dot <= 0;
}

struct polygon {
  int n;
  point p[N];
  void init(int _n) {
    n = _n;
    for (int i = 0; i < n; ++i) scanf("%d%d", &p[i].x, &p[i].y);
    p[n] = p[0];
    if (Area() < 0) std::reverse(p, p + n);
    p[n] = p[0];
  }
  inline int Area() const {
```

```
    int res = 0;
    for (int i = 0; i < n; ++i) res += p[i] * p[i + 1];
    return res;
  }
  bool inner(const point &q) {
    int cnt = 0;
    for (int i = 0; i < n; ++i) {
      if (check(p[i], p[i + 1], q)) return true;
      int d1 = p[i].y - q.y, d2 = p[i + 1].y - q.y;
      int det = (p[i] - q) * (p[i + 1] - q);
      if ((det >= 0 && d1 < 0 && d2 >= 0) ||
          (det <= 0 && d1 >= 0 && d2 < 0)) ++cnt;  // 判断射线是否穿过此边
    }
    return cnt & 1;
  }
} P;

int main() {
  for (int tt = 1; ; ++tt) {
    scanf("%d", &n);
    if (n == 0) break;
    scanf("%d", &m);
    P.init(n);
    if (tt != 1) printf("\n");
    printf("Problem %d:\n", tt);
    while (m--) {
      scanf("%d%d", &q.x, &q.y);
      if (P.inner(q)) puts("Within");
      else puts("Outside");
    }
  }
  return 0;
}
```

练习题

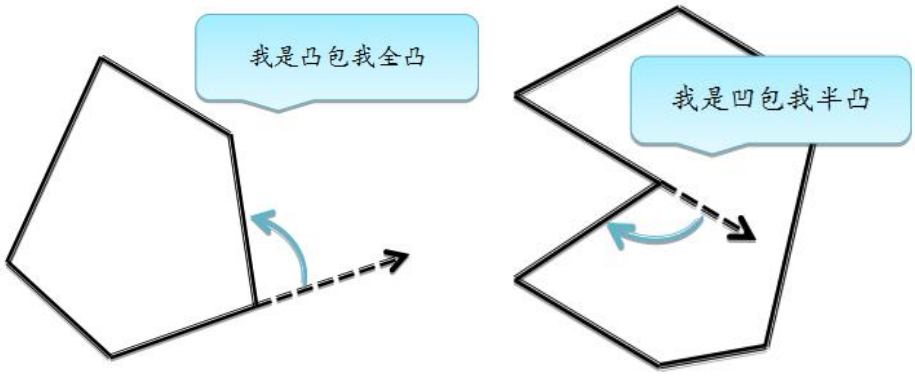
POJ 1269 ; POJ 3907 ; POJ 2318 ; POJ 3304 ; POJ 1066 ; POJ 1039 ; POJ 2074 ; POJ 1375 ;

POJ 3347 ; BZOJ 1033 ; POJ 2826 ; SGU 253

三、凸包

平面上 N 个点，用一个周长最小（面积最小）的凸多边形包含这 N 个点（所有点在其

上或者其内）。所求凸多边形就叫做这个点集的凸包。

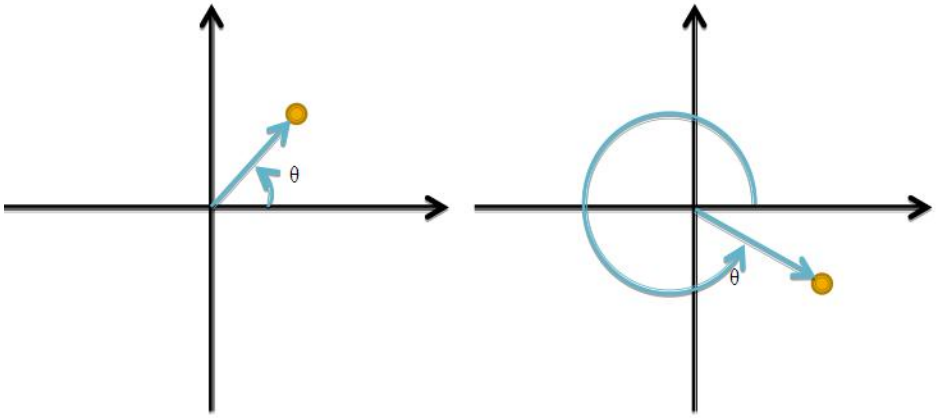


求凸包常用 Graham 扫描法，下面来介绍一下它的算法流程。

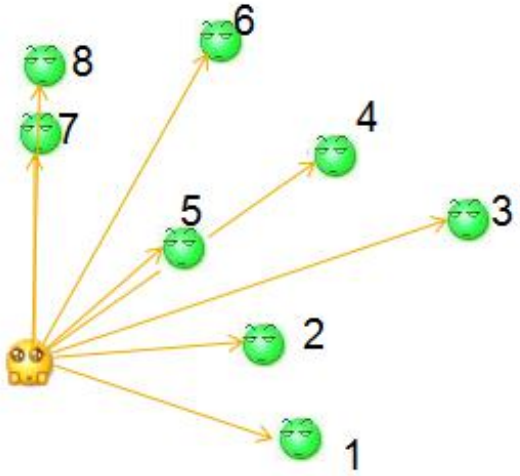
(1) 选出 x 坐标最小（相同情况 y 最小）的点作为极点，这个点必然在凸包上。

(2) 其余点进行极角排序，极角相同的情况下比较与极点距离（近到远）。

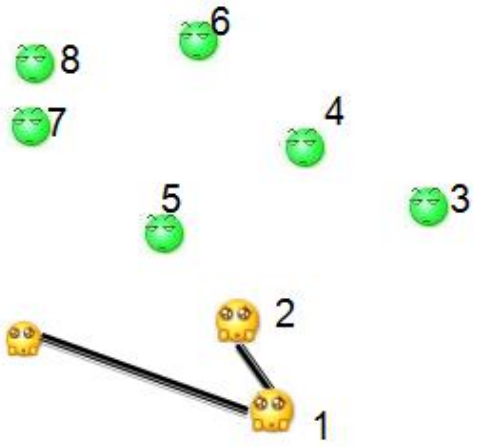
极角如下图所示，是向量从 x 轴正方向逆时针旋转过的角度。平常使用极角时极轴不一定为 x 轴正方向，可以根据情况选择其他轴。



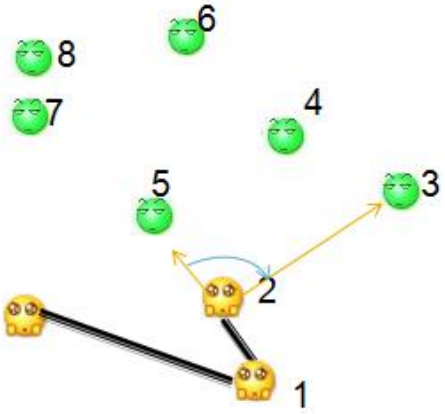
例子如下图，数字大小即排序结果。这一步可以通过叉积来实现，不需要真的求出极角：



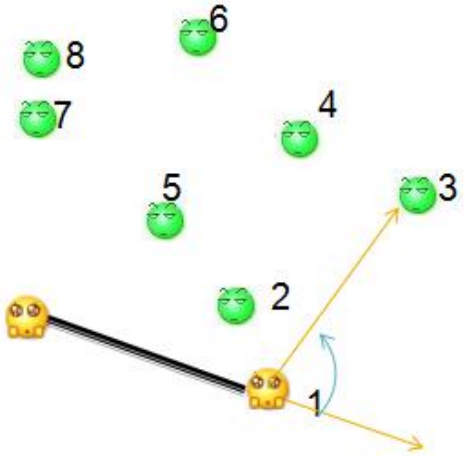
(3) 用一个栈 S 储存凸包上的点，先将选出的极点和极角序最小的 2 个点依次入栈。



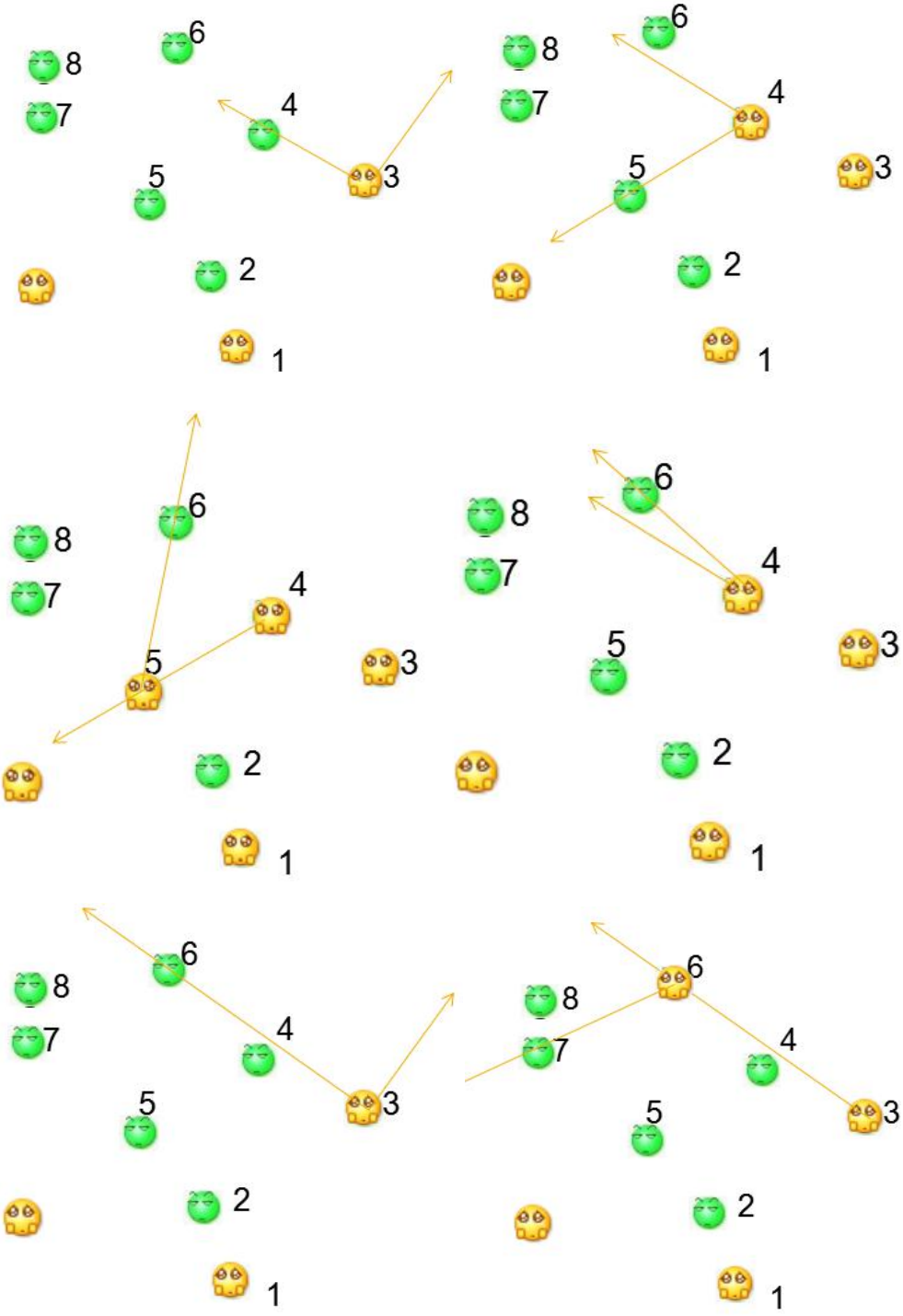
(4) 按序扫描每个点，检查栈顶的前两个元素与这个点构成的折线段是否“拐”向右（顺时针）侧（叉积小于 0）

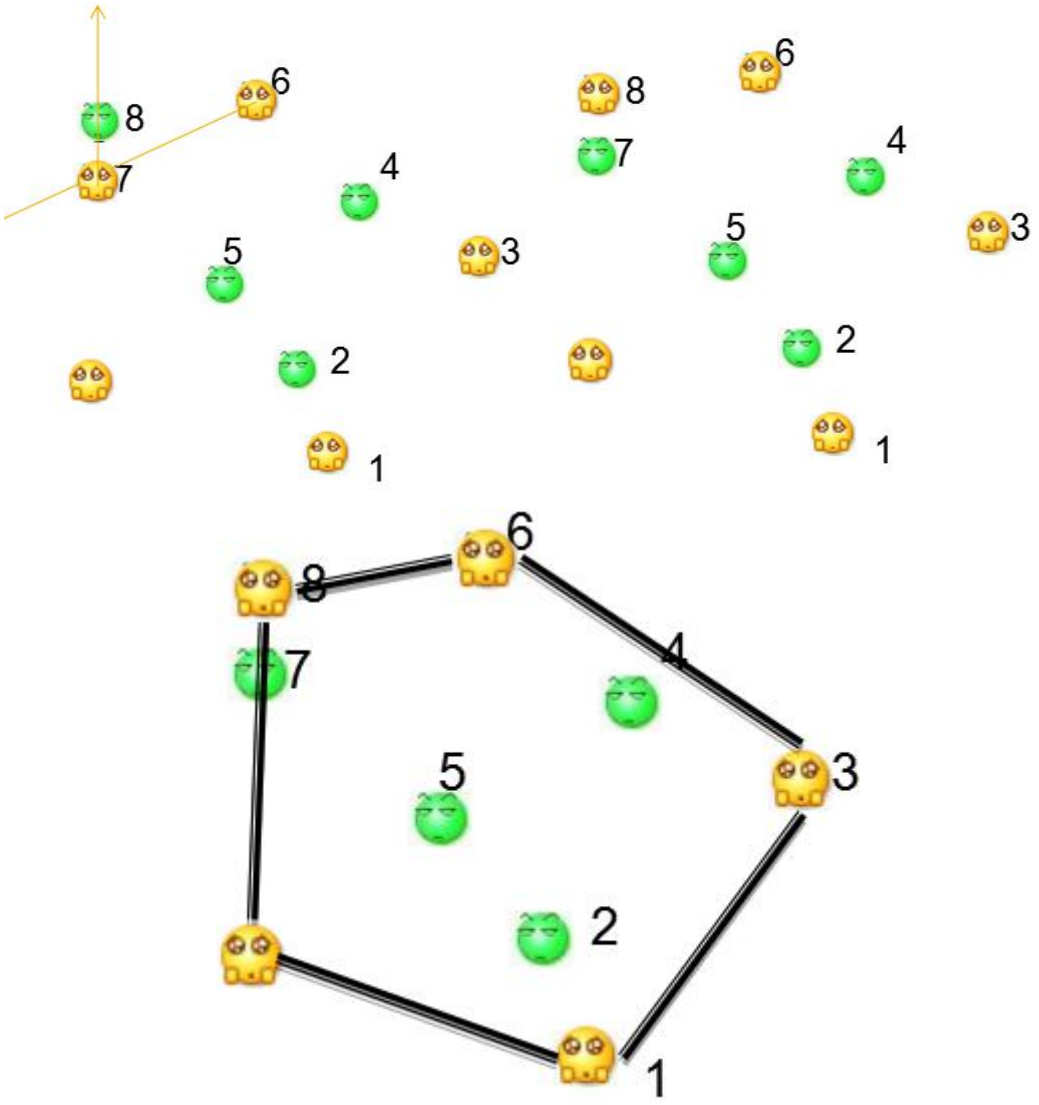


(5) 如果满足则弹出栈顶元素，并再次检查，直至不满足或者栈里面不满 2 个点。



(6) 该点入栈，并对其余点不断执行此操作。


8
6
7
4
5
3
2
1
8
6
7
4
5
3
2
1
6
8
4
7
5
3
2
1
8
6
7
4
5
3
2
1
8
6
7
4
5
3
2
1
8
6
7
4
5
3
2
1



求凸包的过程中，一个点出栈入栈都最多一次，因此扫描部分的时间复杂度为 $O(n)$ 。

算法的瓶颈在于前面的极角排序，因此 Graham 扫描法时间复杂度为 $O(n\log n)$ 。

例题

Cows (POJ 3348)

题目大意：

给定草地上 n 个点的坐标，现在要用绳子圈出一个尽可能大的面积出来养牛，已知每只牛需要 50 单位的面积，问最多能养几只牛。

n <= 10000

分析：

首先求出点集的凸包，再求这个凸包的面积 S，则答案即为 S/50.

代码：

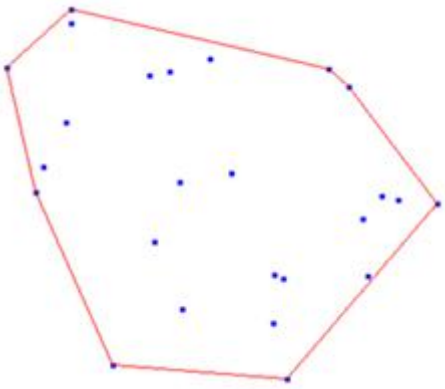
```
#include <cmath>
#include <cstdio>
#include <algorithm>

const int N = 1e4 + 5;
int n, m;
struct point {
  int x, y;
  point() {}
  point(int _x, int _y) :
    x(_x), y(_y) {}
  friend inline point operator - (const point &lhs, const point &rhs) {
    return point(lhs.x - rhs.x, lhs.y - rhs.y);
  }
  friend inline int operator * (const point &lhs, const point &rhs) {
    return lhs.x * rhs.y - lhs.y * rhs.x;  // 叉积
  }
  inline int norm() const {
    return x * x + y * y;
  }
} p[N], q[N];

inline bool cmp(int u, int v) {
  int det = (p[u] - p[1]) * (p[v] - p[1]);
  if (det != 0) return det > 0;     // det>0 则 v 在 u 的逆时针方向
  return (p[u] - p[1]).norm() < (p[v] - p[1]).norm();
}

void Graham() {
  int id = 1;
  for (int i = 2; i <= n; ++i)
    if (p[i].x < p[id].x || (p[i].x == p[id].x && p[i].y < p[id].y))
      id = i;
  if (id != 1) std::swap(p[1], p[id]);

  static int per[N];
  for (int i = 1; i <= n; ++i) per[i] = i;
  std::sort(per + 2, per + n + 1, cmp);

  q[++m] = p[1];
  for (int i = 2; i <= n; ++i) {
```

```
    int j = per[i];
    while (m >= 2 && (p[j] - q[m - 1]) * (q[m] - q[m - 1]) >= 0) --m;
    // 判断拐向
    q[++m] = p[j];
  }
}

int Area() {
  int res = 0;
  q[m + 1] = q[1];
  for (int i = 1; i <= m; ++i)
    res += q[i] * q[i + 1];
  return res / 2;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) scanf("%d%d", &p[i].x, &p[i].y);
  Graham();

  int ans = Area() / 50;
  printf("%d\n", ans);
  return 0;
}
```

四、旋转卡壳

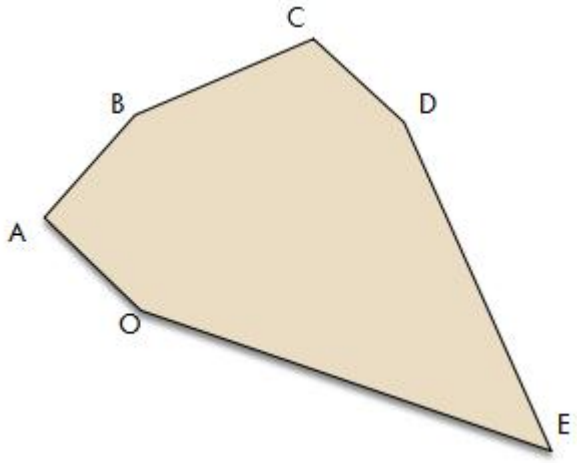
经典问题：求平面上 n 个点的最远点对。

首先有个很直观的结论：最远点对的两点一定在凸包上。如下图，实际上这个最远点对的距离也叫做这个凸包的直径。



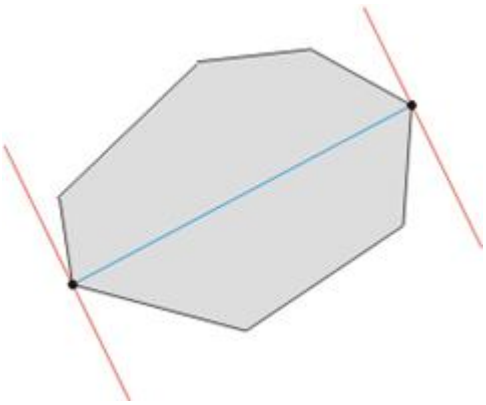
直观想法是凸包中一个点按顺序到其他点的距离是单峰的，因此可以用二分或三分法解决，但这有反例：

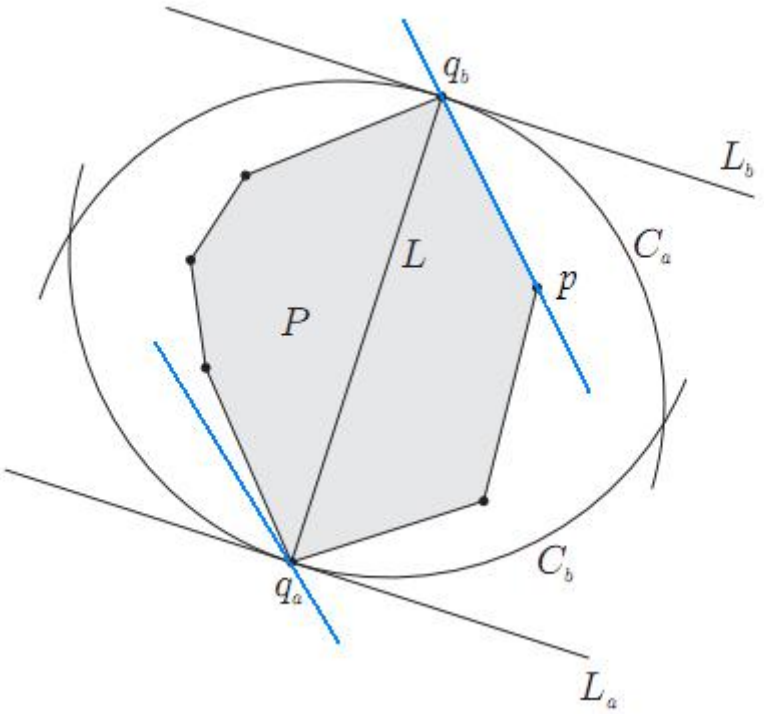
如下图，OA <OB < OC > OD < OE.



求解这个问题可以使用旋转卡壳，下面来介绍一下这个算法。

可以想象有两条平行线，“卡”住这个凸包，然后卡紧的情况下旋转一圈，肯定就能找到凸包直径，也就找到了最远的点对。因此将算法形象的称为“旋转卡壳法”。





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。这其实也是在寻找在这条边的方向上距离最远的两个点，我们称这样两个点为对踵点对。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远点继续计算，于是我们得到了 $O(n)$ 的算法。而计算距离时我们发现边是不变的，因此距离实际上就是三角形的高，所以可以使用叉积计算三角形面积，由于底边相同，三角形面积的大小情况就是距离大小情况。

《挑战》上的解释也类似：

事实上，即使坐标范围变大这道题也能求解。为此我们需要再次用到凸包的性质。假设最远点对是p和q，那么p就是点集中$(p-q)$方向最远的点，而q是点集中$(q-p)$方向最远的点。因此，可以按照逆时针逐渐改变方向，同时枚举出所有对于某个方向上最远的点对[1]，那么最远点对一定也包含于其中。在逐渐改变方向的过程中，对踵点对只有在方向等于凸包某条边的法线方向时发生变化，此时点将向凸包上对应的相邻点移动。令方向逆时针旋转一周，那么对踵点对也在凸包上转了一周，这样就可以在凸包顶点数的线性时间内求得最远点对。像这样，在凸包上旋转扫描的方法又叫做旋转卡壳法[2]。

① 这样的点对又叫做对踵点对。——译者注

② Rotating calipers通常被称为旋转卡（qiǎ）壳（ké），其名称来源于算法的过程就像用游标卡尺卡着凸包旋转一周一样。事实上卡壳这个中文单词并无此意，在这里可能是一个误用。当然如果认为这里的卡壳是短语，壳指代凸包的话，正确的读法应该是卡（kǎ）壳。也有将该方法直译为旋转卡（kǎ）尺的。——译者注

例题

Beauty Contest (POJ 2187)

题目大意：

给定 n 个点，求距离最远的两个点之间的距离，输出最远距离的平方。

分析：

模板题。

代码：

```
#include <cmath>
#include <cstdio>
#include <algorithm>

inline void chkmax(int &a, const int &b) { if (a < b) a = b; }

const int N = 5e4 + 5;
int n, m;
struct point {
  int x, y;
  point() {}
  point(int _x, int _y) :
    x(_x), y(_y) {}
  friend inline point operator - (const point &lhs, const point &rhs) {
    return point(lhs.x - rhs.x, lhs.y - rhs.y);
  }
  friend inline int operator * (const point &lhs, const point &rhs) {
    return lhs.x * rhs.y - lhs.y * rhs.x;
  }
  inline int norm() const {
    return x * x + y * y;
  }
} p[N], q[N];

inline bool cmp(int u, int v) {
  int det = (p[u] - p[1]) * (p[v] - p[1]);
  if (det != 0) return det > 0;
```

```
  return (p[u] - p[1]).norm() < (p[v] - p[1]).norm();
}

void Graham() {
  int id = 1;
  for (int i = 2; i <= n; ++i)
    if (p[i].x < p[id].x || (p[i].x == p[id].x && p[i].y < p[id].y))
      id = i;
  if (id != 1) std::swap(p[1], p[id]);

  static int per[N];
  for (int i = 1; i <= n; ++i) per[i] = i;
  std::sort(per + 2, per + n + 1, cmp);

  q[++m] = p[1];
  for (int i = 2; i <= n; ++i) {
    int j = per[i];
    while (m >= 2 && (p[j] - q[m - 1]) * (q[m] - q[m - 1]) >= 0) --m;
    q[++m] = p[j];
  }
}

inline int Area(const point &x, const point &y, const point &z) {
  return (y - x) * (z - x);
}

inline int nxt(int x) {
  if (x == m) return 1;
  return x + 1;
}

int solve() {
  if (m == 2) return (q[2] - q[1]).norm();
  int res = 0;
  q[m + 1] = q[1];
  for (int i = 1, j = 3; i <= m; ++i) {
    while (nxt(j) != i && Area(q[i], q[i + 1], q[j]) <= Area(q[i], q[i + 1],
q[j + 1])) j = nxt(j);  // 查找距离这条边最远的点
    chkmax(res, (q[j] - q[i]).norm());
    chkmax(res, (q[j] - q[i + 1]).norm());
  }
  return res;
}

```

```
int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) scanf("%d%d", &p[i].x, &p[i].y);
  Graham();
  int ans = solve();
  printf("%d\n", ans);
  return 0;
}
```

练习题

POJ 1228 ; BZOJ 1043 ; BZOJ 1185 ; POJ 3384 ; POJ 2079

上述内容参考：

1.挑战程序设计竞赛