

向量

平面向量

向量：既有大小又有方向的量称为向量，记作 \vec{a} 或 \mathbf{a} 。

有向线段：带方向的线段。用有向线段来直观地表示向量。起点为 A 终点为 B 的有向线段表示的向量，用符号简记为 \overrightarrow{AB} 。

向量的模：向量的大小（或长度），用 $|\overrightarrow{AB}|$ 或 $|\mathbf{a}|$ 表示。

零向量：模为 0 的向量。零向量的方向任意。记为： $\vec{0}$ 或 $\mathbf{0}$ 。

单位向量：模为 1 的向量称为该方向上的单位向量。

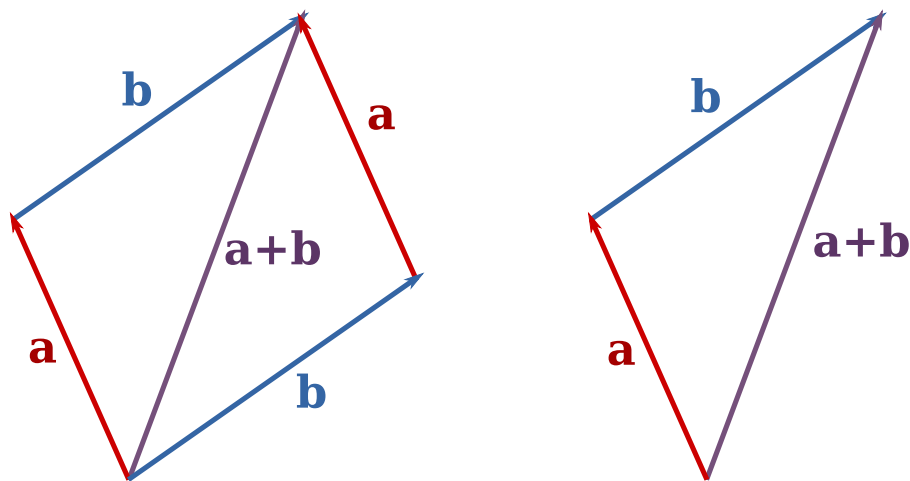
平行向量：方向相同或相反的两个 **非零** 向量。规定 $\mathbf{0}$ 与任意向量平行。 \mathbf{a} 与 \mathbf{b} 平行，记作： $\mathbf{a} \parallel \mathbf{b}$ 。

共线向量：与平行向量的定义相同。任一组平行向量都可以平移到同一直线上。

向量的夹角：已知两个非零向量 \mathbf{a}, \mathbf{b} ，作 $\overrightarrow{OA} = \mathbf{a}, \overrightarrow{OB} = \mathbf{b}$ ，那么 $\theta = \angle AOB$ 就是向量 \mathbf{a} 与向量 \mathbf{b} 的夹角。记作： $\langle \mathbf{a}, \mathbf{b} \rangle$ 。当 $\theta = \frac{\pi}{2}$ 时，称这两个向量垂直，记作 $\mathbf{a} \perp \mathbf{b}$ 。规定 $\theta \in [0, \pi]$ 。

向量的线性运算

• 向量的加法



- 向量加法的三角形法则

对于平面上的任意两个向量 \mathbf{a} 和 \mathbf{b} ，在平面内任取一点 A ，作 $\overrightarrow{AB} = \mathbf{a}$ ， $\overrightarrow{BC} = \mathbf{b}$ ，作向量 \overrightarrow{AC} 。

称向量 \overrightarrow{AC} 为向量 \mathbf{a} 和 \mathbf{b} 的和向量， $\overrightarrow{AB} + \overrightarrow{BC} = \overrightarrow{AC}$ 。

如图1，把向量首尾顺次相连，向量的和为第一个向量的起点指向最后一个向量的终点；

- 向量加法的平行四边形法则

若要求和的两个向量共起点，那么它们的和向量为以这两个向量为邻边的平行四边形的对角线，起点为两个向量共有的起点，方向沿平行四边形对角线方向。

• 向量的减法

减法可以写成加上相反数的形式，即： $a - b = a + (-b)$ ，如图1， $b = c - a$ ， $a = c - b$ 。

• 向量的数乘

给定一个实数 λ 和一个向量 \mathbf{a} ，规定其乘积为一个向量，记作 $\lambda\mathbf{a}$ ，其模与方向定义如下：

1. $|\lambda\mathbf{a}| = |\lambda||\mathbf{a}|$ ；
2. 当 $\lambda > 0$ 时， $\lambda\mathbf{a}$ 与 \mathbf{a} 同向，当 $\lambda = 0$ 时， $\lambda\mathbf{a} = \mathbf{0}$ ，当 $\lambda < 0$ 时， $\lambda\mathbf{a}$ 与 \mathbf{a} 方向相反。

这种运算是向量的数乘运算。

• 坐标表示

$$\mathbf{a} + \mathbf{b} = (a_x + b_x, a_y + b_y)$$

$$\mathbf{a} - \mathbf{b} = (a_x - b_x, a_y - b_y)$$

$$\lambda \mathbf{a} = (\lambda a_x, \lambda a_y)$$

向量的数量积

已知两个向量 \mathbf{a}, \mathbf{b} ，它们的夹角为 θ ，那么：

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos \theta = a_x b_x + a_y b_y$$

就是这两个向量的数量积，也叫点积或内积。其中称 $|\mathbf{a}| \cos \theta$ 为 \mathbf{a} 在 \mathbf{b} 方向上的投影。数量积的几何意义即为：数量积 $\mathbf{a} \cdot \mathbf{b}$ 等于 \mathbf{a} 的模与 \mathbf{b} 在 \mathbf{a} 方向上的投影的乘积。

这种运算得到的结果是一个实数，为标量。

可以方便的计算出 $\cos \theta$ ，于是有如下应用：

1. $\mathbf{a} \perp \mathbf{b} \iff \mathbf{a} \cdot \mathbf{b} = 0$
2. $\mathbf{a} = \lambda \mathbf{b} \iff |\mathbf{a} \cdot \mathbf{b}| = |\mathbf{a}||\mathbf{b}|$
3. $|\mathbf{a}| = \sqrt{m^2 + n^2}$
4. $\cos \theta = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$

向量的向量积

给定两个向量 \mathbf{a}, \mathbf{b} ，规定其向量积为一个向量，记作 $\mathbf{a} \times \mathbf{b}$ ，其模与方向定义如下：

1. $|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin \langle \mathbf{a}, \mathbf{b} \rangle$;
2. $\mathbf{a} \times \mathbf{b}$ 与 \mathbf{a}, \mathbf{b} 都垂直，且 $\mathbf{a}, \mathbf{b}, \mathbf{a} \times \mathbf{b}$ 符合右手法则。

向量积也叫外积，其几何意义是： $|\mathbf{a} \times \mathbf{b}|$ 是以 \mathbf{a}, \mathbf{b} 为邻边的平行四边形的面积。

向量积与 \mathbf{a}, \mathbf{b} 所在平面垂直，其竖坐标为 $a_x b_y - a_y b_x$ 。

我们根据右手法则可以推断出 \mathbf{b} 相对于 \mathbf{a} 的方向，逆时针方向竖坐标为正值，反之为负值。

坐标旋转公式

若将向量 $\mathbf{a} = (x, y)$ 逆时针旋转 α ，得到向量 \mathbf{b} ，则有：

$$\mathbf{b} = (x \cos \alpha - y \sin \alpha, y \cos \alpha + x \sin \alpha)$$

可以通过三角恒等变换证明。

参考代码

```
1  const double pi = 3.1415926535898;
2  const double eps = 1e-8;
3  inline int sgn(double x) {
4      if(fabs(x) < eps) return 0;
5      else return x > 0 ? 1 : -1;
6  }
7  inline int fcmp(double x, double y) {
8      if(fabs(x - y) < eps) return 0;
9      else return x > y ? 1 : -1;
10 }
11 struct Point{
12     double x, y;
13     Point(){};
14     Point(double a, double b): x(a), y(b) {}
15     Point(Point a, Point b): x(b.x - a.x), y(b.y - a.y) {}
16     friend Point operator + (const Point &a, const Point &b) {
17         return Point(a.x + b.x, a.y + b.y);
18     }
19     friend Point operator - (const Point &a, const Point &b) {
20         return Point(a.x - b.x, a.y - b.y);
21     }
22     friend bool operator == (const Point &a, const Point &b) {
23         return fcmp(a.x, b.x) == 0 && fcmp(a.y, b.y) == 0;
24     }
25     friend Point operator * (const double &a, const Point &b) {
26         return Point(a * b.x, a * b.y);
27     }
28     friend Point operator * (const Point &a, const double &b) {
29         return Point(a.x * b, a.y * b);
30     }
31     friend double operator * (const Point &a, const Point &b) {
32         return a.x * b.y - a.y * b.x;
33     }
34     friend double operator & (const Point &a, const Point &b) {
35         return a.x * b.x + a.y * b.y;
36     }
37     friend Point operator / (const Point &a, const double &b) {
38         return Point(a.x / b, a.y / b);
39     }
40     friend bool operator < (const Point &a, const Point &b) {
41         return (a.x < b.x) || (a.x == b.x && a.y < b.y);
```

```

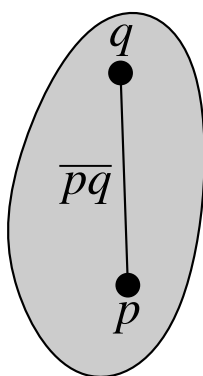
42     }
43     inline double len() {
44         return sqrt(1.0 * x * x + 1.0 * y * y);
45     }
46     friend double area(Point &a, Point &b, Point &c) {
47         return (b - a) * (c - a) / 2.0;
48     }
49 };
50 typedef Point Vec;
51 inline double dis(Point &a, Point &b) {
52     return (a - b).len();
53 }
54 inline double angle(Point &a, Point &b) {
55     return acos((a & b) / a.len() / b.len());
56 }
57 inline Vec rotate(Vec &a, double k) {
58     return Vec(a.x * cos(k) - a.y * sin(k), a.x * sin(k) - a.y * cos(k));
59 }

```

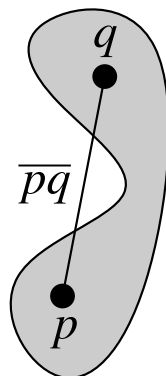
平面凸包

相关概念

凸多边形：所有内角大小都在 $[0, \pi]$ 范围内的简单多边形。

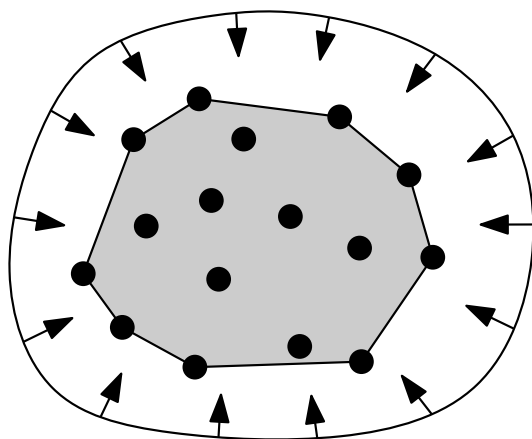


convex



not convex

平面凸包：平面上的一个子集 S 被称为凸的，当且仅当对于任意两点 $p, q \in S$ ，线段 \overline{pq} 都完全属于 S 。集合 S 的凸包 $\mathcal{CH}(S)$ ，是包含 S 的最小凸集，也就是包含 S 的所有凸集的交。



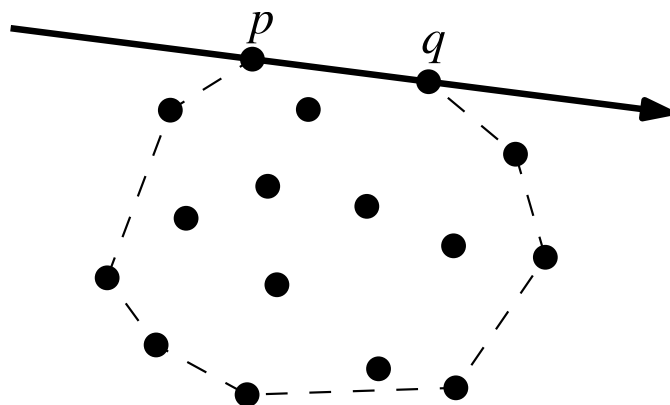
如上图，凸包还可以理解为平面上有若干柱子，用橡皮筋套住所有柱子，绷紧后形成的多边形即为凸包。

所以有更友好的定义（不一定准确）。

凸包：在平面上能包含所有给定点的最小凸多边形叫做凸包。

暴力求解

用二维坐标 (x_i, y_i) 的形式给定点集 P ，考虑如何暴力求解。



注意到，若线段 \overline{pq} 在凸包上，则 P 中的点均位于直线 pq 的同一侧。若我们钦定 $p \rightarrow q$ 按顺时针方向，则有更强的限制，需要 P 中的点都在直线的右侧。

于是可以枚举有序点对 $(p, q) \in P \times P$ ，若 P 中的点都在有向线段 \overrightarrow{pq} 的右侧，则 \overline{pq} 是 $\mathcal{CH}(P)$ 中的一条边。

需要用到向量的叉积，点 t 在 \overrightarrow{pq} 右侧 $\iff \overrightarrow{pt} \times \overrightarrow{pq} > 0$ 。

这样的复杂度是 $\mathcal{O}(n^3)$ 的，有很多可以优化的地方。

Andrew算法

Andrew 算法是一种递增式算法，流程如下。

■ 递增式算法 (incremental algorithm), 在计算几何中常见。算法思想: 逐一加入 P 中的点, 每增加一个点, 都更新一次目前的解, 加入最后一个点后, 即可得到答案。

首先把所有点 排序, 以横坐标为第一关键字, 纵坐标为第二关键字。

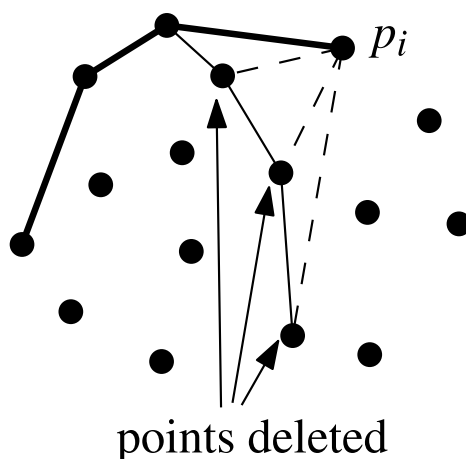
排序后, 第一个点和末尾的点, 一定在凸包上, 容易通过反证法证明。

从左往右看, 上下凸壳斜率的 单调性相反, 即所旋转的方向不同, 所以要分开求。

我们 升序枚举 求出下凸壳, 然后 降序枚举 求出上凸壳, 这样凸包的每条边都是向 逆时针方向 旋转的。

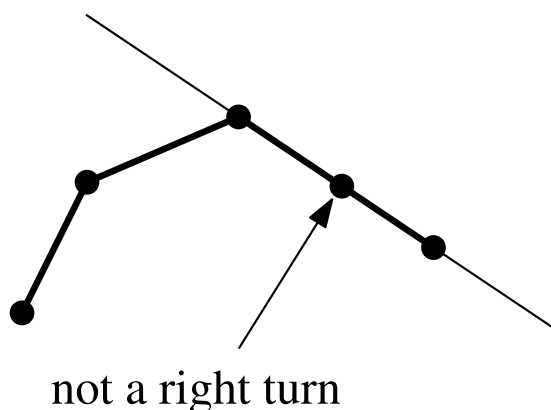
设当前枚举到点 P , 即将把其加入凸包; 当前栈顶的点为 S_1 , 栈中第二个点为 S_2 。

求凸包时, 若 P 与 S_1 构成的新线段是顺时针旋转的, 即叉积满足: $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$, 则弹出栈顶, 继续检查, 直到 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} \geq 0$ 或者栈内仅剩一个元素为止。



上图是一个弹栈的例子, p_i 是新加入的点, 细线是加入 p_i 之前的凸包状态。

记 $n = |P|$, 则时间复杂度为 $\mathcal{O}(n \log n)$, 瓶颈在排序部分。



如上图, 若将弹出栈顶元素的条件改为 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} \leq 0$, 同时停止条件改为 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} > 0$, 则求出的凸包中不存在三点共线。可视情况更改。

下面是参考代码。函数返回值为凸包的点数, `Point ret[]` 的下标从 0 开始。

```

1 inline bool check(Point s1, Point s2, Point p) {
2     return Vec(s2, s1) * Vec(s1, p) > 0;
3 }
4 int Convex_hull_2d(int n, Point *p, Point *ret) {
5     sort(p, p + n, cmp1);
6     int top = -1;
7     for (int i = 0; i < n; i++) {
8         while (top > 0 && !check(ret[top], ret[top - 1], p[i]))
9             top--;
10        ret[++top] = p[i];
11    }
12    int k = top;
13    for (int i = n - 2; i >= 0; i--) {
14        while (top > k && !check(ret[top], ret[top - 1], p[i]))
15            top--;
16        ret[++top] = p[i];
17    }
18    return top;
19 }

```

Graham算法

Andrew 算法是 Graham 算法的改进版本。在 Graham 算法中，点集按照极角序排序。

极角：任取一个顶点 O 作为极点，作射线 OX ，称为极轴。平面上一点 p 的极角，即为向量 \overrightarrow{Op} 与极轴 OX 的夹角。一般地，取 x 轴作为极轴，以逆时针方向为正。

可以利用 `atan2(double y, double x)` 进行极角排序。函数返回值为 (x, y) 与 x 轴的极角，数值 $\in (-\pi, \pi]$ 。

```

1 bool cmp(Point a, Point b) {
2     if(atan2(a.y, a.x) - atan2(b.y, b.x) == 0)
3         return a.x < b.x;
4     return atan2(a.y, a.x) < atan2(b.y, b.x);
5 }

```

另外一种方式是利用叉积排序。

```

1 bool cmp(Point a, Point b) {
2     return a * b > 0;
3 }

```


注意极角排序时，无论用 `atan2` 还是叉积，精度上都会出现不少问题，尽量避免使用这种方法。

平面凸包的周长与面积

先求出按照顺时针排序的，构成凸包的点集 p ，记 $n = |p|$ 。

求周长：把相邻两点组成的向量的模长求和，即：

$$l = \sum_{i=1}^n |\overrightarrow{p_i p_{i+1}}| + |\overrightarrow{p_1 p_n}|$$

```
1 double dis(Point a, Point b) {
2     return (a - b).len();
3 }
4 double Convex_hull_2d_L(int n, Point *p) {
5     Point convex[N];
6     int siz = Convex_hull_2d(n, p, convex);
7     double ans = dis(convex[0], convex[siz - 1]);
8     for (int i = 1; i < siz; i++)
9         ans += dis(convex[i - 1], convex[i]);
10    return ans;
11 }
```

求面积：任取凸包内一点（一般取 p_1 ），则有：

$$s = \sum_{i=2}^{n-1} \text{area}(p_1, p_i, p_{i+1}) = \sum_{i=2}^{n-1} \frac{|(p_i - p_1) \times (p_{i+1} - p_1)|}{2}$$

```
1 double area(Point a, Point b, Point c) {
2     return (b - a) * (c - a) / 2.0;
3 }
4 double Convex_hull_2d_S(int n, Point *p) {
5     Point convex[N];
6     int siz = Convex_hull_2d(n, p, convex);
7     double ans = 0;
8     for (int i = 2; i < siz; i++)
9         ans += area(convex[0], convex[i - 1], convex[i]);
10    return ans;
11 }
```

动态凸包 (CF70D)

维护一个点集 S 的凸包，需要支持如下操作：

- 询问点 p 是否在当前的凸包中，
- 向 S 中添加点 p ，

保证坐标均为整数。

分析

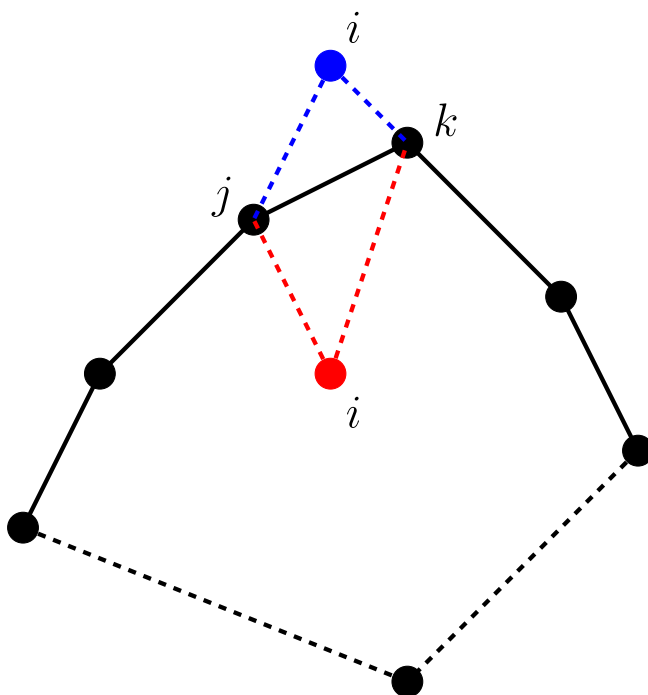
和 Andrew 算法一样，这里的算法按照坐标字典序排序。相对于极角排序，能够减小精度误差。

用两个 `std::map<int, int>`，用 top 记录上凸包，down 记录下凸包。

存储方法：若上凸包中存在横坐标为 x 的点，则这个点的纵坐标为 $\text{top}[x]$ ，down 同理。

询问操作

只需满足：在上凸包之下且在下凸包之上。



以上凸包为例。 i 在上凸包之下，当且仅当 $|\vec{ik} \times \vec{ij}| \geq 0$ 。

```

1  bool check_top(int x, int y) { //是否在上凸包下面
2      auto k = top.lower_bound(x);
3      if(k == top.end())
4          return false;
5      if(k -> first == x)
6          return y <= k->second;
7      if(k == top.begin()) return false;
8      auto j = k; j--;
9      return Point(k->first - x, k->second - y) *
10         Point(j->first - x, j->second - y) >= 0;
11 }

```

下凸包同理, i 在下凸包之上, 当且仅当 $|\vec{ik} \times \vec{ij}| \leq 0$

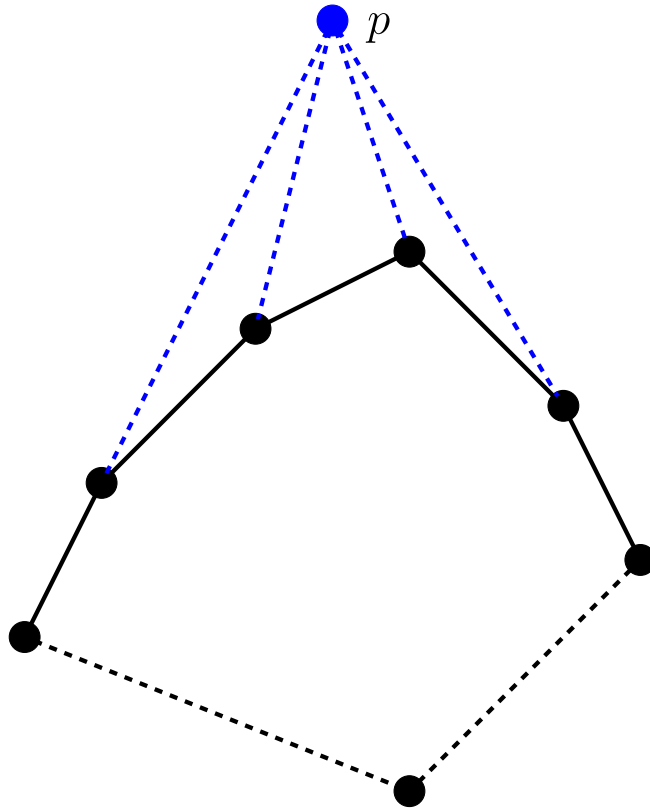
```

1  bool check_down(int x, int y) { //是否在下凸包上面
2      auto k = down.lower_bound(x);
3      if(k == down.end())
4          return false;
5      if(k -> first == x)
6          return y >= k->second;
7      if(k == down.begin()) return false;
8      auto j = k; j--;
9      return Point(k->first - x, k->second - y) *
10         Point(j->first - x, j->second - y) <= 0;
11 }

```

插入操作

把 p 点加入凸包, 上下凸包都要尝试。把加入 p 点后, 删掉不满足凸性的点。



如上图，这些点一定是分布在 p_x 左右的连续段。

因此找到 p 点在上/下凸壳中的位置，向左右分别删点，直到满足凸性。

注意迭代器的边界问题。如果已经删没了，要及时退出循环，否则会 RE。

```

1 void insert_top(int x, int y) {
2     if(check_top(x, y)) return;
3     top[x] = y;
4     auto it = top.find(x);
5     auto jt = it;
6     if(it != top.begin()) { //remove left
7         jt--;
8         while(remove_top(jt++)) jt--;
9     }
10    if(++jt != top.end()) { //remove right
11        while(remove_top(jt--)) jt++;
12    }
13 }
14 void insert_down(int x, int y) {
15     if(check_down(x, y)) return;
16     down[x] = y;
17     auto it = down.find(x);
18     auto jt = it;
19     if(it != down.begin()) { //remove left
20         jt--;
21         while(remove_down(jt++)) jt--;

```

```

22     }
23     if(++jt != down.end()) { //remove right
24         while(remove_down(jt--)) jt++;
25     }
26 }

```

下面的函数用于：判断能否删除当前点，若能删，则执行删除操作。

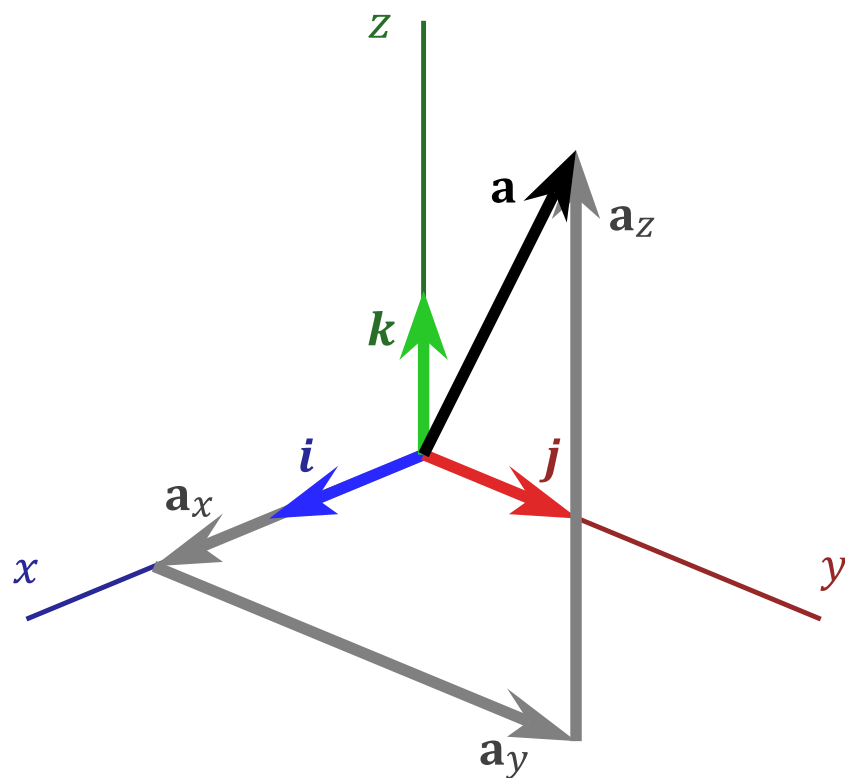
```

1  bool remove_top(map<int, int>::iterator it) {
2      if(it == top.begin()) return false; //到边界就不删了
3      if(++it == top.end()) return false; it--;
4      auto jt = it, kt = it;
5      jt--; kt++;
6      if(Point(it -> first - jt -> first, it->second - jt->second) *
7          Point(it -> first - kt -> first, it->second - kt->second) <= 0) {
8          top.erase(it);
9          return true;
10     }
11     return false;
12 }
13 bool remove_down(map<int, int>::iterator it) {
14     if(it == down.begin()) return false;
15     if(++it == down.end()) return false; it--;
16     auto jt = it, kt = it;
17     --jt; ++kt;
18     if(Point(it -> first - jt -> first, it->second - jt->second) *
19         Point(it -> first - kt -> first, it->second - kt->second) >= 0) {
20         down.erase(it);
21         return true;
22     }
23     return false;
24 }

```

三维凸包

三维向量类



存储：用结构体记录三维坐标，方便重载运算符。

```

1 struct Point3 {
2     double x, y, z;
3     Point3(){};
4     Point3(double a, double b, double c) : x(a), y(b), z(c) {}
5 };

```

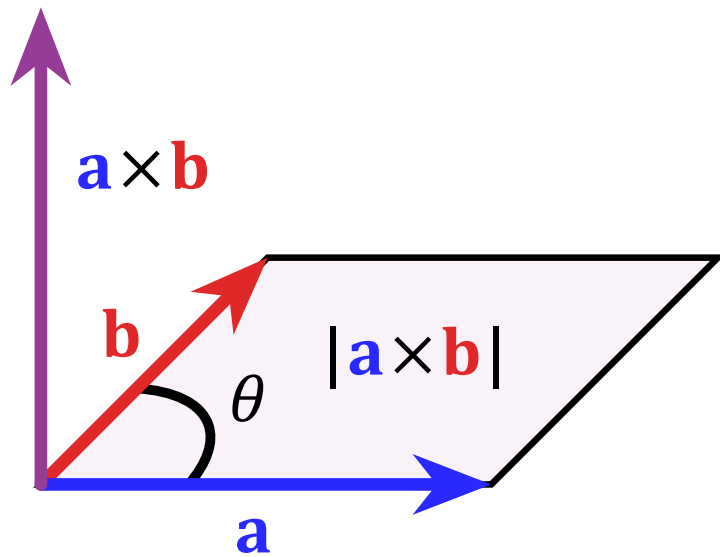
加法、减法和点乘等操作：与二维向量类似。

```

1 Point3 operator + (const Point3 &b) {
2     return Point3(x + b.x, y + b.y, z + b.z);
3 }
4 Point3 operator - (const Point3 &b) {
5     return Point3(x - b.x, y - b.y, z - b.z);
6 }
7 Point3 operator * (const Point3 &b) {
8     return Point3(y*b.z - z*b.y, z*b.x - x*b.z, x*b.y - y*b.x);
9 }
10 bool operator == (const Point3 &b) {
11     return fcmp(x, b.x) == 0 && fcmp(y, b.y) == 0 && fcmp(z, b.z) == 0;
12 }
13 double len() {
14     return sqrt(x * x + y * y + z * z);
15 }

```

叉乘： $\mathbf{a} \times \mathbf{b}$ 的结果为一个三维向量 \mathbf{c} ， $\mathbf{c} \perp \mathbf{a}$ 且 $\mathbf{c} \perp \mathbf{b}$ ，结果向量的模长为 $|\mathbf{a}||\mathbf{b}| \sin\langle \mathbf{a}, \mathbf{b} \rangle$ ，代表以 \mathbf{a} 、 \mathbf{b} 为两边的平行四边形的面积。



在三维向量体系中，我们需要用坐标表示结果向量 \mathbf{a} ，推导过程如下。（来源：[叉积 - 维基百科](#)）

右手坐标系中，基向量 $\mathbf{i}, \mathbf{j}, \mathbf{k}$ 满足以下等式：

$$\begin{aligned} \mathbf{i} \times \mathbf{j} &= \mathbf{k} \\ \mathbf{j} \times \mathbf{i} &= -\mathbf{k} \\ \mathbf{j} \times \mathbf{k} &= \mathbf{i} \\ \mathbf{k} \times \mathbf{j} &= -\mathbf{i} \\ \mathbf{k} \times \mathbf{i} &= \mathbf{j} \\ \mathbf{i} \times \mathbf{k} &= -\mathbf{j} \end{aligned}$$

根据外积的定义可以得出： $\mathbf{i} \times \mathbf{i} = \mathbf{j} \times \mathbf{j} = \mathbf{k} \times \mathbf{k} = \mathbf{0}$.

根据以上等式，结合外积的分配律，就可以确定任意向量的外积。

任取向量 $\mathbf{u} = u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k}$ 和 $\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$ ，两者的外积 $\mathbf{u} \times \mathbf{v}$ 可以根据分配率展开：

$$\begin{aligned} \mathbf{u} \times \mathbf{v} &= (u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k}) \times (v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}) \\ &= u_1v_1(\mathbf{i} \times \mathbf{i}) + u_1v_2(\mathbf{i} \times \mathbf{j}) + u_1v_3(\mathbf{i} \times \mathbf{k}) + \\ &\quad u_2v_1(\mathbf{j} \times \mathbf{i}) + u_2v_2(\mathbf{j} \times \mathbf{j}) + u_2v_3(\mathbf{j} \times \mathbf{k}) + \\ &\quad u_3v_1(\mathbf{k} \times \mathbf{i}) + u_3v_2(\mathbf{k} \times \mathbf{j}) + u_3v_3(\mathbf{k} \times \mathbf{k}) \end{aligned}$$

把前面的 6 个等式代入，则有：

$$\begin{aligned}
\mathbf{u} \times \mathbf{v} &= -u_1v_1\mathbf{0} + u_1v_2\mathbf{k} - u_1v_3\mathbf{j} \\
&\quad - u_2v_1\mathbf{k} - u_2v_2\mathbf{0} + u_2v_3\mathbf{i} \\
&\quad + u_3v_1\mathbf{j} - u_3v_2\mathbf{i} - u_3v_3\mathbf{0} \\
&= (u_2v_3 - u_3v_2)\mathbf{i} + (u_3v_1 - u_1v_3)\mathbf{j} + (u_1v_2 - u_2v_1)\mathbf{k}
\end{aligned}$$

因此结果向量 $\mathbf{s} = \mathbf{u} \times \mathbf{v} = s_1\mathbf{i} + s_2\mathbf{j} + s_3\mathbf{k}$ 的三维坐标为：

$$s_1 = u_2v_3 - u_3v_2$$

$$s_2 = u_3v_1 - u_1v_3$$

$$s_3 = u_1v_2 - u_2v_1$$

```

1 Point3 operator * (const Point3 &b) {
2     return Point3(y*b.z - z*b.y, z*b.x - x*b.z, x*b.y - y*b.x);
3 }

```

平面类

用三个向量表示一个三角形的平面。一个多面体可以通过三角剖分，用若干个三角形表示。

为了节省空间，用 `point3 p[N]` 存储所有可能出现的向量，结构体 `plane` 只记录向量在 `p[]` 中的下标。

记录的三个向量按逆时针首尾相接，这样在判断方向时比较方便。

```

1 struct plane{
2     int v[3]; //逆时针
3     plane(){};
4     plane(int a, int b, int c) { v[0] = a, v[1] = b, v[2] = c; }
5 };

```

平面的法向量：是指垂直于该平面的三维向量。一个平面具有无限个法向量，这些法向量有两个方向。

根据叉积的性质，将三角形的两条邻边叉乘，得到的向量即为法向量。

```

1 Point3 normal() {
2     return (p[v[1]] - p[v[0]]) * (p[v[2]] - p[v[0]]);
3 }

```

利用法向量的模长，也可以算出三角形的面积。


```
1 double area() {  
2     return normal().len() / 2.0;  
3 }
```

三维凸包及性质

由 n 个点构成的凸多面体。

性质：根据欧拉公式，任意包含 n 个顶点的凸多面体，所含的边不会超过 $3n - 6$ 条，所含的小平面不会超过 $2n - 4$ 张。

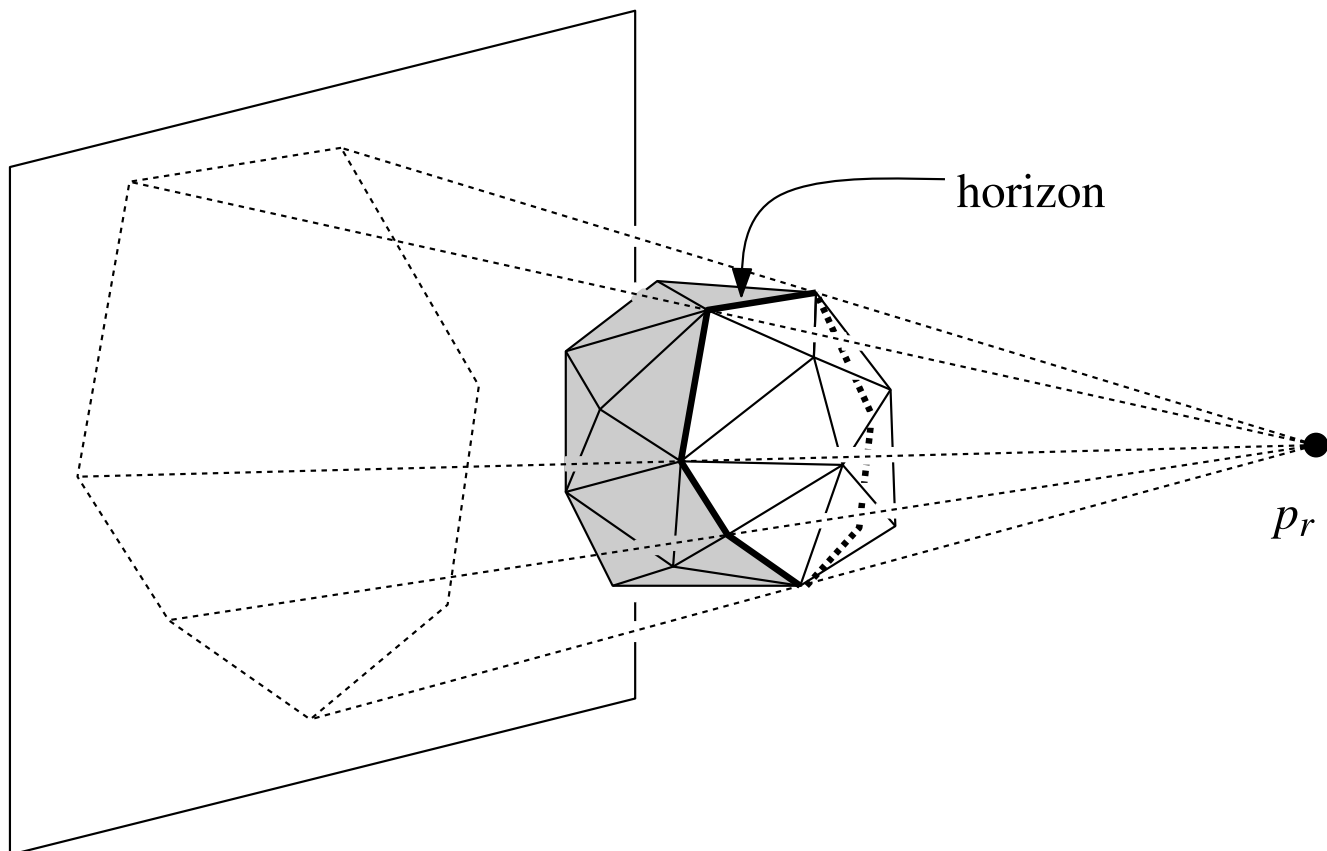
随机增量法

• 算法思想

和 Andrew 算法类似，考虑每次把 p_r 加入到前 $r - 1$ 个点的凸包中，也就是将 $\mathcal{CH}(P_{r-1})$ 转化为 $\mathcal{CH}(P_r)$ 。

第一种情况： p_r 在 $\mathcal{CH}(P_{r-1})$ 的内部或边界上，则 $\mathcal{CH}(P_{r-1}) \rightarrow \mathcal{CH}(P_r)$ 。

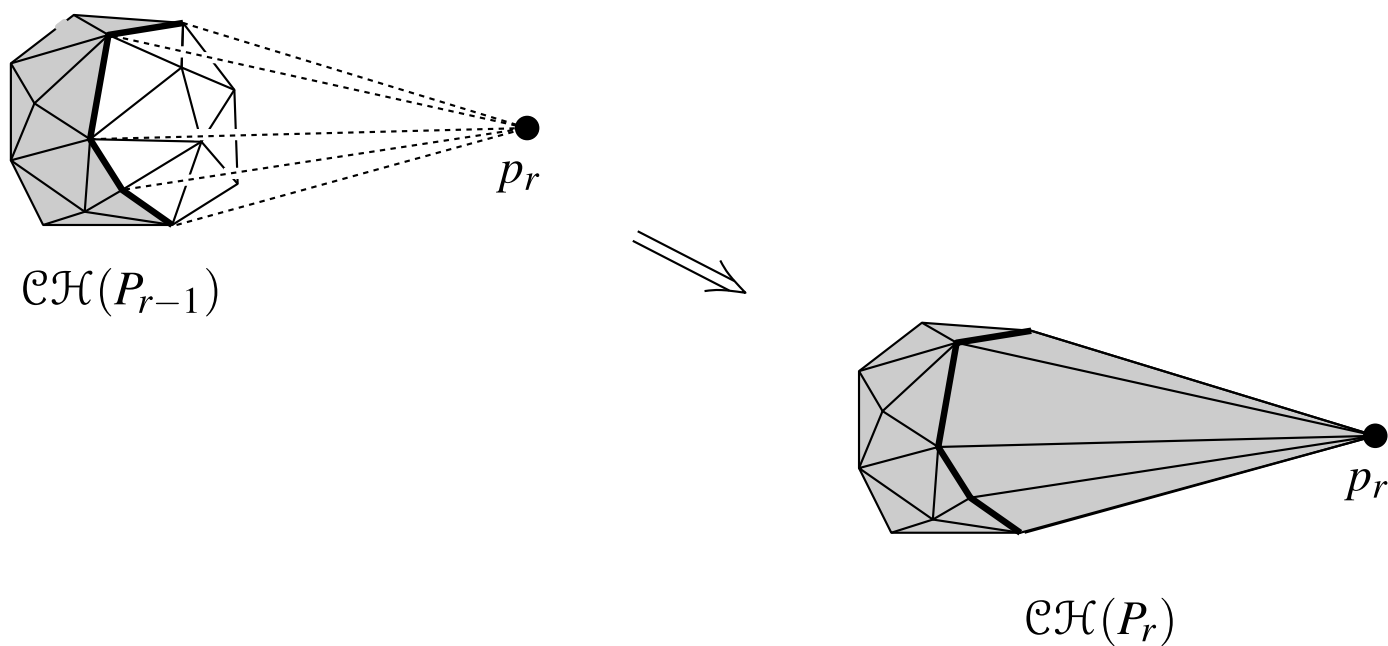
第二种情况： p_r 在 $\mathcal{CH}(P_{r-1})$ 外部。设想你站在 p_r 所在的位置，看向 $\mathcal{CH}(P_{r-1})$ 。其中的某些小平面会被看到，其余在背面的平面不会被看到。如下图，从 p_r 可见的平面构成了一片连通的区域。



这片区域由一条封闭折线围成，称这条线为 p_r 在 $\mathcal{CH}(P_{r-1})$ 上的边界线 (horizon)。

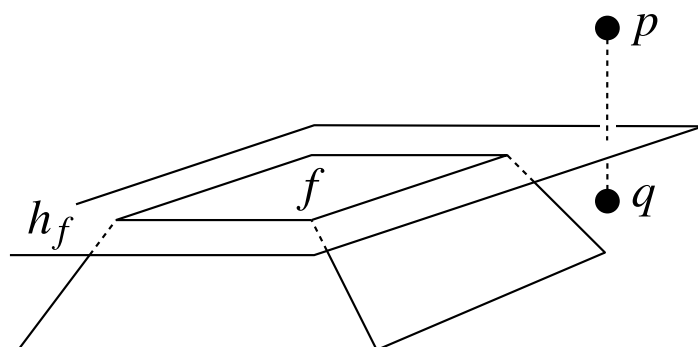
根据这条地平线，我们可以判断出，在原先 $\mathcal{CH}(P_{r-1})$ 表面上的哪些部分需要被保留，哪些需要被替换。

显然，不可见的平面在 $\mathcal{CH}(P_r)$ 中被保留，并且我们用 p_r 与地平线之间连接出新的小平面，来替换所有可见的小平面，如下图。



• 判断平面对点的可见性

如何用几何语言表达：一个平面对 p_r 是可见的？



f is visible from p ,
but not from q

对于一个凸包上的小平面，它可将空间分为两半，一侧是凸包外部，一侧是凸包内部。容易发现，如果点 p 位于小平面的外侧，那么这个平面对于 p 点就是可见的，因为凸包上的其它平面都不会有遮挡。

形式化地，记 a, b, c 为平面三角形的三个顶点，从凸包外部看，三点按照逆时针排列。

利用叉乘的性质，记 $\vec{s} = \vec{ab} \times \vec{ac}$ ，则结果向量 \vec{s} 是一个平面的法向量，且指向凸包外部。

对于空间内任意一点 p ，若 $\vec{ap} \times \vec{s} > 0$ ，则这个平面对点 p 是可见的。

下面是定义在结构体 `plane` 中的函数，用于判断点 A 是否位于平面的外侧。

```
1 bool is_above(Point3 A) {  
2     return (normal() & (A - p[v[0]])) >= 0;  
3 }
```

• 求出边界线

要想把凸包从 $\mathcal{CH}(P_{r-1})$ 转化为 $\mathcal{CH}(P_r)$ ，我们需要准确地求出凸包上的哪些边在边界线上。求出边界线之后，才能用 p 与边界线构成的小平面替换需要被删掉的小平面。

定义 `bool g[N][N]`， $g[i][j]$ 表示 $\vec{p_i p_j}$ 所在的平面是否可见。

若规定平面 (a, b, c) 只包含 $\vec{ab}, \vec{bc}, \vec{ca}$ ，则对于任意有序数对 (i, j) ，向量 $\vec{p_i p_j}$ 最多被包含在一个平面内。

注意到，位于边界线上的向量 $\overrightarrow{p_i p_j}$ 一定满足 $g[i][j] = 1$ 且 $g[j][i] = 0$ 。所以我们只需对每个平面判断其可见性，并更新在 `g[][]` 中对应的数值，即可求出边界线。

• 利用边界线更新凸包

在上一步遍历小平面的时候，若遇到的小平面是不可见的，则把它加入新的凸包中；若可见，则单独记录。

之后遍历所有可见的小平面，若 $\overrightarrow{p_i p_j}$ 在边界线上，则把 (p_i, p_j, p_r) 加入凸包中。 p_r 是新加入凸包的点。这样加入后的三点也满足逆时针排列。

• 参考代码

函数返回值为三维凸包的平面数，`plane ret[]` 的下标从 0 开始。

```
1  int Convex_hull_3d(int n, plane *ret) {
2      plane tmp[N];
3      bool g[N][N];
4      for (int i = 0; i < n; i++) p[i].shake();
5      int top = -1;
6      ret[++top] = plane(0, 1, 2);
7      ret[++top] = plane(0, 2, 1);
8      for (int i = 3; i < n; i++) {
9          int cnt = -1;
10         for (int j = 0; j <= top; j++) {
11             bool flag = ret[j].is_above(p[i]);
12             if (!flag)
13                 tmp[++cnt] = ret[j];
14             for (int k = 0; k < 3; k++)
15                 g[ret[j].v[k]][ret[j].v[(k + 1) % 3]] = flag;
16         }
17         for (int j = 0; j <= top; j++) {
18             for (int k = 0; k < 3; k++) {
19                 int a = ret[j].v[k], b = ret[j].v[(k + 1) % 3];
20                 if (g[a][b] && !g[b][a])
21                     tmp[++cnt] = plane(a, b, i);
22             }
23         }
24         for (int j = 0; j <= cnt; j++) ret[j] = tmp[j];
25         top = cnt;
26     }
27     return (top + 1);
28 }
```

旋转卡壳

概述

旋转卡壳算法用于：在线性时间内，求凸包直径、最小矩形覆盖等于凸包性质相关的问题。线性时间是指求出凸包之后的算法时间复杂度。

求凸包直径

给定平面上的 n 个点，求所有点对之间的最长距离。 $2 \leq n \leq 50000, |x|, |y| \leq 10^4$.

首先，求出这 n 个点的凸包，复杂度可以做到为 $\mathcal{O}(n \log n)$ ，如何求直径？

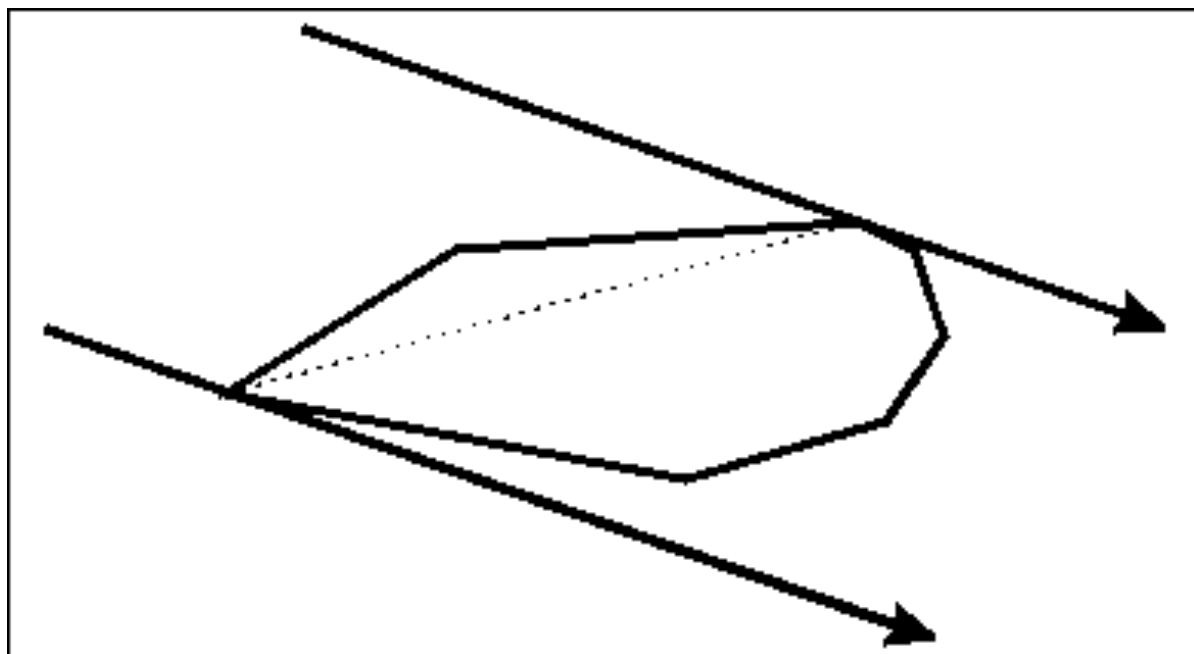
暴力做法：可以遍历每个点对，求出最大距离，复杂度为 $\mathcal{O}(n^2)$ 。

• 算法流程

可以遍历凸包上的边，对每条边 (a, b) ，去找距离这条边最远的点 p 。对于 p 点，距离它最远的点，一定是 a, b 中的一个。

我们发现，若逆时针遍历凸包上的边，那么随着边的转动，对应的最远点也在逆时针旋转。

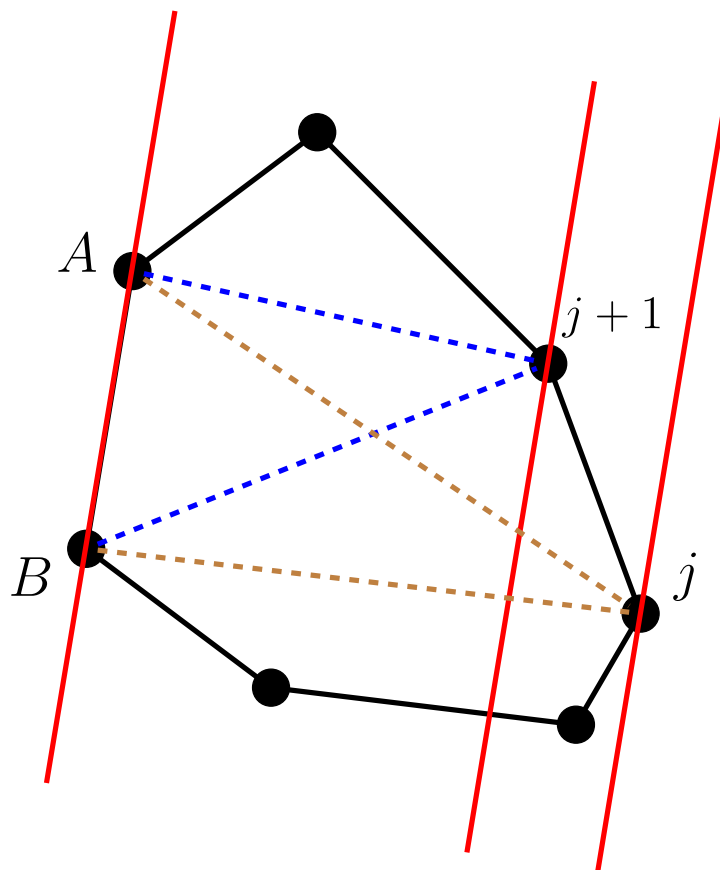
因此，我们可以在逆时针枚举边的同时，实时维护最远点，利用单调性，复杂度为 $\mathcal{O}(n)$ 。



• 算法实现

求凸包时，若使用 Andrew 算法，则凸包上的点已经按照逆时针排序了。问题在如何判断下一个点到当前边的距离是否更大。

一种方法是用点到直线距离公式，但利用叉积的精度更高，也更方便。



```
1 struct Point{
2     int x, y;
3     //.....
4     int sqr_len() { return x * x + y * y; }
5 };
6 inline int sqr_dis(Point a, Point b) { return (a - b).sqr_len(); }
7 int Get_Max(int n, Point *ch) { //传入convex-hull
8     int ret = 0;
9     ch[n] = ch[0];
10    int j = 1;
11    for(int i = 0; i < n; i++) {
12        while((ch[i] - ch[j+1]) * (ch[i+1] - ch[j+1]) >
13              (ch[i] - ch[j]) * (ch[i+1] - ch[j]))
14            j = (j + 1) % n;
15        ret = max(ret, max(sqr_dis(ch[i], ch[j]), sqr_dis(ch[i+1],
16        ch[j]))));
17    }
18    return ret;
19 }
```

最小矩形覆盖

• 题意

给定一些点的坐标，求能够覆盖所有点的最小面积的矩形。

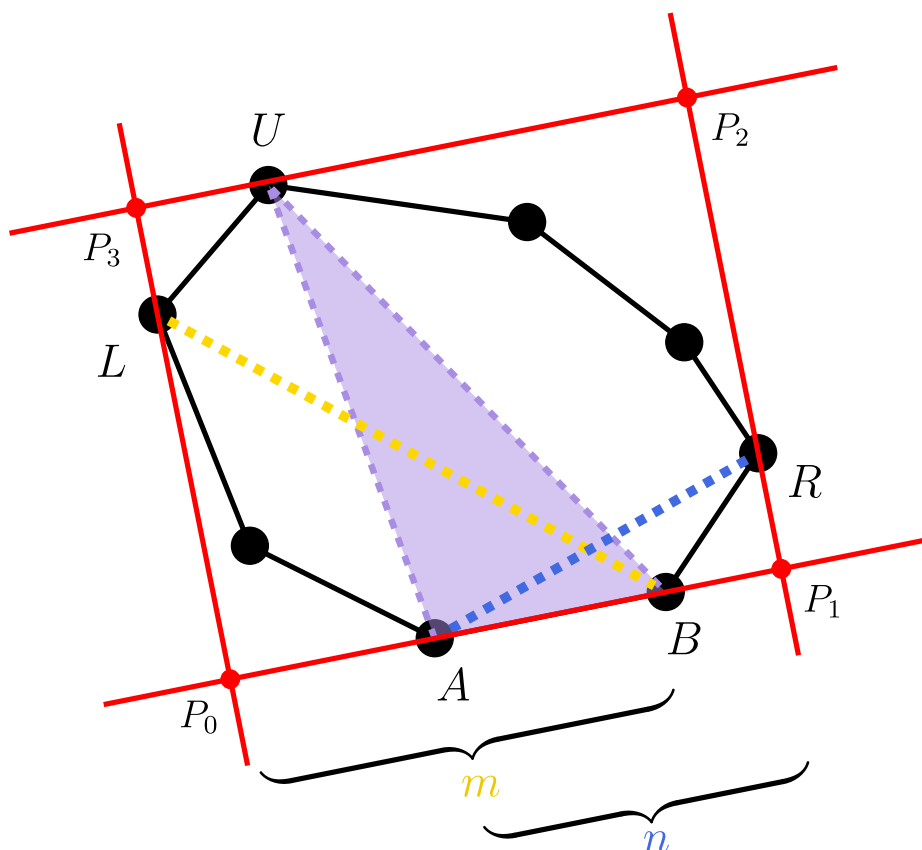
求出矩形面积、顶点坐标。 $3 \leq n \leq 50000$.

• 分析

先求出凸包，之后用旋转卡壳维护三个边界点。

利用叉积和点积，可以求出矩形面积及四个顶点的坐标。

• 实现



下面是一种可行的表示方法。

$$\text{设 } H = |P_0P_3| = \frac{|\vec{AB} \times \vec{BU}|}{|\vec{AB}|}, \quad L = \frac{|\vec{BA} \cdot \vec{AL}|}{|\vec{BA}|}, \quad R = \frac{|\vec{AB} \cdot \vec{BR}|}{|\vec{AB}|}.$$

则矩形面积为：

$$S = H \times (L + |\overrightarrow{AB}| + R)$$

顶点坐标为：

$$\overrightarrow{P_0} = \overrightarrow{A} + L \times \frac{\overrightarrow{BA}}{|\overrightarrow{BA}|}$$

$$\overrightarrow{P_1} = \overrightarrow{B} + R \times \frac{\overrightarrow{AB}}{|\overrightarrow{AB}|}$$

$$\overrightarrow{P_2} = \overrightarrow{P_1} + H \times \frac{\overrightarrow{P_1R}}{|\overrightarrow{P_1R}|}$$

$$\overrightarrow{P_3} = \overrightarrow{P_0} + H \times \frac{\overrightarrow{P_0L}}{|\overrightarrow{P_0L}|}$$

若不要求出顶点坐标，也可以这样表示矩形面积：

$$S = \frac{|\overrightarrow{AB} \times \overrightarrow{BU}| \times (|\overrightarrow{AD} \cdot \overrightarrow{AB}| + |\overrightarrow{BC} \cdot \overrightarrow{BA}| - |\overrightarrow{AB} \cdot \overrightarrow{BA}|)}{|\overrightarrow{AB} \cdot \overrightarrow{BA}|}$$

设 $|\overrightarrow{P_0B}| = m$, $|\overrightarrow{AP_1}| = n$ 后易证。

• 代码

```
1 double Get_Max(int n, Point *ch) {
2     ch[n] = ch[0];
3     int u = 2, l, r = 2;
4     //u是距离AB最远的点；在AB为底时，l和r是两个最靠边的点
5     double ret = 1e100, H, L, R, S;
6     for(int i = 0; i < n; i++) {
7         Point A = ch[i], B = ch[i+1]; Vec AB = B - A, BA = A - B;
8         while((AB * Vec(B, ch[u+1])) >= (AB * Vec(B, ch[u])))
9             u = (u + 1) % n;
10        while((AB & Vec(B, ch[r+1])) >= (AB & Vec(B, ch[r])))
11            r = (r + 1) % n;
12        if(i == 0) l = r;
13        while((AB & Vec(B, ch[l+1])) <= (AB & Vec(B, ch[l])))
14            l = (l + 1) % n;
15        H = (AB * Vec(B, ch[u])) / AB.len(); //以AB所在直线为底边，矩形的高
16        L = (BA & Vec(A, ch[l])) / BA.len(); //A距离左侧顶点的距离
```



```

17     R = (AB & Vec(B, ch[r])) / AB.len(); //B距离右侧顶点的距离
18     S = H * (L + AB.len() + R); //矩形面积
19     if(S < ret) { //求矩形顶点坐标
20         ret = S;
21         P[0] = A + L * BA.unit();
22         P[1] = B + R * AB.unit();
23         P[2] = P[1] + H * (ch[r]-P[1]).unit();
24         P[3] = P[0] + H * (ch[l]-P[0]).unit();
25     }
26 }
27 return ret;
28 }

```

POJ2079 - Triangle

• 题意

给定平面上的 n 个点，从其中任选三个点作为顶点，求能构成的最大三角形面积。

$1 \leq n \leq 50000$.

• 分析

显然，三角形的顶点一定都在这 n 个点的凸包上，所以先求出凸包。

考虑旋转卡壳。

这题中不能固定一条边，再枚举另外两个点，因为三角形的边不一定在凸包上。

因此，先逆时针枚举一个固定点 i ，再逆时针旋转另外两个顶点 j 和 k 。

由于凸包上的一些单峰性，我们旋转 j 时停止的条件是 $S_{\triangle ijk}$ 最大，停止后固定 j ，以相同停止条件旋转 k 。

$S_{\triangle ijk}$ 最大，是指 $S_{\triangle i,j_last,k} < S_{\triangle ijk}$ 且 $S_{\triangle ijk} > S_{\triangle i,j_next,k}$ 。

• 代码

注意特判 $n \leq 2$ 以及凸包大小 $siz \leq 2$ 的情况。用叉积求面积。

```

1 double Get_Max(int n, Point *ch) {
2     double ret = 0;
3     ch[n] = ch[0];

```

```

4   int j = 1, k = 2;
5   for(int i = 0; i < n; i++) {
6       while(Vec(ch[i], ch[j]) * Vec(ch[i], ch[k]) <
7           Vec(ch[i], ch[j]) * Vec(ch[i], ch[k+1]))
8           k = (k + 1) % n;
9       while(Vec(ch[i], ch[j]) * Vec(ch[i], ch[k]) <
10          Vec(ch[i], ch[j+1]) * Vec(ch[i], ch[k]))
11          j = (j + 1) % n;
12       ret = max(ret, Vec(ch[i], ch[j]) * Vec(ch[i], ch[k]));
13   }
14   return ret;
15 }

```

半平面交

半平面

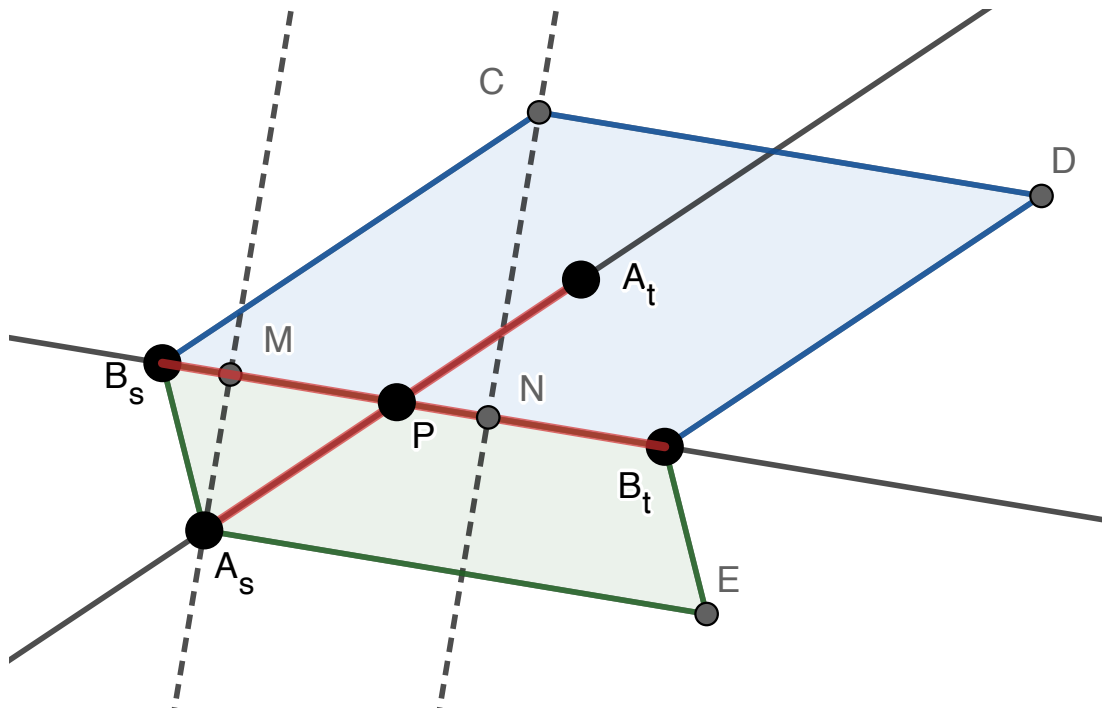
半平面是一条直线和直线的一侧，是一个点集。

当点集包含直线时，称为闭半平面；当不包含直线时，称为开半平面。

直线类

在计算几何中，用 (s, t) 表示直线 st ，一般统一保留向量的左侧半平面。

- 求两直线的交



若交点存在，只需求出 $t = \frac{|A_s P|}{|A_s A_t|}$ ，则 $\vec{P} = \vec{A_s} + t \cdot \vec{A_s A_t}$ 。

注意到 $t = \frac{|A_s P|}{|A_s A_t|} = \frac{|\vec{B_s B_t} \times \vec{B_s A_s}|}{|\vec{B_s B_t} \times \vec{A_s A_t}|}$ ，证明也不难。

如上图，把 $\vec{A_s A_t}$ 平移到 $\vec{B_s C}$ 的位置，则可以把叉乘的模看作平行四边形面积。

记 $S_1 = |\vec{B_s B_t} \times \vec{B_s A_s}|$ ， $S_2 = |\vec{B_s B_t} \times \vec{A_s A_t}|$ ，由于两个平行四边形同底，所以有 $\frac{S_1}{S_2} = \frac{|A_s M|}{|CN|}$ 。

又因为 $\triangle A_s M P \sim \triangle C N B_s$ ，所以 $\frac{S_1}{S_2} = \frac{|A_s M|}{|CN|} = \frac{|A_s P|}{|CB_s|} = \frac{|A_s P|}{|A_s A_t|}$ 。

• 参考代码

```
1 struct Line{
2     Point s, t;
3     Line() {}
4     Line(Point a, Point b) : s(a), t(b) {}
5     double ang() { return atan2((t - s).y, (t - s).x); }
6     Line(double a, double b, double c) { //ax + by + c = 0
7         if(sgn(a) == 0) s = Point(0, -c/b), t = Point(1, -c/b);
8         else if(sgn(b) == 0) s = Point(-c/a, 0), t = Point(-c/a, 1);
9         else s = Point(0, -c/b), t = Point(1, (-c-a)/b);
```

```

10     }
11     friend bool parallel(const Line &A, const Line &B) {
12         return sgn((A.s - A.t) * (B.s - B.t)) == 0;
13     }
14     friend bool Calc_intersection(const Line &A, const Line &B, Point
&res) {
15         if(parallel(A, B)) return false;
16         double s1 = (B.t - B.s) * (B.s - A.s);
17         double s2 = (B.t - B.s) * (A.t - A.s);
18         res = A.s + (A.t - A.s) * (s1 / s2);
19         return true;
20     }
21 };

```

半平面交

半平面交是多个半平面的交集。

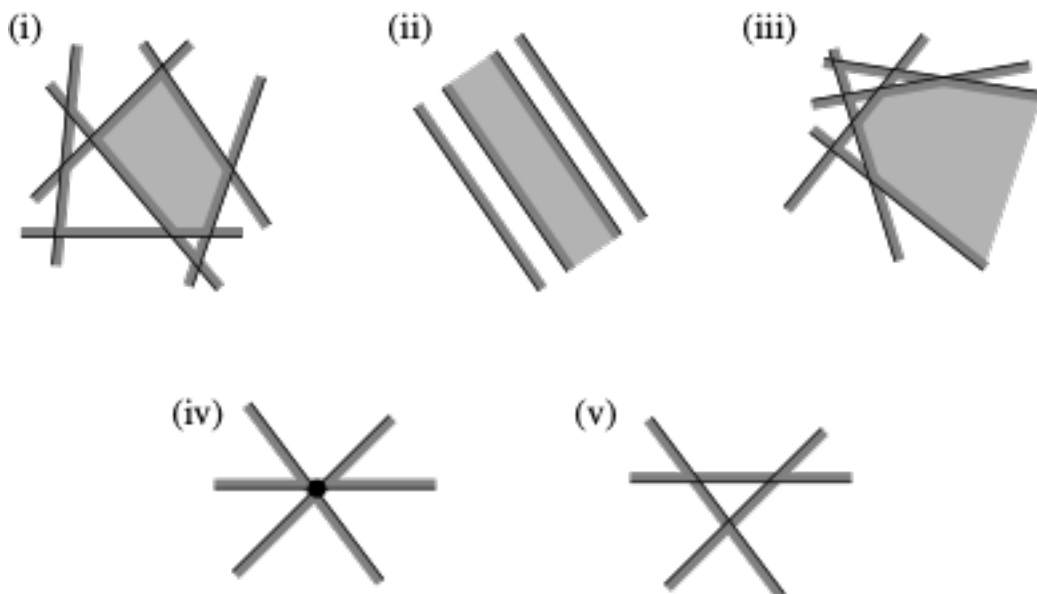
半平面交是一个点集，并且是一个凸集。在直角坐标系上是一个区域。

半平面交在代数意义下，是若干个线性约束条件，每个约束条件形如：

$$a_i x + b_i y \leq c_i$$

其中 a_i, b_i, c_i 为常数，且 a_i 和 b_i 不都为零。

半平面交有下面五种可能情况，每个半平面位于边界直线的阴影一侧。



(iii) 和 (v) 比较特殊，我们一般假设产生的交集总是有界或空的。

性质

注意到，位于半平面交边界上的任何一个点，必定来自某张半平面的边界。

并且，因为半平面交是凸集，所以每条边界线最多只能为边界贡献一条边。

因此：由 n 张半平面相交而成的凸多边形，其边界最多由 n 条边围成。

Sort-and-Incremental Algorithm

S&I 算法，排序增量法。

• 算法思想

S&I 算法利用了性质：半平面交是凸集。

因为半平面交是凸集，所以我们维护凸壳。

若我们把半平面按极角排序，那么在过程中，只可能删除队首或队尾的元素，因此使用双端队列维护。

下面是一个例子。

TODO.

• 算法流程

1. 把半平面按照极角序排序，需要 $\mathcal{O}(n \log n)$ 的时间。
2. 对每个半平面，执行一次增量过程，每次根据需要弹出双端队列的头部或尾部元素。这是线性的，因为每个半平面只会被增加或删除一次。
3. 最后，通过双端队列中的半平面，在线性时间内求出半平面交（一个凸多边形，用若干顶点描述）。

这样，我们得到了一个时间复杂度为 $\mathcal{O}(n \log n)$ 的算法，瓶颈在于排序。因此，若题目给定的半平面满足极角序，则我们可以在线性的时间内求出半平面交。

• 极角排序

注意判断不要写成 `<=` 或 `>=`。

```

1 inline bool cmp(Line A, Line B) {
2     //极角相等时, 位置靠右的排在前面
3     if(!sgn(A.ang - B.ang)) return (A.t - A.s) * (B.t - A.s) > 0;
4     return A.ang < B.ang;
5 }

```

• 求半平面交

```

1 bool Halfplane_intersection(int n, Line *hp, Point *p) {
2     if(n < 3) return false;
3     sort(hp, hp + n, cmp);
4     Halfplane_unique(n, hp);
5     st = 0; ed = 1;
6     que[0] = 0; que[1] = 1;
7     if(parallel(hp[0], hp[1])) return false;
8     Calc_intersection(hp[0], hp[1], p[1]);
9     for(int i = 2; i < n; i++) {
10         while(st < ed &&
11             sgn((hp[i].t - hp[i].s) * (p[ed] - hp[i].s)) < 0)
12             ed--;
13         while(st < ed &&
14             sgn((hp[i].t - hp[i].s) * (p[st + 1] - hp[i].s)) < 0)
15             st++;
16         que[++ed] = i;
17         assert(ed >= 1);
18         if(parallel(hp[i], hp[que[ed - 1]])) return false;
19         Calc_intersection(hp[i], hp[que[ed - 1]], p[ed]);
20     }
21     while(st < ed &&
22         sgn((hp[que[st]].t - hp[que[st]].s) * (p[ed] - hp[que[st]].s))
23         < 0)
24         ed--;
25     while(st < ed &&
26         sgn((hp[que[ed]].t - hp[que[ed]].s) * (p[st + 1] -
27         hp[que[ed]].s)) < 0)
28         st++;
29     if(st + 1 >= ed) return false;
30     return true;
31 }

```

• 求凸多边形的顶点

```
1 int Get_convex_hull(Line *hp, Point *p, Point *ch) {
2     Calc_intersection(hp[que[st]], hp[que[ed]], p[st]);
3     for(int i = 0, j = st; j <= ed; i++, j++) ch[i] = p[j];
4     return ed - st + 1;
5 }
```

• 计算面积

```
1 double Calc_area(int n, Point *ch) {
2     double ans = 0;
3     for (int i = 2; i < n; i++)
4         ans += area(ch[0], ch[i - 1], ch[i]);
5     return ans;
6 }
```

P3256 [JLOI2013]赛车

题目描述

赛场上一共有 n 辆车，分别称为 g_1, g_2, \dots, g_n 。

赛道是一条无限长的直线。最初， g_i 位于距离起跑线前进 x_i 的位置。比赛开始后，车辆 g_i 将会以 v_i 单位每秒的恒定速度行驶。

过程中，如果一辆赛车曾经处于领跑位置的话，即没有其他的赛车跑在他的前面，这辆赛车最后就可以得奖。

求哪些赛车将会得奖。输出赛车编号。

$1 \leq n \leq 10^4$, $0 \leq x_i \leq 10^9$, $0 \leq v_i \leq 10^9$ 。

分析

可以想到，每辆车的 $x - t$ 图像都是一条直线。在平面直角坐标系上画出这些直线。

若取每条直线的左侧半平面，再与 x 轴上侧平面、 y 轴右侧平面一起求交，则所有凸多边形上的直线可以获奖。

直接跑排序增量法即可，时间复杂度 $\mathcal{O}(n \log n)$ 。

注意到这题 x_i 和 v_i 的值域均为 10^9 ，所以运算过程中的数值会达到 $10^9 \times 10^9 = 10^{18}$ ，

因此需要使用 `long double`，并且需要 `const long double eps = 1e-18`。

代码

这道题不能直接去重，因为需要输出获奖车辆的编号，所以用 `std::map` 维护。

同时，还需要在 `Line` 类中存储所有重合直线的编号，用 `std::vector` 维护。

```
1 struct Line{
2     Point s, t;
3     vector<int> id;
4     Line() {}
5     Line(Point a, Point b) : s(a), t(b) {}
6     Line(Point a, Point b, vector<int> c) : s(a), t(b), id(c) {}
7     //...
8 };
9 //...
10 signed main() {
11     int n, x[MAXN], v[MAXN], N = 0;
12     map<pair<int, int>, vector<int> > mp;
13     cin >> n;
14     for(int i = 1; i <= n; i++) cin >> x[i];
15     for(int i = 1; i <= n; i++) cin >> v[i];
16
17     for(int i = 1; i <= n; i++) mp[make_pair(x[i], v[i])].push_back(i);
18     L[N++] = Line(Point(0, 1), Point(0, 0));
19     L[N++] = Line(Point(0, 0), Point(1, 0));
20     for(auto it = mp.begin(); it != mp.end(); it++) {
21         pair<int, int> tmp = it -> first;
22         L[N++] = Line(Point(0, tmp.first),
23                     Point(1, tmp.first+tmp.second), it -> second);
24     }
25
26     Halfplane_intersection(N, L, p);
27
28     int ans[MAXN], cnt_ans = 0;
29     for(int i = st + 1; i <= ed; i++) {
30         for(int j = 0; j < L[que[i]].id.size(); j++) {
31             ans[cnt_ans++] = L[que[i]].id[j];
```



```
32     }
33 }
34 sort(ans, ans + cnt_ans);
35 cout << cnt_ans << endl;
36 for(int i = 0; i < cnt_ans; i++) cout << ans[i] << " "; cout << endl;
37 return 0;
38 }
```

P4250 [SCOI2015]小凸想跑步

题目描述

小凸晚上喜欢到操场跑步，今天他跑完两圈之后，他玩起了这样一个游戏。

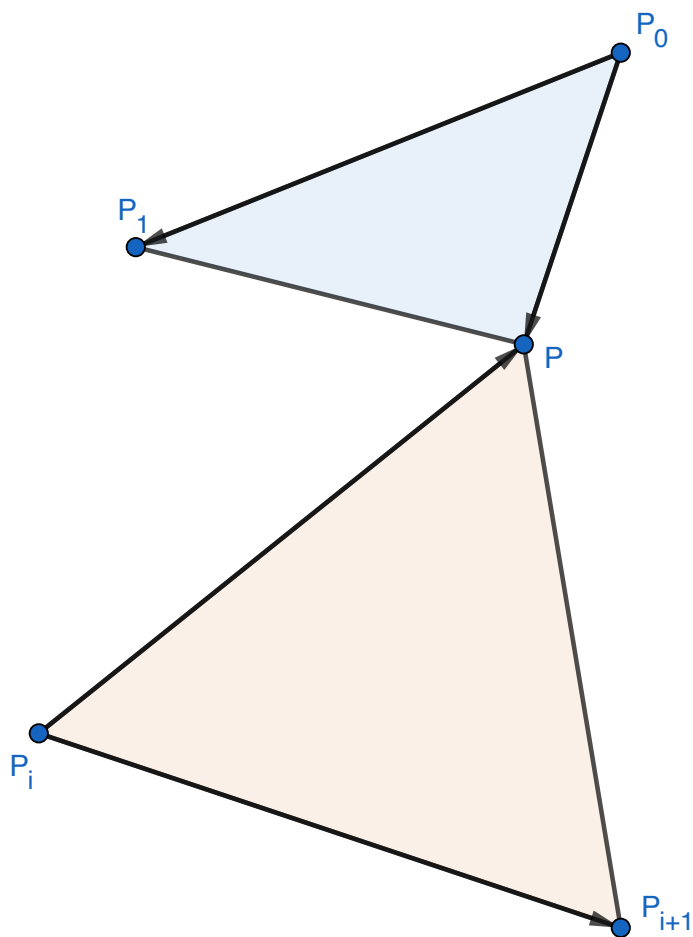
操场是个凸 n 边形， n 个顶点按照逆时针从 $0 \sim n - 1$ 编号。

现在小凸随机站在操场中的某个位置，标记为 p 点。将 p 点与 n 个顶点各连一条边，形成 n 个三角形。

如果这时 p 点， 0 号点， 1 号点形成的三角形的面积是 n 个三角形中最小的一个，则认为这是一次正确站位。

现在小凸想知道他一次站位正确的概率是多少。

分析



推一点式子。

设 $\vec{A} = (x_a, y_a)$, $\vec{B} = (x_b, y_b)$, $\vec{C} = (x_c, y_c)$, $\vec{D} = (x_d, y_d)$, $\vec{P} = (x, y)$.

由题意,

$$\begin{aligned} |\vec{AB} \times \vec{AP}| &< |\vec{CD} \times \vec{CP}| \\ (x_b - x_a)(y - y_a) - (y_b - y_a)(x - x_a) &< (x_d - x_c)(y - y_c) - (y_d - y_c)(x - x_c) \end{aligned}$$

展开得,

$$\begin{aligned} \text{左式} &= (x_b - x_a)y - x_by_a + x_ay_a - (y_b - y_a)x + x_ay_b - x_ay_a \\ \text{右式} &= (x_d - x_c)y - x_dy_c + x_cy_c - (y_d - y_c)x + x_cy_d - x_cy_c \end{aligned}$$

合并同类项,

$$\begin{aligned} (x_b - x_a)(y - y_a) - (y_b - y_a)(x - x_a) &< (x_d - x_c)(y - y_c) - (y_d - y_c)(x - x_c) \\ \Rightarrow (x_b - x_a + x_c - x_d)y + (y_a - y_b + y_d - y_c)x + (y_bx_a - x_by_a - y_dx_c + x_dy_c) &< 0 \end{aligned}$$

注意到, 这是直线解析式的形式, 于是转化为了半平面交问题。

参考资料

<https://cp-algorithms.com/geometry/halfplane-intersection.html>