# 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 $\boldsymbol{a}$。

**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 $|\boldsymbol{a}|$ 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。

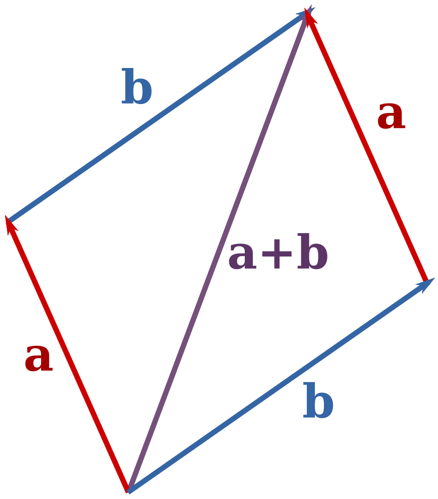
**平行向量**：方向相同或相反的两个 **非零** 向量。规定 $\vec{0}$ 与任意向量平行。$\boldsymbol{a}$ 与 $\boldsymbol{b}$ 平行，记作：$\boldsymbol{a} \parallel \boldsymbol{b}$ 。

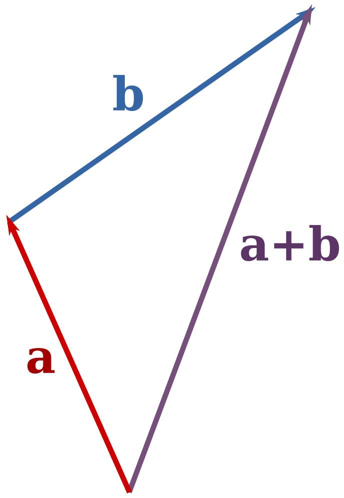
**共线向量**：与平行向量的定义相同。任一组平行向量都可以平移到同一直线上。

**向量的夹角**：已知两个非零向量 $\boldsymbol{a}, \boldsymbol{b}$，作 $\overrightarrow{OA} = \boldsymbol{a}, \overrightarrow{OB} = \boldsymbol{b}$，那么 $\theta = \angle AOB$ 就是向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 的夹角。记作：$\langle \boldsymbol{a}, \boldsymbol{b} \rangle$。当 $\theta = \frac{\pi}{2}$ 时，称这两个向量垂直，记作 $\boldsymbol{a} \perp \boldsymbol{b}$。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 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，在平面内任取一点 $A$，作 $\overrightarrow{AB}=\boldsymbol{a}$，$\overrightarrow{BC}=\boldsymbol{b}$，作向量 $\overrightarrow{AC}$.

称向量 $\overrightarrow{AC}$ 为向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 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 $\lambda$ 和一个向量 $\boldsymbol{a}$，规定其乘积为一个向量，记作 $\lambda\boldsymbol{a}$，其模与方向定义如下：

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

这种运算是向量的数乘运算。

- 坐标表示

$$
\begin{gathered}
\boldsymbol{a}+\boldsymbol{b}=(\boldsymbol{a}_x+\boldsymbol{b}_x,\ \boldsymbol{a}_y+\boldsymbol{b}_y)\\
\boldsymbol{a}-\boldsymbol{b}=(\boldsymbol{a}_x-\boldsymbol{b}_x,\ \boldsymbol{a}_y-\boldsymbol{b}_y)\\
\lambda\boldsymbol{a}=(\lambda\boldsymbol{a}_x,\ \lambda\boldsymbol{a}_y)
\end{gathered}
$$

# 向量的数量积

已知两个向量 $\boldsymbol{a},\boldsymbol{b}$，它们的夹角为 $\theta$，那么：

$$
\boldsymbol{a}\cdot\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\cos\theta=\boldsymbol{a}_x\boldsymbol{b}_x+\boldsymbol{a}_y\boldsymbol{b}_y
$$

就是这两个向量的 **数量积**，也叫 **点积** 或 **内积**。其中称 $|\boldsymbol{a}|\cos\theta$ 为 $\boldsymbol{a}$ 在 $\boldsymbol{b}$ 方向上的投影。数量积的几何意义即为：数量积 $\boldsymbol{a}\cdot\boldsymbol{b}$ 等于 $\boldsymbol{a}$ 的模与 $\boldsymbol{b}$ 在 $\boldsymbol{a}$ 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. $\boldsymbol{a}\perp\boldsymbol{b}\iff\boldsymbol{a}\cdot\boldsymbol{b}=0$
2. $\boldsymbol{a}=\lambda\boldsymbol{b}\iff|\boldsymbol{a}\cdot\boldsymbol{b}|=|\boldsymbol{a}||\boldsymbol{b}|$
3. $|\boldsymbol{a}|=\sqrt{m^2+n^2}$
4. $\cos\theta=\dfrac{\boldsymbol{a}\cdot\boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|}$

# 向量的向量积

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

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

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

向量积与 $\boldsymbol{a},\boldsymbol{b}$ 所在平面垂直，其竖坐标为 $\boldsymbol{a}_x\boldsymbol{b}_y-\boldsymbol{a}_y\boldsymbol{b}_x$.

我们根据右手法则可以推断出 $\boldsymbol{b}$ 相对于 $\boldsymbol{a}$ 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 $\boldsymbol{a}=(x,\ y)$ 逆时针旋转 $\alpha$，得到向量 $\boldsymbol{b}$，则有：

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

可以通过三角恒等变换证明。

# 参考代码

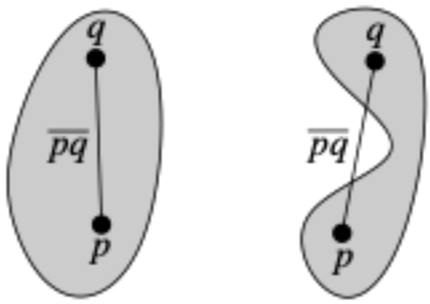
```
const double eps = 1e-10;
inline int fcmp(double x, double y) {
    if(fabs(x - y) < eps) return 0;
    else return x > y ? 1 : -1;
}
struct Point{
    double x, y;
    Point(){};
    Point(double a, double b): x(a), y(b) {}
    Point(Point a, Point b): x(b.x - a.x), y(b.y - a.y) {}
    Point operator + (const Point &b) {
        return Point(x + b.x, y + b.y);
    }
    Point operator - (const Point &b) {
        return Point(x - b.x, y - b.y);
    }
    double operator * (const Point &b) {
        return x * b.y - y * b.x;
    }
    double operator & (const Point &b) {
        return x * b.x + y * b.y;
    }
    bool operator == (const Point &b) {
        return fcmp(x, b.x) == 0 && fcmp(y, b.y) == 0;
    }
    double len() {
        return sqrt(x * x + y * y);
    }
};
typedef Point Vec;
inline double dis(Point &a, Point &b) {
    return (a - b).len();
```

```
}
inline double angle(Point &a, Point &b) {
    return acos((a & b) / a.len() / b.len());
}
inline Vec rotate(Vec &a, double k) {
    return Vec(a.x * cos(k) - a.y * sin(k), a.x * sin(k) - a.y * cos(k));
}
```

# 平面凸包

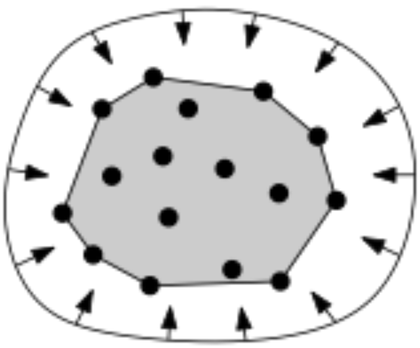
## 相关概念

**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。



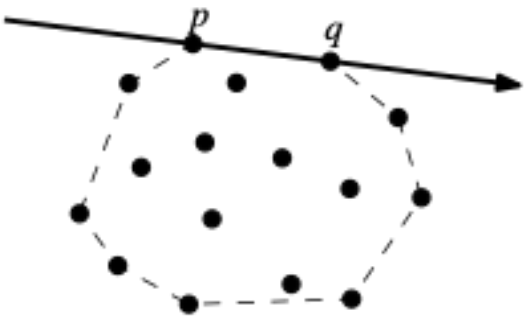
如上图，凸包还可以理解为平面上有若干柱子，**用橡皮筋套住所有柱子**，绷紧后形成的多边形即为凸包。

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 $p \to q$ 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 $\overline{pq}$ 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。

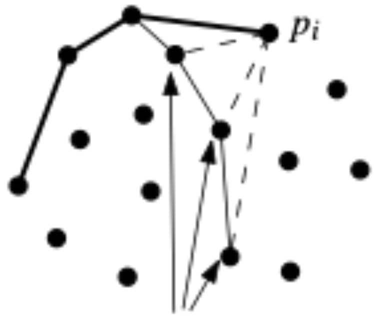
排序后，第一个点和末尾的点，一定在凸包上，容易通过反证法证明。

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。

```
inline bool check(Point s1, Point s2, Point p) {
    return Vec(s2, s1) * Vec(s1, p) > 0;
}
int Convex_hull_2d(int n, Point *p, Point *ret) {
    sort(p, p + n, cmp1);
    int top = -1;
    for (int i = 0; i < n; i++) {
        while (top > 0 && !check(ret[top], ret[top - 1], p[i]))
            top--;
        ret[++top] = p[i];
    }
    int k = top;
    for (int i = n - 2; i >= 0; i--) {
        while (top > k && !check(ret[top], ret[top - 1], p[i]))
```

```
            top--;
        ret[++top] = p[i];
    }
    return top;
}
```

## 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 $\in(-\pi,\pi]$.

```
bool cmp(Point a, Point b) {
    if(atan2(a.y, a.x) - atan2(b.y, b.x) == 0)
        return a.x < b.x;
    return atan2(a.y, a.x) < atan2(b.y, b.x);
}
```

另外一种方式是利用叉积排序。

```
bool cmp(Point a, Point b) {
    return a * b > 0;
}
```

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：

$$l=\sum_{i=1}^{n}|\overrightarrow{p_ip_{i+1}}|+|\overrightarrow{p_1p_n}|$$

```cpp
double dis(Point a, Point b) {
    return (a - b).len();
}
double Convex_hull_2d_L(int n, Point *p) {
    Point convex[N];
    int siz = Convex_hull_2d(n, p, convex);
    double ans = dis(convex[0], convex[siz - 1]);
    for (int i = 1; i < siz; i++)
        ans += dis(convex[i - 1], convex[i]);
    return ans;
}
```

**求面积：** 任取凸包内一点（一般取 $p_1$），则有：

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

```cpp
double area(Point a, Point b, Point c) {
    return (b - a) * (c - a) / 2.0;
}
double Convex_hull_2d_S(int n, Point *p) {
    Point convex[N];
    int siz = Convex_hull_2d(n, p, convex);
    double ans = 0;
    for (int i = 2; i < siz; i++)
        ans += area(convex[0], convex[i - 1], convex[i]);
    return ans;
}
```

## 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

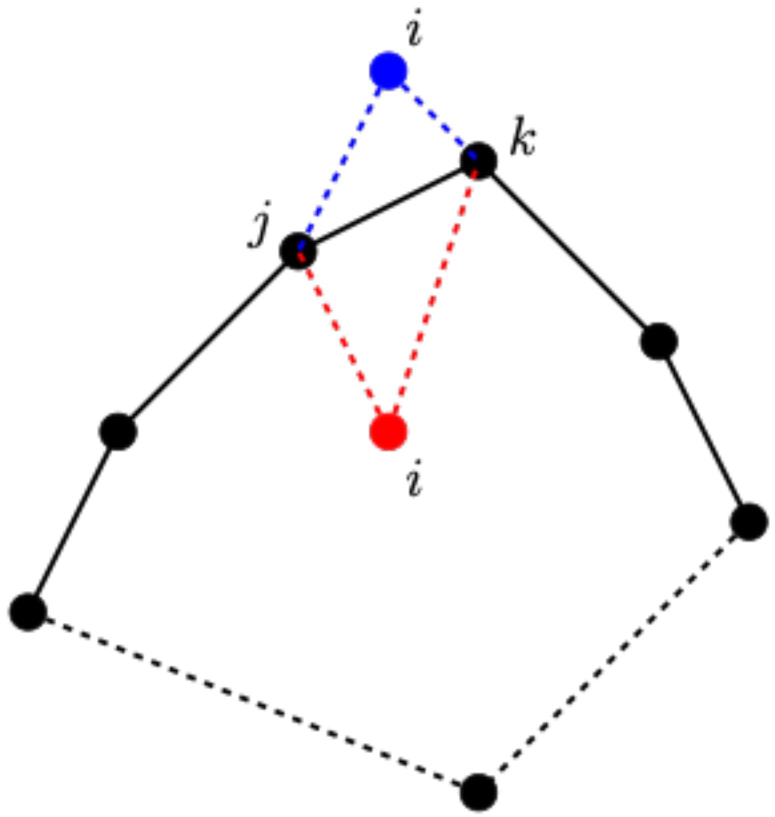
和 Andrew 算法一样，这里的算法按照坐标字典序排序。相对于极角排序，能够减小精度误差。

用两个 `std::map<int, int>`，用 $\mathrm{top}$ 记录上凸包，$\mathrm{down}$ 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 $\mathrm{top}[x]$，$\mathrm{down}$ 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 $|\overrightarrow{ik} \times \overrightarrow{ij}| \geq 0$.

```cpp
bool check_top(int x, int y) { //是否在上凸包下面
    auto k = top.lower_bound(x);
    if(k == top.end())
        return false;
    if(k -> first == x)
        return y <= k->second;
    if(k == top.begin()) return false;
    auto j = k; j--;
    return Point(k->first - x, k->second - y) *
           Point(j->first - x, j->second - y) >= 0;
}
```

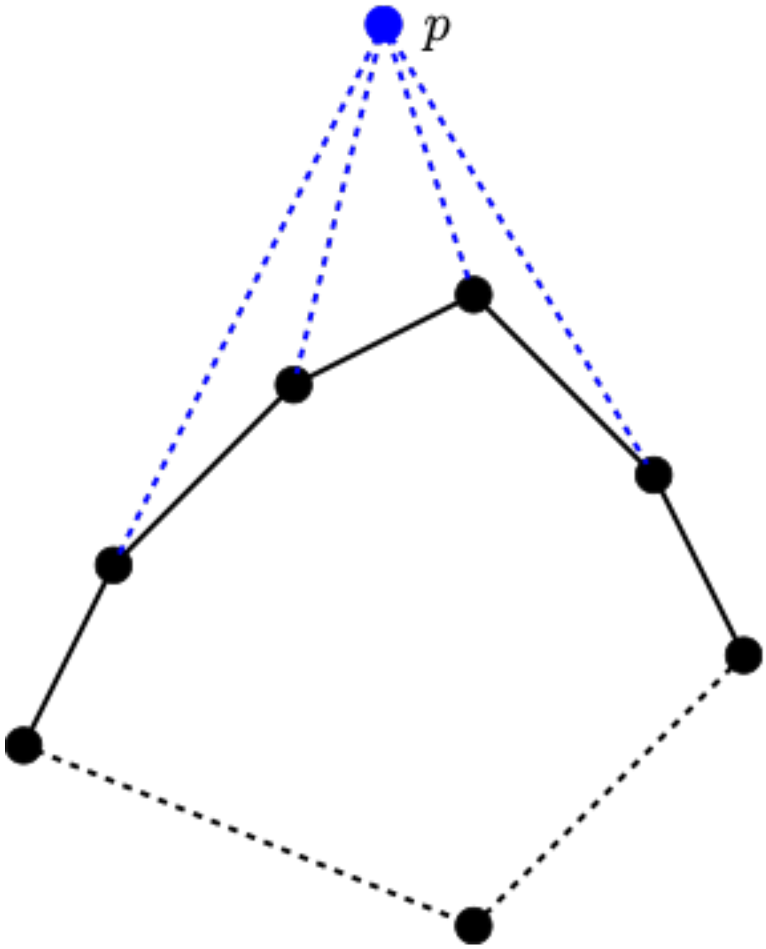
下凸包同理，$i$ 在下凸包之上，当且仅当 $|\overrightarrow{ik} \times \overrightarrow{ij}| \leq 0$

```cpp
bool check_down(int x, int y) { //是否在下凸包上面
    auto k = down.lower_bound(x);
    if(k == down.end())
        return false;
    if(k -> first == x)
        return y >= k->second;
    if(k == down.begin()) return false;
    auto j = k; j--;
    return Point(k->first - x, k->second - y) *
           Point(j->first - x, j->second - y) <= 0;
}
```

- ## 插入操作


- ## 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。

```
void insert_top(int x, int y) {
    if(check_top(x, y)) return;
    top[x] = y;
    auto it = top.find(x);
    auto jt = it;
    if(it != top.begin()) { //remove left
        jt--;
        while(remove_top(jt++)) jt--;
    }
    if(++jt != top.end()) { //remove right
        while(remove_top(jt--)) jt++;
    }
}
void insert_down(int x, int y) {
    if(check_down(x, y)) return;
    down[x] = y;
    auto it = down.find(x);
```

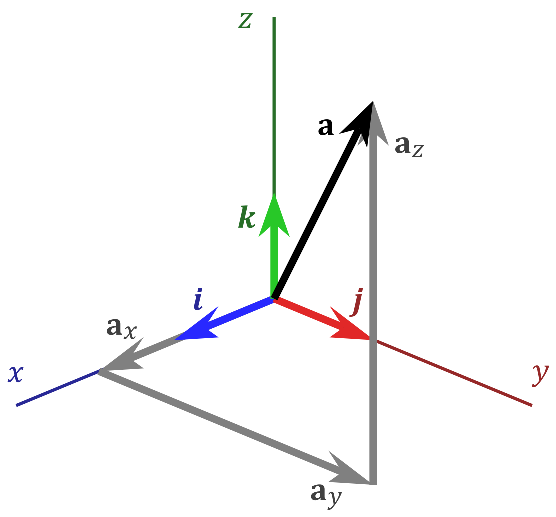
```
    auto jt = it;
    if(it != down.begin()) { //remove left
        jt--;
        while(remove_down(jt++)) jt--;
    }
    if(++jt != down.end()) { //remove right
        while(remove_down(jt--)) jt++;
    }
}
```

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。

```
bool remove_top(map<int, int>::iterator it) {
    if(it == top.begin()) return false; //到边界就不删了
    if(++it == top.end()) return false; it--;
    auto jt = it, kt = it;
    jt--; kt++;
    if(Point(it -> first - jt -> first, it->second - jt->second) *
       Point(it -> first - kt -> first, it->second - kt->second) <= 0) {
            top.erase(it);
            return true;
    }
    return false;
}
bool remove_down(map<int, int>::iterator it) {
    if(it == down.begin()) return false;
    if(++it == down.end()) return false; it--;
    auto jt = it, kt = it;
    --jt; ++kt;
    if(Point(it -> first - jt -> first, it->second - jt->second) *
       Point(it -> first - kt -> first, it->second - kt->second) >= 0) {
            down.erase(it);
            return true;
    }
    return false;
}
```

## 三维凸包

# 三维向量类



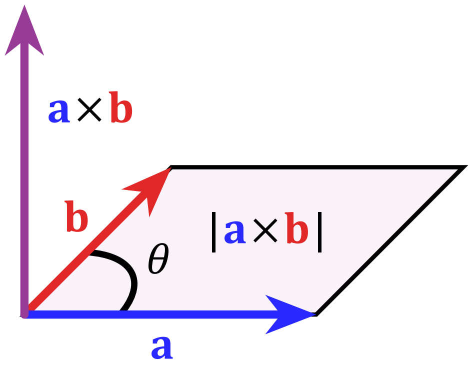

**存储**：用结构体记录三维坐标，方便重载运算符。

```
struct Point3 {
    double x, y, z;
    Point3(){};
    Point3(double a, double b, double c) : x(a), y(b), z(c) {}
};
```

**加法、减法和点乘等操作**：与二维向量类似。

```
Point3 operator + (const Point3 &b) {
    return Point3(x + b.x, y + b.y, z + b.z);
}
Point3 operator - (const Point3 &b) {
    return Point3(x - b.x, y - b.y, z - b.z);
}
Point3 operator * (const Point3 &b) {
    return Point3(y*b.z - z*b.y, z*b.x - x*b.z, x*b.y - y*b.x);
}
bool operator == (const Point3 &b) {
    return fcmp(x, b.x) == 0 && fcmp(y, b.y) == 0 && fcmp(z, b.z) == 0;
}
double len() {
    return sqrt(x * x + y * y + z * z);
}
```

**叉乘**：$\vec{a} \times \vec{b}$ 的结果为一个三维向量 $\vec{c}$，$\vec{c} \perp \vec{a}$ 且 $\vec{c} \perp \vec{b}$，结果向量的模长为 $|\vec{a}||\vec{b}| \sin\langle\vec{a}, \vec{b}\rangle$，代表以 $\vec{a}$、$\vec{b}$ 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 $\boldsymbol{a}$，推导过程如下。（来源：[叉积 - 维基百科]()）

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

$$
\begin{gathered}
\boldsymbol{i} \times \boldsymbol{j} = \boldsymbol{k} \\
\boldsymbol{j} \times \boldsymbol{i} = -\boldsymbol{k} \\
\boldsymbol{j} \times \boldsymbol{k} = \boldsymbol{i} \\
\boldsymbol{k} \times \boldsymbol{j} = -\boldsymbol{i} \\
\boldsymbol{k} \times \boldsymbol{i} = \boldsymbol{j} \\
\boldsymbol{i} \times \boldsymbol{k} = -\boldsymbol{j}
\end{gathered}
$$

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

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

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

把前面的 6 个等式代入，则有：

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

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

$$
\begin{aligned}
s_1 &= u_2v_3 - u_3v_2 \\
s_2 &= u_3v_1 - u_1v_3 \\
s_3 &= u_1v_2 - u_2v_1
\end{aligned}
$$

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

# 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。

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

**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。

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

利用法向量的模长，也可以算出三角形的面积。

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

# 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。

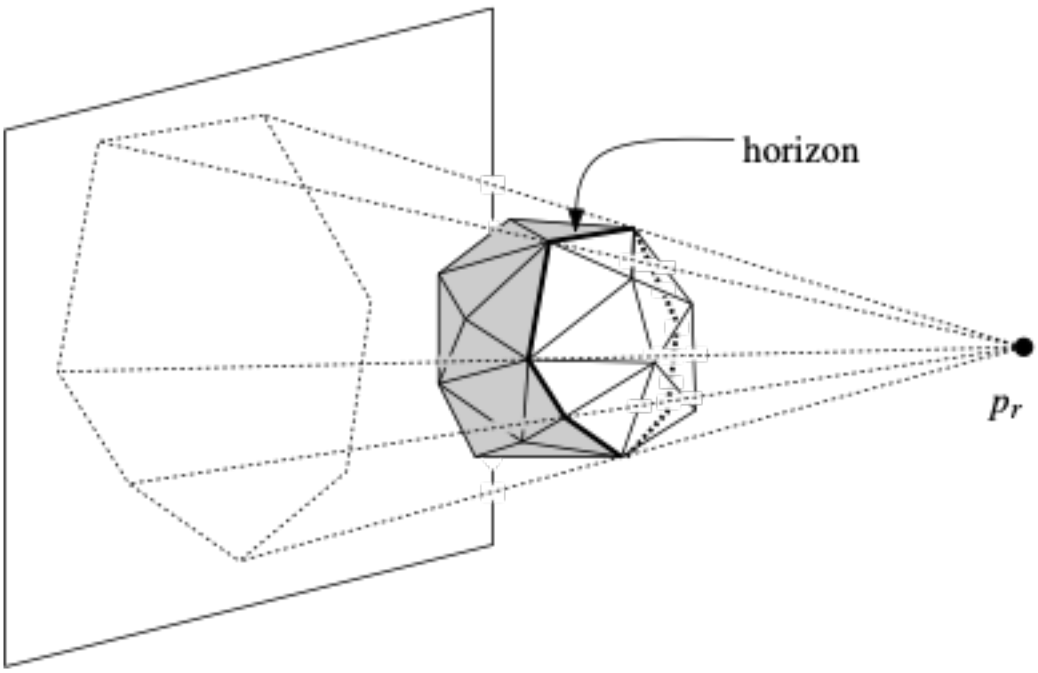
# 随机增量法

## 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 $\mathcal{CH}(P_{r-1}) \to \mathcal{CH}(P_r)$.

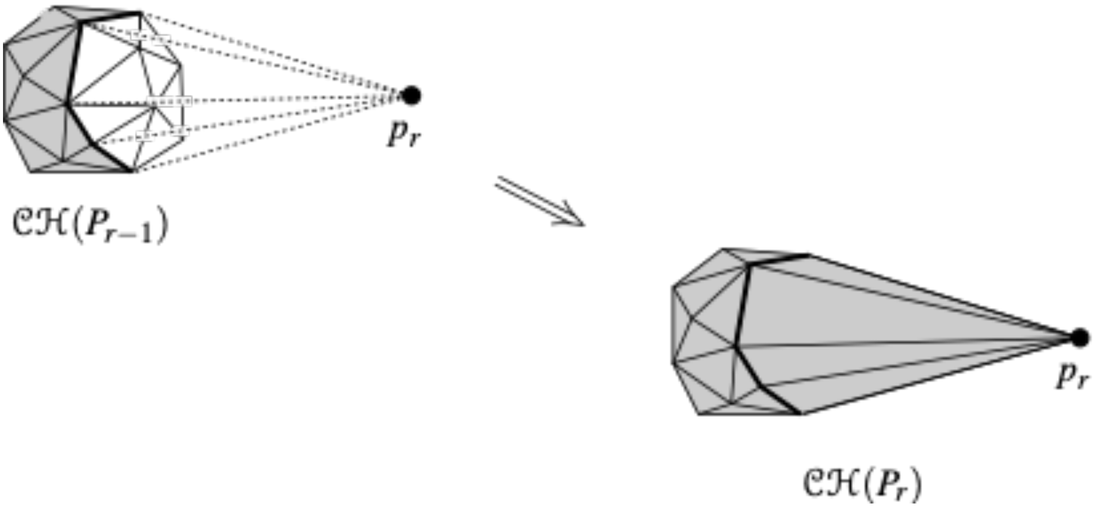
**第二种情况**：$p_r$ 在 $\mathcal{CH}(P_{r-1})$ 外部。设想你站在 $p_r$ 所在的位置，看向 $\mathcal{CH}(P_{r-1})$. $\mathcal{CH}(P_{r-1})$ 中的某些小平面会被看到，其余在背面的平面不会被看到。如下图，从 $p_r$ 可见的平面构成了一片连通的区域。




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)。

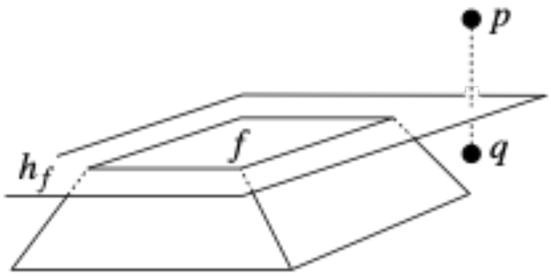
根据这条地平线，我们可以判断出，在原先 $\mathcal{CH}(P_{r-1})$ 表面上的哪些部分需要被保留，哪些需要被替换。

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，它能将空间分为两半，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 $\boldsymbol{s} = \overrightarrow{ab} \times \overrightarrow{ac}$，则结果向量 $\boldsymbol{s}$ 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 $\overrightarrow{ap} \times \boldsymbol{s} > 0$，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。

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

- ## 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 $\overrightarrow{p_ip_j}$ 所在的平面是否可见。

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

注意到，位于边界线上的向量 $\overrightarrow{p_ip_j}$ 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时，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 $\overrightarrow{p_ip_j}$ 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。

```
int Convex_hull_3d(int n, plane *ret) {
    plane tmp[N];
    bool g[N][N];
    for (int i = 0; i < n; i++) p[i].shake();
    int top = -1;
    ret[++top] = plane(0, 1, 2);
    ret[++top] = plane(0, 2, 1);
    for (int i = 3; i < n; i++) {
        int cnt = -1;
        for (int j = 0; j <= top; j++) {
            bool flag = ret[j].is_above(p[i]);
            if (!flag)
                tmp[++cnt] = ret[j];
            for (int k = 0; k < 3; k++)
                g[ret[j].v[k]][ret[j].v[(k + 1) % 3]] = flag;
        }
```

```
            for (int j = 0; j <= top; j++) {
                for (int k = 0; k < 3; k++) {
                    int a = ret[j].v[k], b = ret[j].v[(k + 1) % 3];
                    if (g[a][b] && !g[b][a])
                        tmp[++cnt] = plane(a, b, i);
                }
            }
            for (int j = 0; j <= cnt; j++) ret[j] = tmp[j];
            top = cnt;
        }
    return (top + 1);
}
```

# 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。$2 \le n \le 50000, |x|, |y| \le 10^4$.

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?

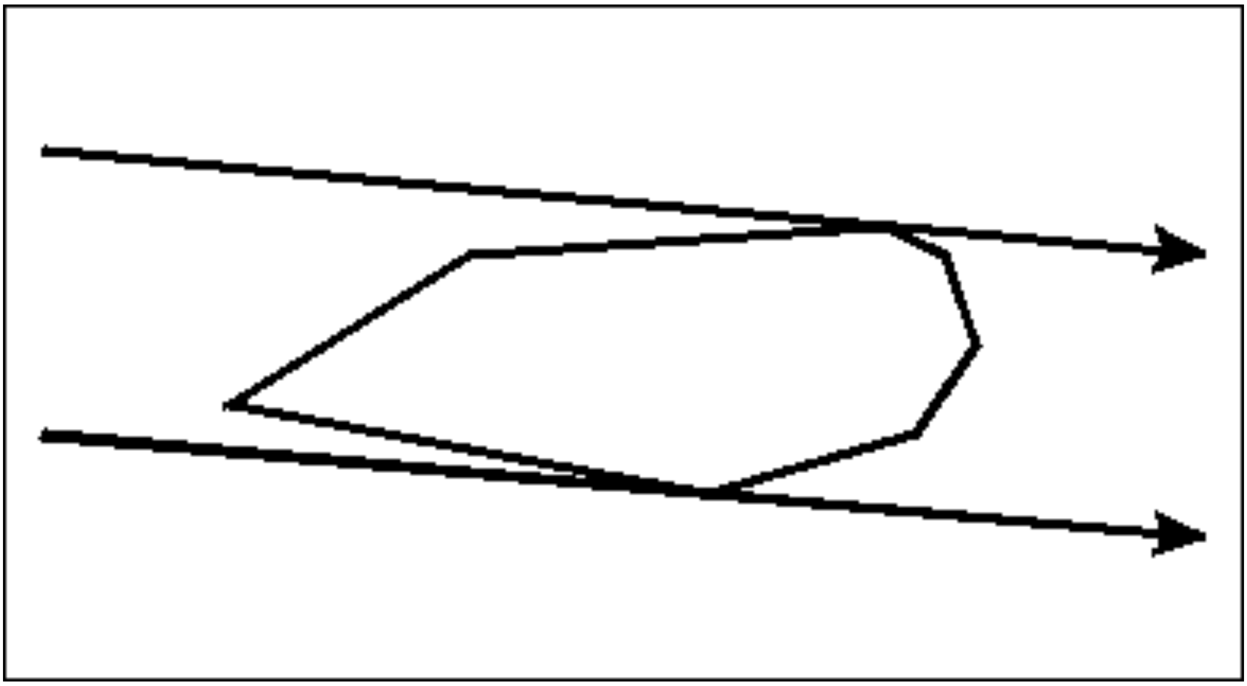
**暴力做法**：可以遍历每个点对，求出最大距离，复杂度为 $\mathcal{O}(n^2)$.

### 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。

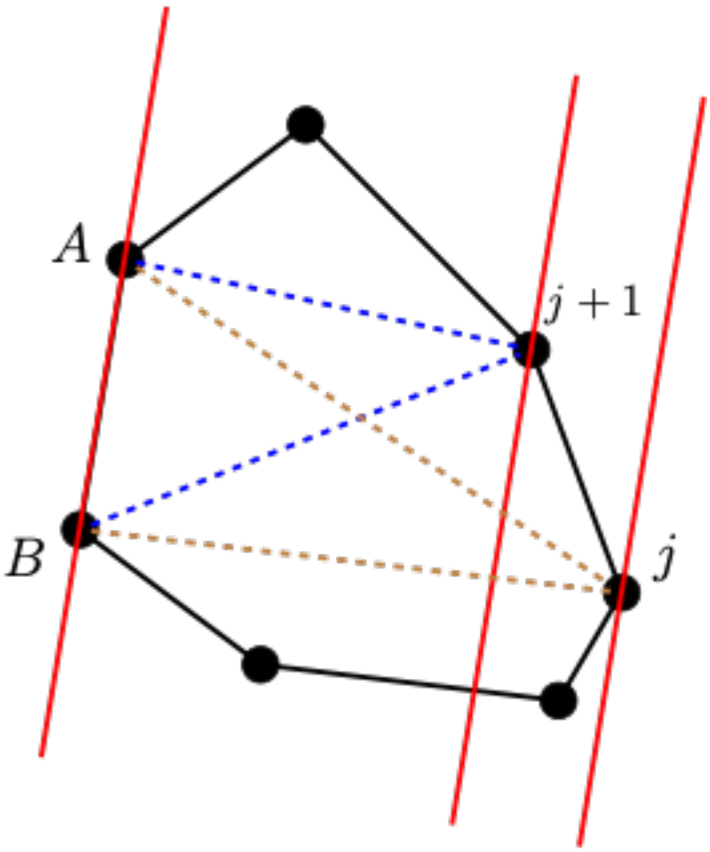
因此，我们可以在逆时针枚举边的同时，实时维护最远点，利用单调性，复杂度为 $\mathcal{O}(n)$.



## 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。




```cpp
struct Point{
```

```
    int x, y;
    //......
    int sqr_len() { return x * x + y * y; }
};
inline int sqr_dis(Point a, Point b) { return (a - b).sqr_len(); }
int Get_Max(int n, Point *ch) {//传入convex-hull
    int ret = 0;
    ch[n] = ch[0];
    int j = 1;
    for(int i = 0; i < n; i++) {
        while((ch[i] - ch[j+1]) * (ch[i+1] - ch[j+1]) >
              (ch[i] - ch[j]) * (ch[i+1] - ch[j]))
            j = (j + 1) % n;
        ret = max(ret, max(sqr_dis(ch[i], ch[j]), sqr_dis(ch[i+1], ch[j])));
    }
    return ret;
}
```

## 最小矩形覆盖

给定一些点的坐标，求能够覆盖所有点的最小面积的矩形。$3 \le n \le 50000$.