

# Advanced Competitive Programming

---

國立成功大學ACM-ICPC程式競賽培訓隊  
[nckuacm@imslab.org](mailto:nckuacm@imslab.org)

Department of Computer Science and Information Engineering  
National Cheng Kung University  
Tainan, Taiwan

# 計算幾何

---

# 解析幾何

---

- 國高中數學大部分都是解析幾何
- 通常會用方程式來表示圖形
  - e.g. 二維平面直線方程式  $y = ax + b$
- 某些函數計算上容易有誤差，例如：三角函數

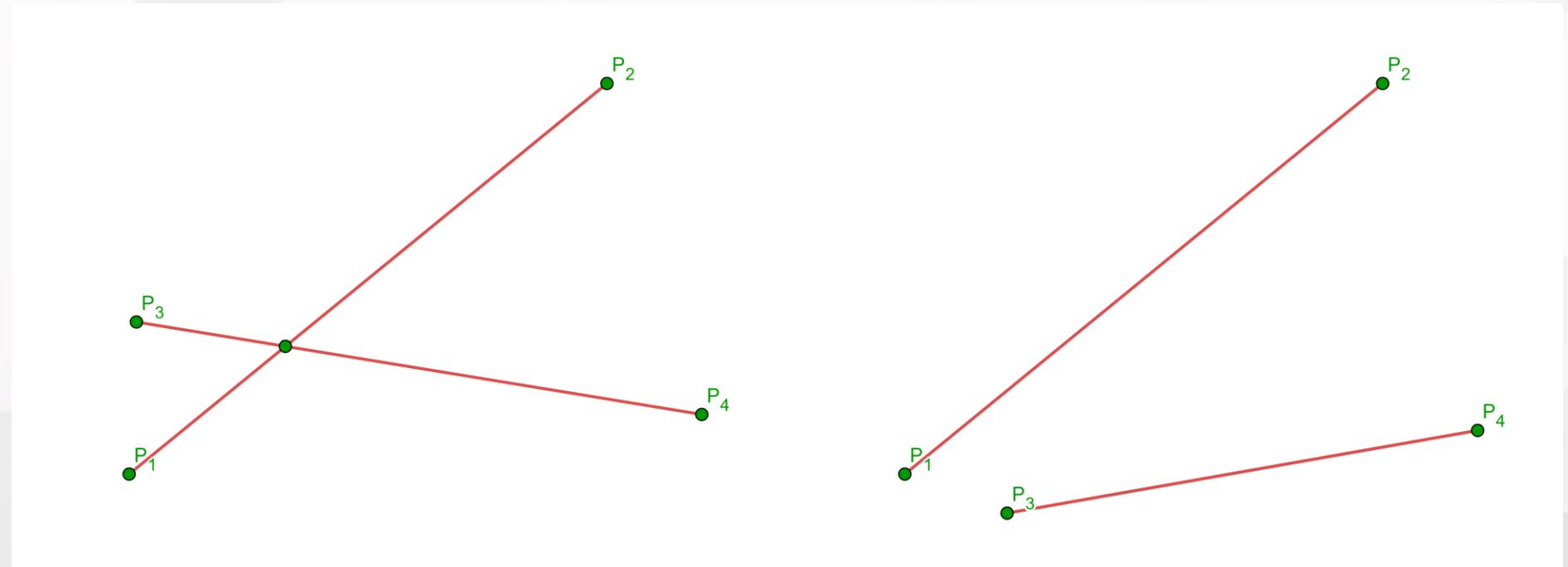
# 計算幾何

---

- 計算幾何 = 解析幾何 + 演算法 + 電腦精度問題
- 為了降低浮點數的誤差，會儘量少用乘法、除法、以及一些可能產生無理數的函數
- 某些題目也可以用計算幾何先簡化後較好操作

# 計算幾何

- 給定兩個線段，請問這兩個線段是否相交，如果相交則輸出其交點。



# 線段表示方法

---

- 要容易計算
- 執行速度快
- 產生誤差小

# 線段表示方法

---

- $y = ax + b$  ?
  - 垂直線無法表示
- 特判 ?
  - 這也要特判那也要特判，不好寫又很容易出錯

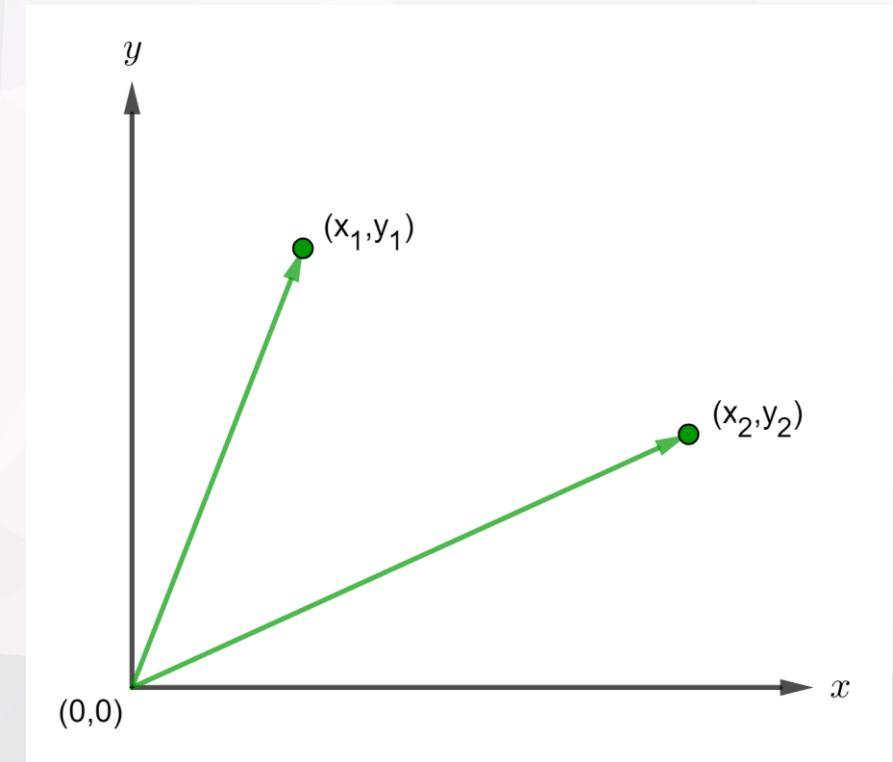
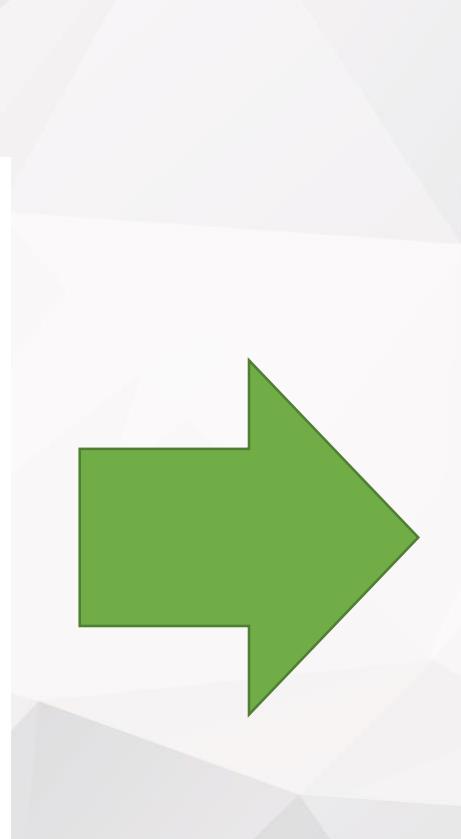
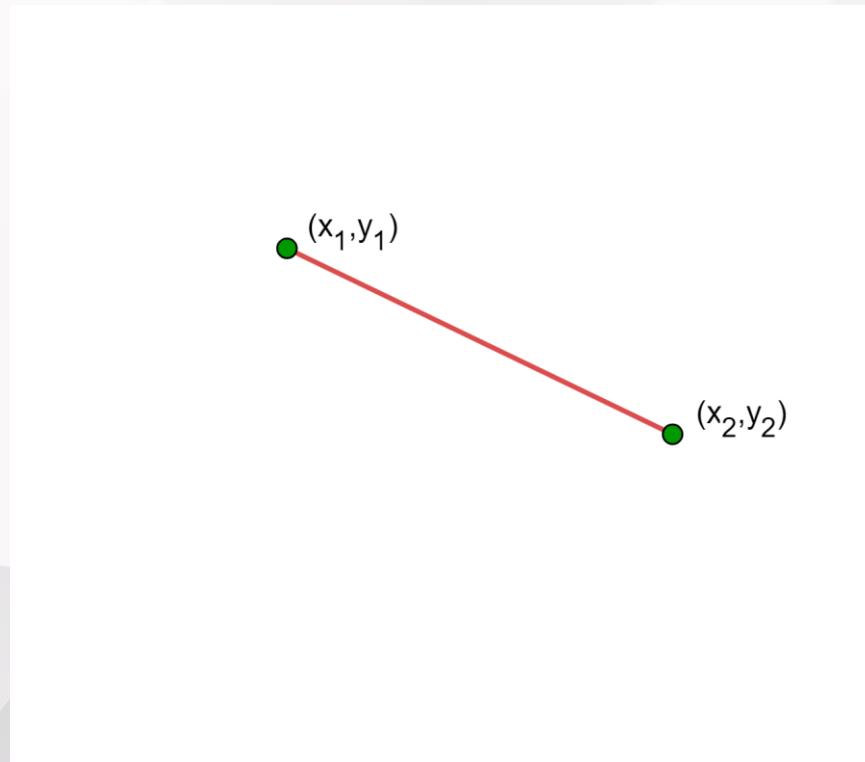
# 點與向量

---

- 講線段前先從點開始講起
- 兩個點可以決定一條線段或直線，稱為兩點式
- 正確來說，這是兩個位置向量

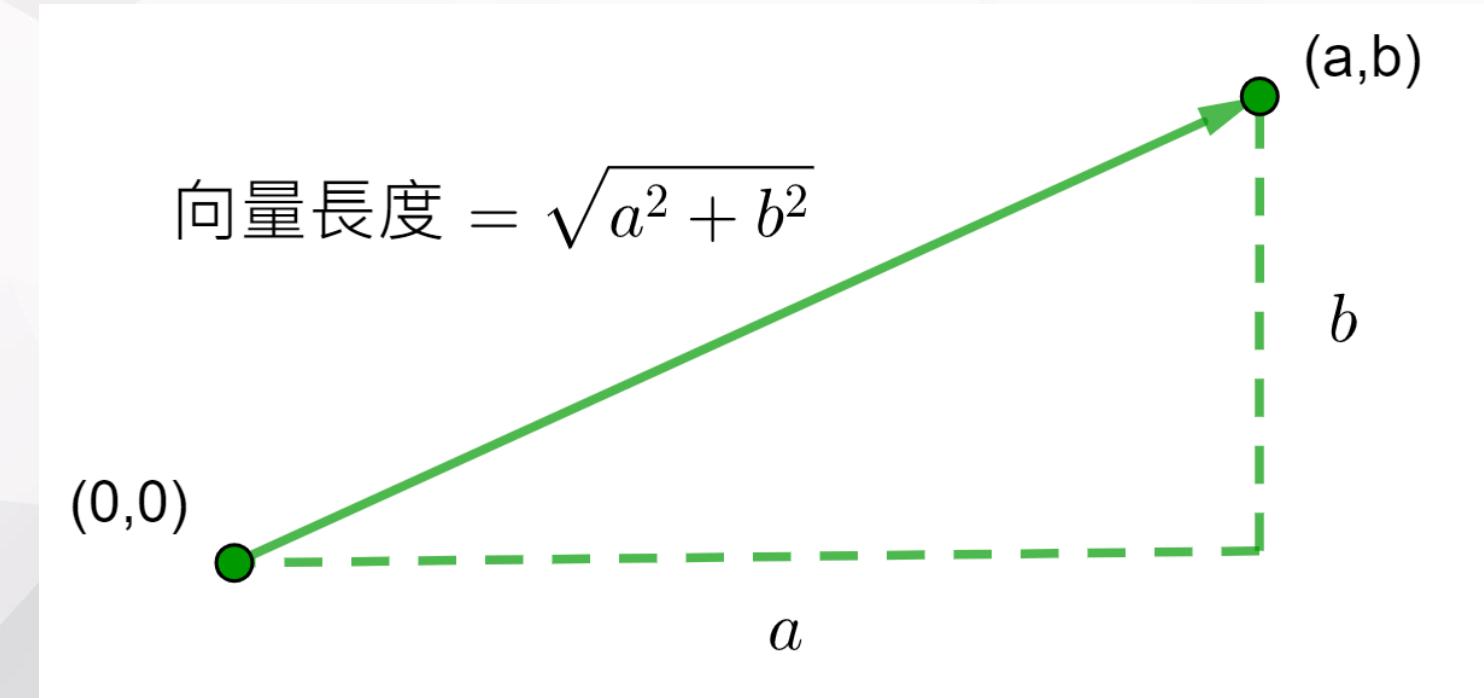
# 點與向量

- 位置向量



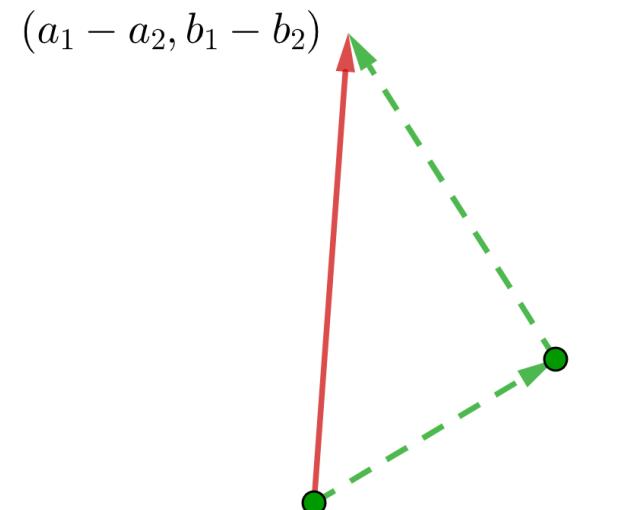
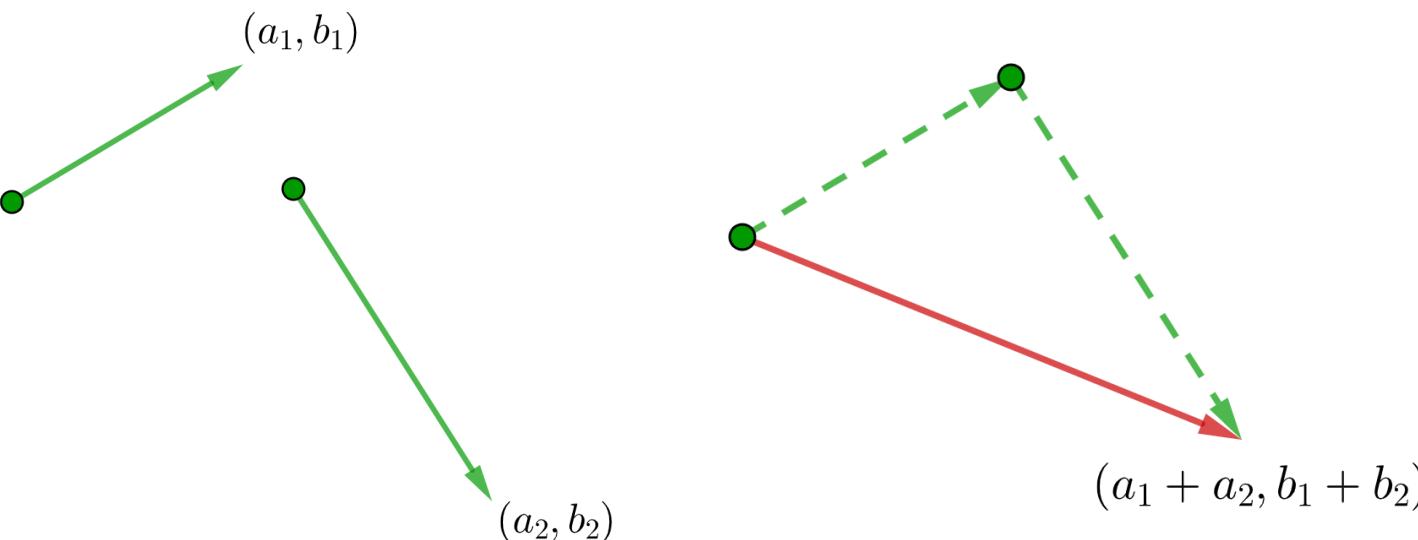
# 點與向量

- 向量可以表示方向與長度
- 通常會使用  $(a, b)$  的方式紀錄一個向量



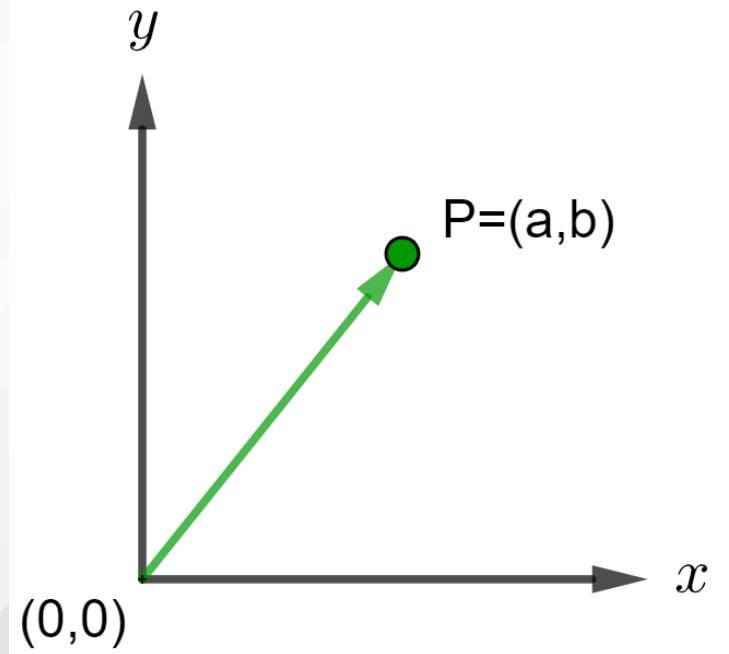
# 點與向量

- 向量可以做加減的運算
- $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$
- $(a_1, b_1) - (a_2, b_2) = (a_1 - a_2, b_1 - b_2)$



# 點與向量

- 平面上的一個點可以視為從原點發射的向量
- 在計算幾何中：點(point) = 向量(vector)



# 點與向量

- 程式碼通常寫成一個 struct 表示

```
struct Point{
    double x, y;
    Point(double x = 0, double y = 0) : x(x), y(y){}
    Point operator+(const Point &b) const{ // 向量相加
        return Point(x + b.x, y + b.y);
    }
    Point operator-(const Point &b) const{ // 向量相減
        return Point(x - b.x, y - b.y);
    }
    Point operator*(double d) const{ // 伸長
        return Point(x * d, y * d);
    }
    Point operator/(double d) const{ // 縮短
        return Point(x / d, y / d);
    }
    double dot(const Point &b) const{} // 內積
    double cross(const Point &b) const{} // 外積
};
```

# 點與向量

---

- 也可使用 C++ 的 `<complex>` 函式庫 (since C++11)
- 將二維座標用複數平面表示，實部為  $x$ ，虛部為  $y$
- 本身就支援相加相減等操作
  - 筆者還是習慣自己寫 `struct`，較好撰寫函數

# 點與向量

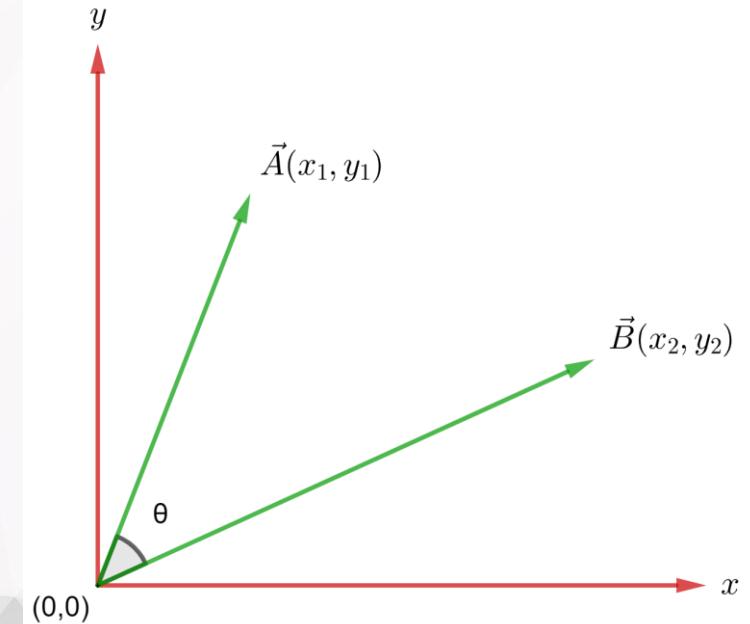
---

- 變數型態根據題目的要求而定
  - int
  - long long
  - double
- 課程中均使用 double 為例
- 請勿使用 float

# 內積 (dot)

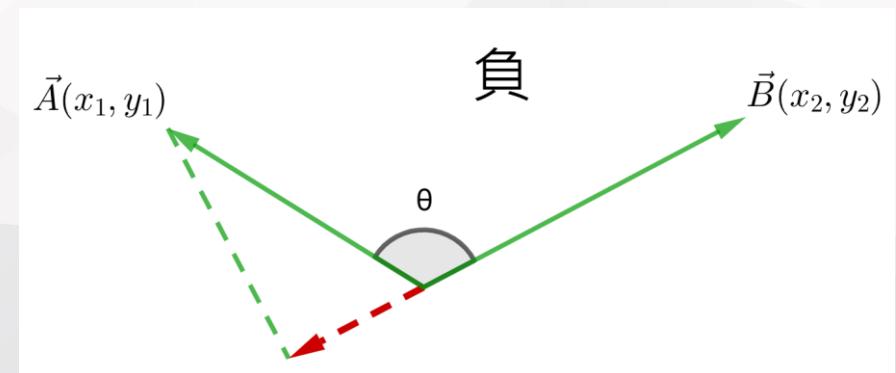
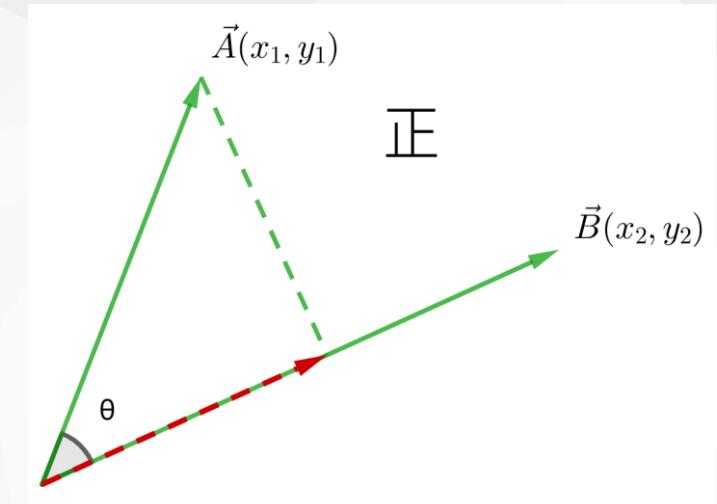
---

- 定義：  
給定兩向量  $\vec{A} = (x_1, y_1)$ ,  $\vec{B} = (x_2, y_2)$
- $\vec{A} \cdot \vec{B} = |\vec{A}| \times |\vec{B}| \times \cos \theta$   
 $= x_1 \times x_2 + y_1 \times y_2$
- 性質：
  - $\vec{A} \cdot \vec{B} = \vec{B} \cdot \vec{A}$
  - $\vec{A} \cdot (\vec{B} + \vec{C}) = \vec{A} \cdot \vec{B} + \vec{A} \cdot \vec{C}$



# 內積 (dot)

- 幾何意義：  
 $\vec{A}$  在  $\vec{B}$  的投影長度乘上  $\vec{B}$  的向量長度
- $\vec{A}$  與  $\vec{B}$  同向，內積為正
- $\vec{A}$  與  $\vec{B}$  反向，內積為負
- $\vec{A}$  與  $\vec{B}$  垂直，內積為 0



# 內積 (dot)

---

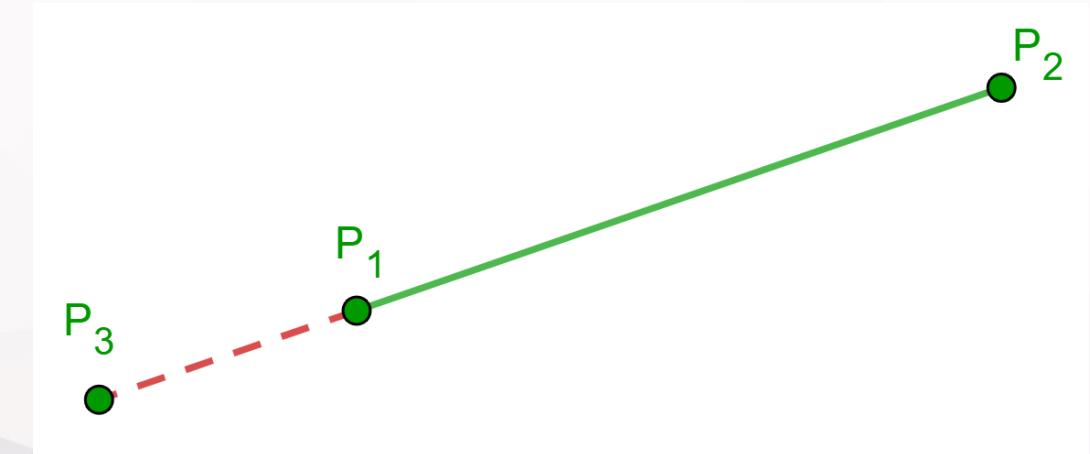
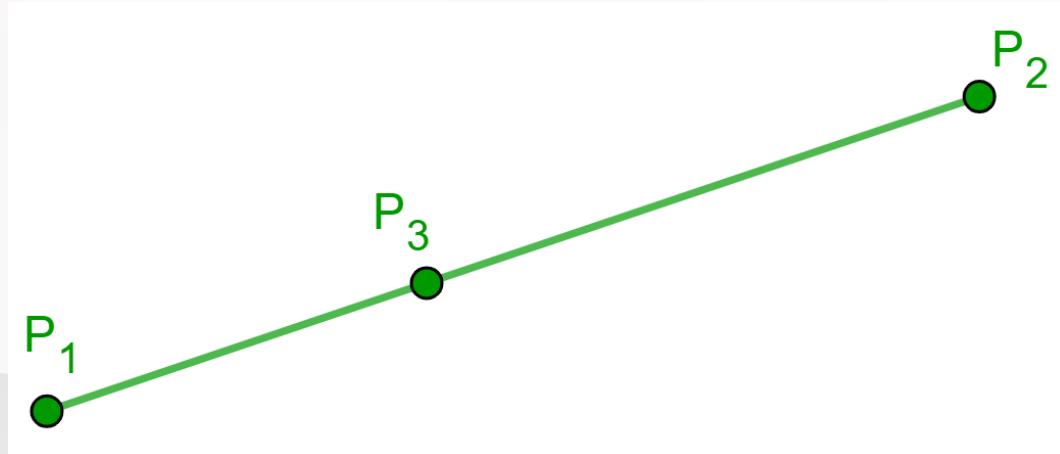
- 程式碼

```
double dot(const Point &b) const{
    return x * b.x + y * b.y;
}

Point p1(5, 6);
Point p2(3, 4);
cout << p1.dot(p2); // 輸出 39
```

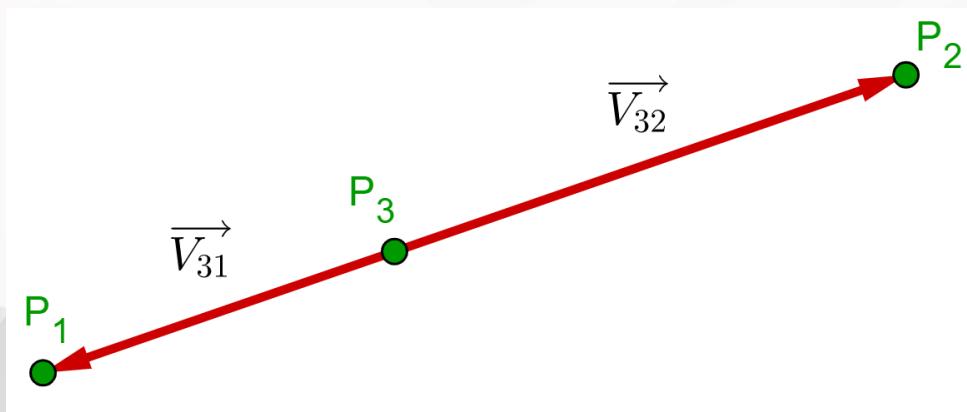
# 判斷點是否在線段內

- 一條直線上  $P_1, P_2, P_3$  三點，其中  $(P_1, P_2)$  形成一條線段，請問  $P_3$  是否在線段內

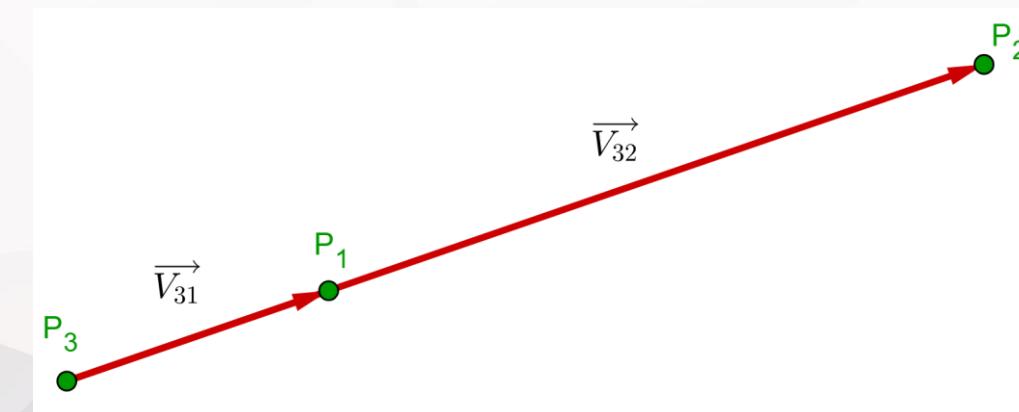


# 判斷點是否在線段內

- 利用內積(dot)
- 令兩向量  $\overrightarrow{V_{31}} = P_1 - P_3$ 、 $\overrightarrow{V_{32}} = P_2 - P_3$
- 可以發現若  $\overrightarrow{V_{31}} \cdot \overrightarrow{V_{32}} \leq 0$ ，則  $P_3$  位於線段  $(P_1, P_2)$  中



$$\overrightarrow{V_{31}} \cdot \overrightarrow{V_{32}} \leq 0$$



$$\overrightarrow{V_{31}} \cdot \overrightarrow{V_{32}} > 0$$

# 判斷點是否在線段內

---

- 程式碼

```
// 三點共線才能使用
bool btw(const Point &p1, const Point &p2, const Point &p3){
    return (p1 - p3).dot(p2 - p3) <= 0;
}
```

# 外積 (cross)

- 二維的外積定義：

給定兩向量  $\vec{A} = (x_1, y_1)$ ,  $\vec{B} = (x_2, y_2)$

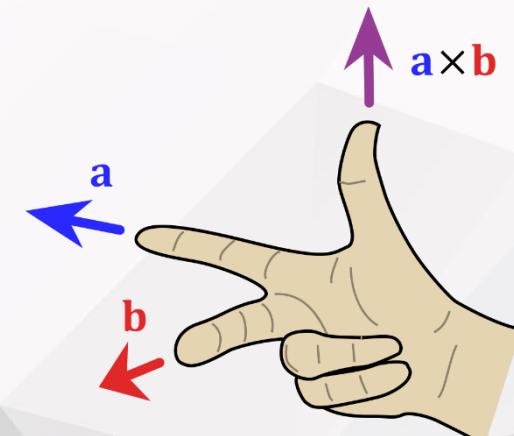
- $\vec{A} \times \vec{B} = |\vec{A}| \times |\vec{B}| \times |\sin \theta| \times n = x_1 \times y_2 - y_1 \times x_2$
- 其中  $n$  可以為 1 或是  $-1$

- 若  $\vec{A}$  到  $\vec{B}$  為逆時針方向，則  $n = 1$

- 若  $\vec{A}$  到  $\vec{B}$  為順時針方向，則  $n = -1$

- 性質：

-  $\vec{A} \times \vec{B} = -\vec{B} \times \vec{A}$

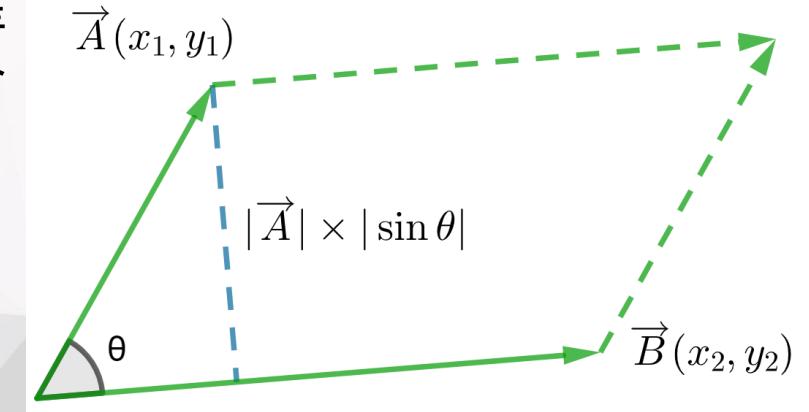


From Wikipedia

# 外積 (cross)

- 幾何性質：

- $|\vec{A}| \times |\sin \theta|$  為圖藍色線段
- $|\vec{B}| \times |\vec{A}| \times |\sin \theta|$  為  $\vec{A}, \vec{B}$  構成之平行四邊形面積
- $|\vec{A} \times \vec{B}|$  為  $\vec{A}, \vec{B}$  所構成平行四邊形的面積  
除以 2 即為  $\vec{A}, \vec{B}$  所構成三角形的面積



# 外積 (cross)

---

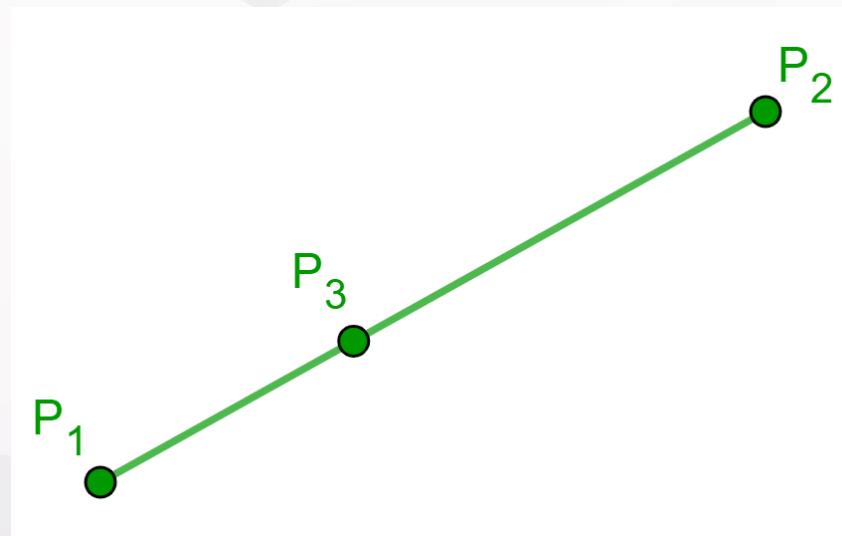
- 程式碼

```
double cross(const Point &b) const{
    return x * b.y - y * b.x;
}

Point p1(5, 6);
Point p2(3, 4);
cout << p1.cross(p2); // 輸出 2
```

# 判斷三點是否共線

- 給定  $P_1, P_2, P_3$  三點，請問這三點是否在同一條直線上



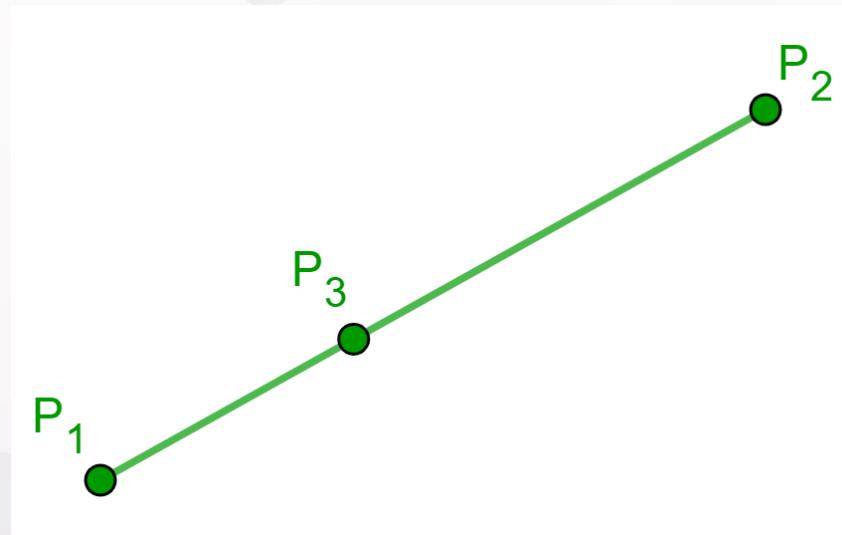
共線



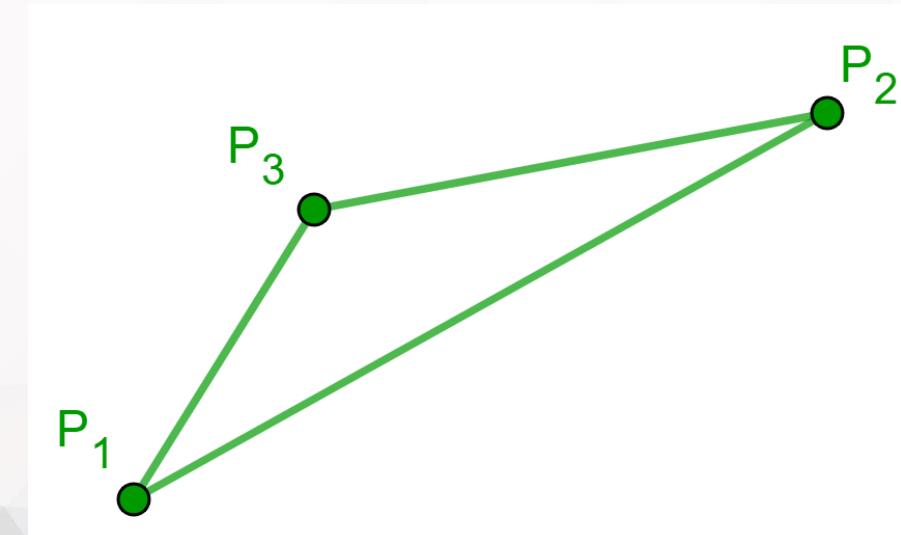
不共線

# 判斷三點是否共線

- 紿定  $P_1, P_2, P_3$  三點，請問這三點是否在同一條直線上
- 判斷三點所形成三角形之面積是否為 0



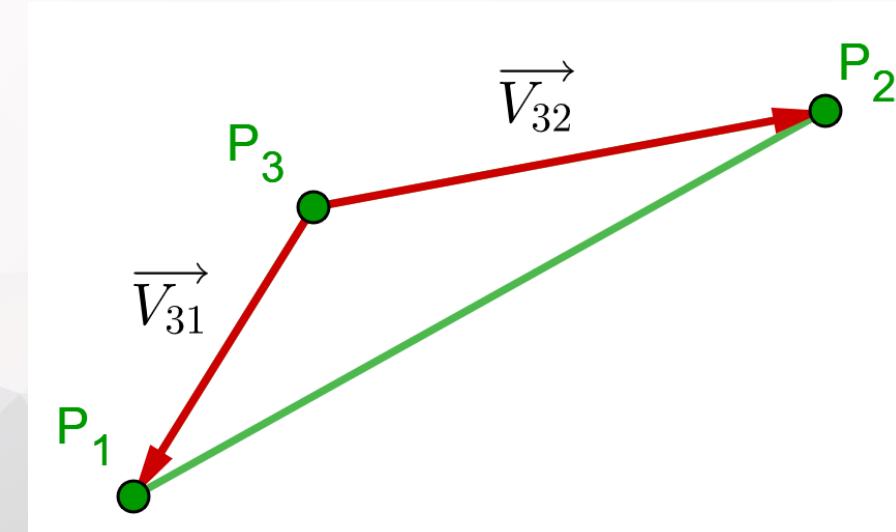
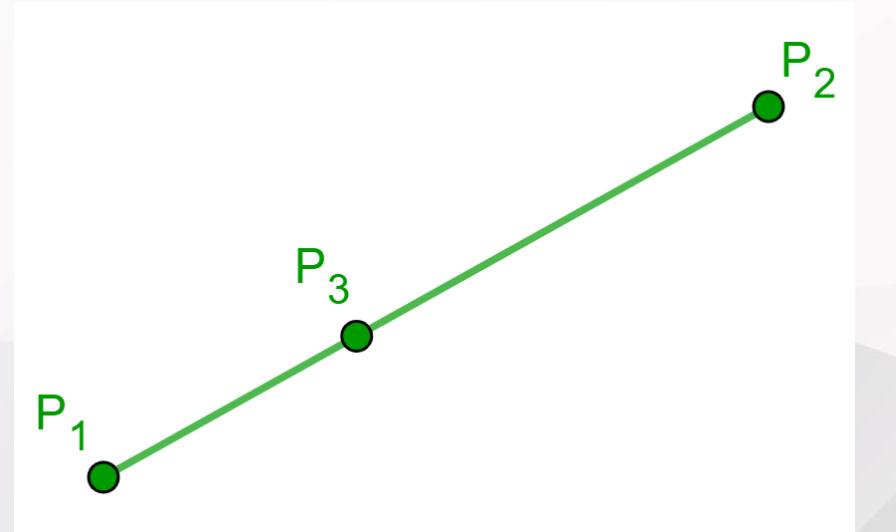
面積為 0



面積不為 0

# 判斷三點是否共線

- 利用外積(cross)
- 令兩向量  $\overrightarrow{V_{31}} = P_1 - P_3$ 、 $\overrightarrow{V_{32}} = P_3 - P_2$
- 若  $\overrightarrow{V_{31}} \times \overrightarrow{V_{32}} = 0$  則三點共線



# 判斷三點是否共線

---

- 程式碼

```
bool collinearity(const Point &p1, const Point &p2, const Point &p3){  
    return (p1 - p3).cross(p2 - p3) == 0;  
}
```

# 判斷 $P_3$ 是否在線段 $(P_1, P_2)$ 中

---

- 需要先判斷是否共線才能判斷是否在線段中
- 程式碼

```
bool pointOnSegment(const Point &p1, const Point &p2, const Point &p3){  
    return collinearity(p1, p2, p3) && btw(p1, p2, p3);  
}
```

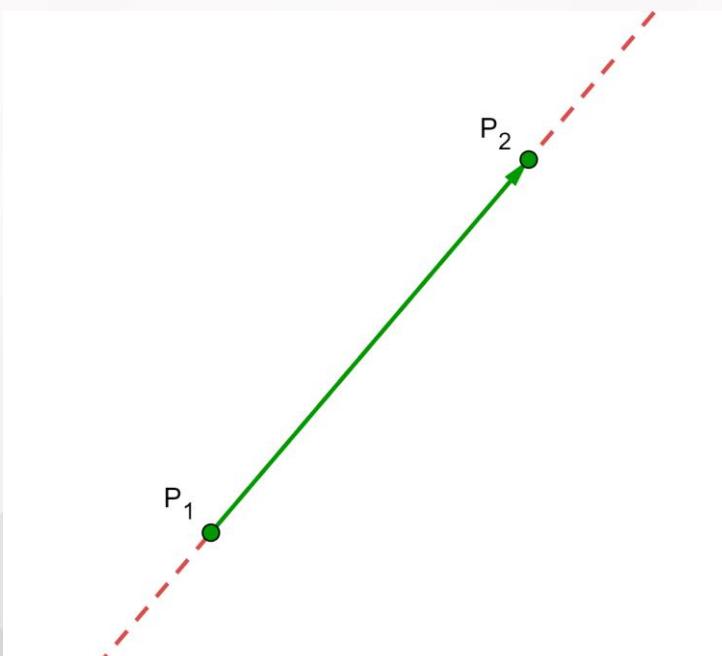
# 線段與有向面積

---

# 有向線段

---

- 對於一條線段  $\overline{P_1P_2}$ ，本單元會使用  $(P_1, P_2)$  表示
- 這樣代表這條線段的方向是從  $P_1$  到  $P_2$
- $(P_1, P_2)$  延伸的直線會將平面切成兩個部分



# 有向線段

---

- 定義：  
對於任一點  $P$ 
  - $(P_2 - P_1) \times (P - P_1) > 0$ ，則  $P$  位於  $(P_1, P_2)$  的正方向
  - $(P_2 - P_1) \times (P - P_1) < 0$ ，則  $P$  位於  $(P_1, P_2)$  的負方向
- 想像你站在  $P_1$  且面向  $P_2$ ，左手邊都屬於  $(P_1, P_2)$  正方向

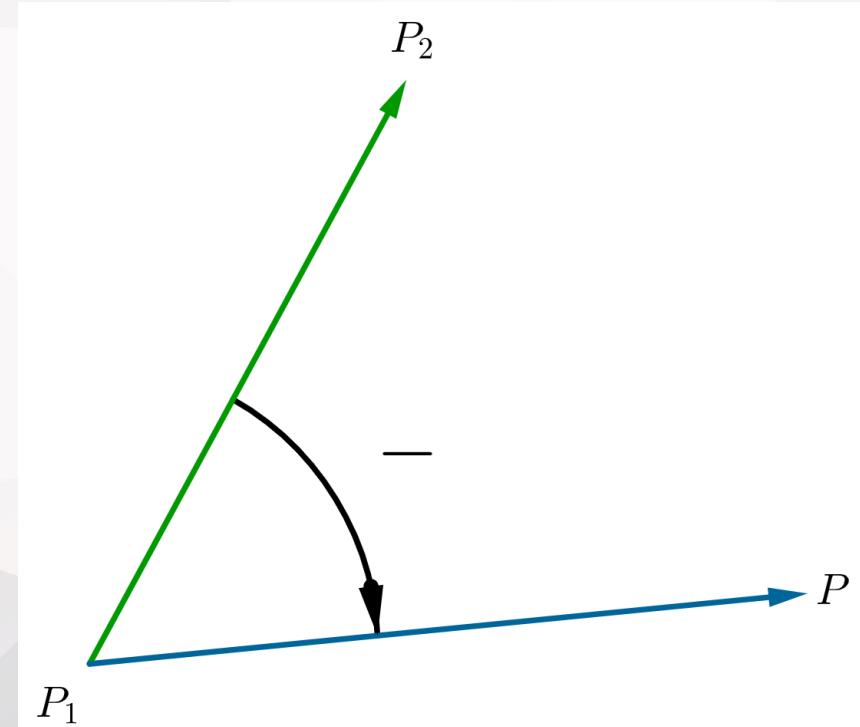
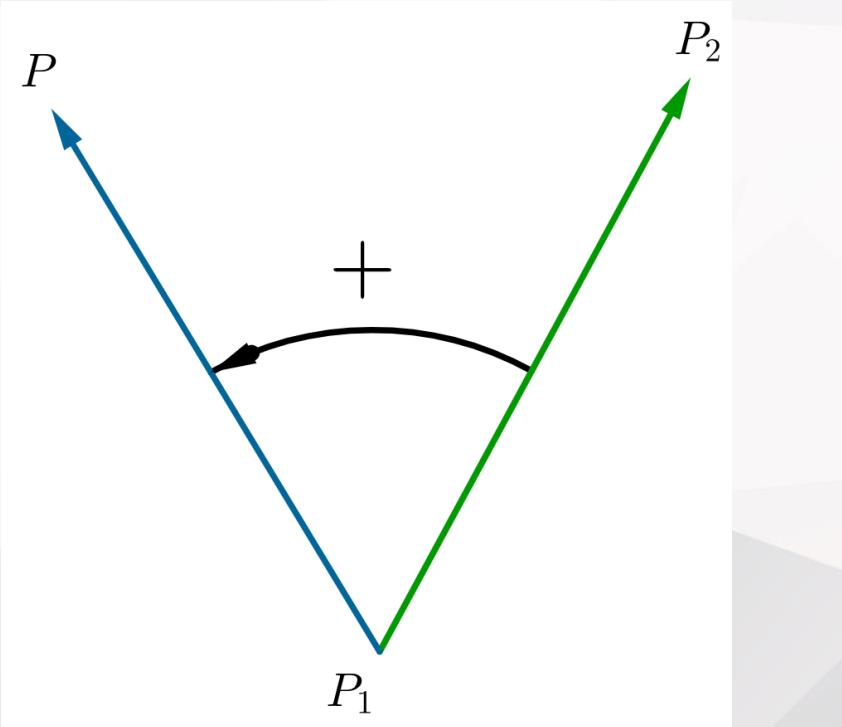
# 有向面積

---

- $\frac{|(P_2 - P_1) \times (P - P_1)|}{2}$  即為三點所組成之三角形面積
  - 由於外積有可能是負的，因此須加上絕對值
- 將絕對值去掉即為有向面積

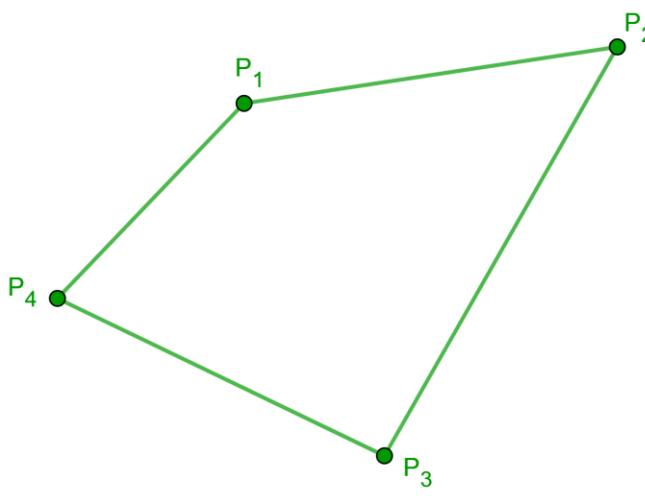
# 有向面積

- 逆時針旋轉為正，順時針為負

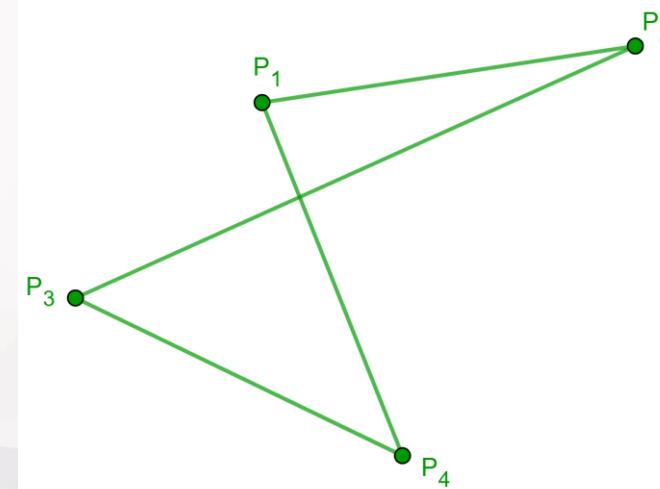


# 簡單多邊形面積

- 簡單多邊形
  - 除了相鄰邊交於頂點以外，邊不相交的多邊形
- 利用有向面積的正負性質可以方便計算簡單多邊形面積



簡單多邊形



非簡單多邊形

# 簡單多邊形面積

---

- 紿定一個  $n$  點的簡單多邊形，其點的逆時針順序依序為  $P_1, P_2, \dots, P_n$ ，計算該多邊形面積
- 任選一點  $P$ ，則多邊形面積可由以下公式得出

$$\sum_{i=0}^{n-1} \frac{(P_i - P) \times (P_{(i+1)\%n} - P)}{2}$$

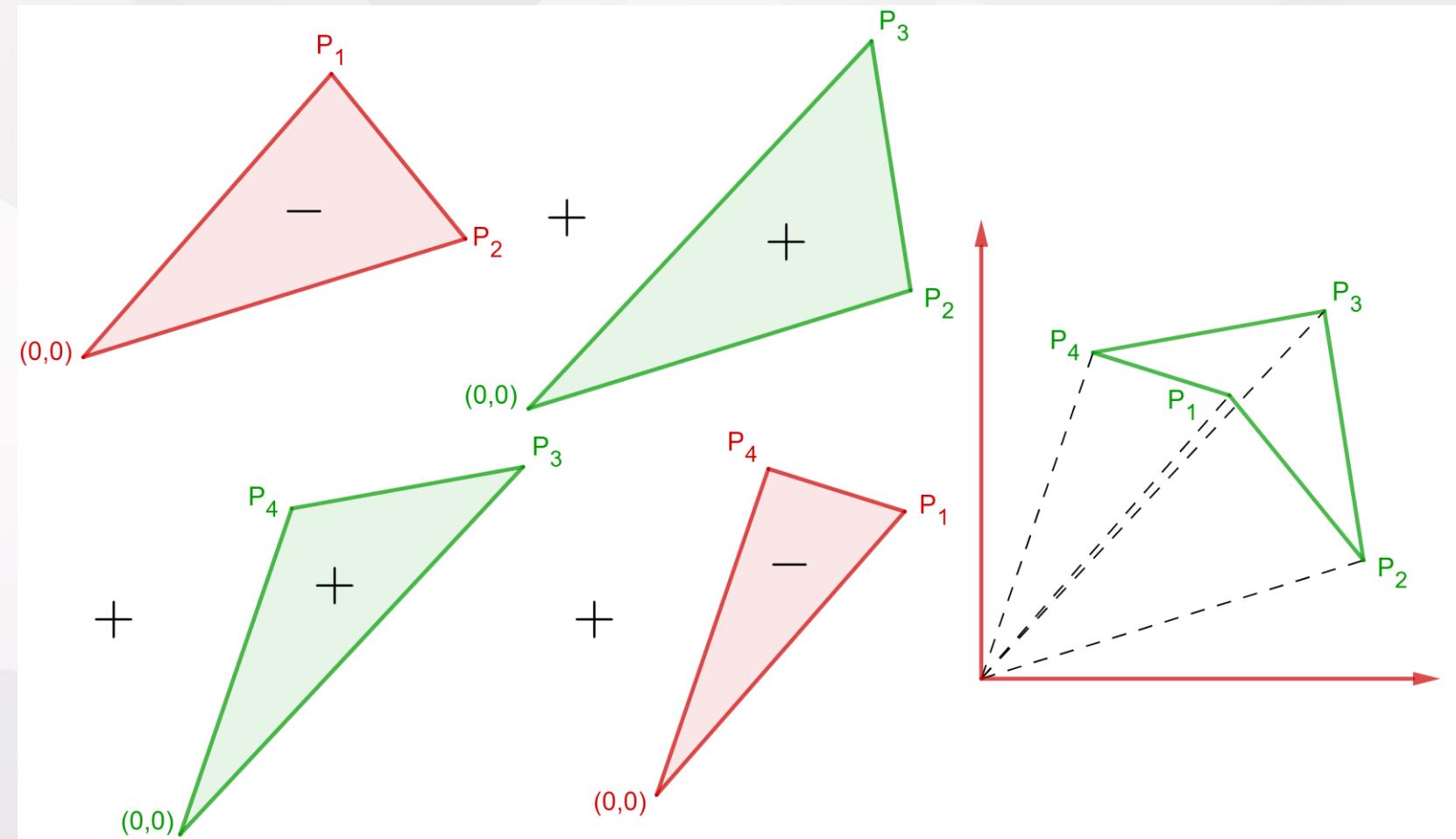
# 簡單多邊形面積

---

- 若取  $P = (0,0)$ ，則公式簡化為

$$\sum_{i=0}^{n-1} \frac{P_i \times P_{(i+1)\%n}}{2}$$

# 簡單多邊形面積



# 有向面積正負

---

- 因為有向面積正負判斷會常用到，因此會寫成函數
  - `ori(P1, P2, P3)`回傳  $(P_2 - P_1) \times (P_3 - P_1)$  的正負

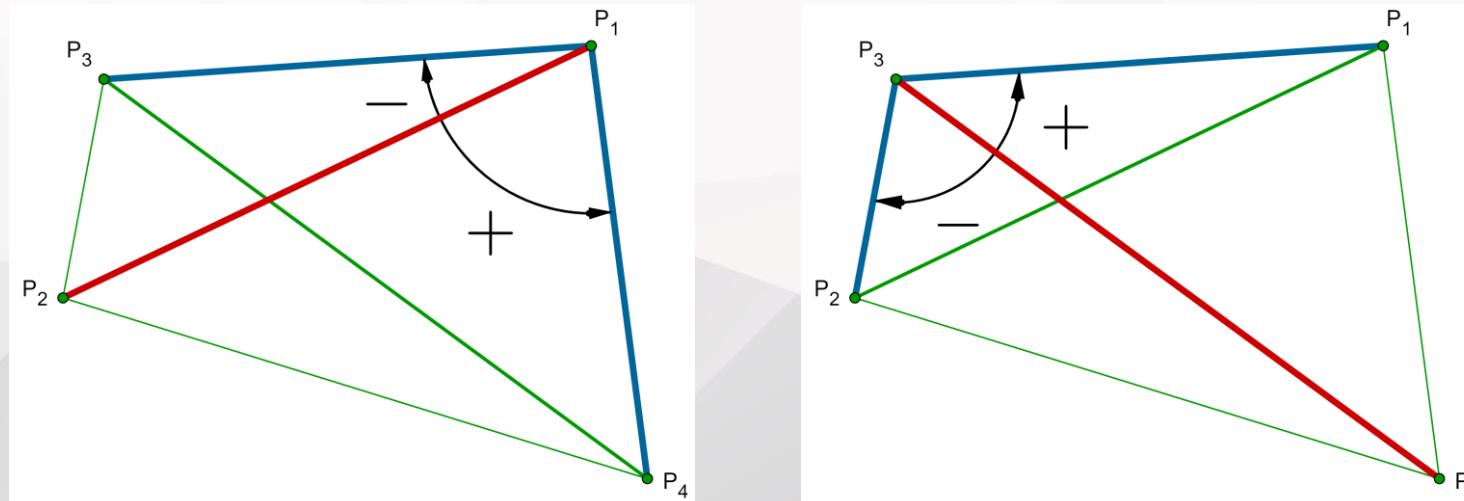
```
int ori(const Point &p1, const Point &p2, const Point &p3){  
    double d = (p2 - p1).cross(p3 - p1);  
    if(d == 0)  
        return 0;  
    return d > 0 ? 1 : -1;  
}
```

# 判斷線段相交、找出直線交點

# 判斷線段相交

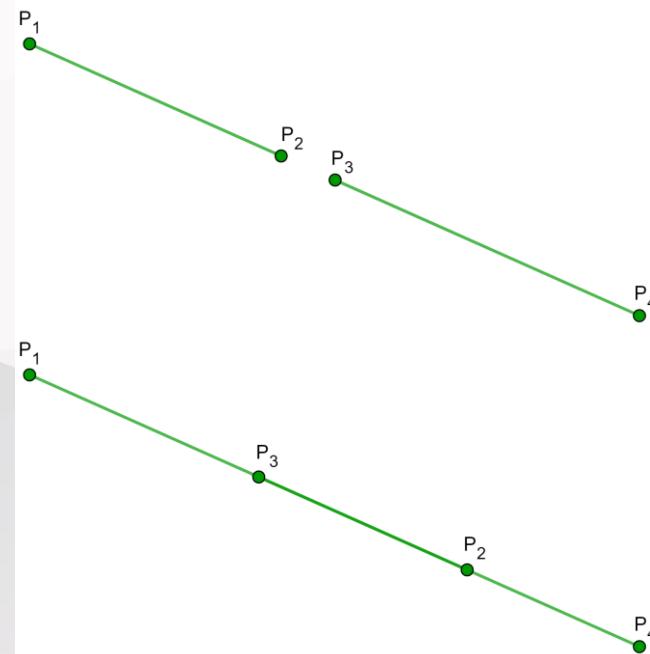
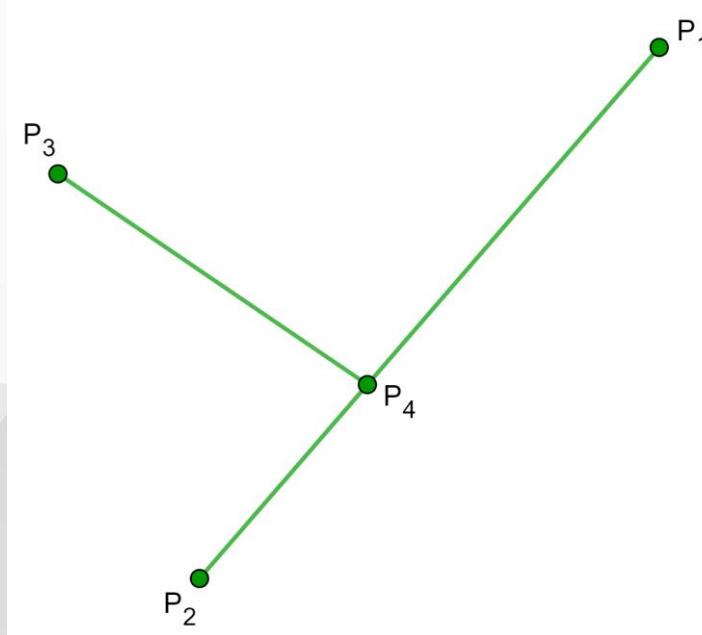
- 有了判斷有向面積正負的函數 `ori`，則可以很簡單的判斷兩線段  $(P_1, P_2), (P_3, P_4)$  是否相交
  - 若以下兩條件均成立，則兩線段相交

$$\text{ori}(P1, P2, P3) \times \text{ori}(P1, P2, P4) < 0$$
$$\text{ori}(P3, P4, P1) \times \text{ori}(P3, P4, P2) < 0$$



# 特例

- 下圖左邊的 case 會被判斷成不相交  
- 將  $<$  改成  $\leq$
- 下圖右邊的 case 都會被判斷成相交



# 特例

---

- 共線時若有相交，則一定會存在一線段的某一段點包含在另一條線段中
  - `btw` 函數
- 下面其中一條成立，則表示兩線段在共線時有相交
  - `btw(P1, P2, P3)`
  - `btw(P1, P2, P4)`
  - `btw(P3, P4, P1)`
  - `btw(P3, P4, P2)`

# 特例

---

- 判斷共線
  - 當 $\text{ori}(P1, P2, P3)$ 與 $\text{ori}(P1, P2, P4)$ 均為 0，則兩線段共線

# 判斷線段相交

---

- 程式碼

```
bool seg_intersect(const Point &p1, const Point &p2, const Point &p3, const Point &p4){  
    int a123 = ori(p1, p2, p3);  
    int a124 = ori(p1, p2, p4);  
    int a341 = ori(p3, p4, p1);  
    int a342 = ori(p3, p4, p2);  
    if(a123 == 0 && a124 == 0)  
        return btw(p1, p2, p3) || btw(p1, p2, p4) || btw(p3, p4, p1) || btw(p3, p4, p2);  
    return a123 * a124 <= 0 && a341 * a342 <= 0;  
}
```

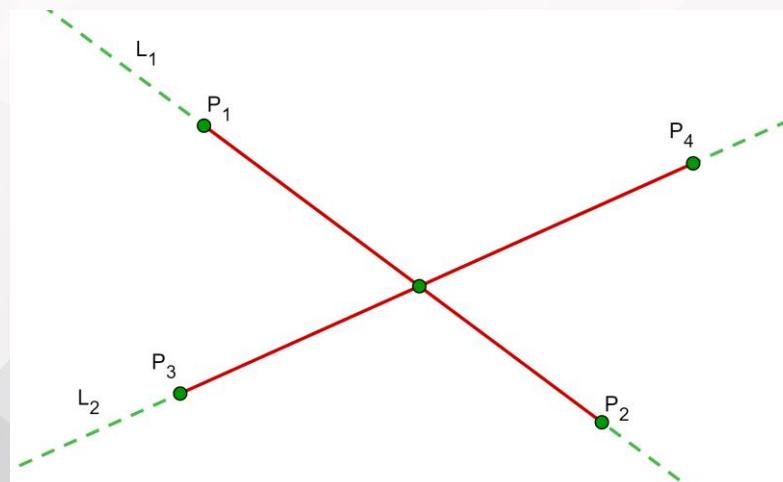
# 找出直線交點

---

- 解聯立方程式？
  - 最前面說過解聯立會有太多不好處理的問題
- 分點公式

# 找出直線交點

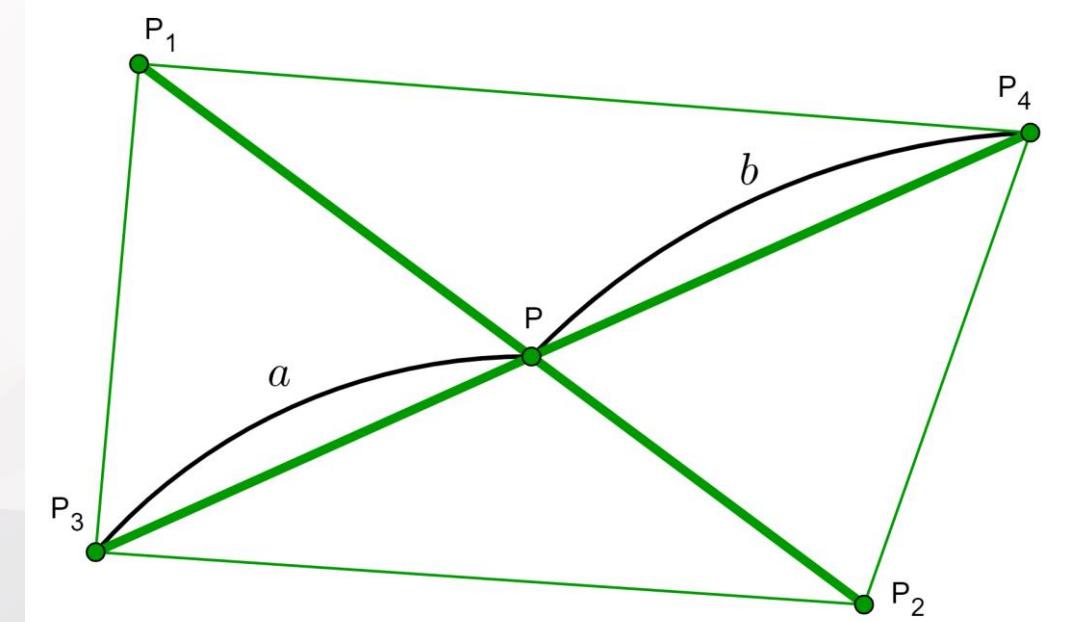
- 計算幾何中，直線會用線上的兩點表示
  - 紿定四個點  $P_1, P_2, P_3, P_4$   
其中經過  $P_1, P_2$  的直線為  $L_1$ ，經過  $P_3, P_4$  的直線為  $L_2$
  - 求出  $L_1, L_2$  的交點
- 分點公式只要利用外積即可求出兩直線交點



# 找出直線交點

- $a : b = \Delta P_1 P_2 P_3 : \Delta P_1 P_2 P_4$   
 $= (P_2 - P_1) \times (P_3 - P_1) : -(P_2 - P_1) \times (P_4 - P_1)$

$$P = \frac{\Delta P_1 P_2 P_3 \times P_4 + \Delta P_1 P_2 P_4 \times P_3}{\Delta P_1 P_2 P_3 + \Delta P_1 P_2 P_4}$$



# 找出直線交點

---

- 利用 `cross` 求出面積
  - 因為是有向面積，因此其中一邊要加上負號
- 交點不存在的話會有除以 0 的錯誤

# 找出直線交點

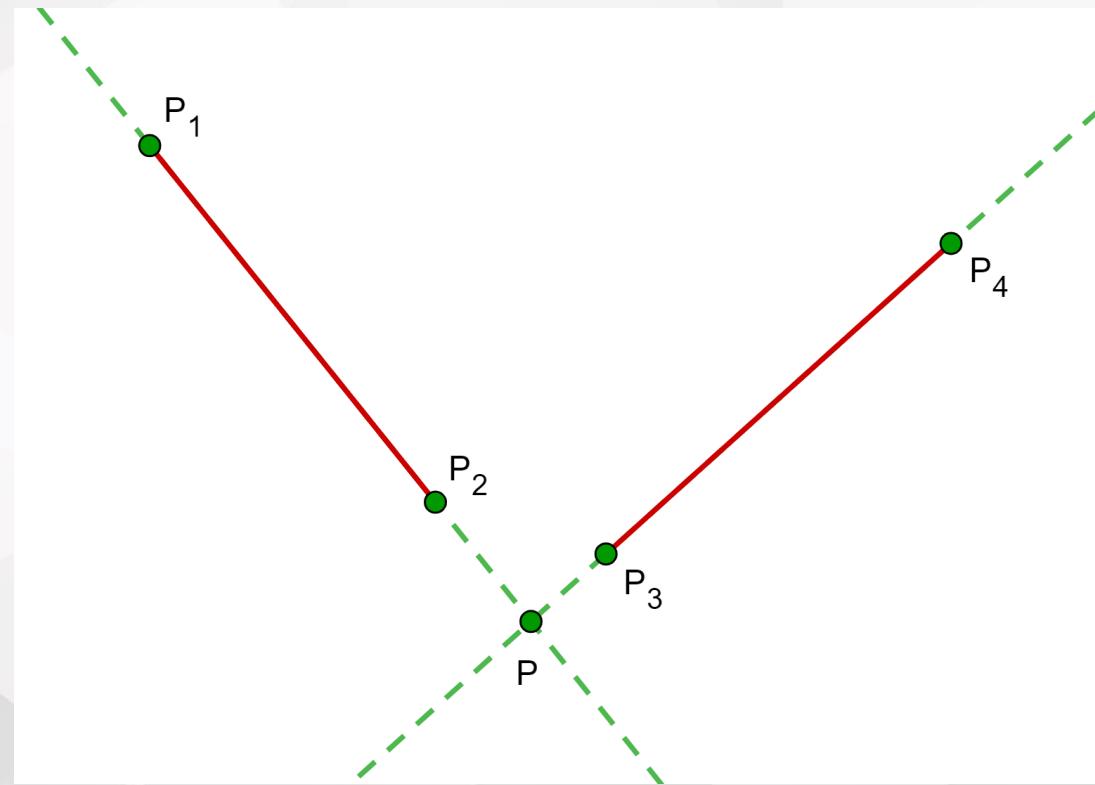
---

- 程式碼

```
Point intersect(const Point &p1, const Point &p2, const Point &p3, const Point &p4){  
    int a123 = (p2 - p1).cross(p3 - p1);  
    int a124 = (p2 - p1).cross(p4 - p1);  
    return (p4 * a123 - p3 * a124) / (a123 - a124);  
}
```

# 找出直線交點

- 即使是下圖這種 case 也可以求出來



# 計算幾何模板

---

# 計算幾何模板

---

- 通常遇到計算幾何的題目，都是直接把模板打上去要用到什麼就直接用
- 資料型態依照題目需求定義
  - 使用 `double` 型態時需要注意浮點數誤差，例如：  
 $(a == 0) \rightarrow (\text{abs}(a) \leq \text{EPS})$   
 $(a < 0) \rightarrow (a < \text{EPS})$
- 模板：<https://reurl.cc/vgVQ7k>

# 凸包與最遠點對

# 最遠點對

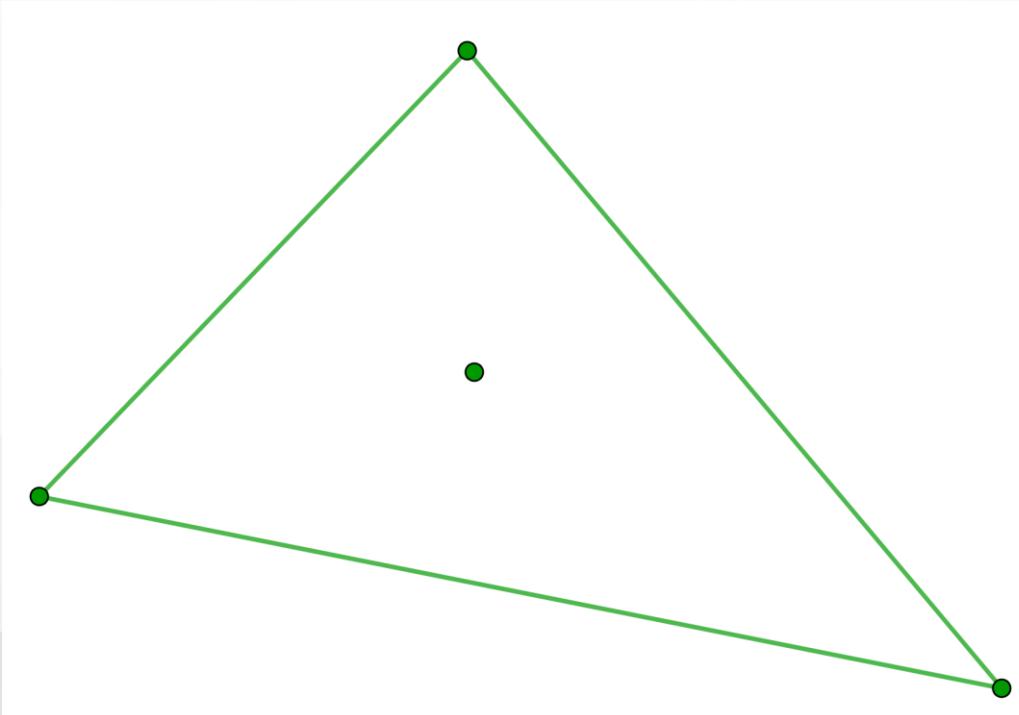
---

- 紿定二維平面上  $N$  個點，請求出最遠兩個點的距離平方
  - $2 \leq N \leq 2 \times 10^5$
  - 點座標  $(x, y)$  :  $-10^4 \leq x, y \leq 10^4$
- 這題  $N$  的範圍顯然無法使用  $O(N^2)$  枚舉所有點對的方法

# 最遠點對

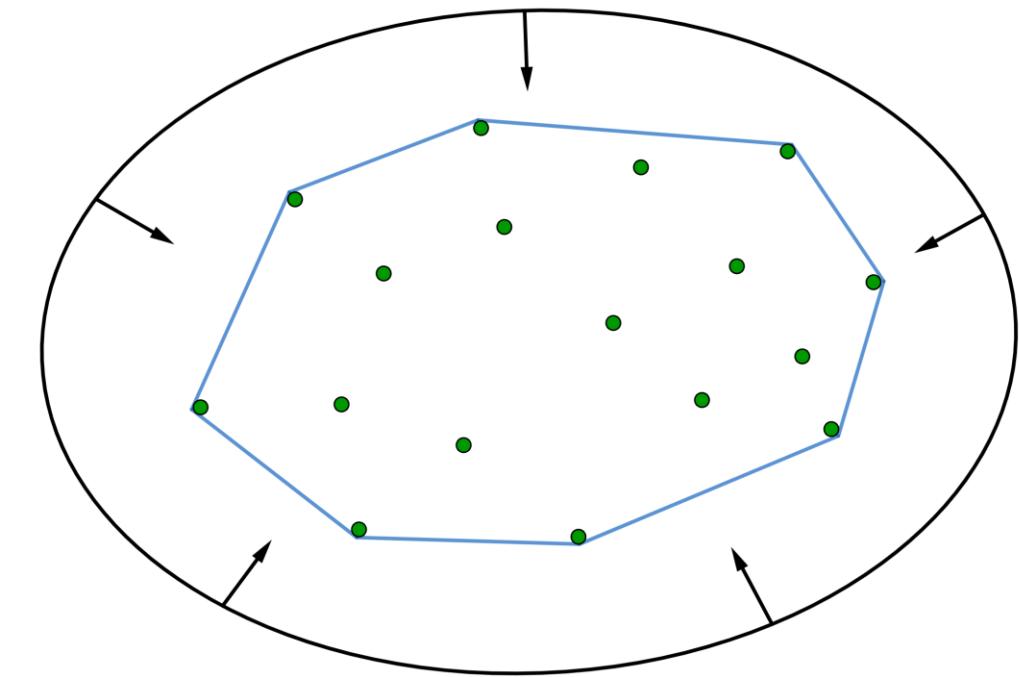
---

- 可以觀察到，三角形內部的點絕對不會是最遠點對的點
- 不在三角形內部的點  $\Rightarrow$  最外側的點



# 最遠點對

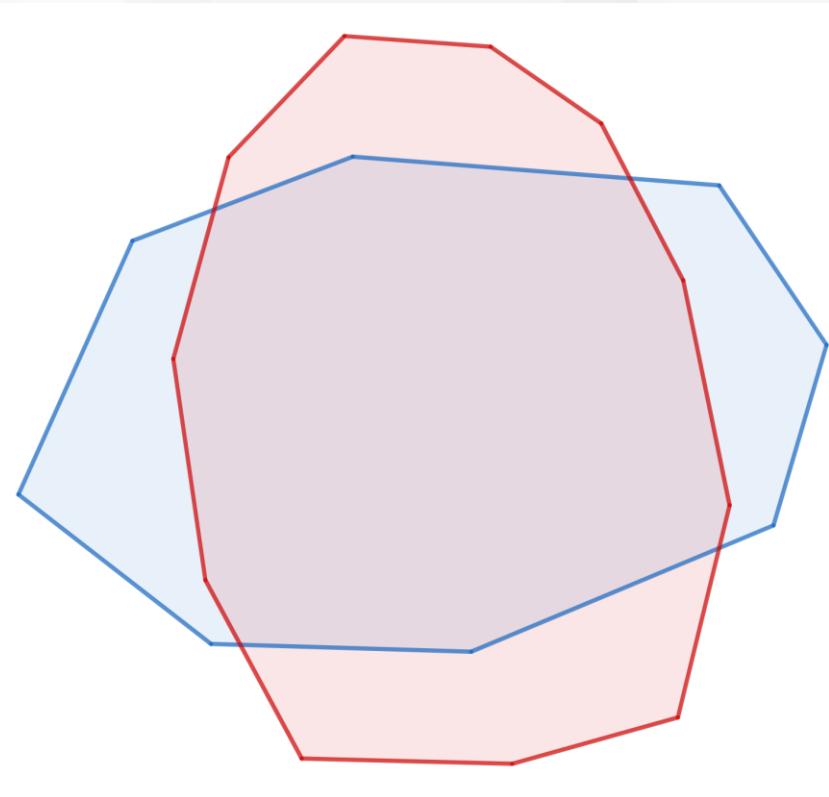
- 將所有點用橡皮筋圈起來，撐住橡皮筋的點就是最外側的點
- 這些點的集合稱為凸包
- 通常會用逆時針順序儲存



# 凸集合性質

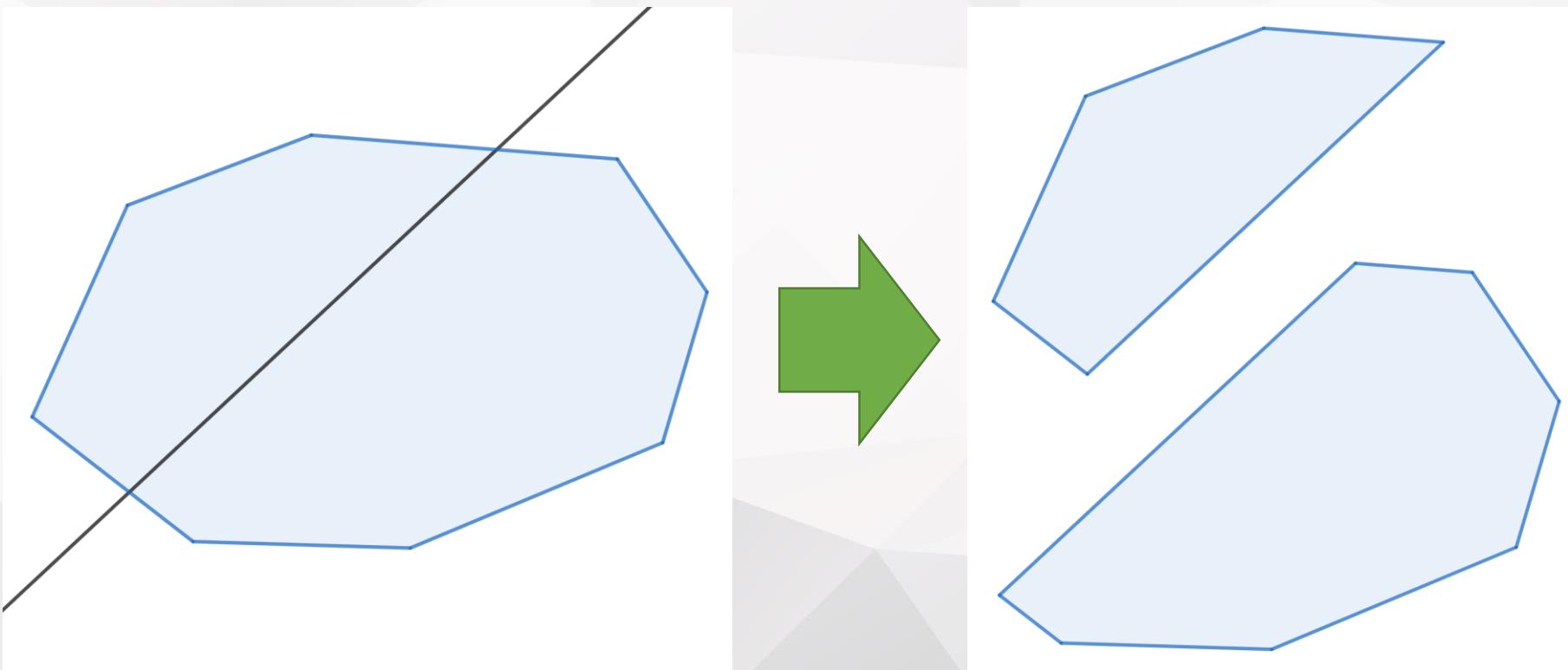
---

- 兩個凸包交集依舊是凸包



# 凸集合性質

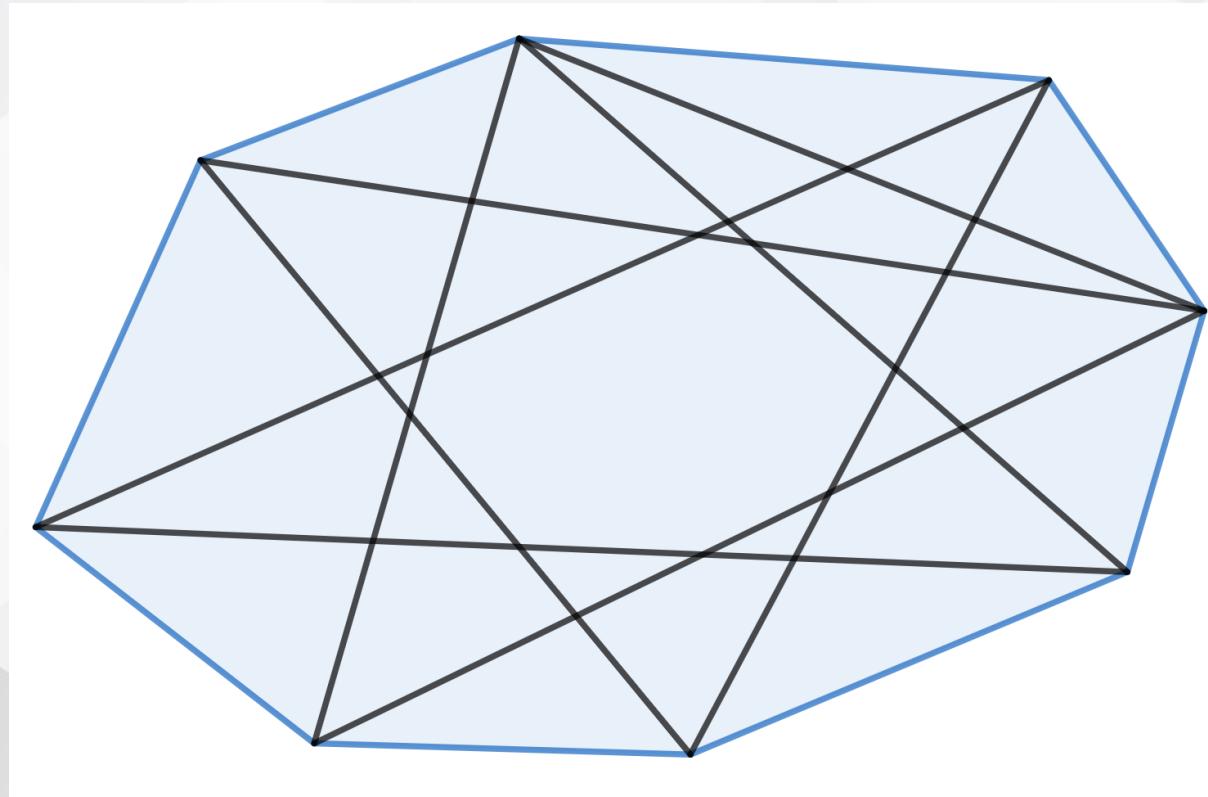
- 將一個凸包用直線切割還會是凸包



# 凸集合性質

---

- 凸包中任兩點連線都會在凸包內



# 整數點數限制

---

- 假設所有點座標均位於  $[0, w]$  的整數點  
則凸包上最多只會有  $O(\sqrt{w})$  個點  
因為凸包不會有三點共線的情況
- 用凸包上的點枚舉最遠點對複雜度為  $O(\sqrt{w}^2) = O(w)$

# 邊斜率單調

---

- 凸包上面的邊斜率會依序增加  
如果需要在邊上枚舉的話可以利用二分搜來降低複雜度
- 例如：判斷一個點是不是在凸包上  $O(\log N)$

# 找出凸包

# 找出凸包

---

- 建構凸包有很多的方法
- 這邊使用 Graham 掃描法

# 排序

---

- 一開始先將所有的點依照  $x$  座標由小到大排序，如果  $x$  座標一樣就依照  $y$  座標排序

```
bool cmp(const Point &a, const Point &b){  
    return (a.x < b.x) || (a.x == b.x && a.y < b.y);  
}
```

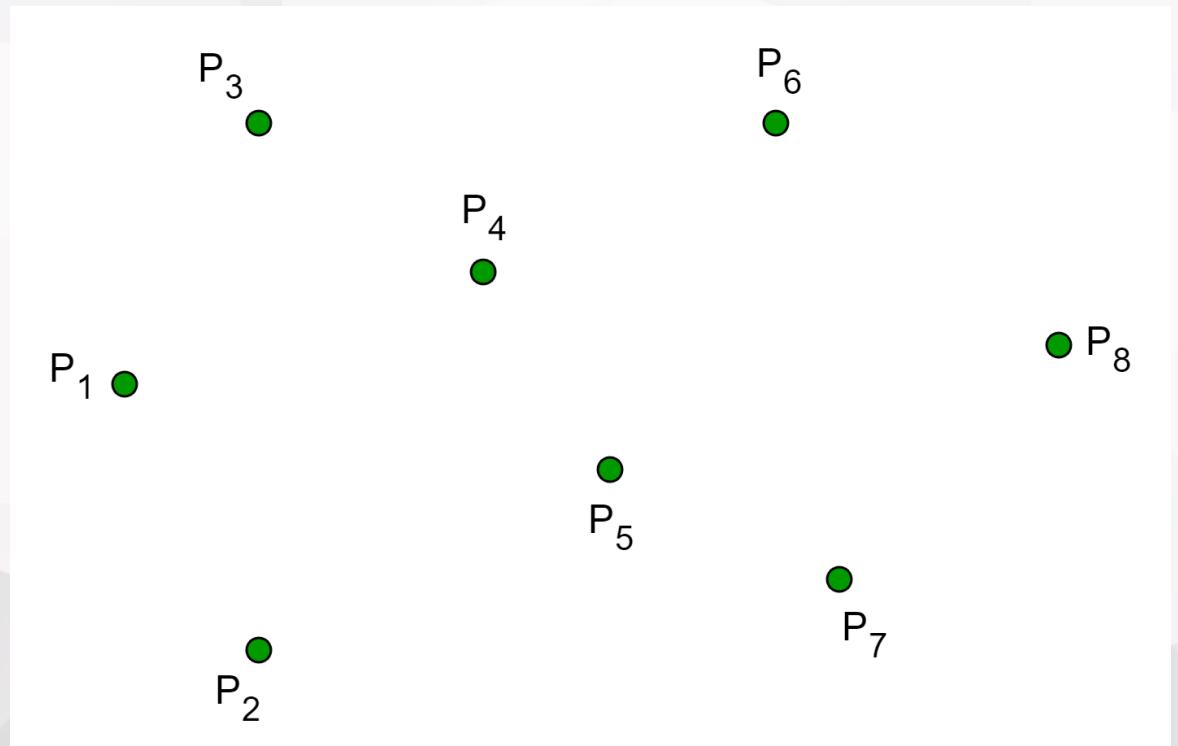
# 找出凸包

---

- 排序完後的第一個點與最後一個點一定會在凸包上
- 這兩個點連線可以將整個凸包分成上側凸包與下側凸包
- 找出上下側凸包的方法大同小異，這邊先以下側凸包為例

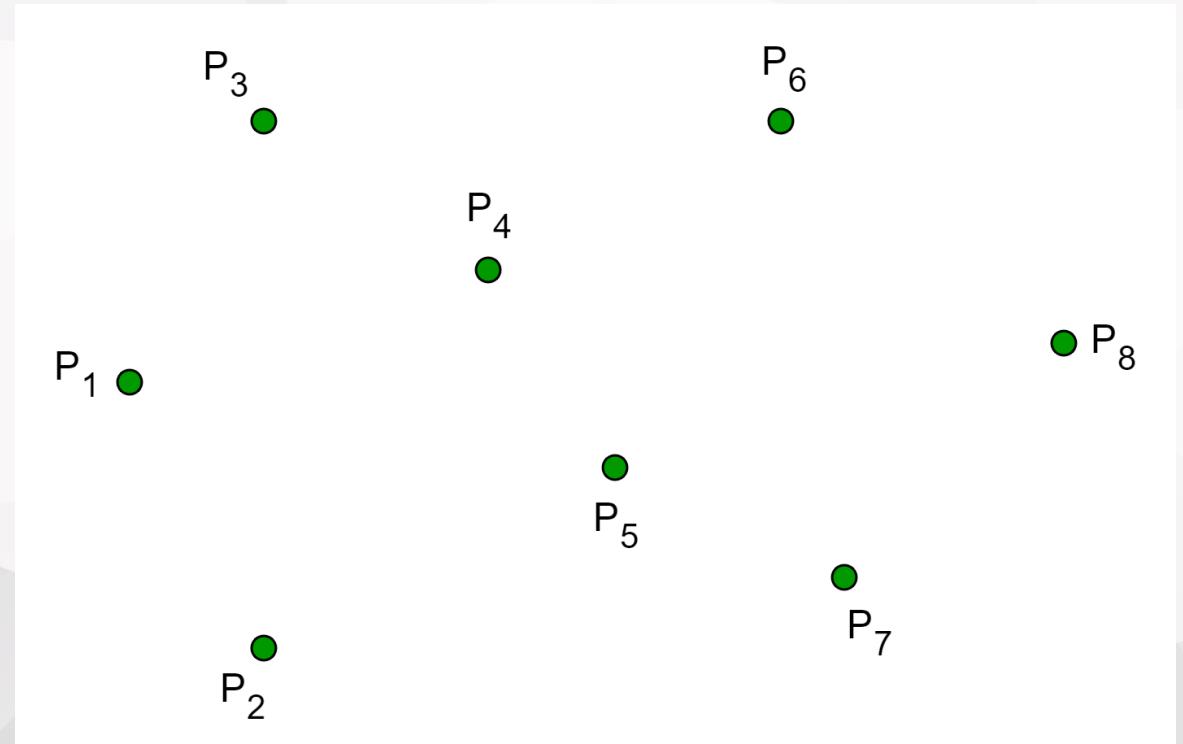
# 找出下側凸包

- 尋找以下點集的凸包



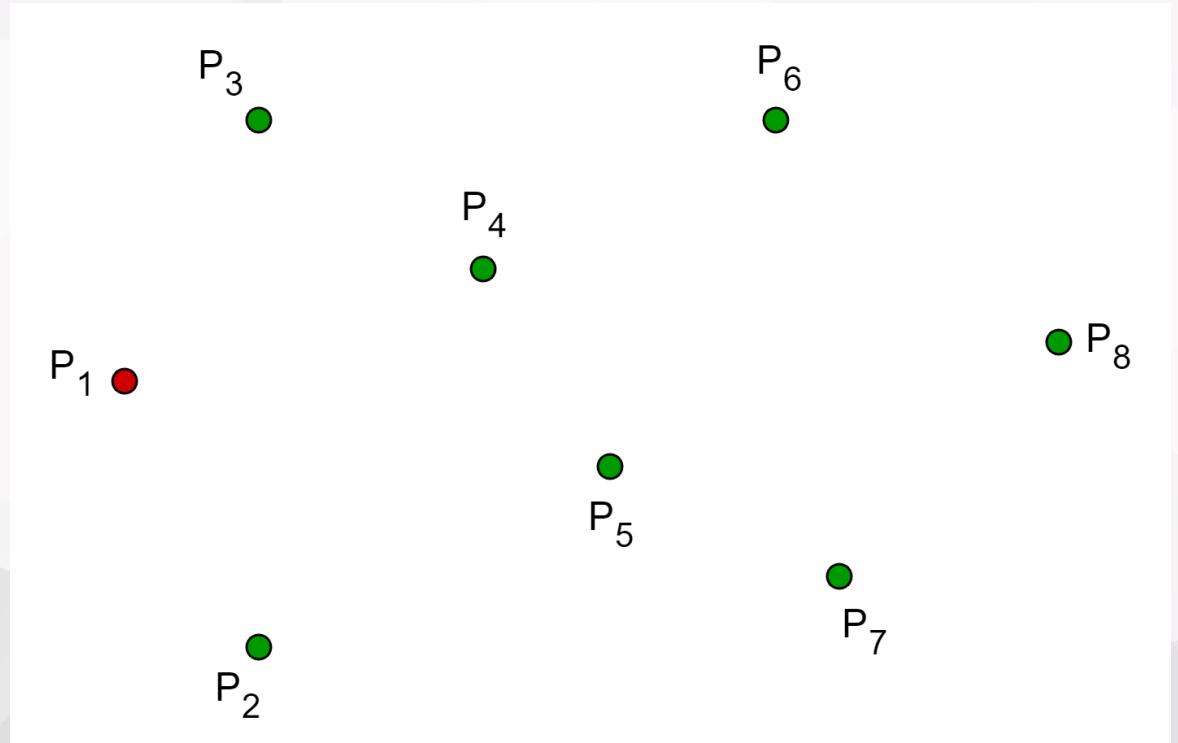
# 找出下側凸包

- 一開始什麼點都沒有



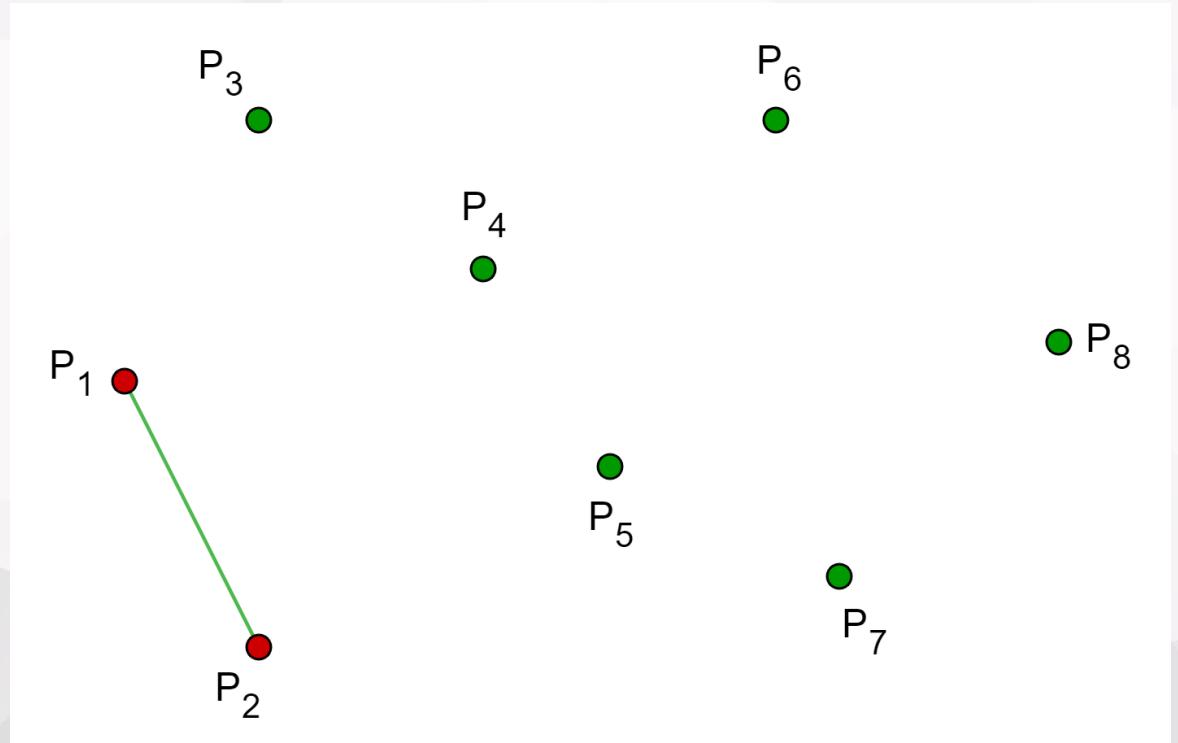
# 找出下側凸包

- 將  $P_1$  加入凸包



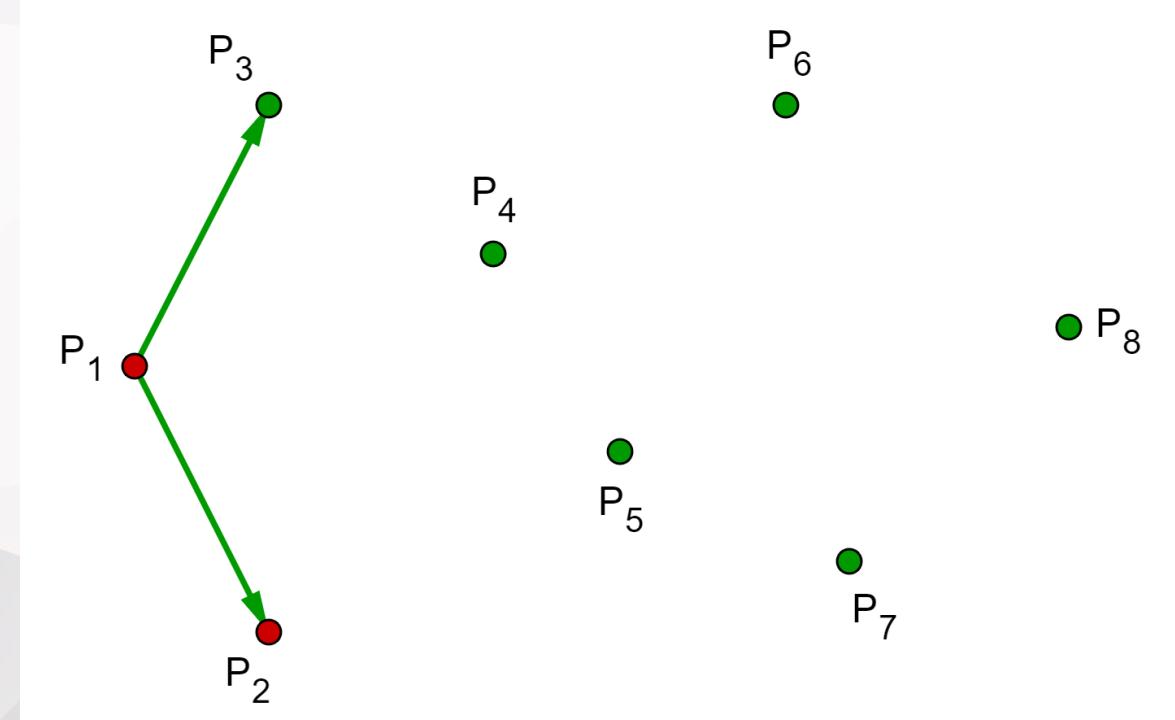
# 找出下側凸包

- 將  $P_2$  加入凸包



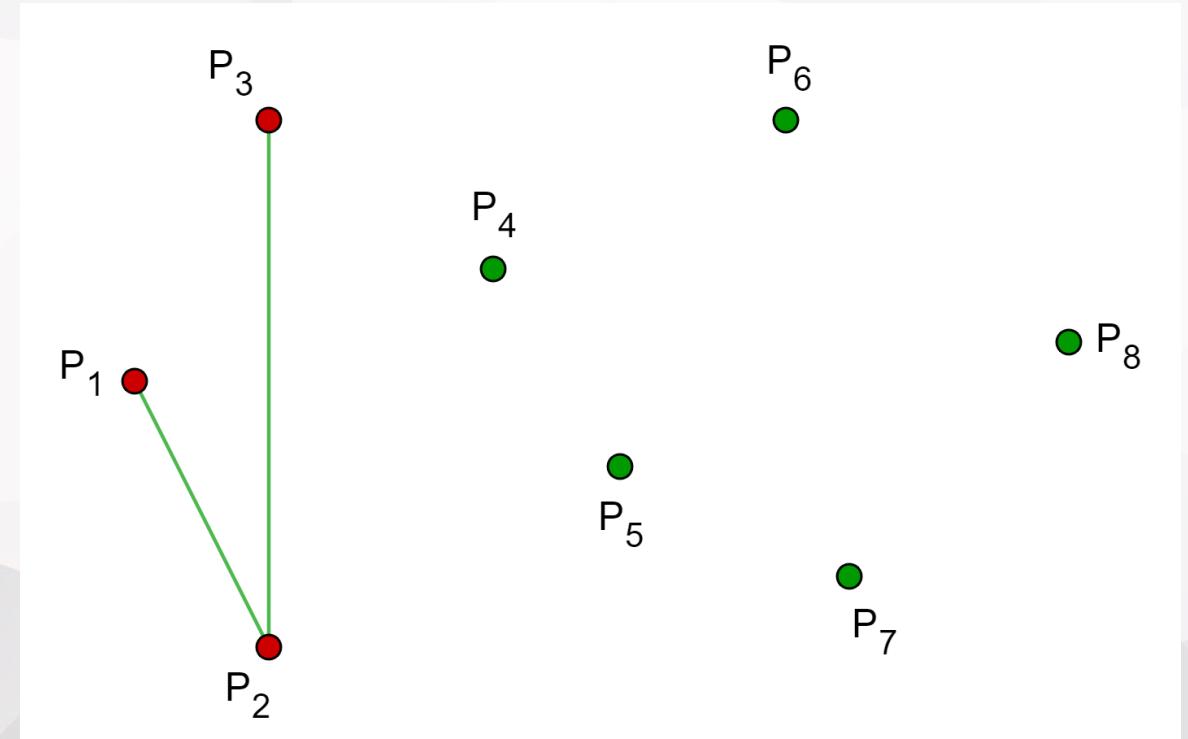
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



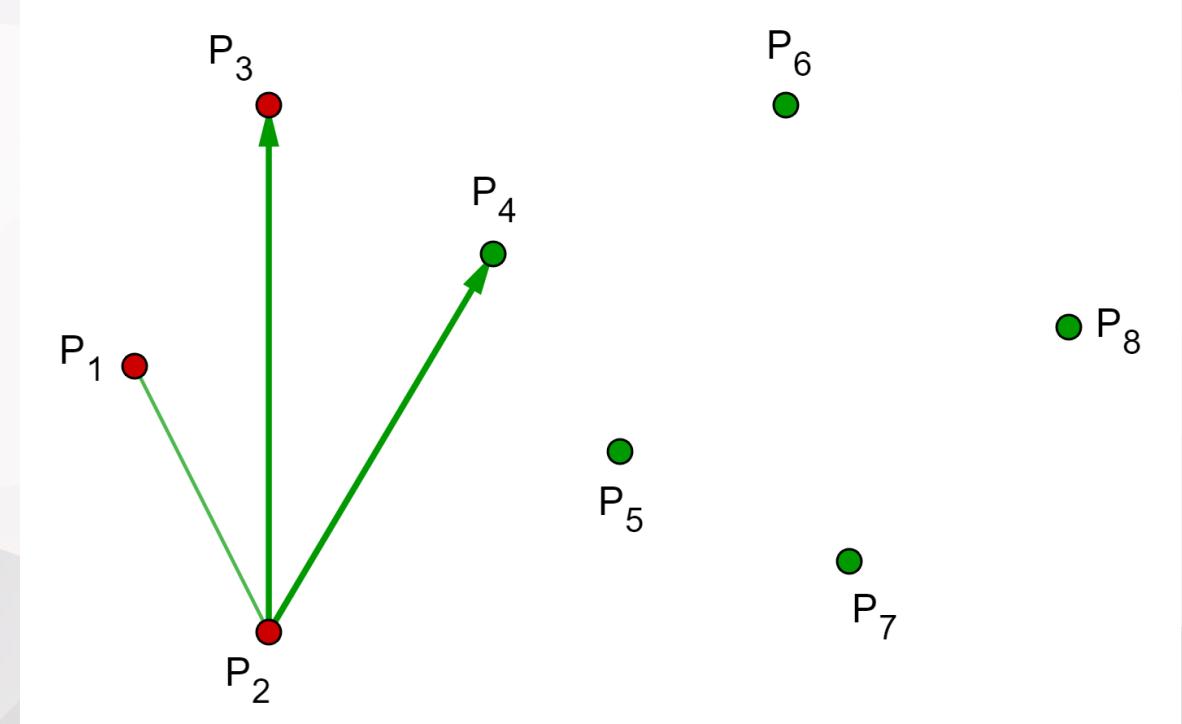
# 找出下側凸包

- $\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3} > 0$ ，不會造成凹陷
- 將  $P_3$  加入凸包



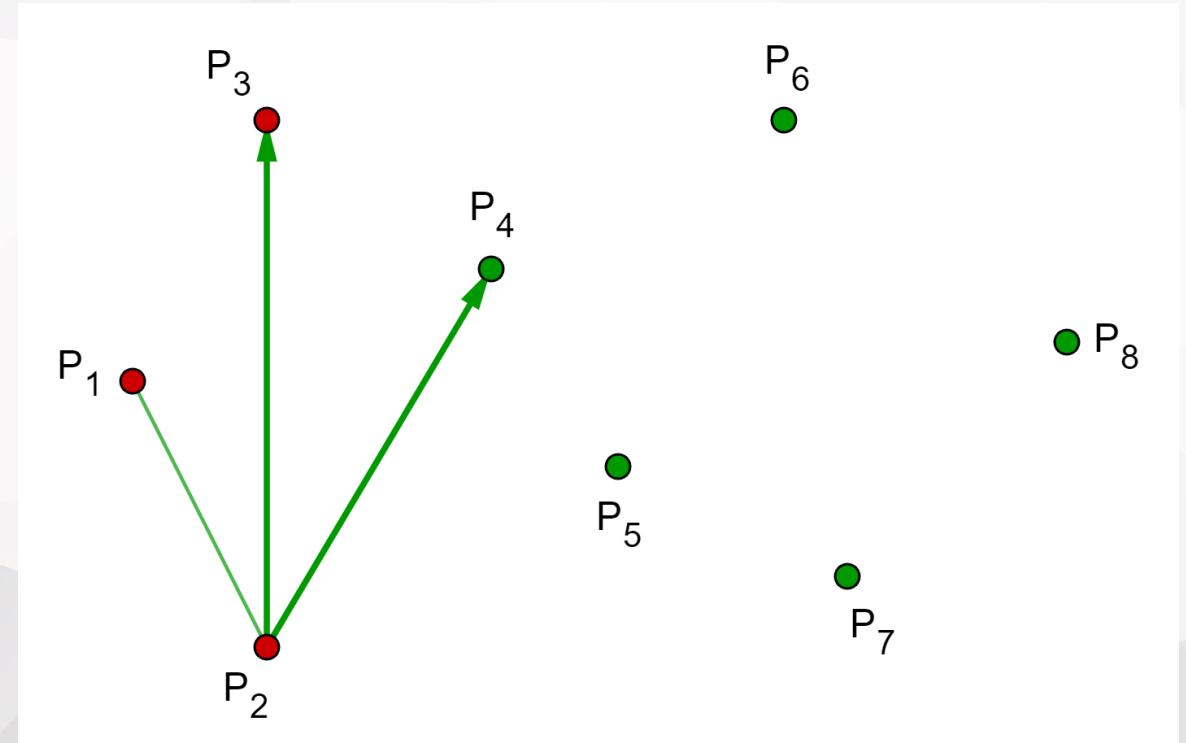
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



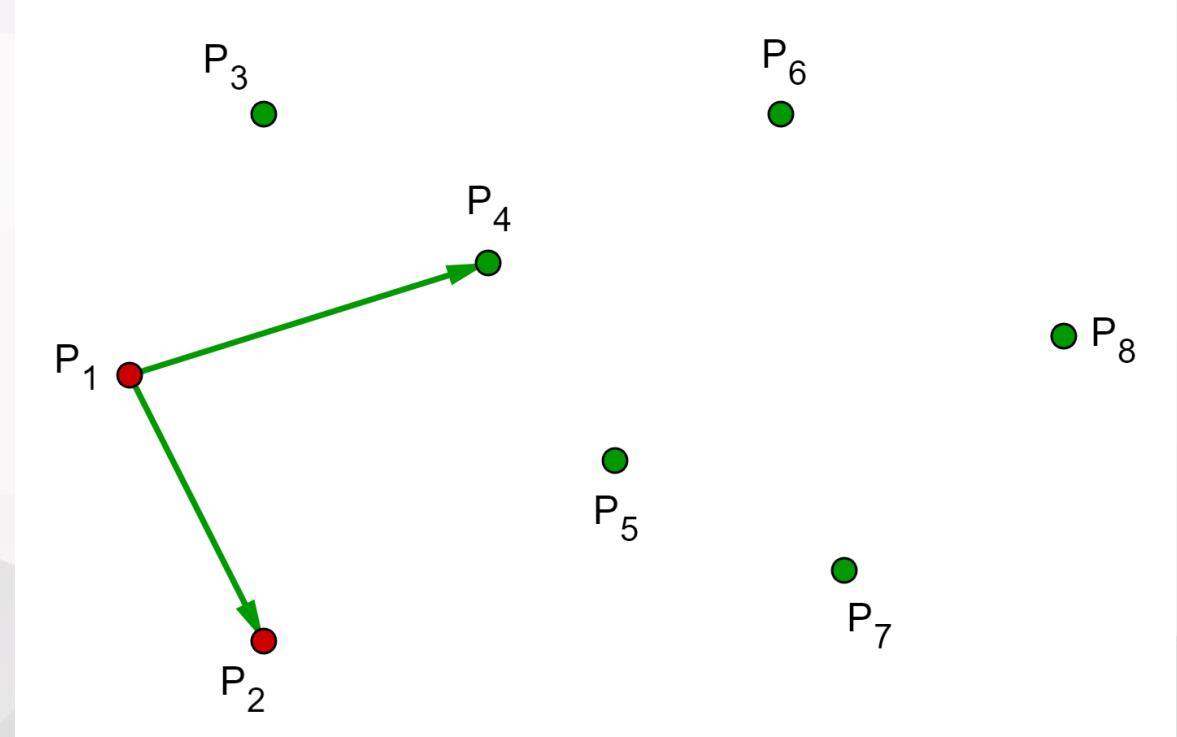
# 找出下側凸包

- $\overrightarrow{P_2P_3} \times \overrightarrow{P_2P_4} \leq 0$ ，會造成凹陷
- 將  $P_3$  移出凸包



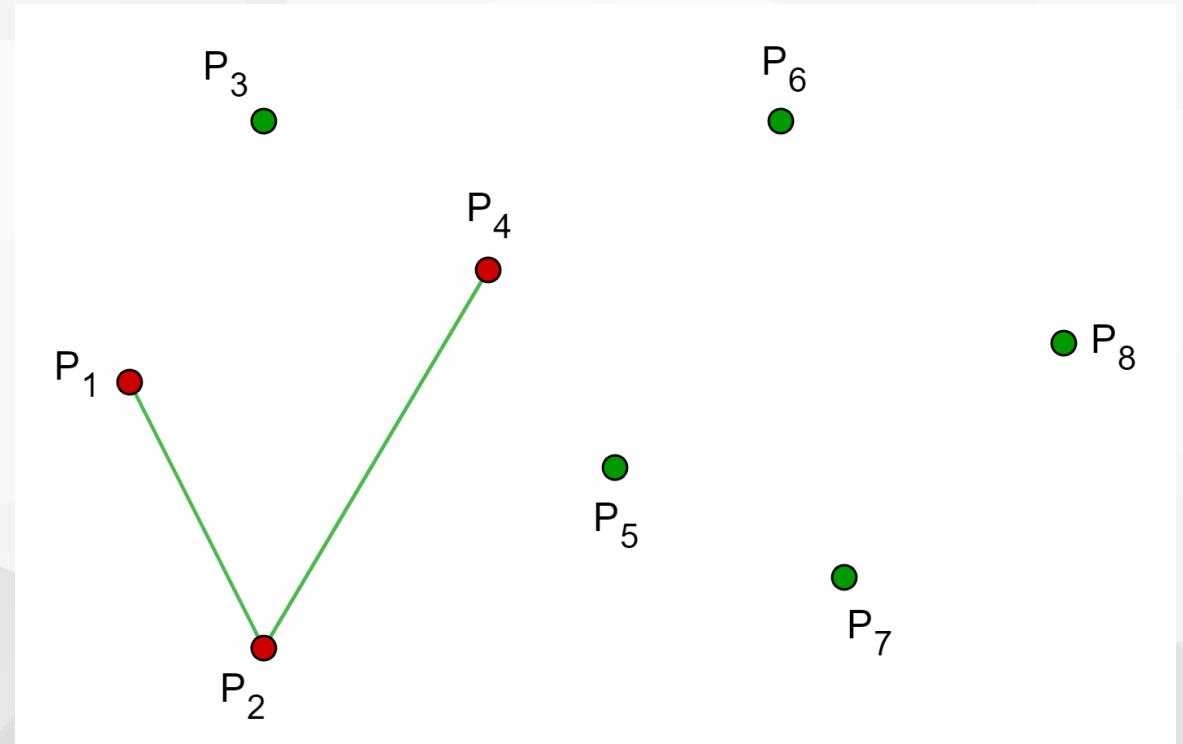
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



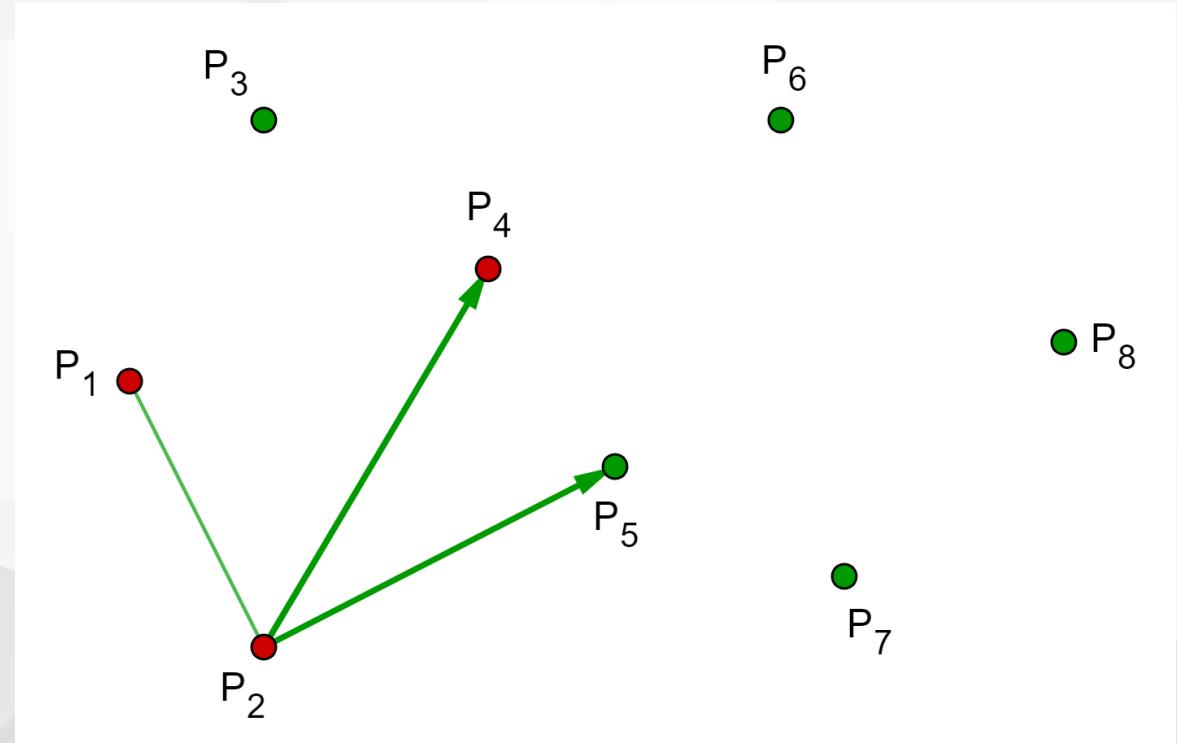
# 找出下側凸包

- $\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_4} > 0$ ，不會造成凹陷
- 將  $P_4$  加入凸包



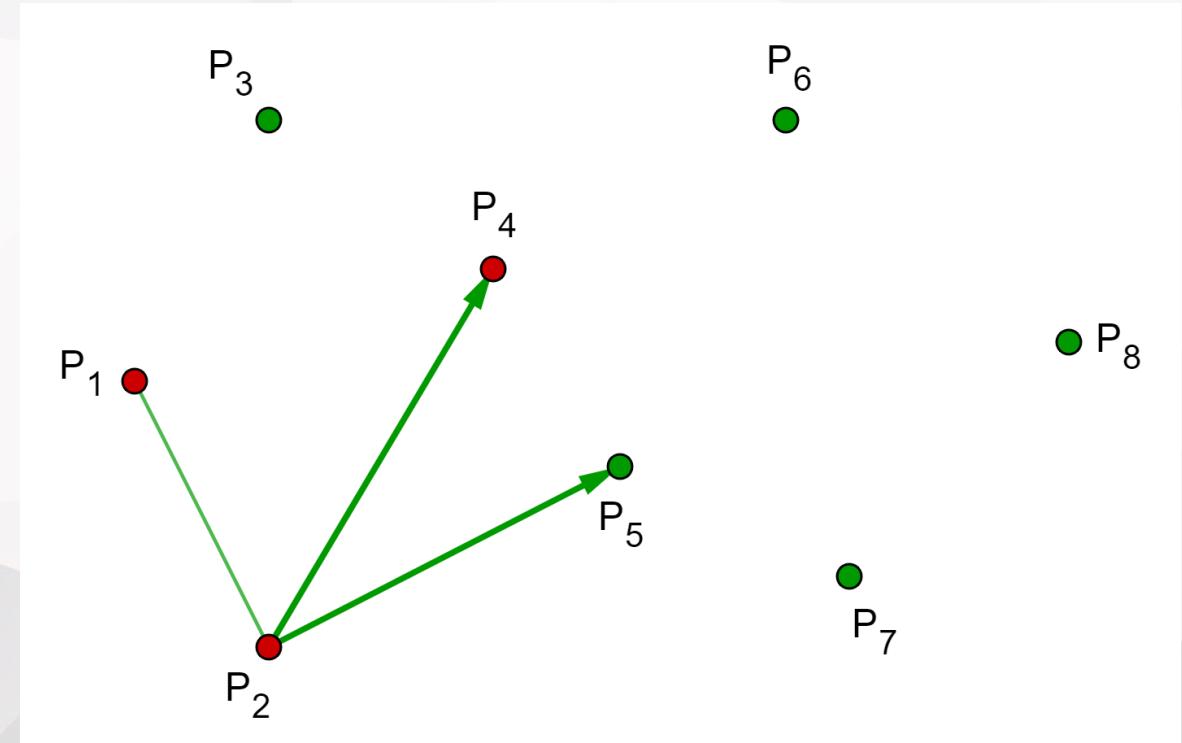
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



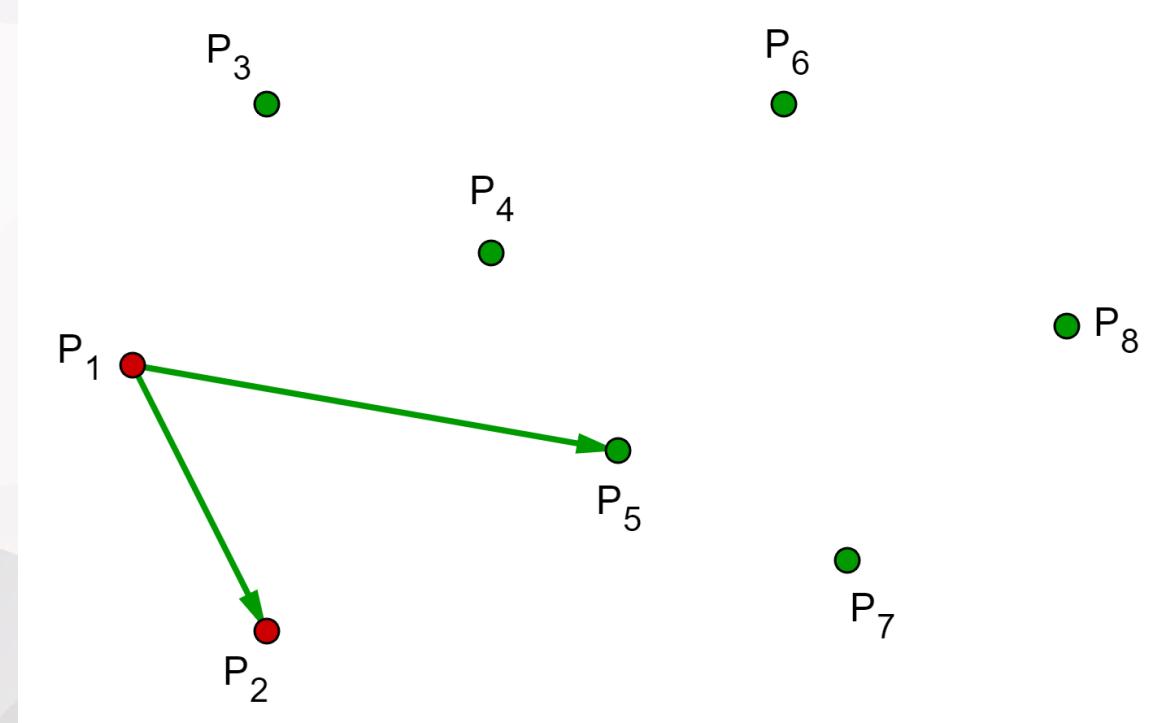
# 找出下側凸包

- $\overrightarrow{P_2P_3} \times \overrightarrow{P_2P_4} \leq 0$ ，會造成凹陷
- 將  $P_4$  移出凸包



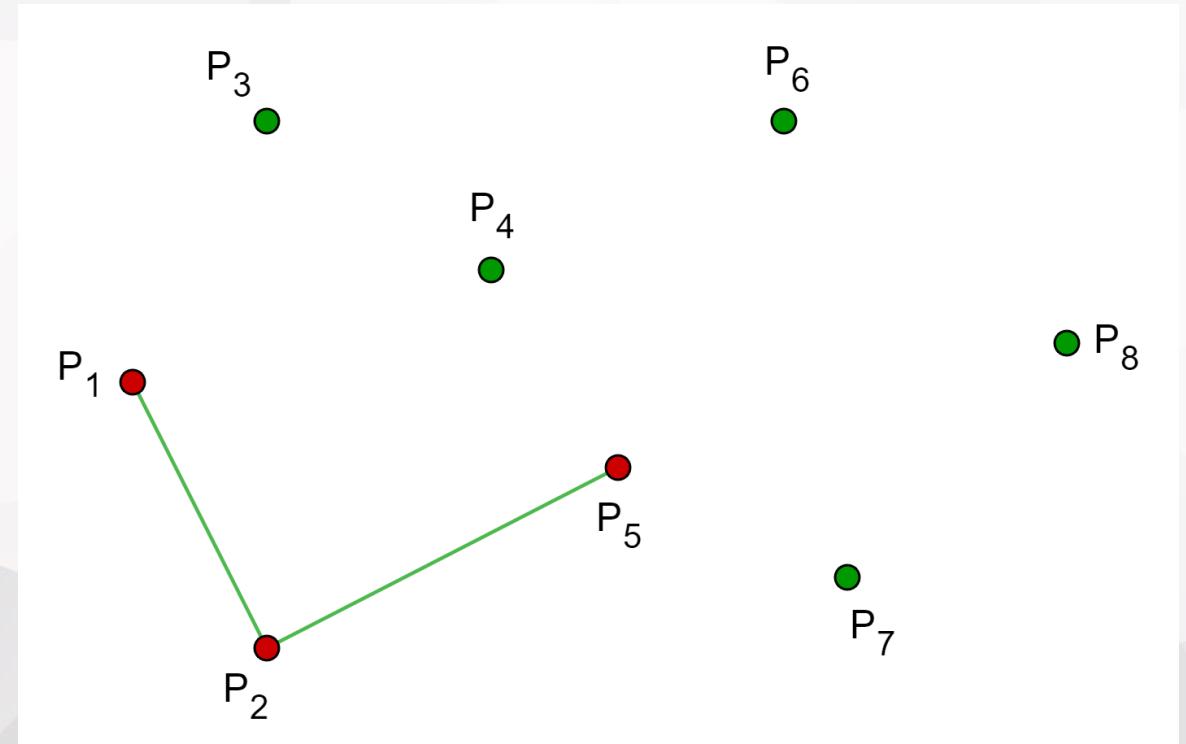
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



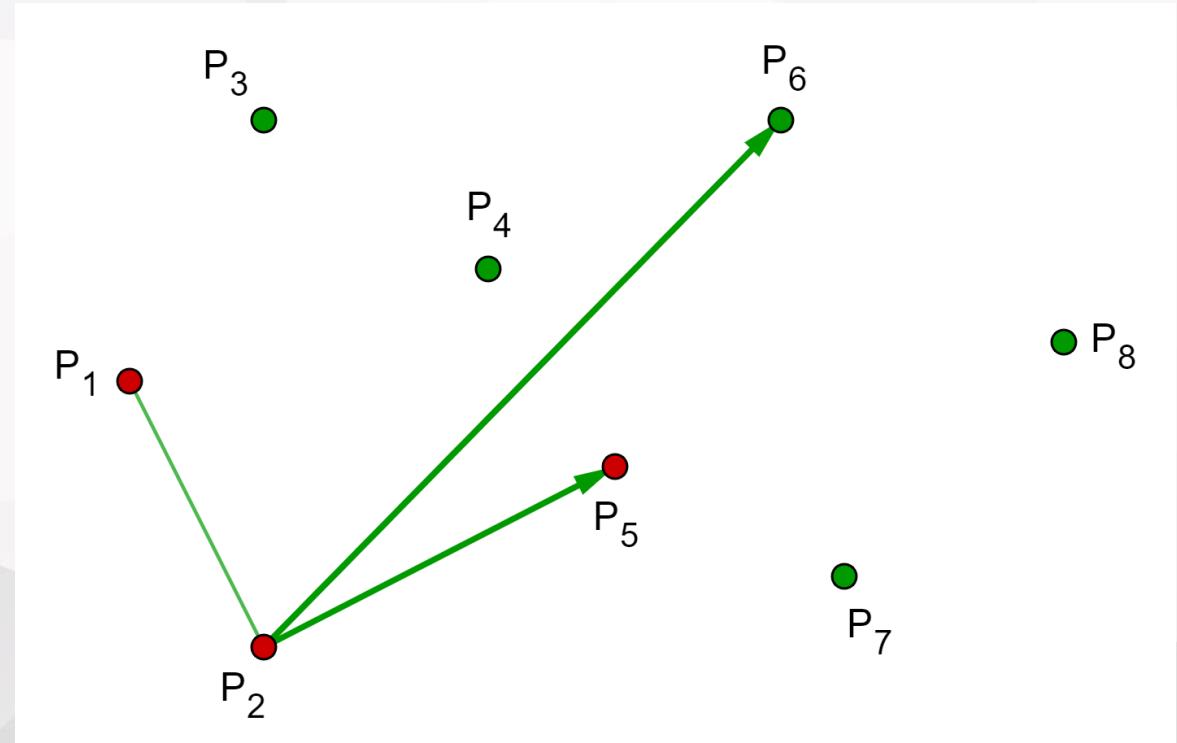
# 找出下側凸包

- $\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_5} > 0$ ，不會造成凹陷
- 將  $P_5$  加入凸包



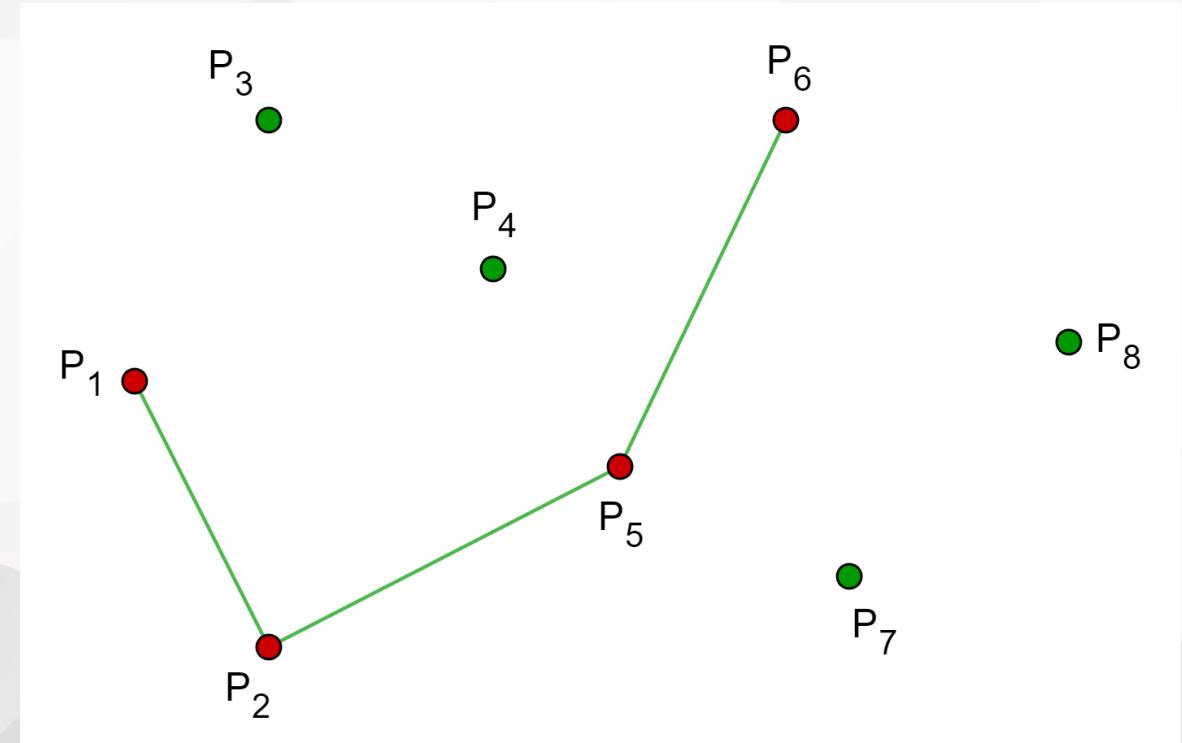
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



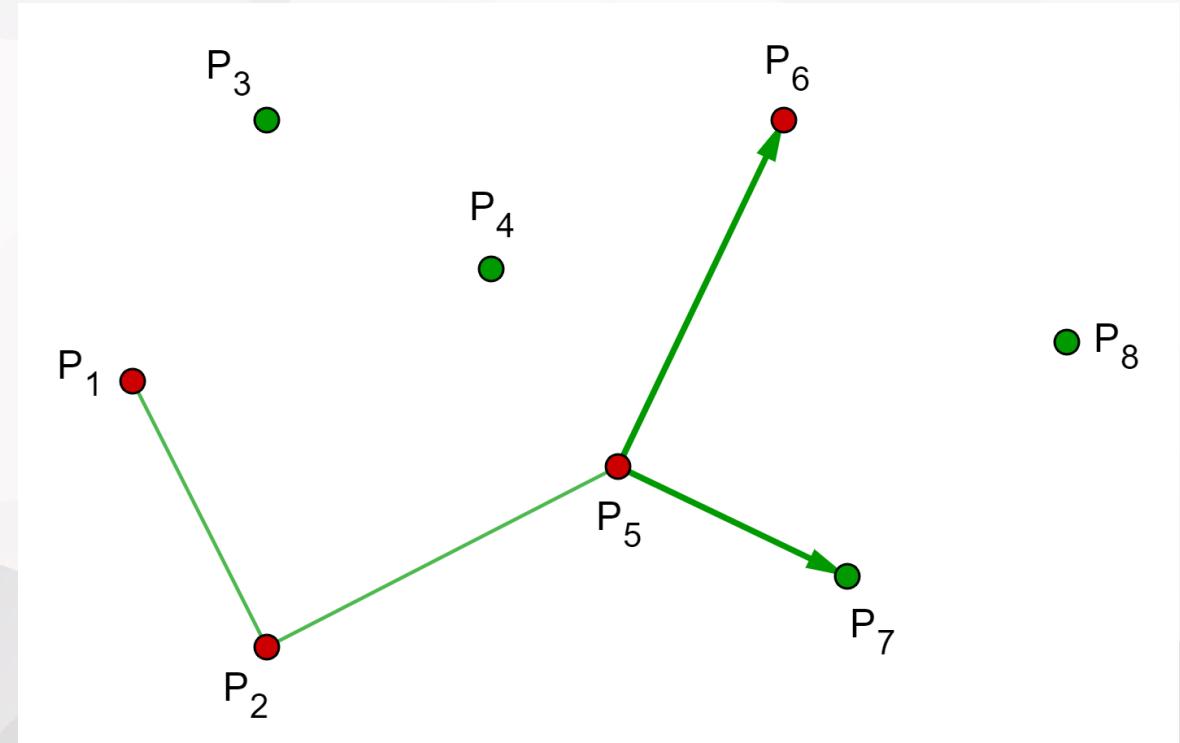
# 找出下側凸包

- $\overrightarrow{P_2P_5} \times \overrightarrow{P_2P_6} > 0$ ，不會造成凹陷
- 將  $P_6$  加入凸包



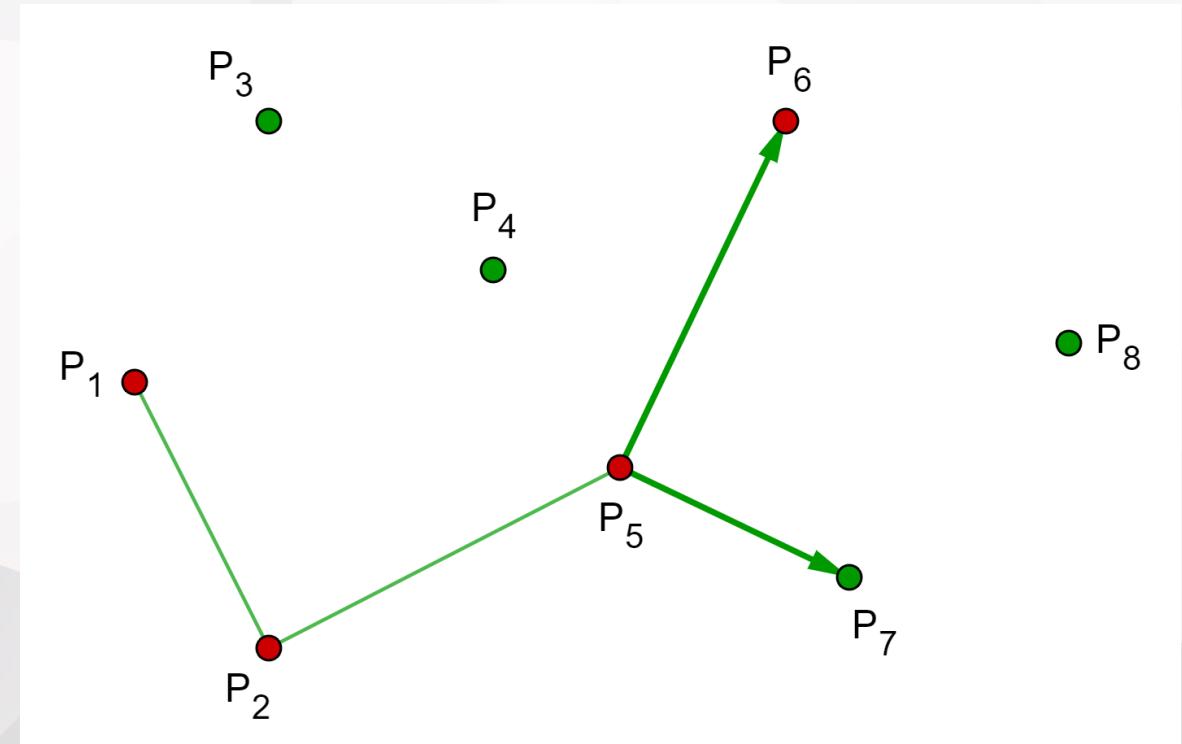
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



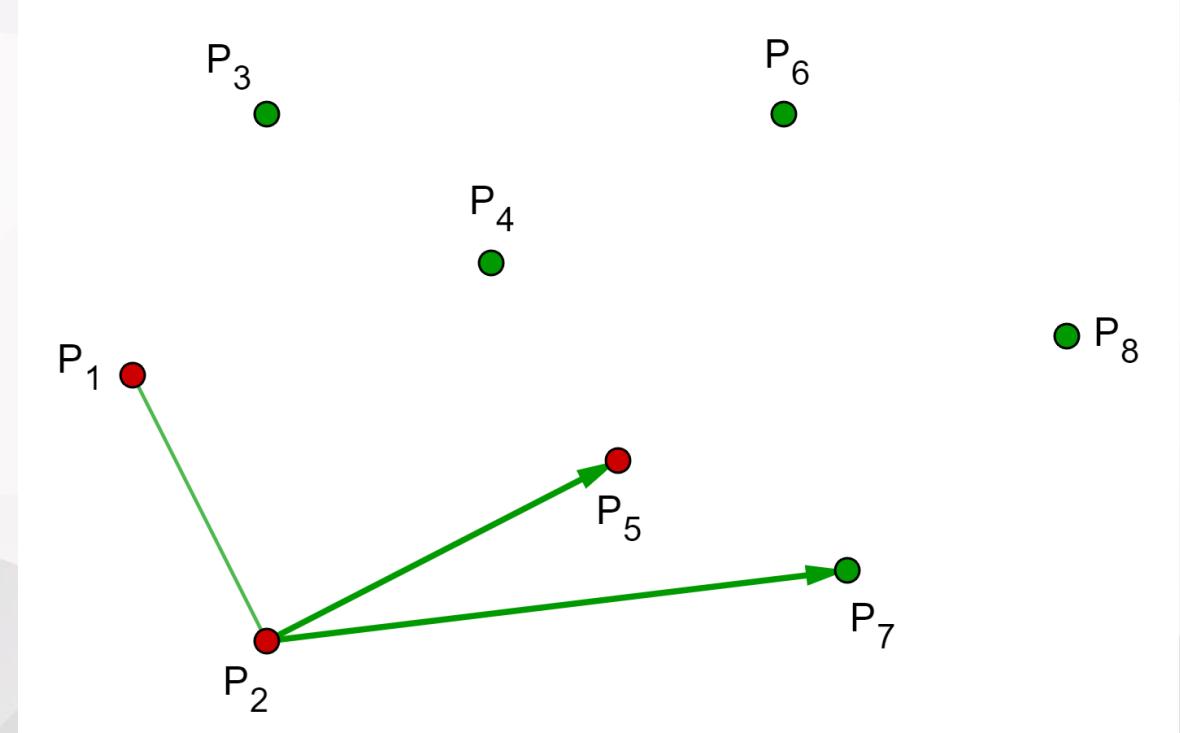
# 找出下側凸包

- $\overrightarrow{P_5P_6} \times \overrightarrow{P_5P_7} \leq 0$ ，會造成凹陷
- 將  $P_6$  移出凸包



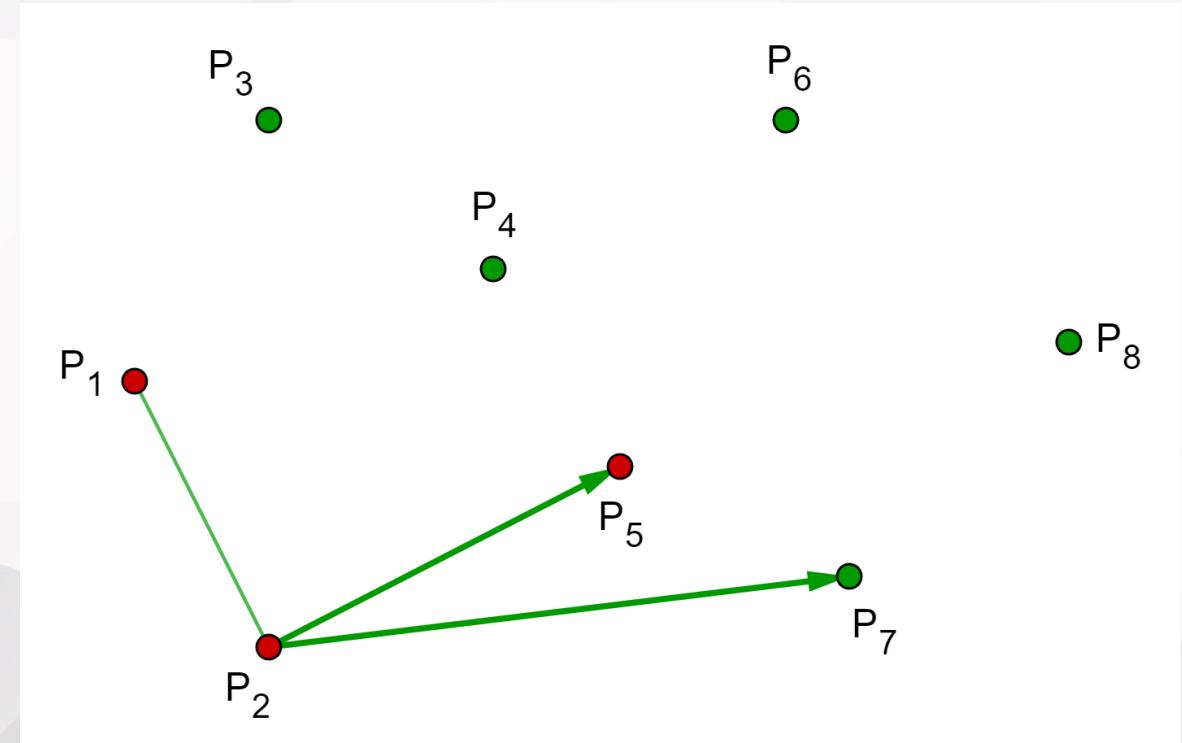
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



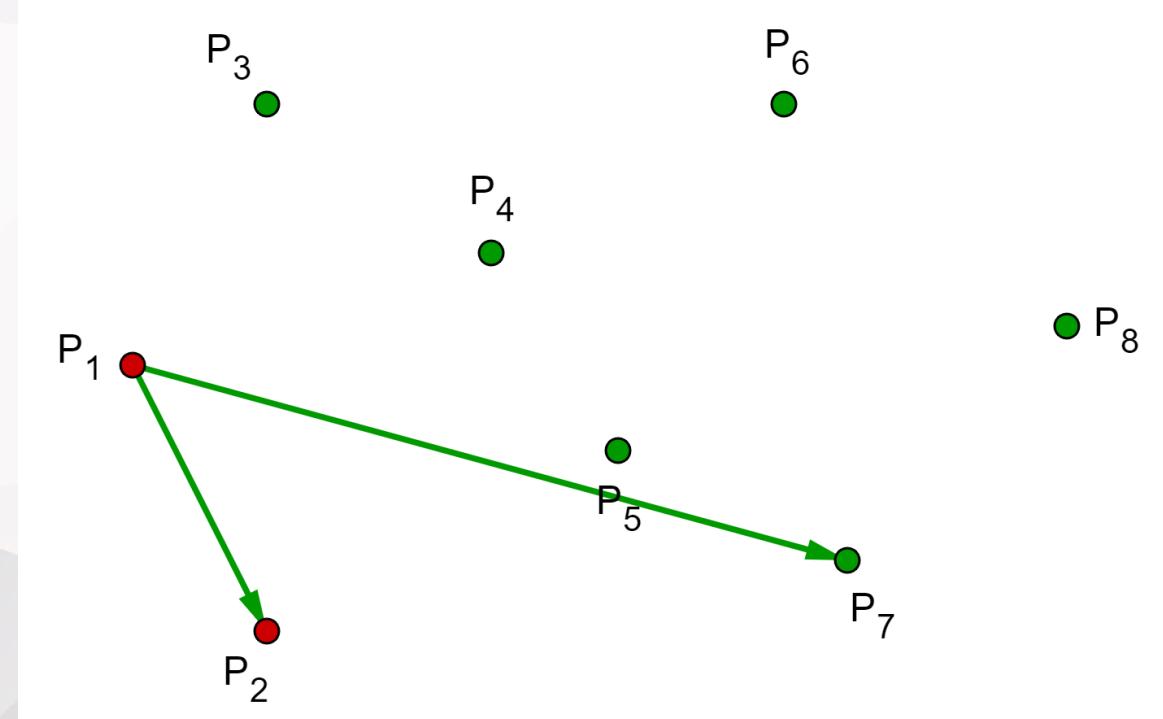
# 找出下側凸包

- $\overrightarrow{P_2P_5} \times \overrightarrow{P_2P_7} \leq 0$ ，會造成凹陷
- 將  $P_5$  移出凸包



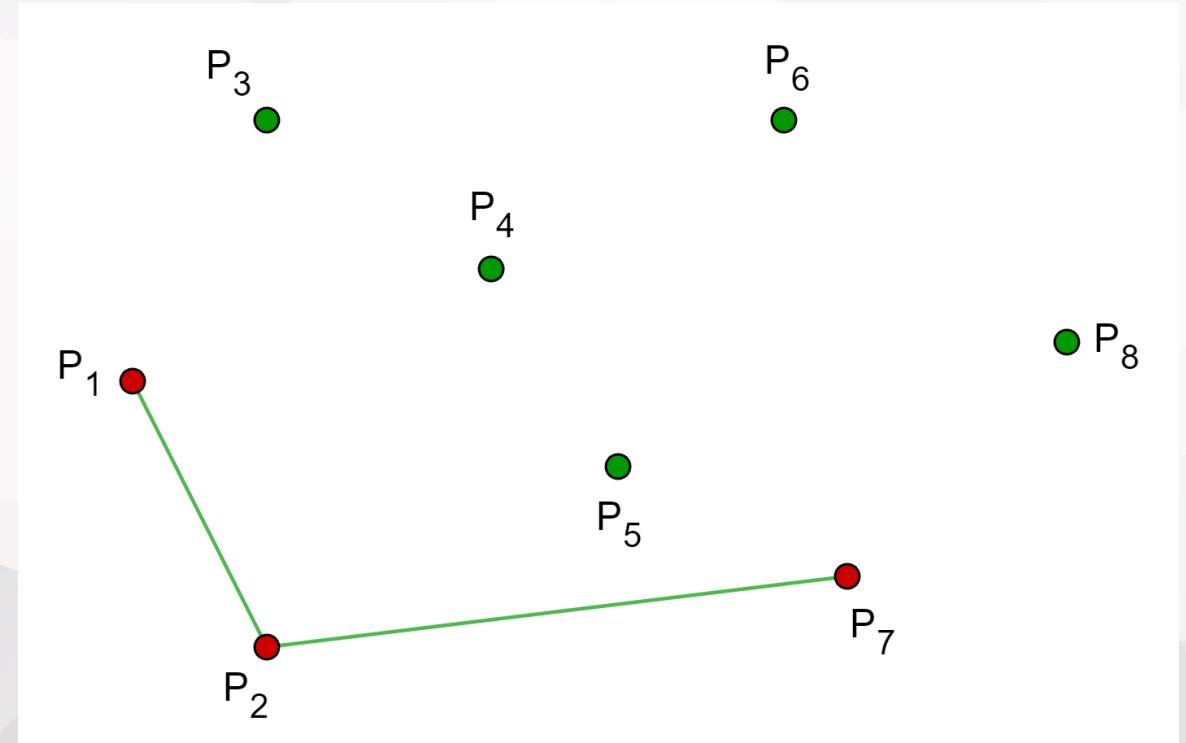
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



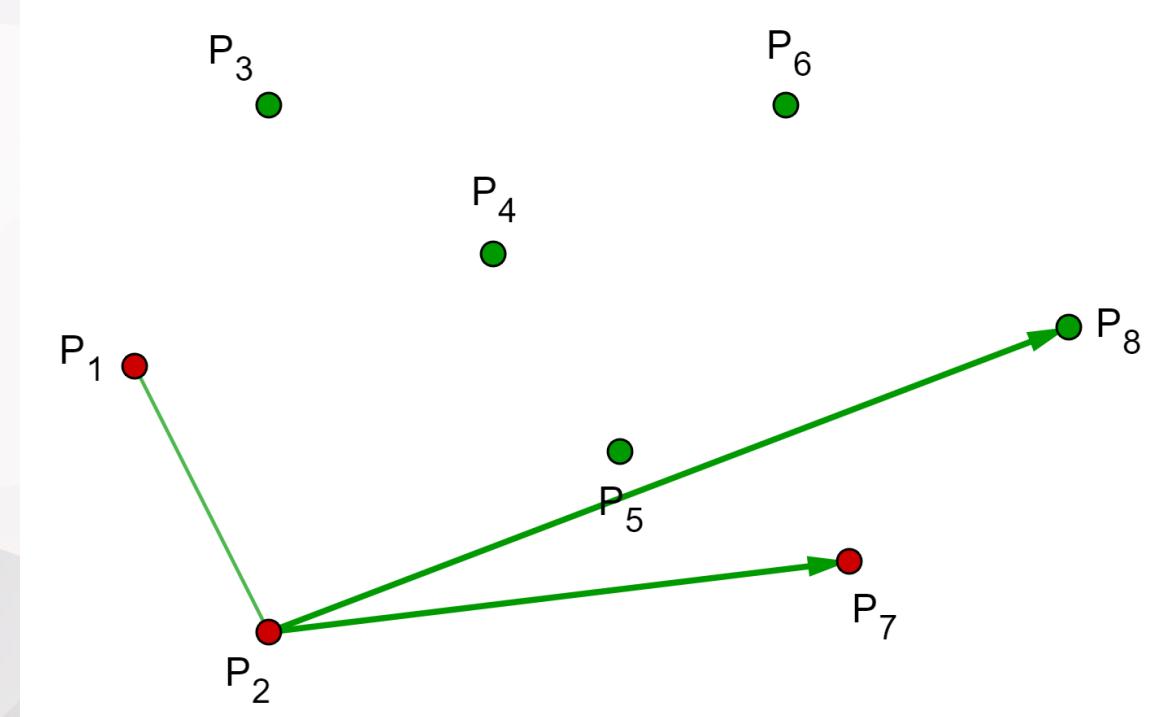
# 找出下側凸包

- $\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_7} > 0$ ，不會造成凹陷
- 將  $P_7$  加入凸包



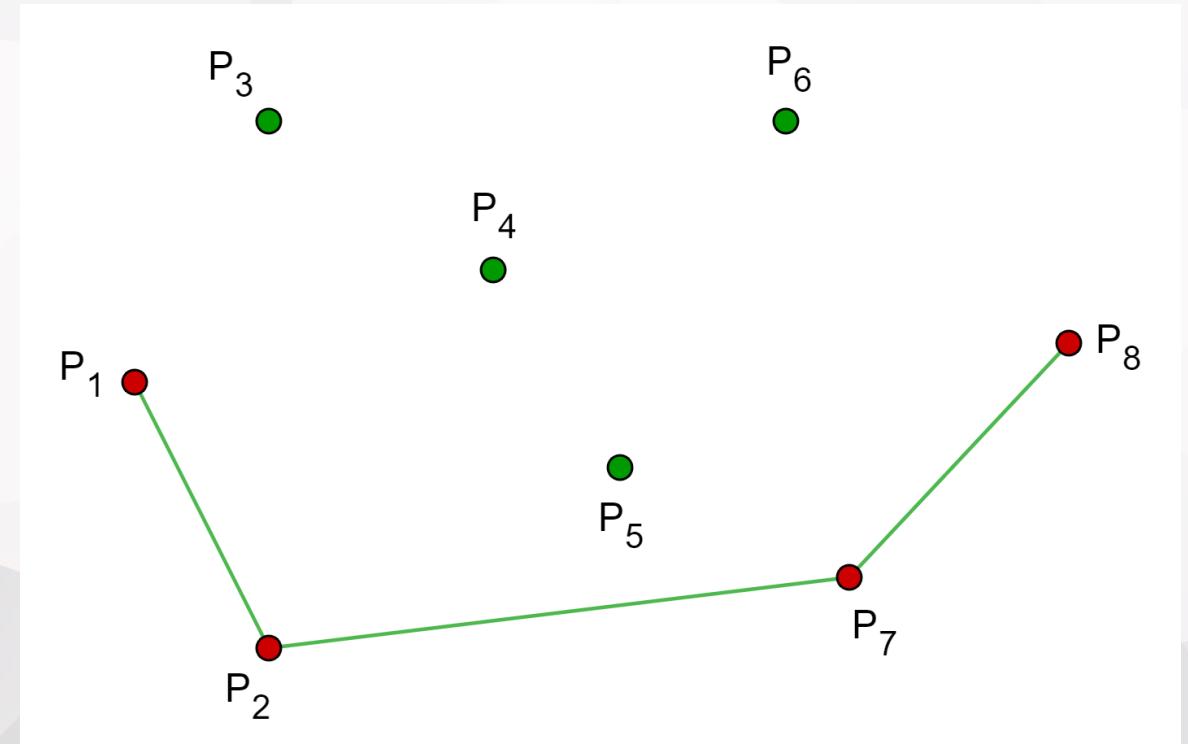
# 找出下側凸包

- 因為大於等於兩個點了
- 加入下個點之前要判斷是否會造成凹陷



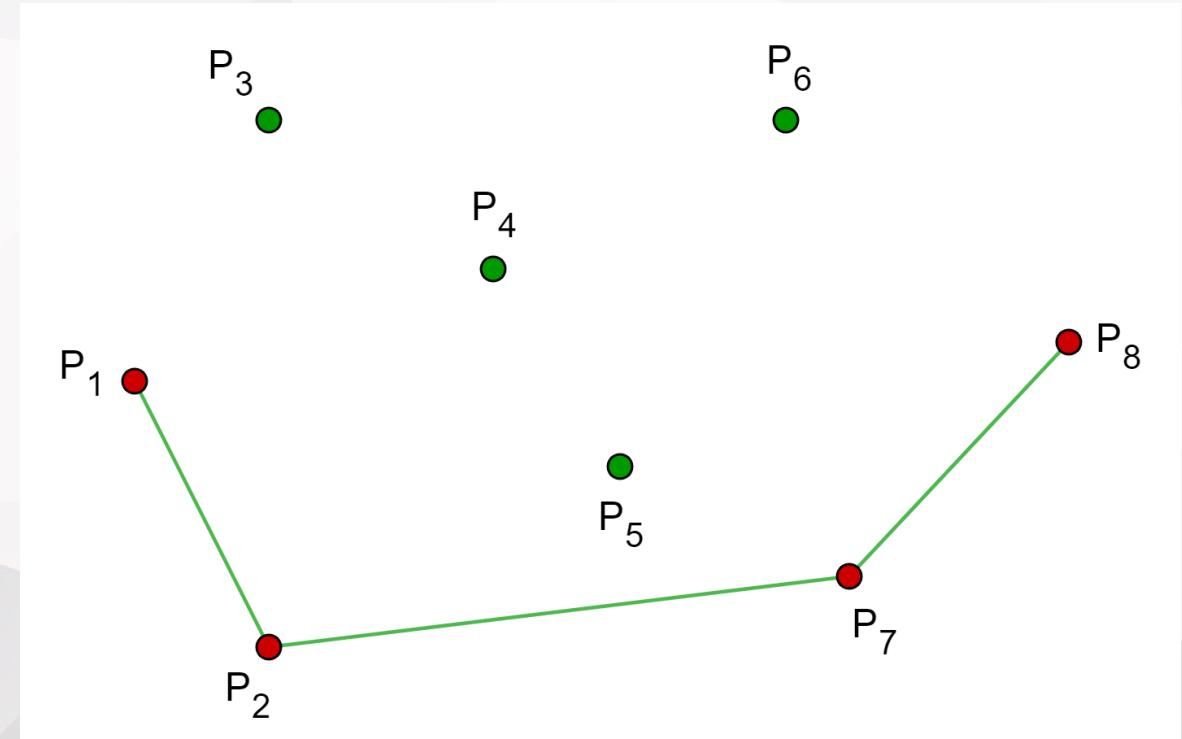
# 找出下側凸包

- $\overrightarrow{P_2P_7} \times \overrightarrow{P_2P_8} > 0$ ，不會造成凹陷
- 將  $P_8$  加入凸包



# 找出下側凸包

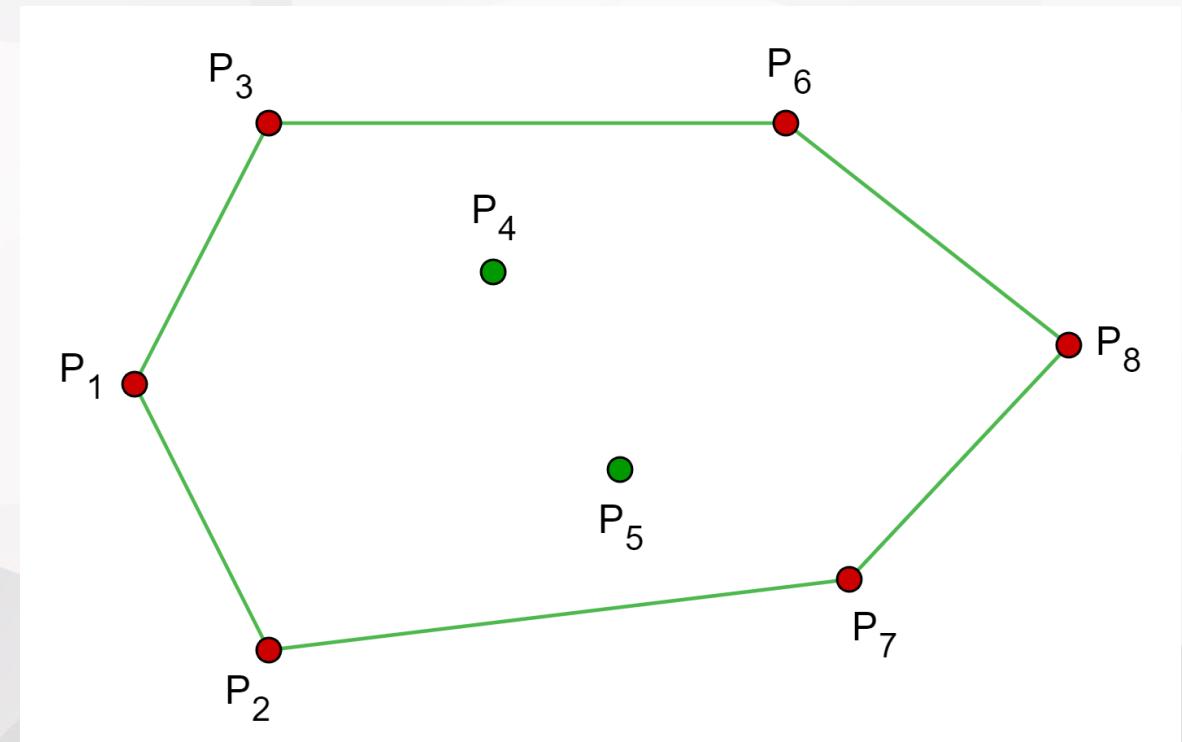
- 下側凸包完成
- 上側凸包唯一不同的地方只是從後面掃到前面，其他步驟均與找出下側凸包相同



# 找出凸包

---

- 要特別注意  $P_1$  會被算到兩次
- 如果點的數量大於 1 時要在最後把凸包點集中最後一個點去掉



# 找出凸包

---

- 程式碼

```
vector<Point> convexHull(vector<Point> v){  
    int sz = 0;  
    vector<Point> ans;  
    sort(v.begin(), v.end(), cmp);  
    for(size_t i = 0; i < v.size(); i++){  
        while(sz >= 2 && (ans[sz - 1] - ans[sz - 2]).cross(v[i] - ans[sz - 2]) <= 0)  
            ans.pop_back(), sz--;  
        ans.push_back(v[i]), sz++;  
    }  
    for(int i = int(v.size()) - 2, t = sz + 1; i >= 0; i--){  
        while(sz >= t && (ans[sz - 1] - ans[sz - 2]).cross(v[i] - ans[sz - 2]) <= 0)  
            ans.pop_back(), sz--;  
        ans.push_back(v[i]), sz++;  
    }  
    if(v.size() > 1)  
        ans.pop_back();  
    return ans;  
}
```

# 旋轉卡尺

---

# 旋轉卡尺

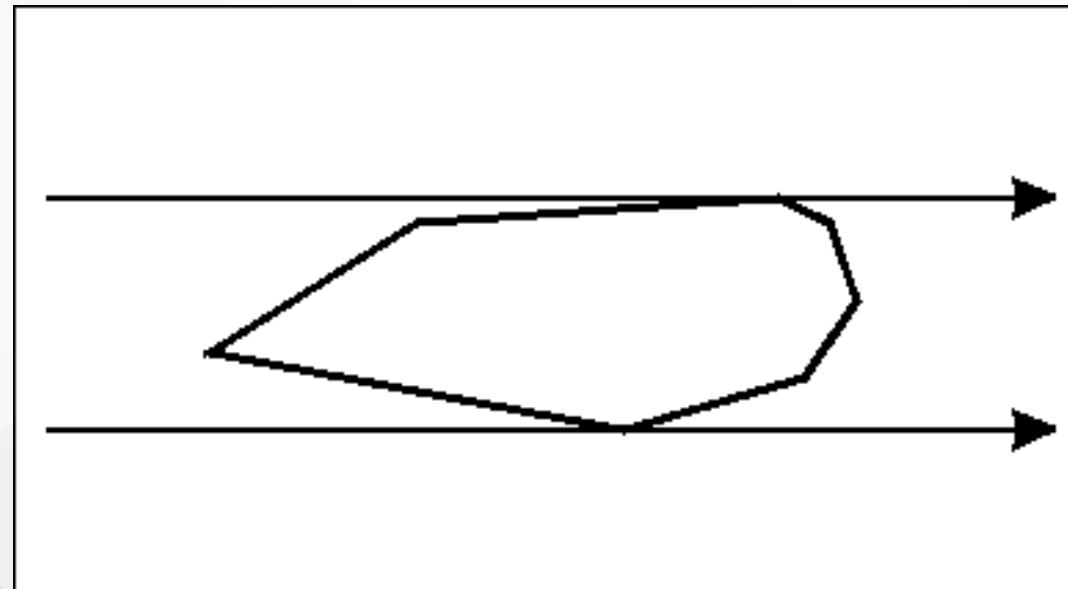
---

- 雖然可以在  $O(w)$  的時間內找到最遠點對，但是如果座標範圍到  $10^9$  的話，這樣的時間複雜度顯然會 TLE
- 旋轉卡尺

# 旋轉卡尺

---

- 用兩條平行線夾住凸包旋轉
- 剛好被卡住的兩個點就是對踵點



From Internet

# 如何旋轉？

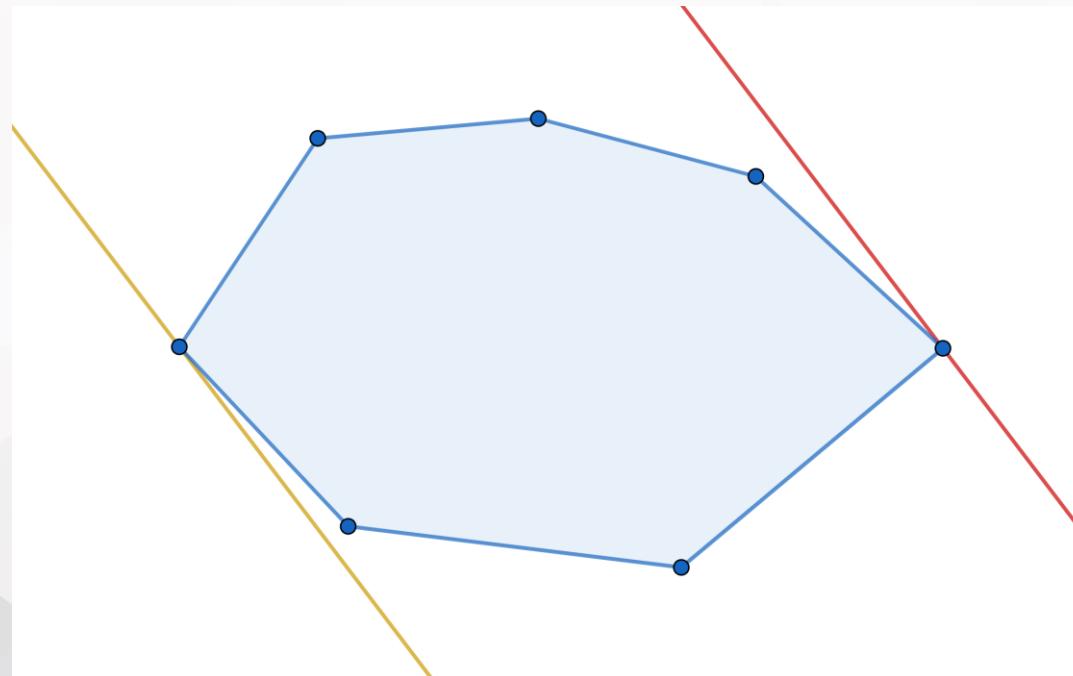
---

- 在程式實作上要做到旋轉有點困難
- 有一個比較方便的做法
  - 讓一條直線卡在某個點上
  - 對踵點則是另一條平行直線夾住的點
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點；距離最遠的即為最遠點對

# 如何旋轉？

---

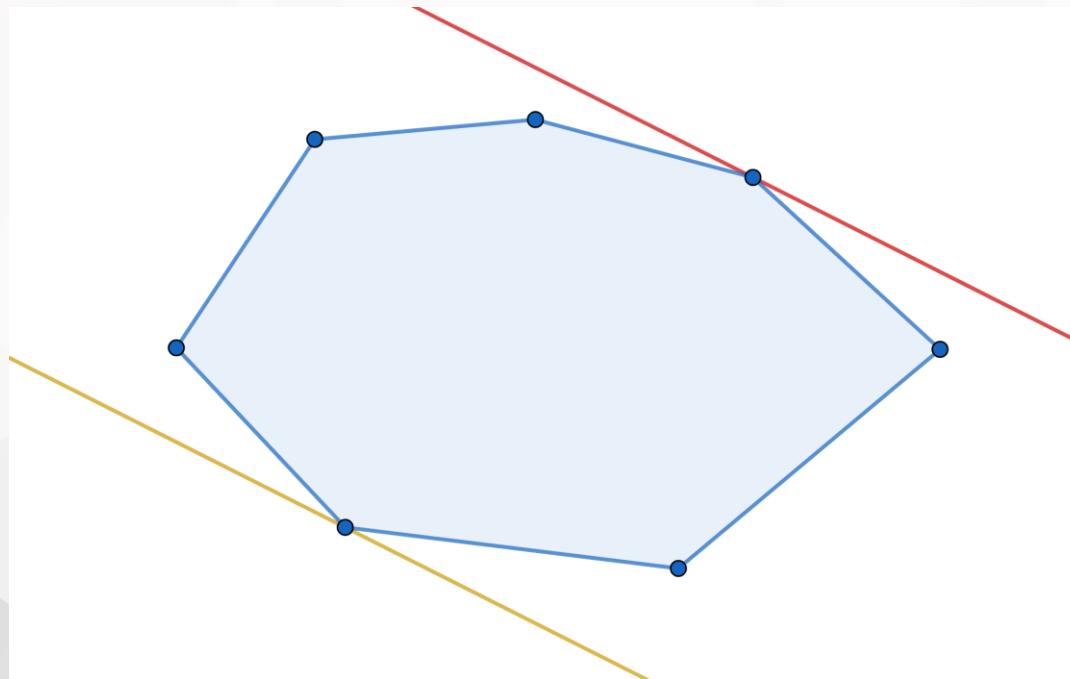
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

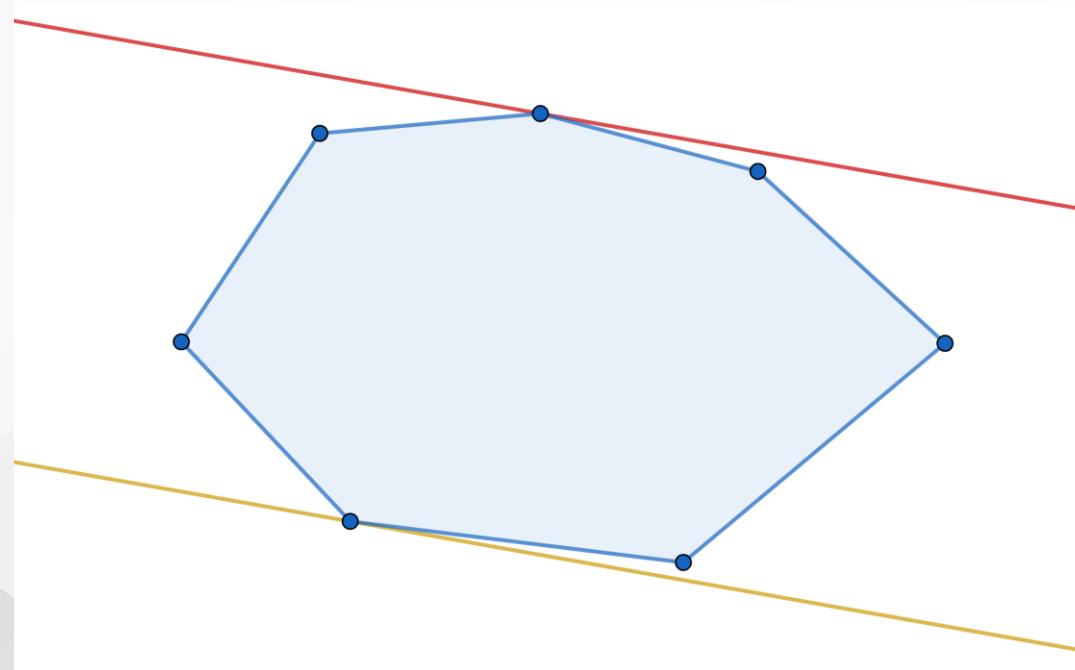
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

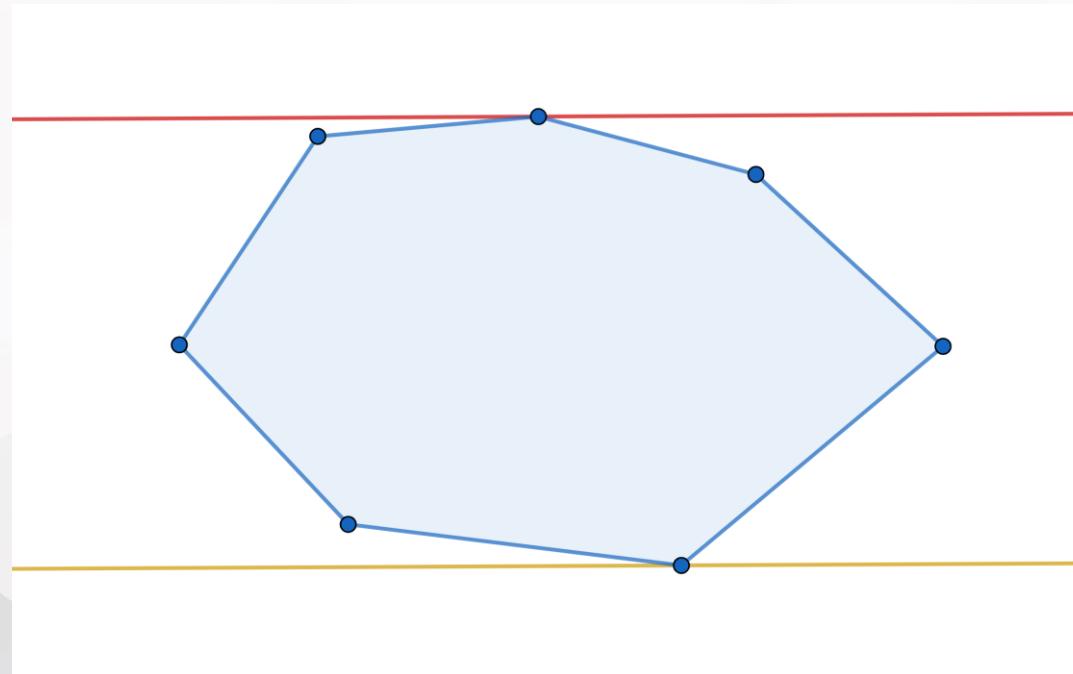
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

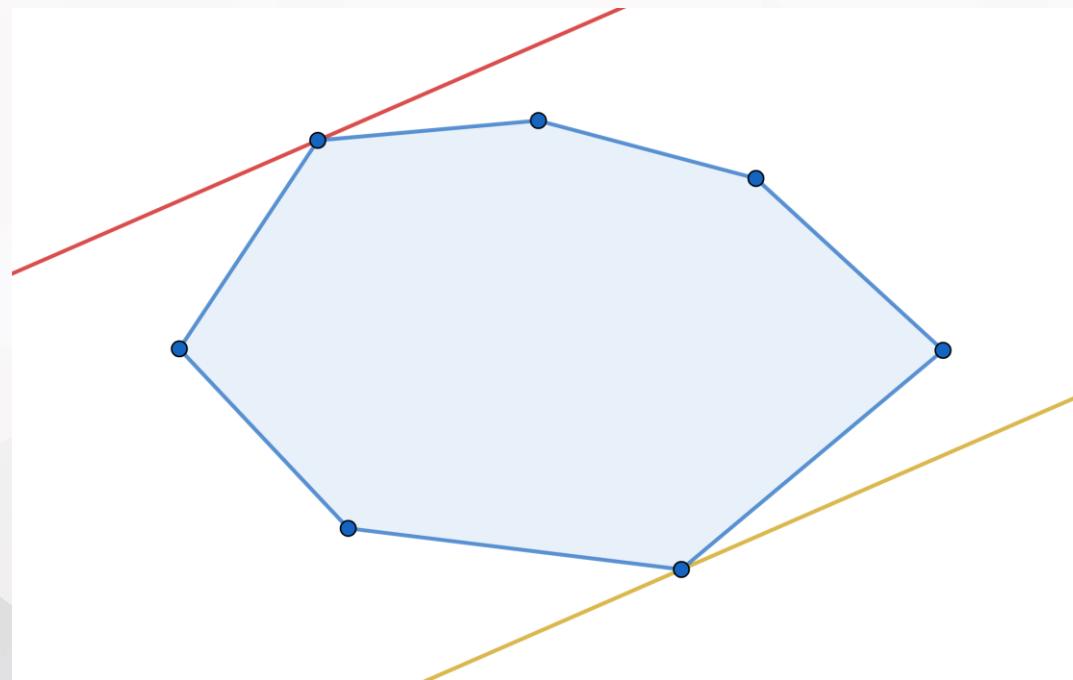
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

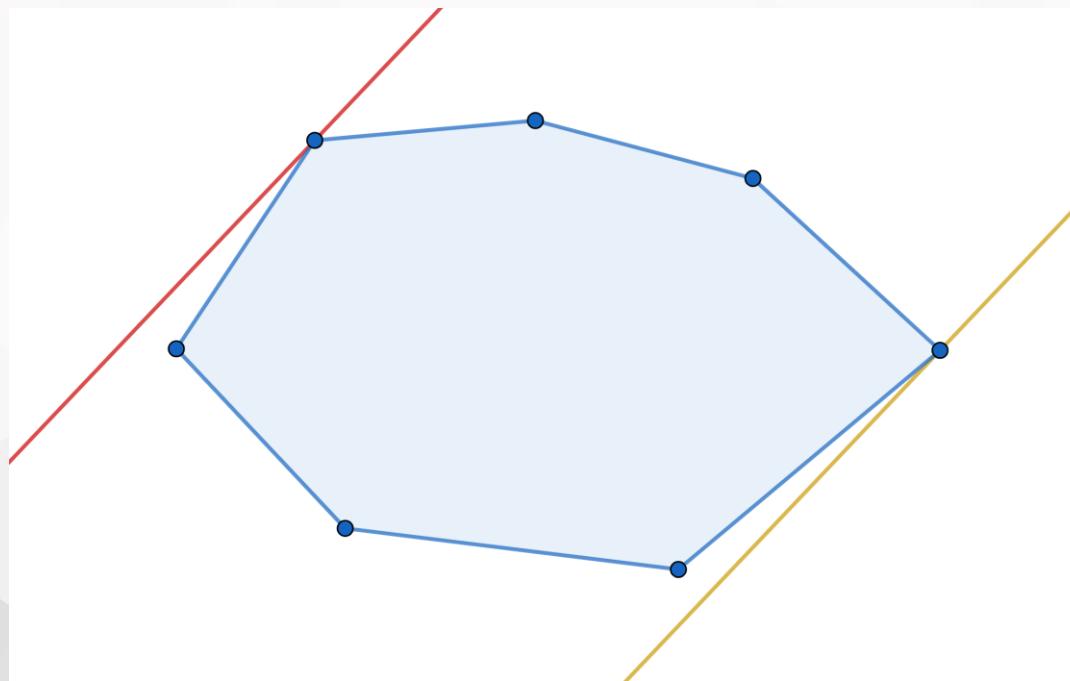
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

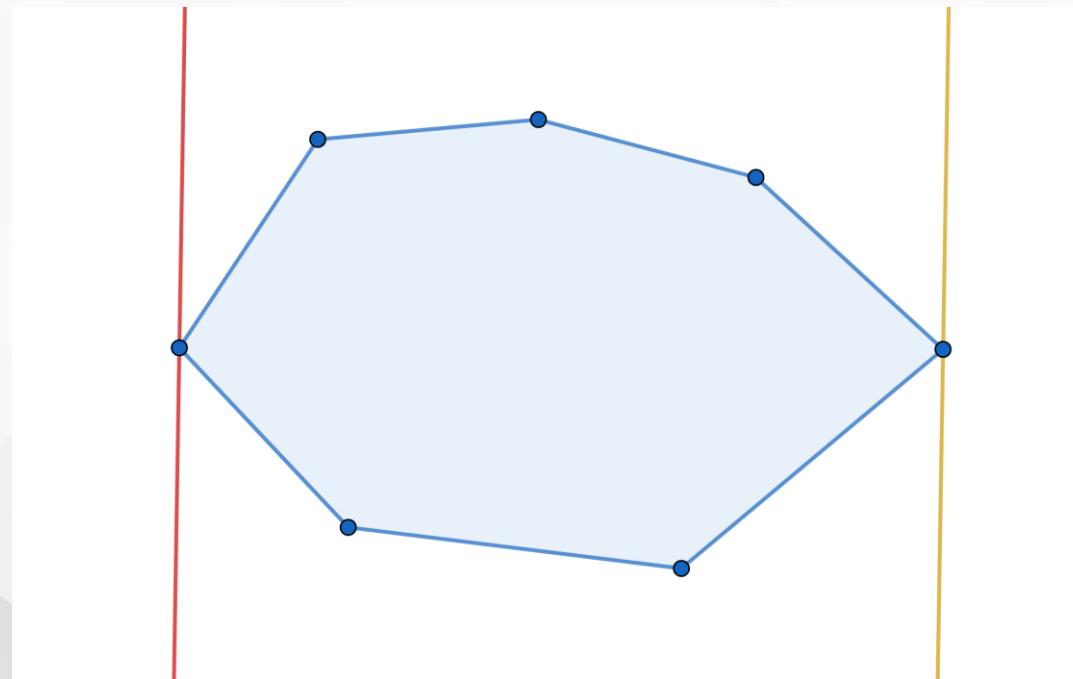
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

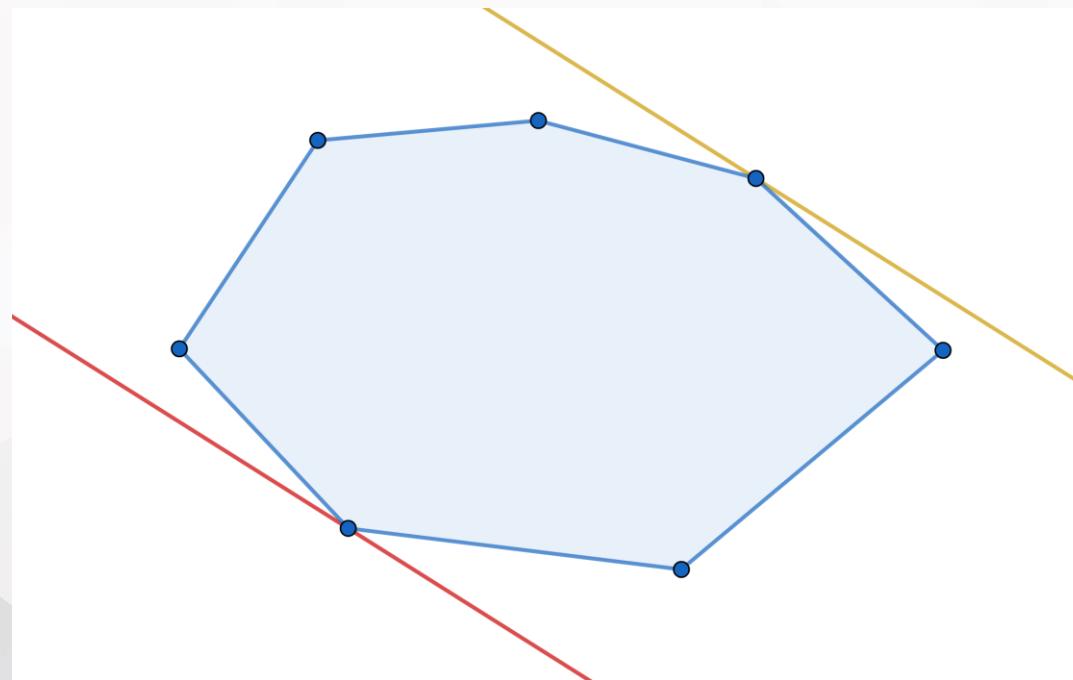
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

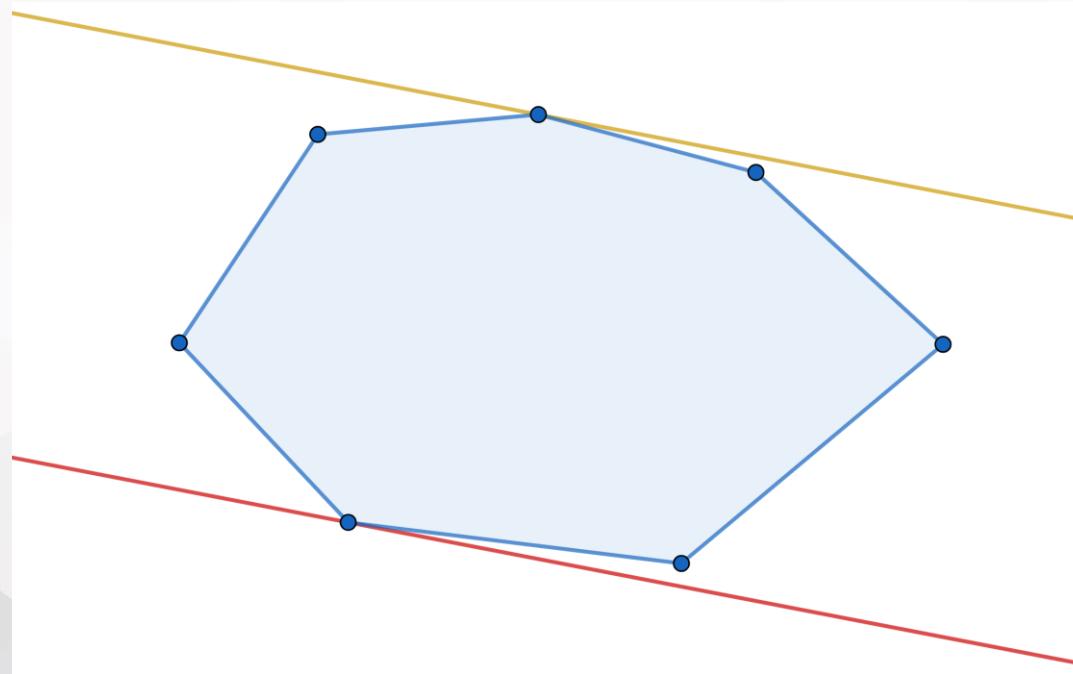
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

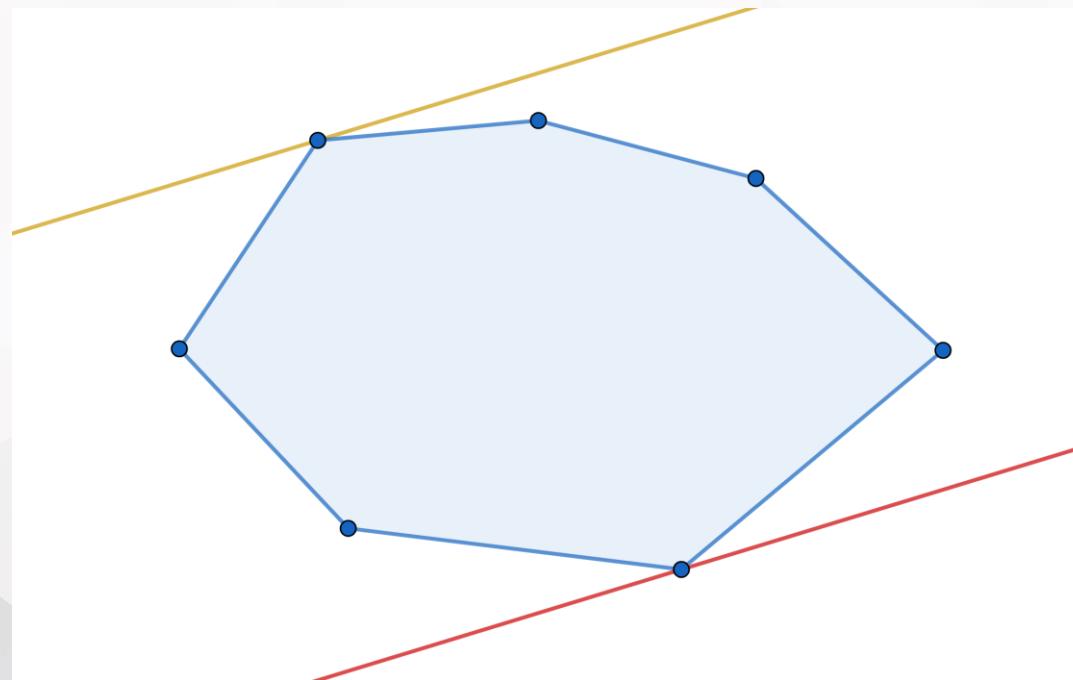
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

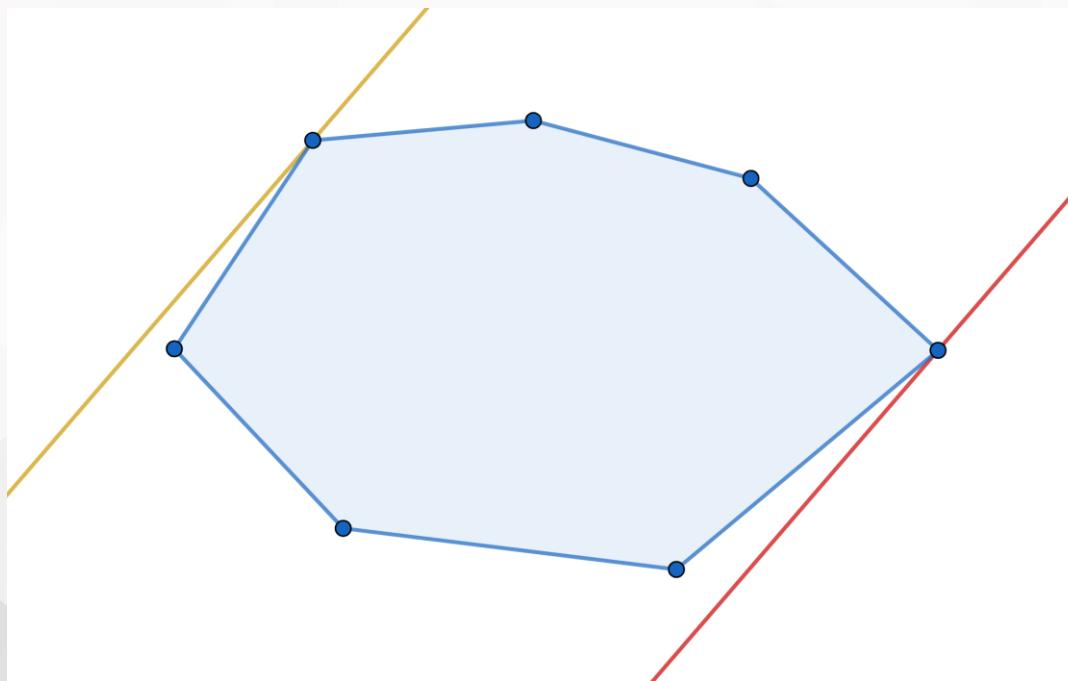
- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 如何旋轉？

---

- 依照逆時針順序枚舉所有的點，對於每個點找到最遠的對踵點，距離最遠的即為最遠點對



# 旋轉卡尺

---

- 枚舉的過程花費  $O(N)$  的時間找到最遠的點
  - 總共枚舉  $O(N)$  個點
  - 總複雜度  $O(N^2)$
- 那不如直接枚舉所有點對

# 旋轉卡尺

---

- 枚舉點的順序是逆時針旋轉，因此最遠點對也是逆時針旋轉
- 只要從上一個點的最遠點對逆時針旋轉即可找到現在這個點的最遠點

# 旋轉卡尺

---

- 逆時針枚舉所有點
- 每次枚舉從上一個最遠點開始，向逆時針方向找最遠點
- 計算距離，找出最大的都是答案

# 旋轉卡尺

---

- 程式碼
  - abs2 函數計算兩點距離之平方
  - 最後回傳長度前再開根號

```
double rotatingCaliper(const vector<Point> &v){  
    double ans = 0;  
    for(int i = 0, j = 0; i < v.size(); i++){  
        while(abs2(v[i], v[j]) <= abs2(v[i], v[(j + 1) % v.size()]))  
            j = (j + 1) % v.size();  
        ans = max(ans, abs2(v[i], v[j]));  
    }  
    return sqrt(ans);  
}
```

# Questions?

# Multiplication algorithm

快 還要更快

# Multiplication algorithm

---

- Gauss's complex multiplication algorithm
- Karatsuba algorithm

# 複數乘法

---

對於  $a + bi, c + di$

# 複數乘法

---

對於  $a + bi, c + di$

相乘得  $(ac - bd) + (bc + ad)i$

# 複數乘法

---

對於  $a + bi, c + di$

相乘得  $(ac - bd) + (bc + ad)i$

一般需要計算  $ac, bd, bc, ad$  共四次乘法  
才能計算出兩數相乘

# 複數乘法

---

$$(ac - bd) + (bc + ad)i$$

對於  $(ac - bd) = ac + bc - bc - bd$

# 複數乘法

---

$$(ac - bd) + (bc + ad)i$$

對於  $(ac - bd) = \underline{ac + bc} - \underline{bc - bd}$

令

$$k_1 = ac + bc = c(a+b)$$

$$k_2 = bc + bd = b(c+d)$$

# 複數乘法

---

$$(ac - bd) + (bc + ad)i$$

對於  $(bc + ad) = ac + bc + ad - ac$

# 複數乘法

---

$$(ac - bd) + (bc + ad)i$$

對於  $(bc + ad) = \underline{ac} + \underline{bc} + \underline{ad} - \underline{ac}$

令

$$k_1 = ac + bc = c(a+b)$$

$$k_3 = ad - ac = a(d-c)$$

# 複數乘法

---

$$(ac - bd) + (bc + ad)i$$

$$= (k_1 - k_2) + (k_1 + k_3)i$$

$$k_1 = ac + bc = c(a+b)$$

$$k_2 = bc + bd = b(c+d)$$

$$k_3 = ad - ac = a(d-c)$$

# 複數乘法

---

$$(ac - bd) + (bc + ad)i$$

$$= (k_1 - k_2) + (k_1 + k_3)i$$

總共只需要三次乘法

似乎變快了一點



# Multiplication algorithm

---

- Gauss's complex multiplication algorithm
- Karatsuba algorithm

# Karatsuba algorithm

---

對於剛剛的複數乘法

似乎對於整數  $x, y$  相乘有些啟示

# Karatsuba algorithm

---

對於剛剛的複數乘法

似乎對於整數  $x, y$  相乘有些啟示

也就是令

$$x = am + b$$

$$y = cm + d$$



# Karatsuba algorithm

---

$$x = am + b$$

$$y = cm + d$$

$$xy = acm^2 + (ad + bc)m + bd$$

# Karatsuba algorithm

---

$$xy = acm^2 + \underline{(ad + bc)m} + bd$$

其中

$$\begin{aligned}(ad + bc) &= ad + ac + bc + bd - bd - ac \\ &= (a + b)(c + d) - bd - ac\end{aligned}$$

# Karatsuba algorithm

---

$$xy = acm^2 + \underline{(ad + bc)m} + bd$$

$$\begin{aligned}(ad + bc) &= ad + ac + bc + bd - bd - ac \\ &= (a + b)(c + d) - bd - ac\end{aligned}$$

$$z_1 = ac$$

$$z_2 = bd$$

$$z_3 = (a + b)(c + d)$$

# Karatsuba algorithm

---

$$xy = acm^2 + \underline{(ad + bc)m} + bd$$

也就是說

$$xy = z_1m^2 + (z_3 - z_1 - z_2)m + z_2$$

$$z_1 = ac$$

$$z_2 = bd$$

$$z_3 = (a + b)(c + d)$$

# Karatsuba algorithm

---

$$xy = acm^2 + \underline{(ad + bc)m} + bd$$

也就是說

$$xy = z_1m^2 + (z_3 - z_1 - z_2)m + z_2$$

$$z_1 = ac$$

$$z_2 = bd$$

$$z_3 = (a + b)(c + d)$$

# Karatsuba algorithm

---

$$xy = z_1m^2 + (z_3 - z_1 - z_2)m + z_2$$

假設數字長度為  $n$

總共只需要三次乘法

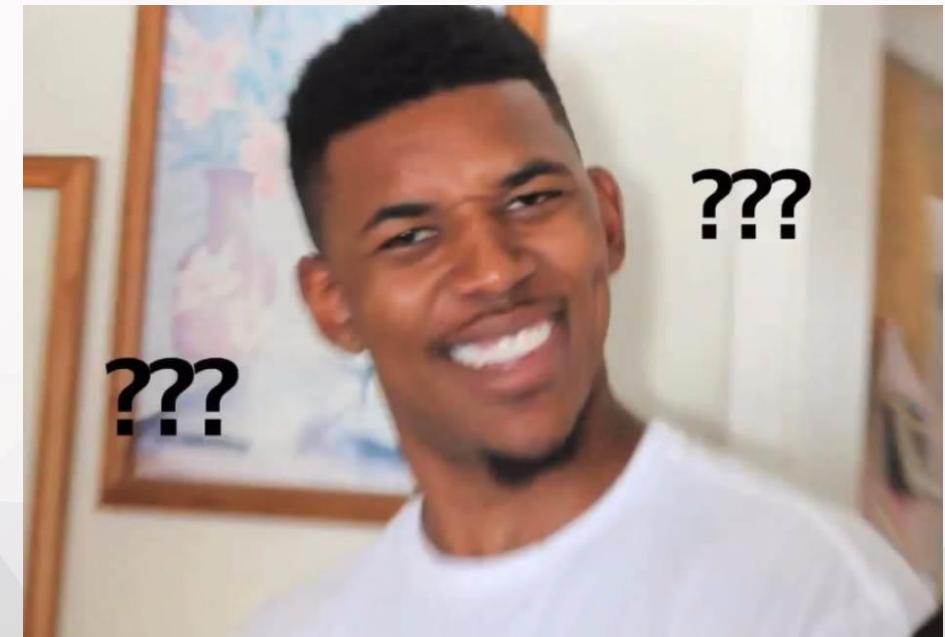
相較於直式乘法複雜度  $O(n^2)$

這個算法有複雜度  $3 \cdot O(n^2)$

# Karatsuba algorithm

---

到底在幹三小?

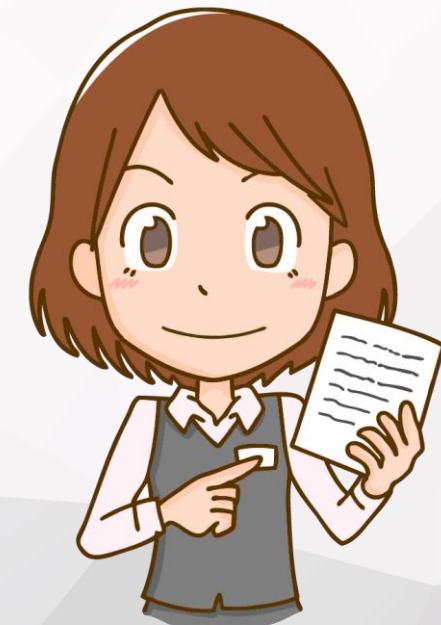


# 分治法

---

對於上述演算法，  
凡是遇到乘法運算，都使用同樣的演算法

感恩分治 讚嘆分治



# 分治法

---

對於上述演算法，  
凡是遇到乘法運算，都使用同樣的演算法

並且對於數字的分割，總是分成均等的**兩半**

例如 6789 分成 67 和 89

$$6789 = 67 \cdot 100 + 89$$

# 分治法

---

```
int k(int x, int y) {
    if(x < 10 || y < 10) return x*y;

    int len = min(log10(x), log10(y));
    int m = pow(10, len/2 + 1);

    auto [a, b] = div(x, m); // since c++17
    auto [c, d] = div(y, m);

    int z1 = k(a, c), z2 = k(b, d), z3 = k(a+b, c+d);
    return z1*m*m + (z3-z1-z2)*m + z2;
}
```

# 分治法

---

時間成本  $T(n)$

$$T(n) = 3 \cdot T(n/2) + c_n$$

$$T(1) = 1 + c_1$$

複雜度為  $O(3^{\lg n}) = O(n^{\lg 3})$



# Floating-Point Precision

---

浮點數誤差以及競程處理技巧

# 形成原因：IEEE 754 的浮點數的儲存

---

- 用 0 跟 1 表示浮點數
- 表示方式： $1.1101010_{(2)} \times 2^4_{(10)}$
- 優點：在精度內可以表達好，通常精度夠用
- 缺點：超出精度就無法表示
- 直接 WA

# 舉例

---

- $0.69_{(\text{double})} \times 10 \neq 6.9_{(\text{double})}$
- 結果可能是 6.89999.....
- 在 `print` 的時候不容易出錯，但在比較大小或判斷相等時常會出問題

# 解決方法

---

- 假設今天有個閾值是  $6.9$ ， $6.899999$  這樣的表示會有問題
- 解決方案一：四捨五入
- 解決方案二：自己用 `int` 做運算
- 解決方案三：乘上  $1.000\dots001$ (數量根據精度調整) ->  
**較推薦**

# 解決方法(比較大小)

---

```
if (0.69 * 10 * 1.000...1 >= 6.9) {  
    do something  
}
```

這裡比較  $0.69 \times 10$  是否大於等於 6.9  
所以直接將左邊乘大一些，給他一點誤差空間

# 解決方法(比較相等)

---

```
if (abs((0.69*10) - 6.9) <= 6.9*1e-14) {  
    do something  
}
```

這裡比較  $0.69*10$  是否等於  $6.9$

$1e-14$  是一個很小的數，需大於資料型態的精度  
代表  $0.69*10$  跟  $6.9$  的差值是否小於自身的某個比例

# AC Get

---