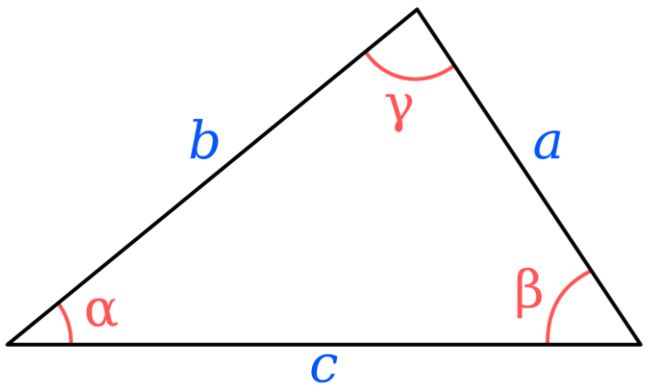
Computational Geometry

日月卦長

學校教的 – 解析幾何 (Analytic Geometry)

- 座標
- 方程式
- 距離、角度
- 旋轉

計算過程容易出現無理數



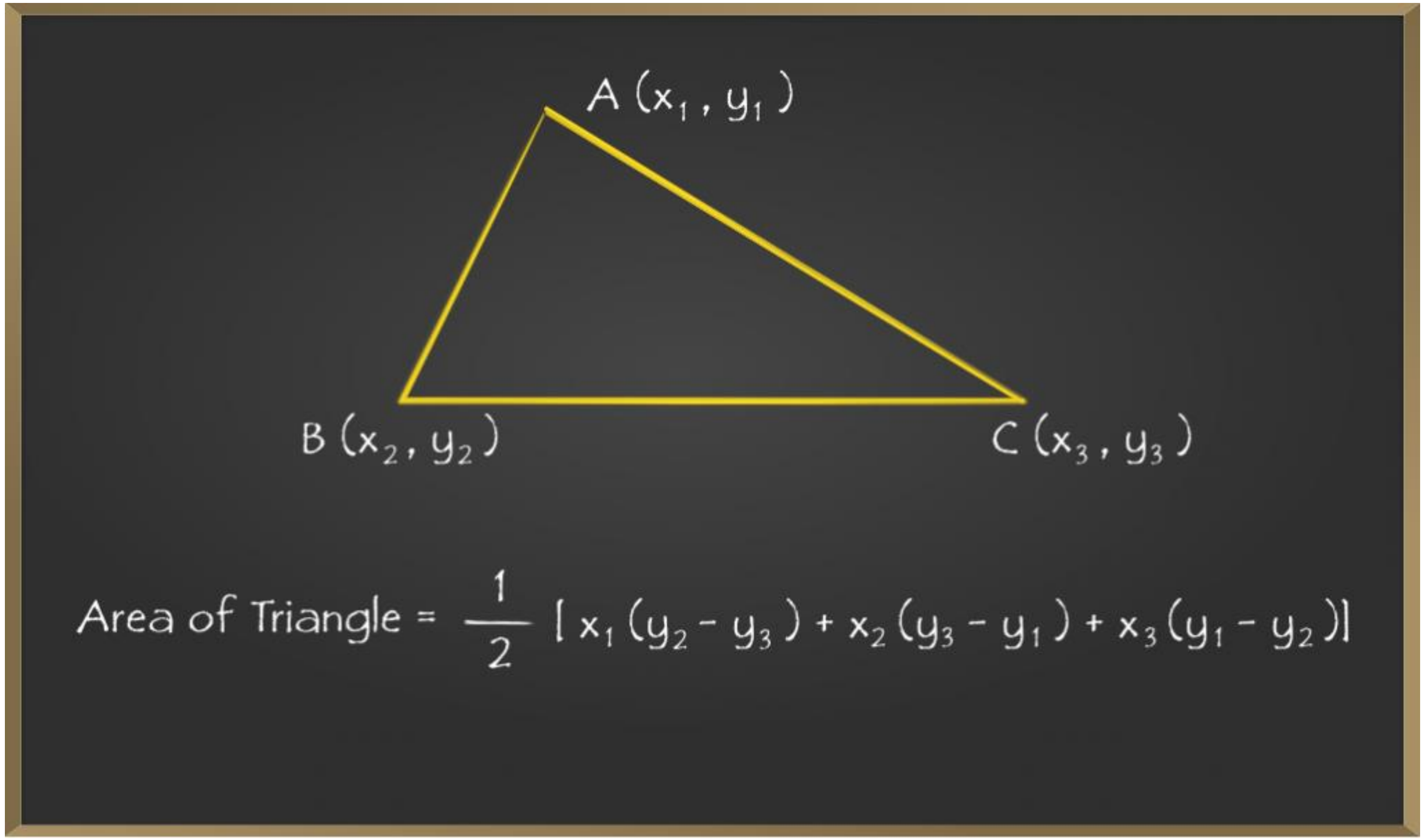
設 $s=\frac{a+b+c}{2}$

三角形面積 $A=\sqrt{s(s-a)(s-b)(s-c)}$

避免無用的無理數、除法

電腦算的 – 計算幾何 (Computational Geometry)

- 座標
- 向量
- 內積
- 外積



第一道題：計算線段相交

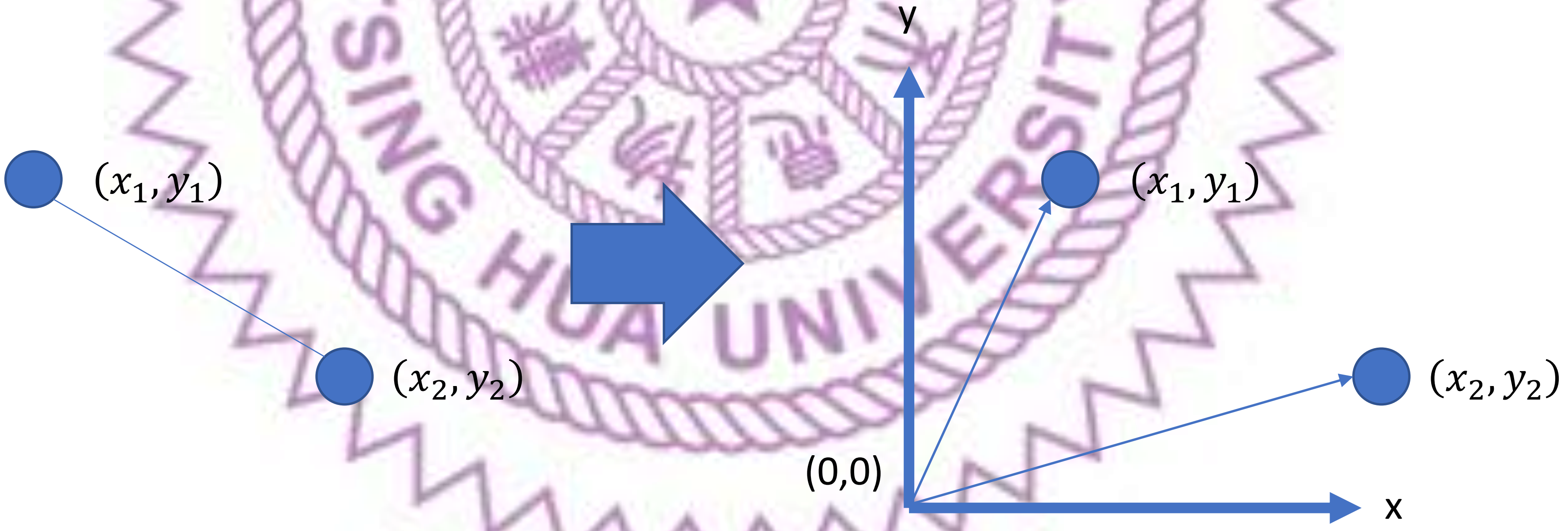
- 給你兩個線段，請判斷他們有沒有相交
  如果有相交的話找出他們的交點

高中學過的：$y = ax + b$

- 最大的問題就是垂直線無法表示
- 你說寫程式時垂直線可以特判掉？
  那這樣到處會有「特例」，最後程式會充滿這些垃圾然後爆炸！

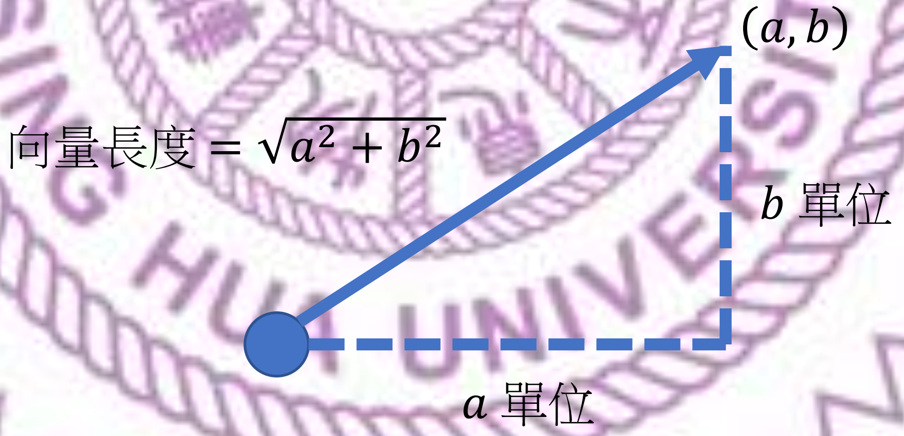
點與向量

- 兩個點就可以決定一條線段(或是直線)，稱為兩點式
- 但正確來說，這其實是兩個位置向量



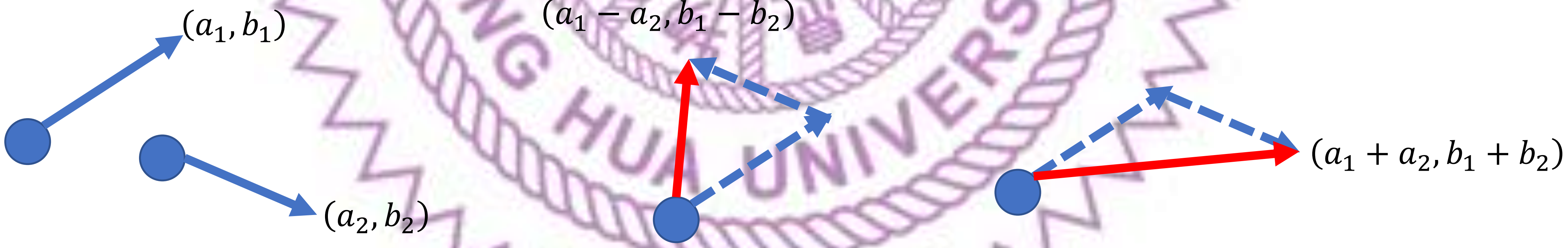
點與向量

- 向量可以表示方向和長度
- 通常我們會這樣紀錄一個平面向量：(a, b)



點與向量

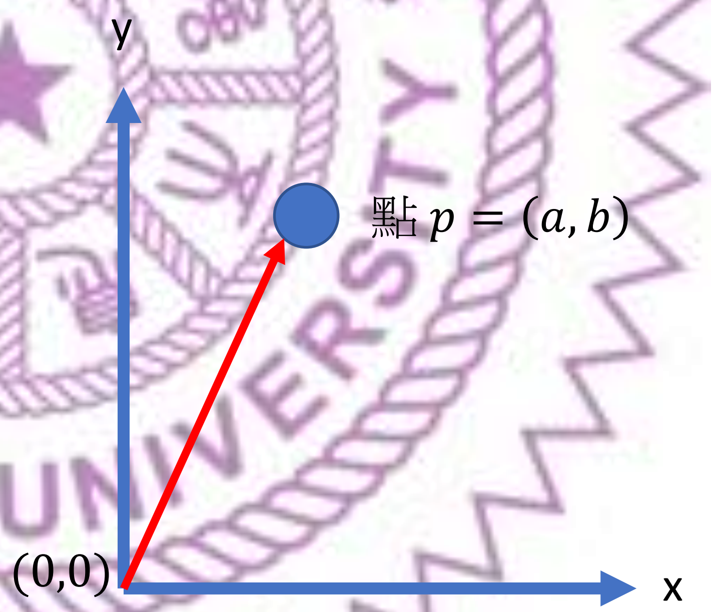
- 兩向量之間可以互相做加減
- $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$
- $(a_1, b_1) - (a_2, b_2) = (a_1 - a_2, b_1 - b_2)$



點與向量

- 平面上的一個點可以看成是由原點發射出去的一個向量

- 因此在計算幾何中
  點 (point) = 向量 (vector)！



點與向量

```cpp
struct PT {
  double x, y;
  PT(double x = 0, double y = 0) : x(x), y(y) {}
  PT operator+(const PT &b) const { // 相加
    return PT(x + b.x, y + b.y);
  }
  PT operator-(const PT &b) const { // 相減
    return PT(x - b.x, y - b.y);
  }
  PT operator*(double b) const { // 放大
    return PT(x * b, y * b);
  }
  PT operator/(double b) const { // 縮小
    return PT(x / b, y / b);
  }
  double dot(const PT &b) const { ... } // 內積 (等一下會教怎麼寫)
  double cross(const PT &b) const { ... } // 外積(叉積) (等一下會教怎麼寫)
};
```

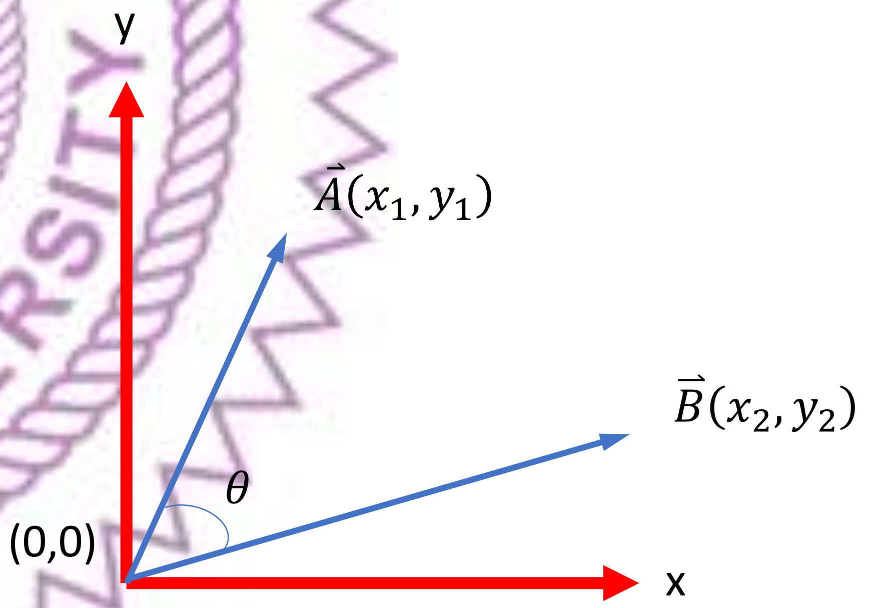
double、int、long long

- 寫題目時要根據題目要求適當的使用不同的資料型態
- 本單元的範例程式碼都會使用 double 作為範例
- **千萬不要用 float**

dot(內積)

- 定義：

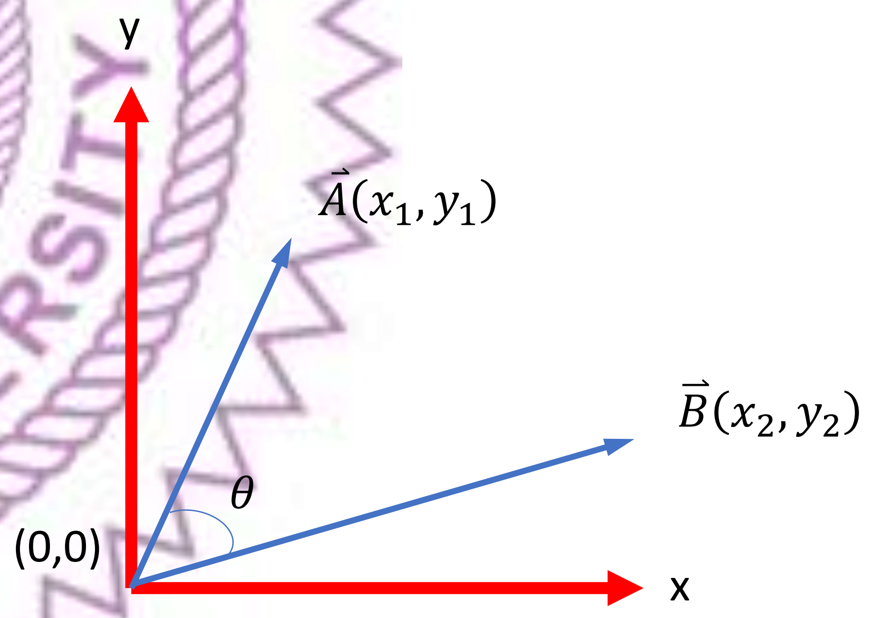
- 給兩個向量 $\vec{A} = (x_1, y_1), \vec{B} = (x_2, y_2)$
  - $\vec{A} \cdot \vec{B} = x_1 \times x_2 + y_1 \times y_2$
  - $\vec{A} \cdot \vec{B} = \|\vec{A}\| \times \|\vec{B}\| \times \cos\theta$



dot(內積)

- 定義：
  - $\vec{A} \cdot \vec{B} = x_1 \times x_2 + y_1 \times y_2 = \|\vec{A}\| \times \|\vec{B}\| \times \cos\theta$

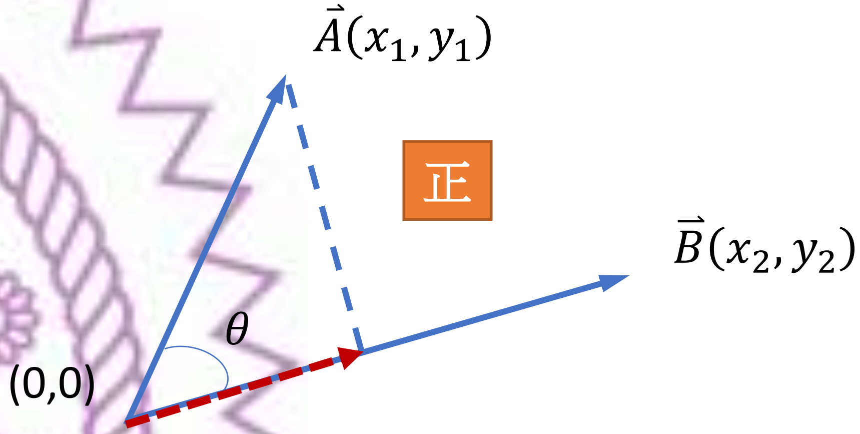
- 性質：
  - $\vec{A} \cdot \vec{B} = \vec{B} \cdot \vec{A}$
  - $\vec{A} \cdot (\vec{B} + \vec{C}) = \vec{A} \cdot \vec{B} + \vec{A} \cdot \vec{C}$

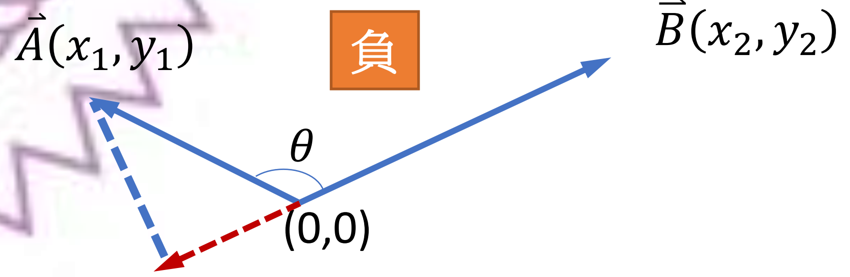


dot(內積)

- 定義：
  - $\vec{A} \cdot \vec{B} = x_1 \times x_2 + y_1 \times y_2 = \|\vec{A}\| \times \|\vec{B}\| \times \cos\theta$

- 幾何意義：
  - $\|\vec{A}\| \times \cos\theta$ 是向量 $\vec{A}$ 到 $\vec{B}$ 的投影長度
  - 可以理解為向量 $\vec{A}$ 在 $\vec{B}$ 的投影長度乘上向量 $\vec{B}$ 的長度
  - 當向量 $\vec{A}$ 在 $\vec{B}$ 的投影和向量 $\vec{B}$ 同向時會是正的
    反之會是負的
    如果 $\vec{A}, \vec{B}$ 夾角是 $90°$ 時會是 0



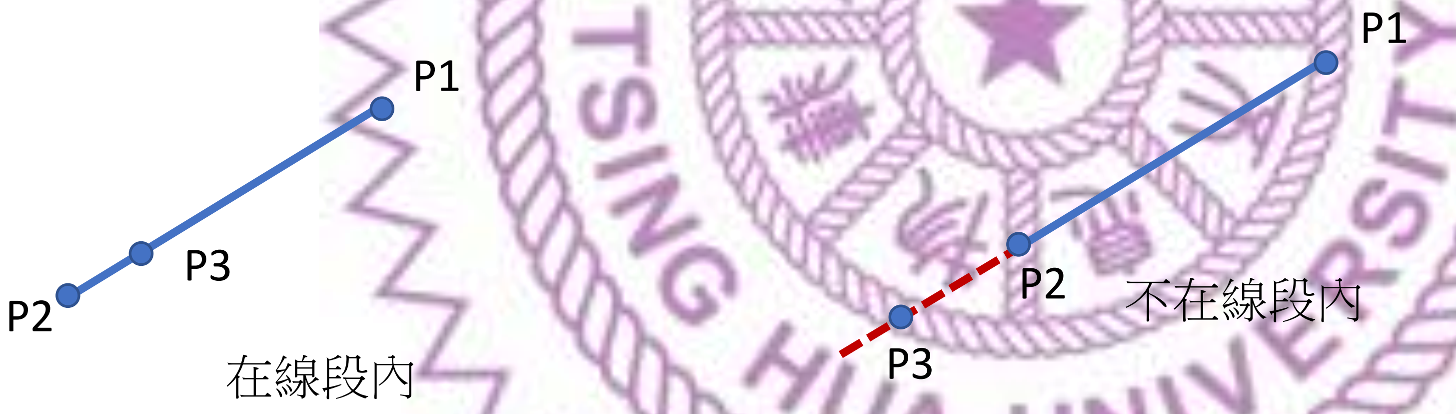


dot(內積) 程式碼

```
double dot(const PT &b) const { // 內積
  return x * b.x + y * b.y;
}
...
PT A(7, 1), B(2, 2);
cout << A.dot(B) << endl; // 輸出16
```

應用：判斷點是否在線段內

- 在一條直線上有三個點 $P1, P2, P3$，其中 $(P1, P2)$ 形成一條線段
- 目標是要判斷 $P3$ 是否落在線段 $(P1, P2)$ 之間



應用：判斷點是否在線段內

- 利用 dot 可以巧妙的解決這個問題
- 設向量 $\overrightarrow{V32} = P2 - P3, \overrightarrow{V31} = P1 - P3$
- 可以簡單的發現若 $\overrightarrow{V32} \cdot \overrightarrow{V31} \leq 0$
  則 $P3$ 就會落在線段 $(P1, P2)$ 之間

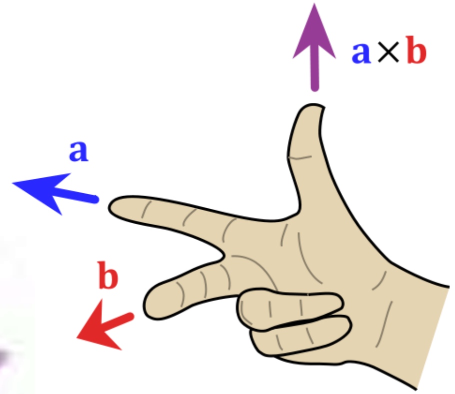
非常短的程式碼 between

三點共線有其他判斷方式，之後會說

```
// 使用之前要先確定三點要共線才會是正確的
bool btw(const PT &p1, const PT &p2, const PT &p3) {
  return (p1 - p3).dot(p2 - p3) <= 0;
}
```

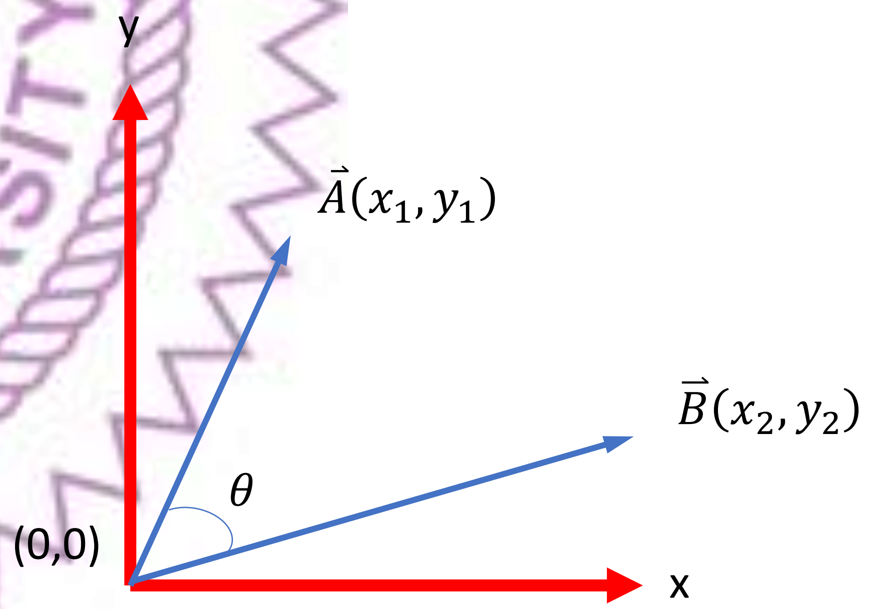
cross(叉積)

- 定義(只考慮二維)：
  - $\vec{A} \times \vec{B} = \mathrm{x}1 \times y2 - y1 \times x2 = \|\vec{A}\| \times \|\vec{B}\| \times |\sin\theta| \times n$
  - n 可以是 1 or -1，用右手定則判斷



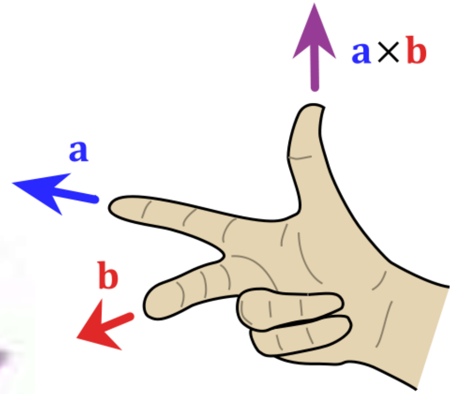
圖片來自於維基百科

- 性質：
  - $\vec{A} \times \vec{B} = -\vec{B} \times \vec{A}$
  - $\vec{A} \cdot (\vec{B} \times \vec{C}) = \vec{B} \cdot (\vec{C} \times \vec{A}) = \vec{C} \cdot (\vec{A} \times \vec{B})$



cross(叉積) – 右手定則

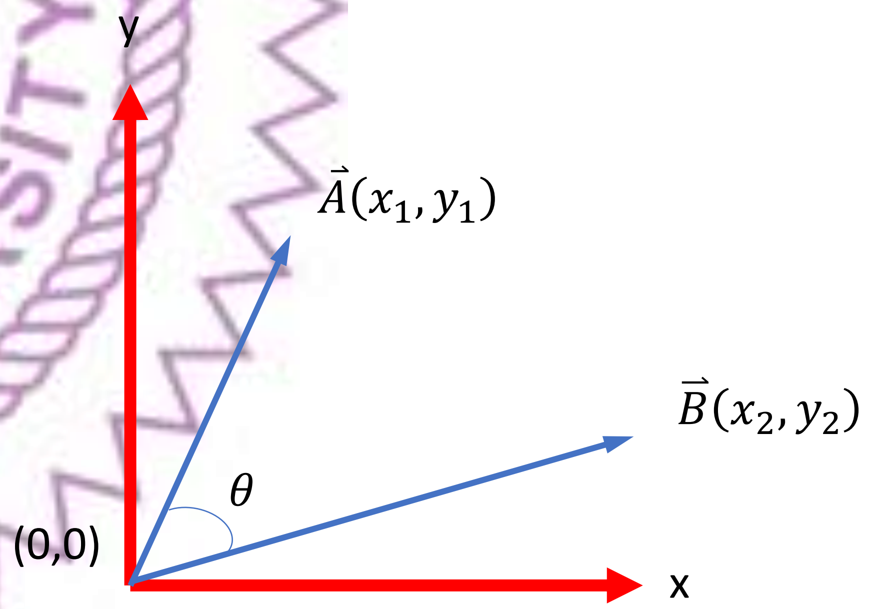
- 定義(只考慮二維)：
  - $\vec{A} \times \vec{B} = \mathrm{x1} \times y2 - y1 \times x2 = \|\vec{A}\| \times \|\vec{B}\| \times |\sin\theta| \times n$



圖片來自於維基百科

- 將中指表示 $\|\vec{B}\|$，食指表示 $\|\vec{A}\|$
  拇指的方向即是 n 的方向
  往上指表示 1；往下指表示 -1

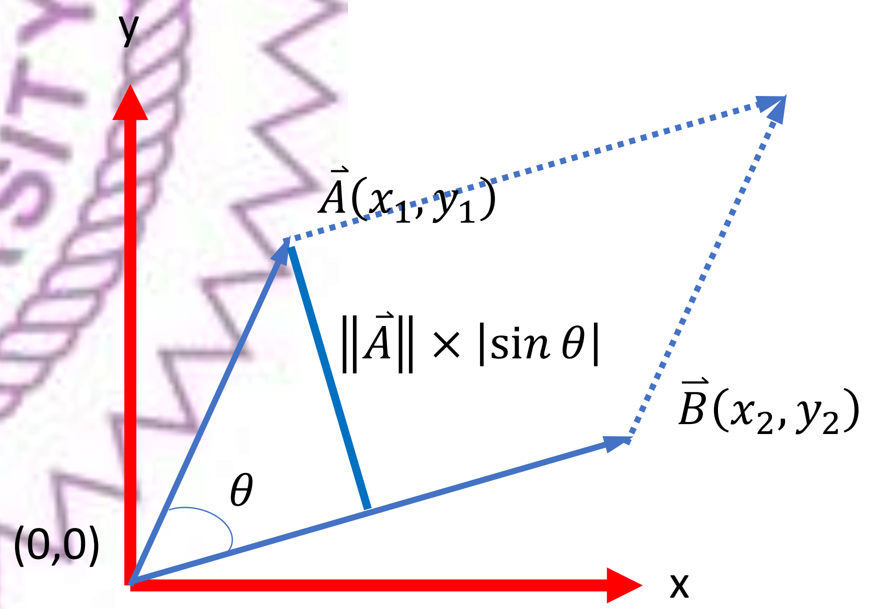
- 如果$\vec{A}$ 到$\vec{B}$是逆時針方向
  $\vec{A} \times \vec{B}$結果為正，反之$\vec{A} \times \vec{B}$結果為負



cross(叉積)

- 定義：
  - $\vec{A} \times \vec{B} = \mathrm{x1} \times y2 - y1 \times x2 = \|\vec{A}\| \times \|\vec{B}\| \times |\sin\theta| \times n$

- 幾何意義：
  - $\|\vec{A}\| \times |\sin\theta|$ 為圖中藍色部分長度
  - $\|\vec{B}\| \times \|\vec{A}\| \times |\sin\theta|$ 就是 $\vec{A}, \vec{B}$ 構成的平行四邊形面積！
  - $|\vec{A} \times \vec{B}| = \vec{A}, \vec{B}$ 構成的平行四邊形面積 (除以二就是兩向量構成的三角形面積)



cross(叉積) 程式碼

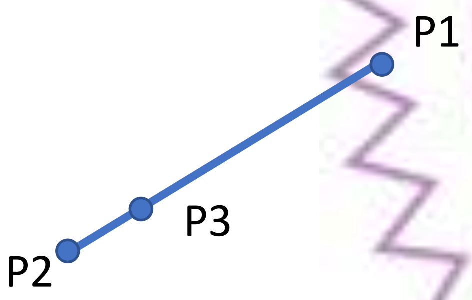
```
double cross(const PT &b) const { // 外積
  return x * b.y - y * b.x;
}
...
PT A(7, 1), B(2, 2);
cout << A.cross(B) << endl; // 輸出12
```

應用：判斷三點是否共線

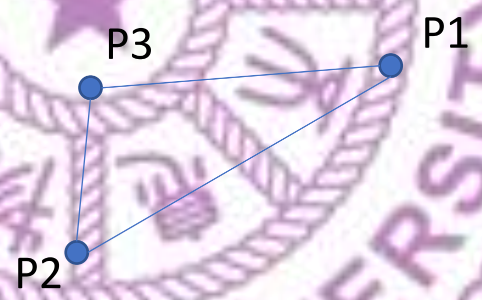
- 給你三個點$P1, P2, P3$，你要判斷這三個點是不是在同一條直線上

應用：判斷三點是否共線

- 給你三個點$P1, P2, P3$，你要判斷這三個點是不是在同一條直線上
- 判斷三個點構成的三角形面積是否為 0 就可以了



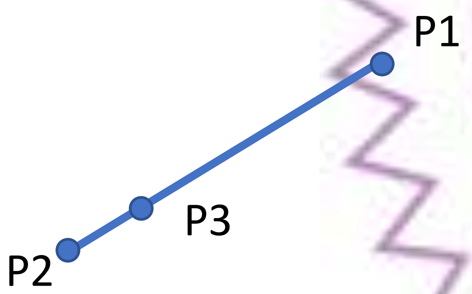
面積為0



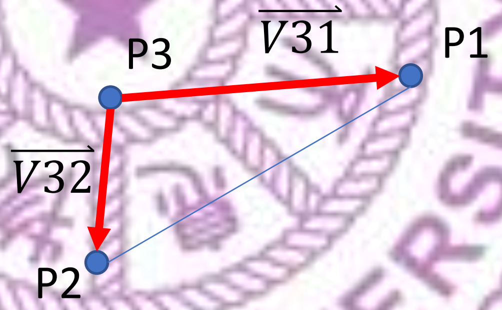
面積不為0

應用：判斷三點是否共線

- 用 cross 可以輕鬆做到
- 設向量 $\overrightarrow{V32} = P2 - P3, \overrightarrow{V31} = P1 - P3$
- $\overrightarrow{V32} \times \overrightarrow{V31} = 0$ 表示三點共線



面積為0



面積不為0

非常短的程式碼

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

結合：判斷 $P3$ 是否在線段 $(P1, P2)$ 上
三點不用共線！

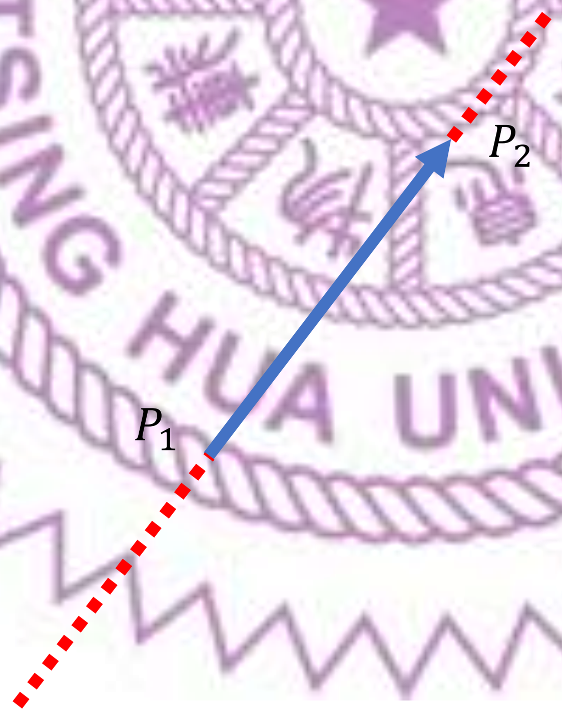
```
// 結合!!
bool pointOnSegment(const PT &p1, const PT &p2, const PT &p3) {
  return collinearity(p1, p2, p3) && btw(p1, p2, p3);
}
```

線段與有向面積

Area vector direction

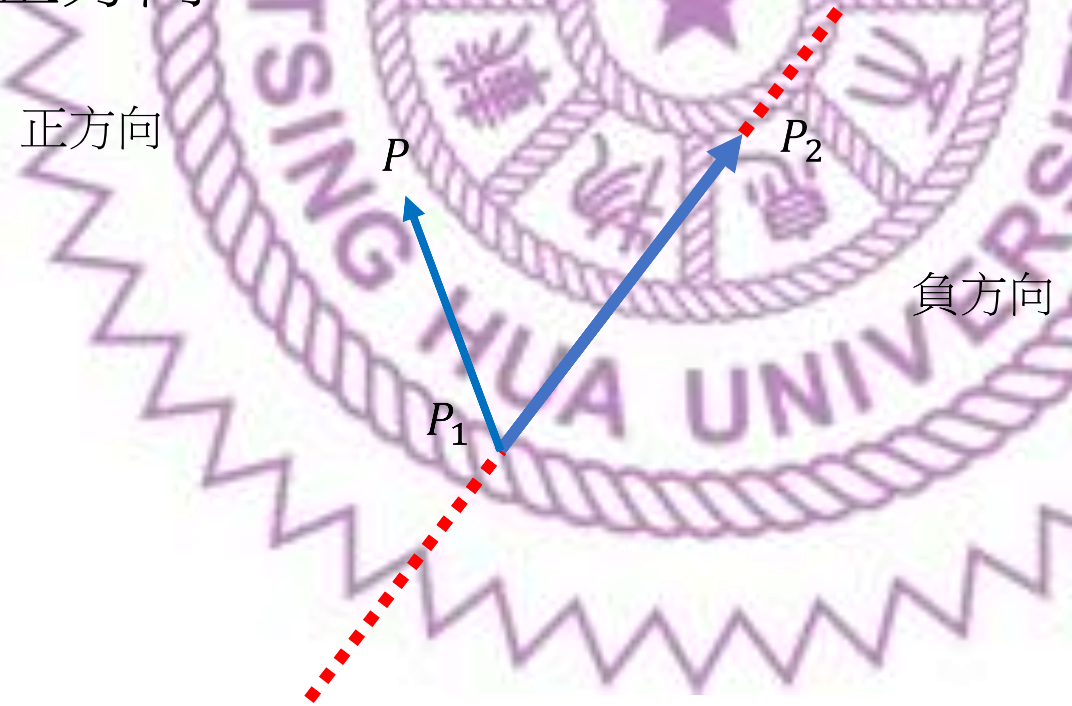
有向線段

- 對於一條線段 $\overline{P_1P_2}$，在本單元中會用 (P_1, P_2) 來表示
- 這樣表示的意義是這個線段的方向是 P_1 到 P_2
- (P_1, P_2) 的延伸直線會把平面切成兩個部分



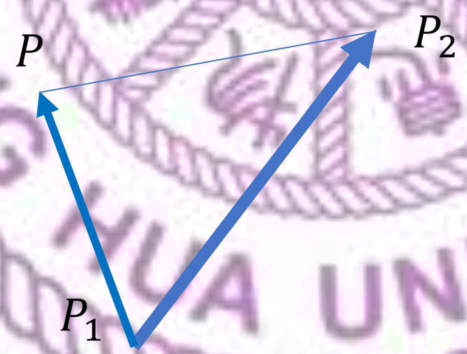
有向線段

- 對於任何一個點 P，若滿足 $(P_2 - P_1) \times (P - P_1) > 0$ 則它所在的區域為 (P_1, P_2) 的正方向
- 想像你現在站在 P_1，往 P_2 的方向看，左手邊的區域都屬於 (P_1, P_2) 的正方向



有向面積

- 其實 $\frac{|(P_2-P_1)\times(P-P_1)|}{2}$ 就是 P_1, P_2, P 三點所組成的三角形面積
  - 用 $\Delta P_1 P_2 P$ 表示
- 由於 cross 運算有可能是負的，因此要加上絕對值
- 將絕對值拿掉後就稱為有向面積



有向面積

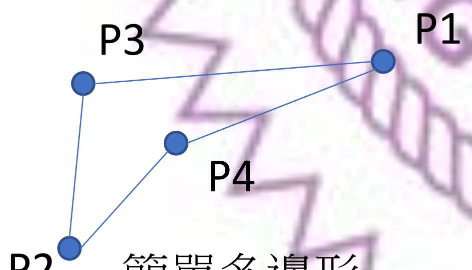
- 逆時針旋轉時是正的，順時針旋轉時是負的

$(P_2 - P_1) \times (P - P_1) > 0$

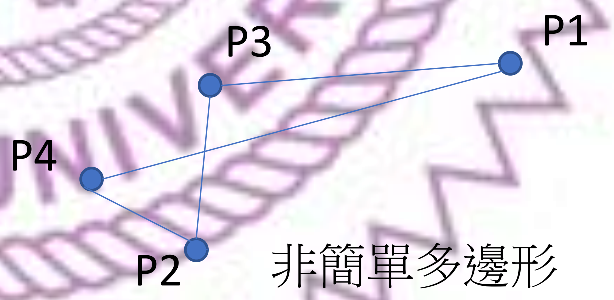
$(P_2 - P_1) \times (P - P_1) < 0$

簡單多邊形面積

- 除了相鄰邊可以交於多邊形的頂點外，邊不相交的多邊形
- 利用有向面積的正負性質可以方便的計算簡單多邊形的面積



簡單多邊形



非簡單多邊形

簡單多邊形面積

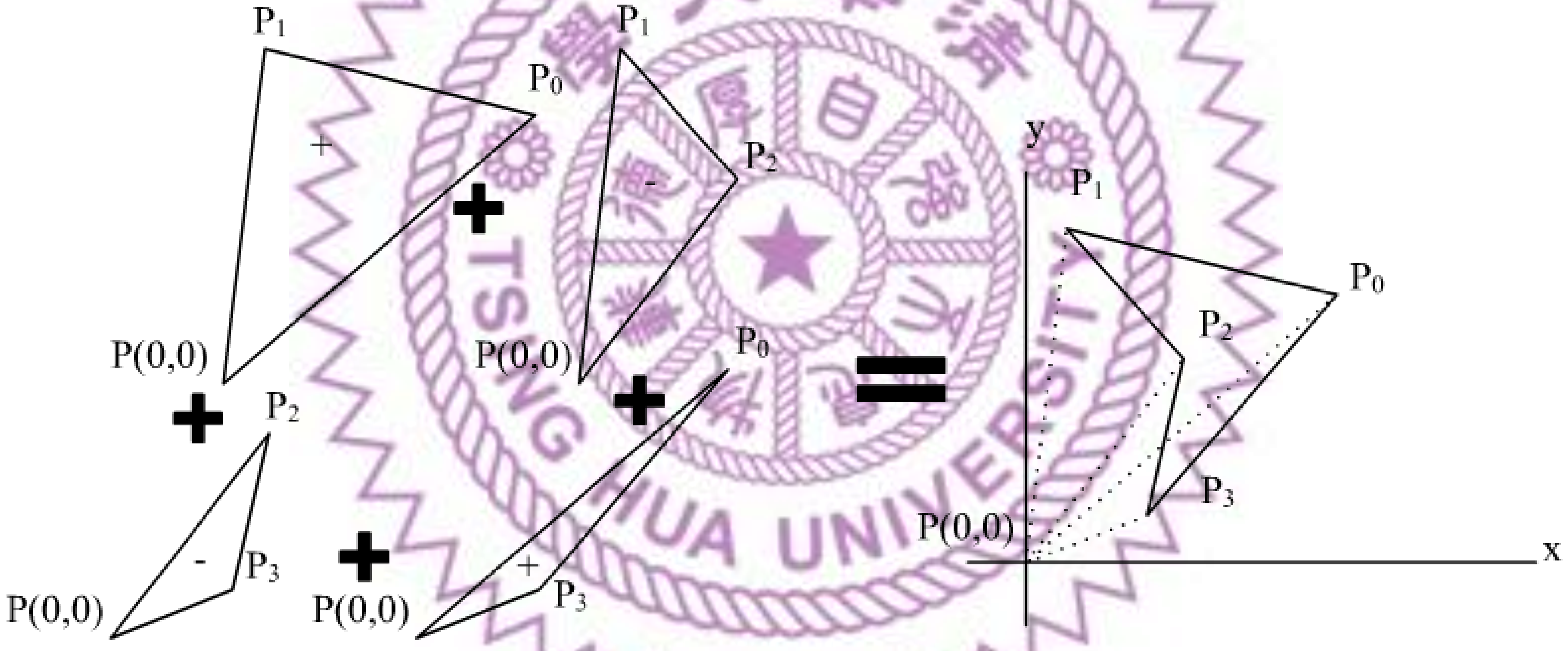
- 給一個 n 個點的簡單多邊形，其點的逆時針順序為$P_0, P_1, \dots, P_{n-1}$
- **"任選"**一個點 P，其面積可以用以下的公式算出來：

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

- 通常為了方便 P 會選 (0,0)

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

簡單多邊形面積



簡單多邊形面積

```cpp
double area(const vector<PT> &Polygon) {
  if (Polygon.size() <= 1)
    return 0;
  double ans = 0;
  for (auto a = --Polygon.end(), b = Polygon.begin(); b != Polygon.end();
       a = b++)
    ans += a->cross(*b);
  return ans / 2;
}
```

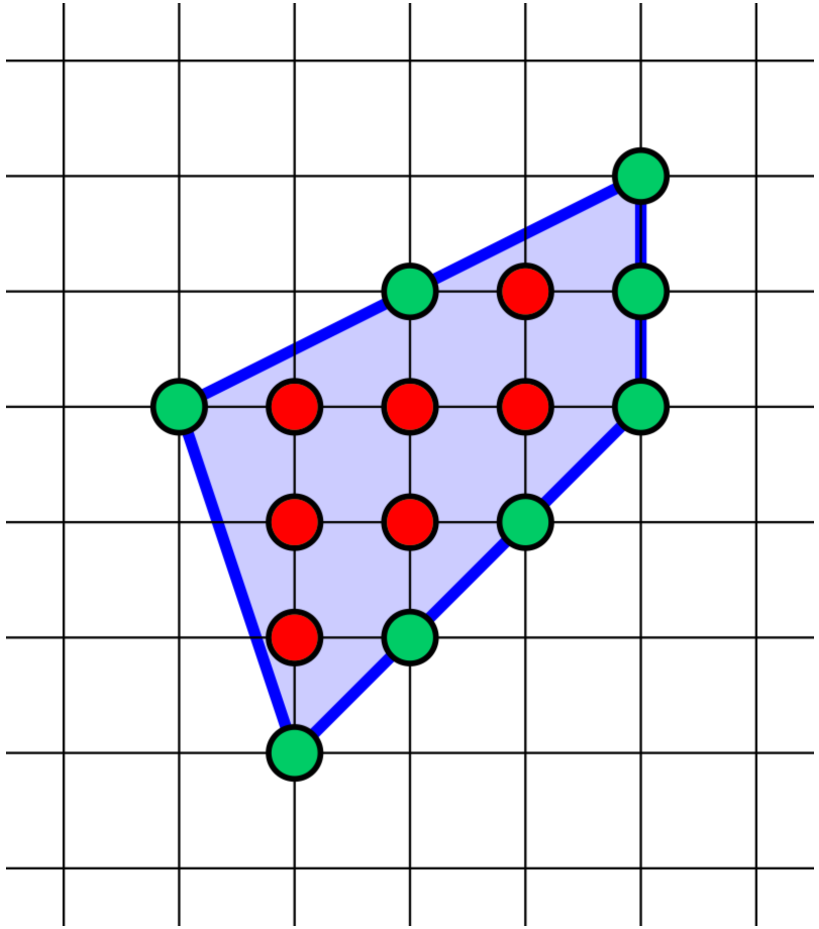
有向面積正負 orientation

- 有向面積的正負判斷之後會經常遇到
  我們會以以下的函數計算它：

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

$$i = 7, b = 8$$

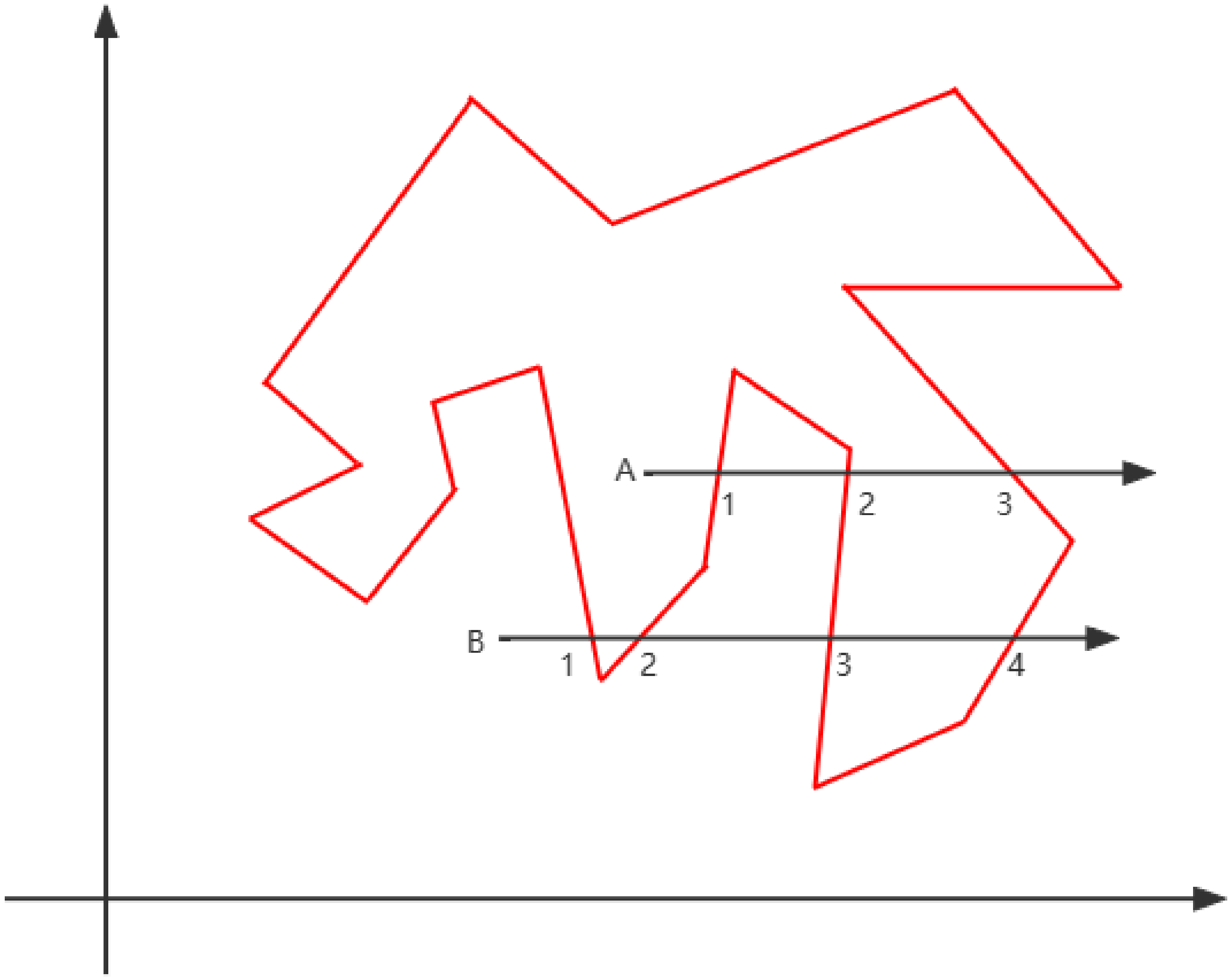
$$A = 7 + \frac{8}{2} - 1 = 10$$



皮克定理

- 若簡單多邊形的所有頂點都在整數點上
  - 設邊上的整數點數為 b
  - 多邊形內部的整數點數為 i
  - 多邊形面積為 A
- 則：

$$A = i + \frac{b}{2} - 1$$



點是否位於簡單多邊形內部

- 任何位於多邊形內部的點
- 延伸出的任何射線
- 只會穿過奇數條邊

程式碼

```cpp
// 點是否在簡單多邊形內，是的話回傳 1、在邊上回傳 -1、否則回傳 0
int pointInPolygon(const vector<PT> &Polygon, const PT &p) {
  int ans = 0;
  for (auto a = --Polygon.end(), b = Polygon.begin(); b != Polygon.end();
       a = b++) {
    if (pointOnSegment(*a, *b, p))
      return -1;
    if ((a->y > p.y) != (b->y > p.y) &&
        (p.x - b->x) < (a->x - b->x) * (p.y - b->y) / (a->y - b->y)) {
      ans = !ans;
    }
  }
  return ans;
}
```

用從 p 射出
無限向右延伸的線段來判斷

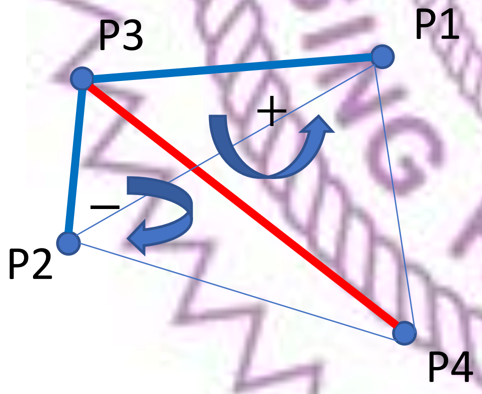
判斷線段相交、找出直線交點

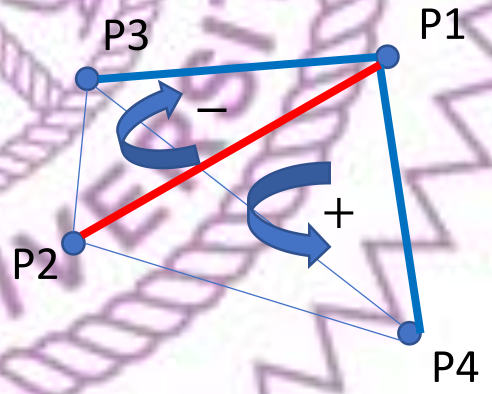
Segment intersection

判斷線段相交

- 有了判斷有向面積正負的函數 ori
  線段 $(P1, P2), (P3, P4)$ 的相交判定函數就可以變得很簡單：

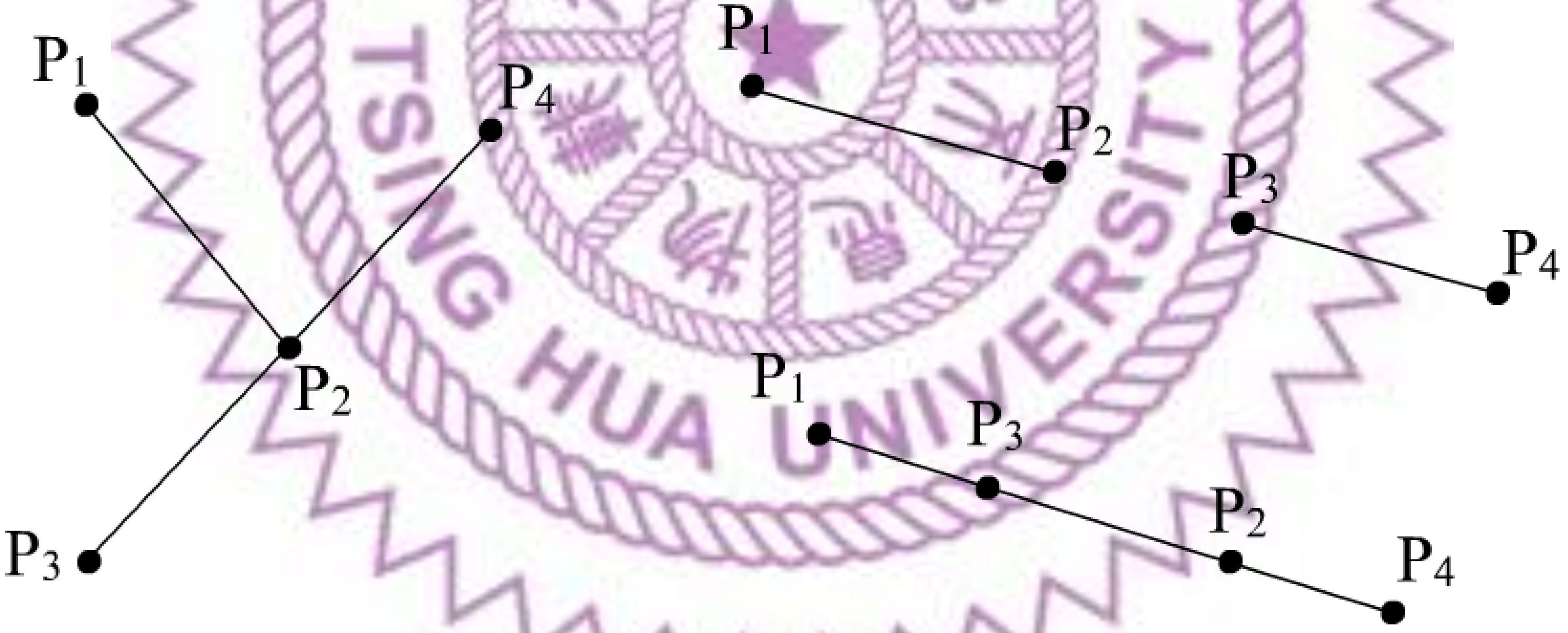
$$ori(P1, P2, P3) \times ori(P1, P2, P4) < 0 \land ori(P3, P4, P1) \times ori(P3, P4, P2) < 0$$





特例

- 左邊解決方法只要把判斷式的 $<$ 改成 $\leq$ 就行了
- 但右邊的情形都會被判斷成相交，要分開來處理

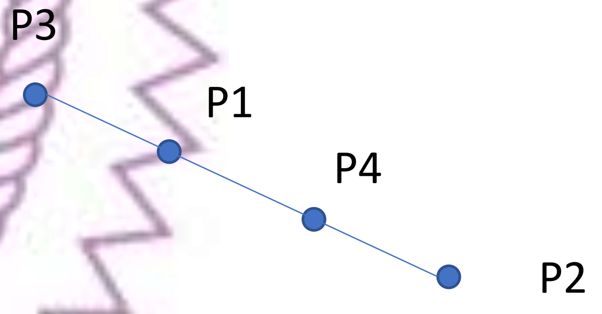


共線時判斷

- 共線時如果有相交的話
  一定會存在另一條線段的某個點被包含在某線段中
- 共線時判斷某點是否被包含在某線段中可以用之前的 *btw* 函數
- 只要

```
btw(p1,p2,p3) //p3介於(p1,p2)中
btw(p1,p2,p4) //p4介於(p1,p2)中
btw(p3,p4,p1) //p1介於(p3,p4)中
btw(p3,p4,p2) //p2介於(p3,p4)中
```

  其中一條成立
  那就表示兩線段在共線時有相交

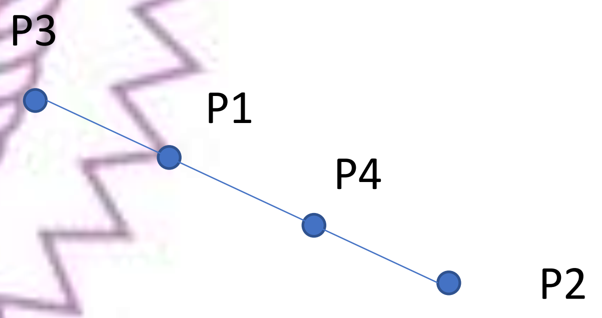


程式碼

```
bool seg_intersect(const PT &p1, const PT &p2, const PT &p3, const PT &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 (如果兩線段共線)
    return btw(p1, p2, p3) || btw(p1, p2, p4) || btw(p3, p4, p1) ||
           btw(p3, p4, p2);
  else if (a123 * a124 <= 0 && a341 * a342 <= 0)
    return true;
  return false;
}
```

如何判斷共線？

- **if(如果兩線段共線)?**
- 這裡可以用簡單的方式進行判斷
- 根據觀察，可以發現共線時 $\Delta P_1P_2P_3$ 和 $\Delta P_1P_2P_4$ 面積都是 0
- 也就是 a123==0 且 a124==0

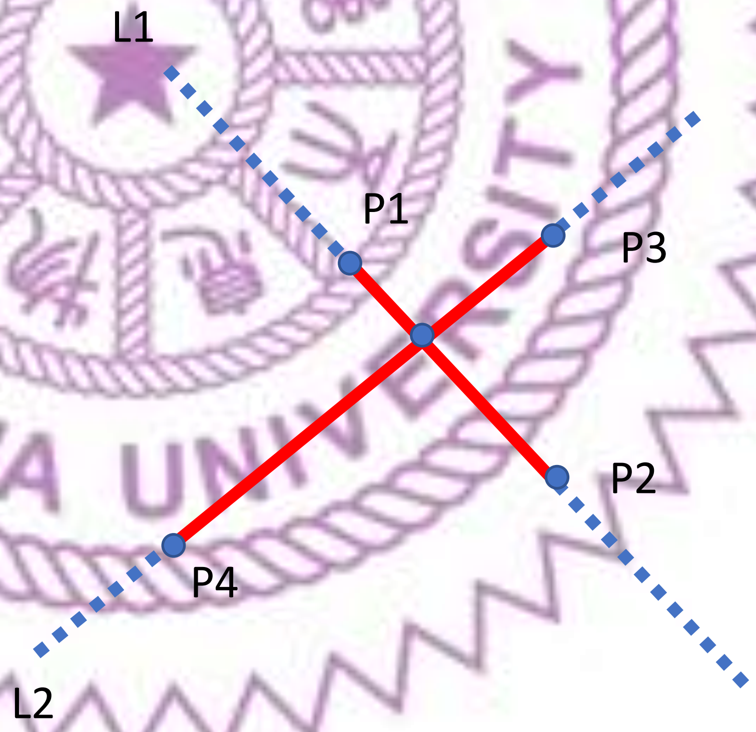


完整程式碼

```
bool seg_intersect(const PT &p1, const PT &p2, const PT &p3, const PT &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) // 共線情況
    return btw(p1, p2, p3) || btw(p1, p2, p4) || btw(p3, p4, p1) ||
           btw(p3, p4, p2);
  else if (a123 * a124 <= 0 && a341 * a342 <= 0)
    return true;
  return false;
}
```

兩直線交點

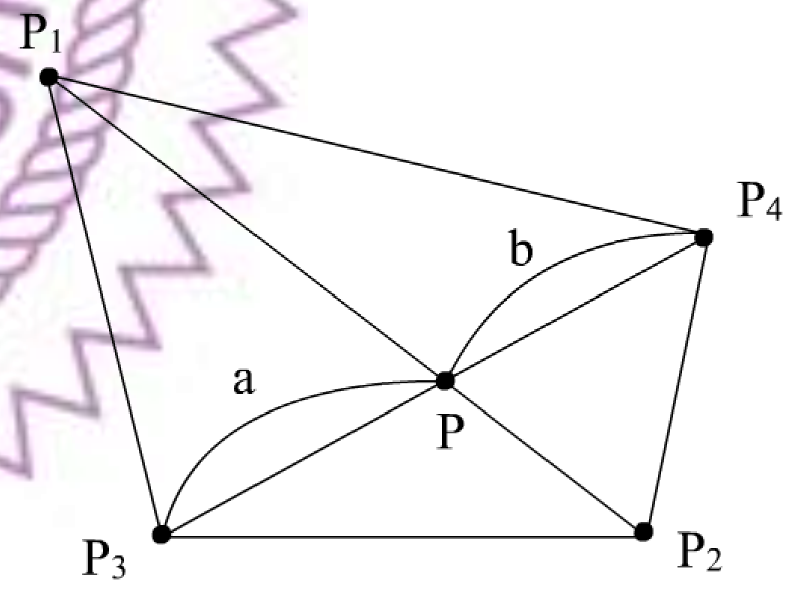
- 給定四個點 P_1, P_2, P_3, P_4
  其中經過 P_1, P_2 的直線稱為 L_1，經過 P_3, P_4 的直線稱為 L_2
- 請求出 L_1, L_2 的交點



兩直線交點 – 分點公式

$$a:b \;=\Delta P_1P_2P_3:\Delta P_1P_2P_4$$
$$= (P_2-P_1)\times(P_3-P_1):-(P_2-P_1)\times(P_4-P_1)$$
$$\Rightarrow P=\frac{\Delta P_1P_2P_3\times P_4+\Delta P_1P_2P_4\times P_3}{\Delta P_1P_2P_3+\Delta P_1P_2P_4}$$

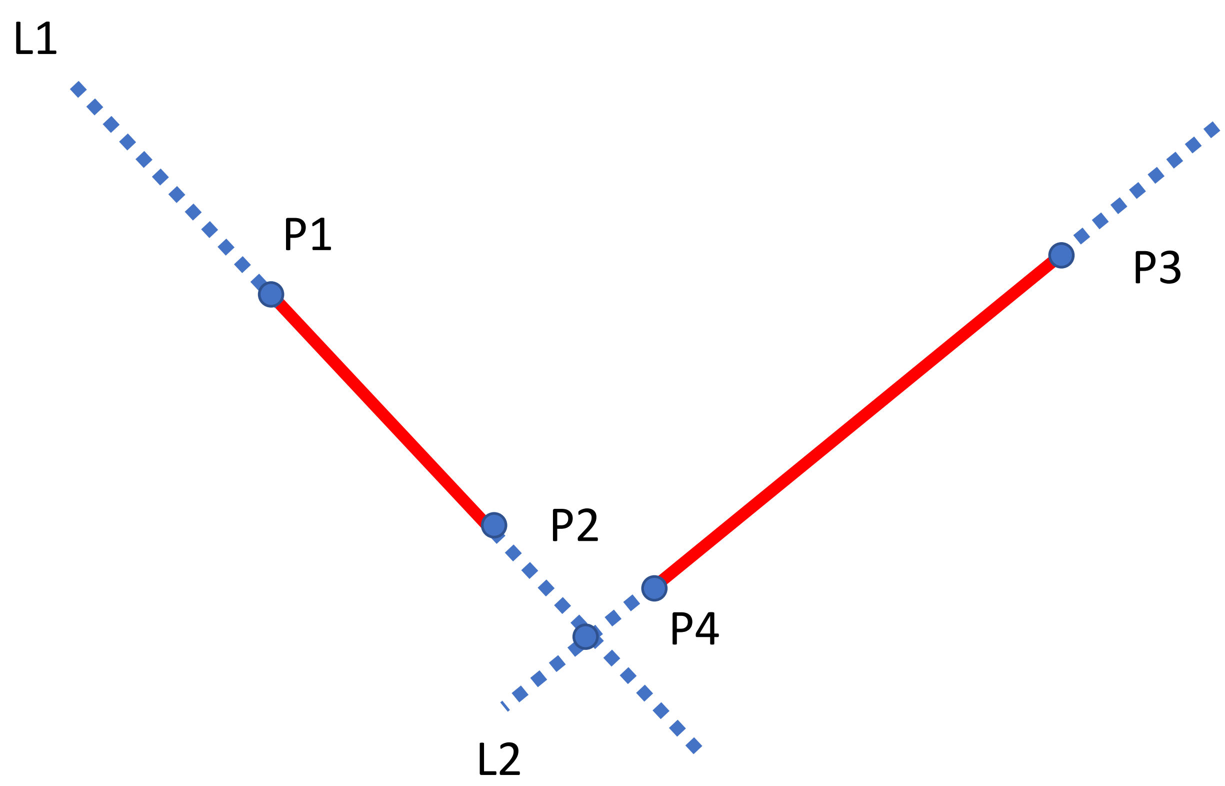
- 但要注意會一正一負
  因此其中一邊要加上負號
- 注意交點不存在的話
  會發生除以 0 的錯誤



很短的程式碼

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

就算這樣也算得出來！



浮點數誤差

round-off error

計算誤差

- 就算使用 double
  經過很多次的乘法、除法算之後，還是有可能會有誤差

- 經常造成需要判斷大小時發生錯誤

處理誤差

- 通常我們會設一個很小的數 $\varepsilon(eps)$
- 通常會設 $\varepsilon = 10^{-7} \sim 10^{-9}$
- 當判斷大小時只要差距不超過ε就當作滿足條件

範例

```
const double EPS = 1e-9;
double a = 1.0000000007122;
if( a == 1 ) cout<<"GG\n"; //不會輸出

if( abs(a-1) <= EPS ) cout<<"EPS!\n";
// 1-EPS <= a <= 1+EPS
```

常用判斷加上浮點誤差處理

- $a < 0 \rightarrow a < -\varepsilon$
- $a \leq 0 \rightarrow a \leq \varepsilon$
- $a > 0 \rightarrow a > \varepsilon$
- $a \geq 0 \rightarrow a \geq -\varepsilon$
- $a = 0 \rightarrow abs(a) \leq \varepsilon$

凸包
最遠點對

日月卦長

題目：最遠點對

- 平面上有 $n(2 \leq n \leq 50000)$ 個點
  請你找出距離最遠的兩個點，輸出其距離的平方

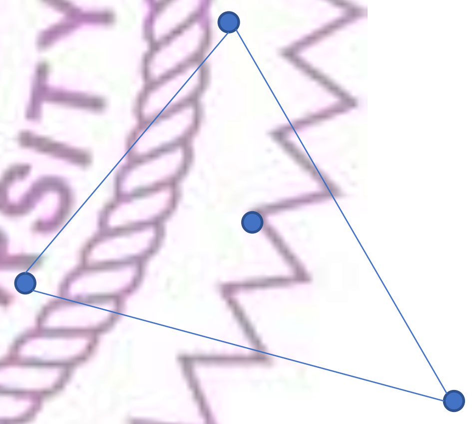
- 點座標範圍：$-10000 \leq x, y \leq 10000$

- 顯然直覺上可以想到 $O(n^2)$ 的暴力法
  但顯然這題的範圍不予許這樣做

觀察

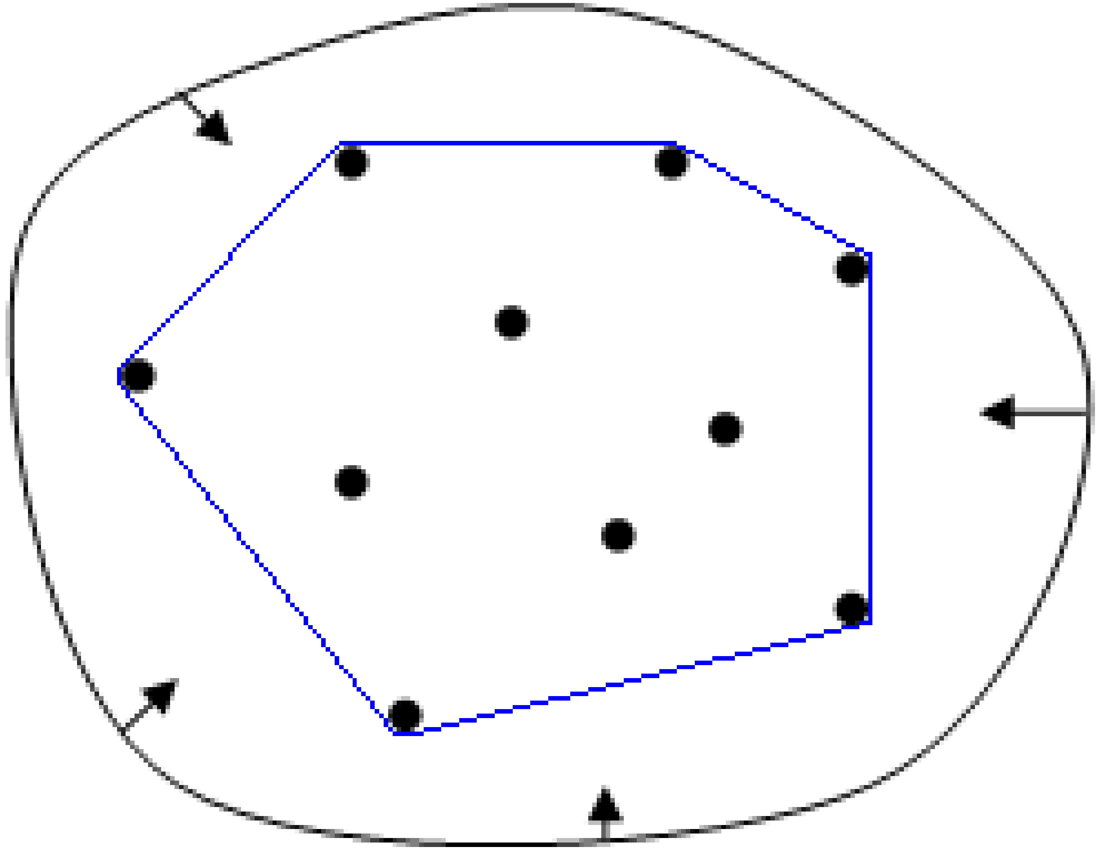
- 只要有一個點在另外三個點構成的三角形中
  這個點絕對不會是距離最遠的兩個點之一

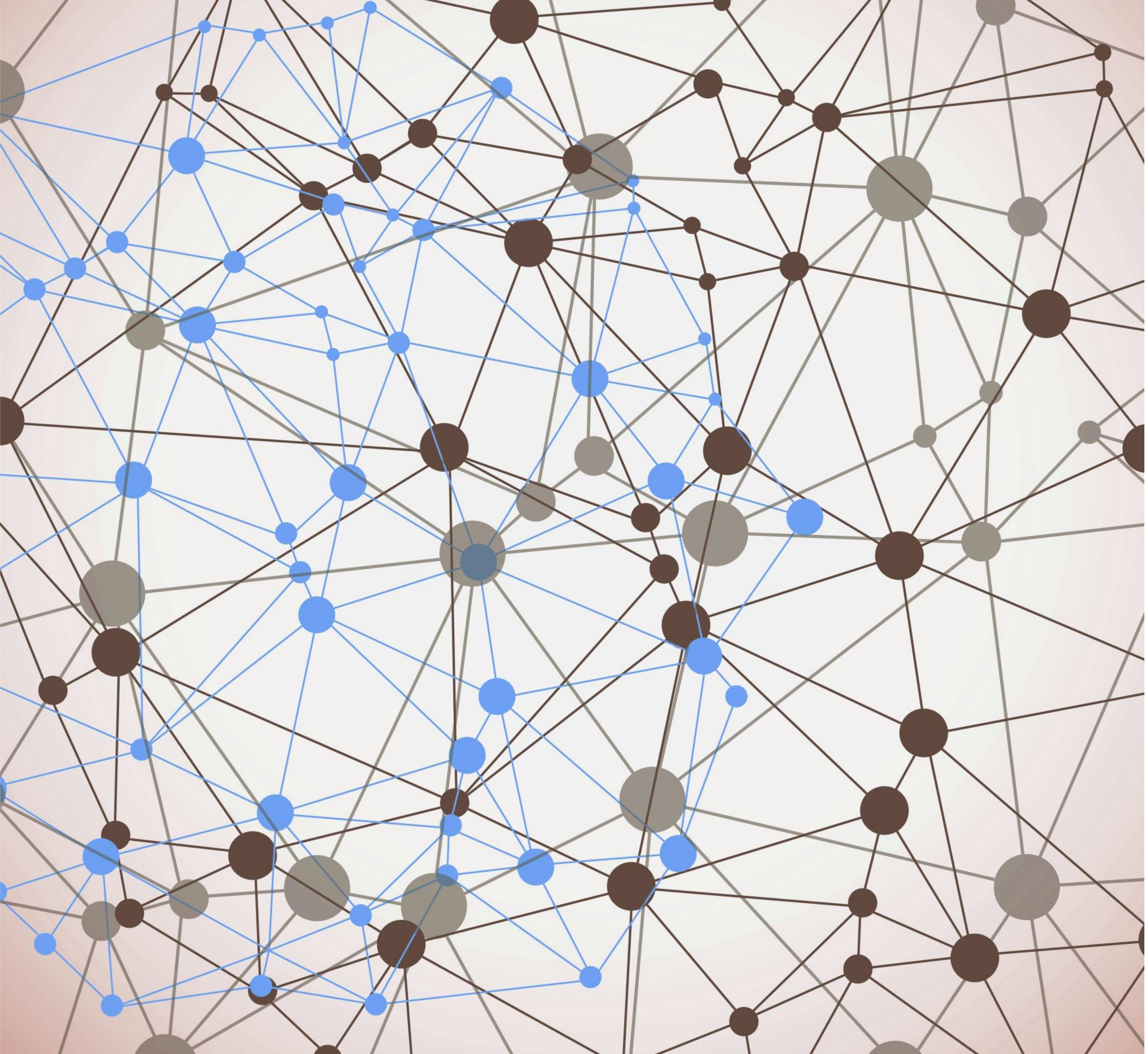
- 不再任何三角形內的點➔最外側的點



最外側的點集合 – 凸包

- 將所有點用繩子圈起來然後拉緊，圍成的多邊形的頂點，就是最外側的點
- 這些點的集合稱為**凸包**
- 通常我們會用逆時針的順序來存這些點以及其他多邊形



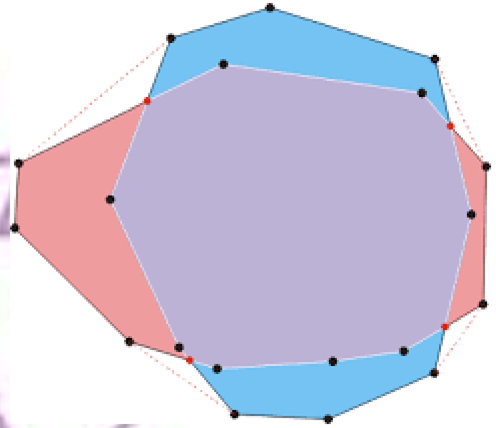


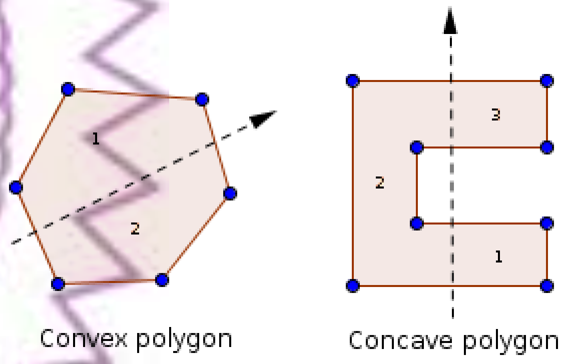
性質

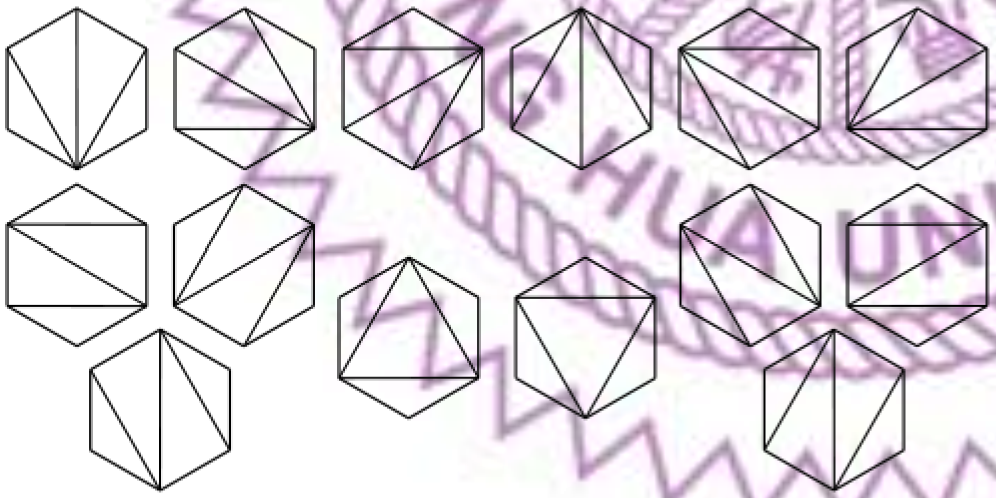
凸集合性質
整數點數限制
邊斜率單調

凸集合性質

- 兩凸包的交集還是凸包
- 用直線切割凸包還會是凸包
- 凸包中任兩點構成的線段一定會在凸包內







圖片來自於網路

整數點數量限制

- 若所有點的座標範圍都是限制在 $[0, W]$ 的整數那凸包上最多只會有 $O(\sqrt{W})$ 個點
- 原因是凸包中沒有三點共線的情況，具體證明在這裡略過
- 如果找出凸包候用凸包的點枚舉最遠點對的話複雜度變成
$$O\left(\sqrt{W}^2\right) = O(W)$$
- 最遠點對問題因為座標範圍是
$$-10000 \leq x, y \leq 10000$$
所以 $W = 20000$，不會 TLE！

邊斜率單調

- 凸包上每條邊的斜率 (角度) 是依序增加的
  需要在邊上枚舉時可以用 binary search 降低複雜度

- Ex: 判斷點是否在凸包中或線段是否與凸包相交
  複雜度可以做到 $O(\log n)$

- 更為進階的旋轉卡尺會用到此性質

點是否在凸包內 $O(\log n)$

```
// 點是否在凸包內，是的話回傳 1、在邊上回傳 -1、否則回傳 0
int point_in_convex(const vector<PT> &ConvexHull, const point<T> &p) const {
  int l = 1, r = (int)ConvexHull.size() - 2;
  while (l <= r) {
    int m = (l + r) / 2;
    auto a1 = (ConvexHull[m] - ConvexHull[0]).cross(p - ConvexHull[0]);
    auto a2 = (ConvexHull[m + 1] - p[0]).cross(p - ConvexHull[0]);
    if (a1 >= 0 && a2 <= 0) {
      auto res = (ConvexHull[m + 1] - ConvexHull[m]).cross(p - ConvexHull[m]);
      return res > 0 ? 1 : (res >= 0 ? -1 : 0);
    }
    if (a1 < 0) r = m - 1;
    else l = m + 1;
  }
  return 0;
}
```

找出凸包

Andrew‘s Monotone Chain Algorithm

方法很多

這個比較簡單

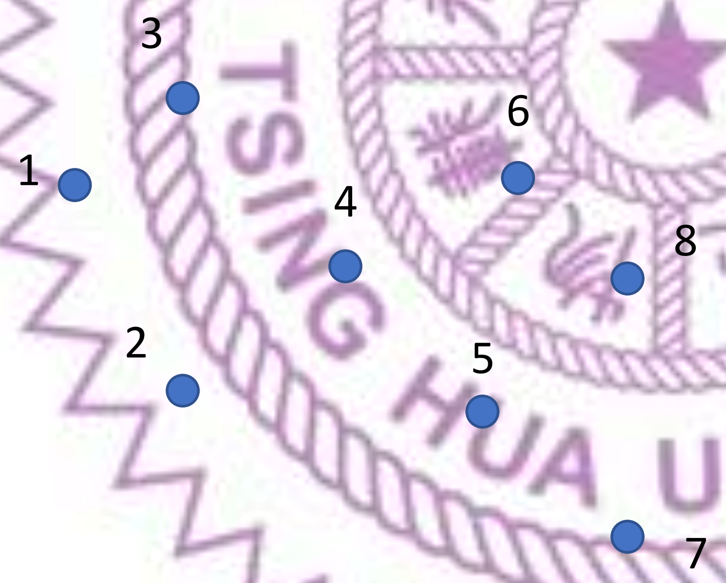
- Jarvis 步進法
- Graham 掃描法
- 單調鏈 (Andrew‘s Monotone Chain)
- 分治法

步驟

排序 → 構造下半部分 → 構造上半部分

排序

- 把所有點按造 x 座標由小到大排，如果 x 相同的就根據 y 排



排序比較函數

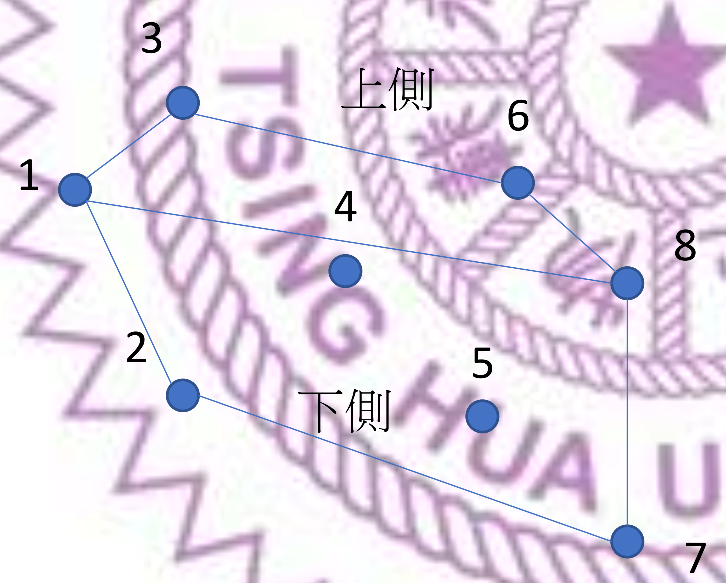
```
bool x_cmp(const PT &a, const PT &b) {
  if (a.x != b.x)
    return a.x < b.x;
  return a.y < b.y;
}
```

把判斷寫成一行

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

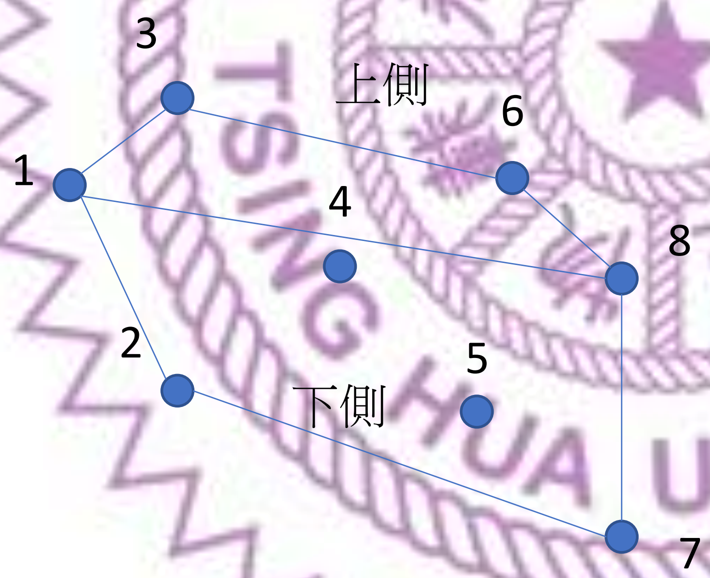
找出下(上)部分

- 排序好後最小和最大的點一定會在凸包裡面
- 這兩個點連線後可以將凸包分成上下兩部分



找出下(上)部分

- 上下兩部份分別找出來後凸包就完成了
- 找出上下部分的做法差不多，因此會以下側進行說明



找出下側

- 一開始甚麼都沒有

找出下側

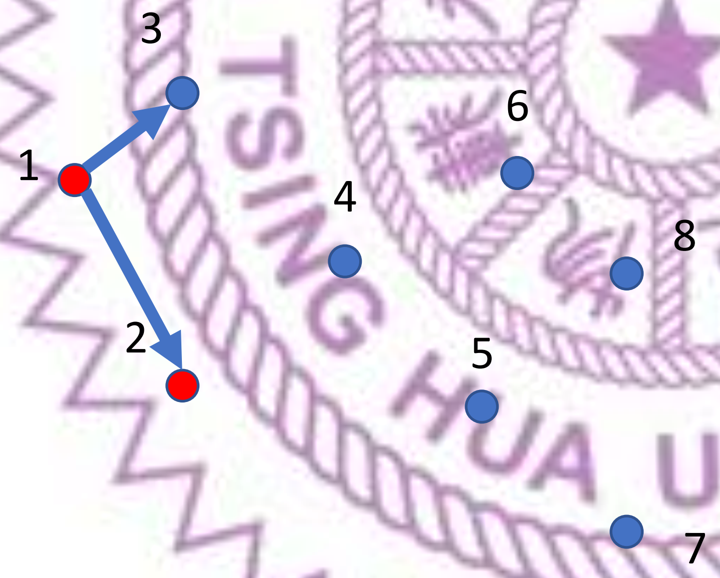
- 將第1個點加進來

找出下側

- 加入第2個點

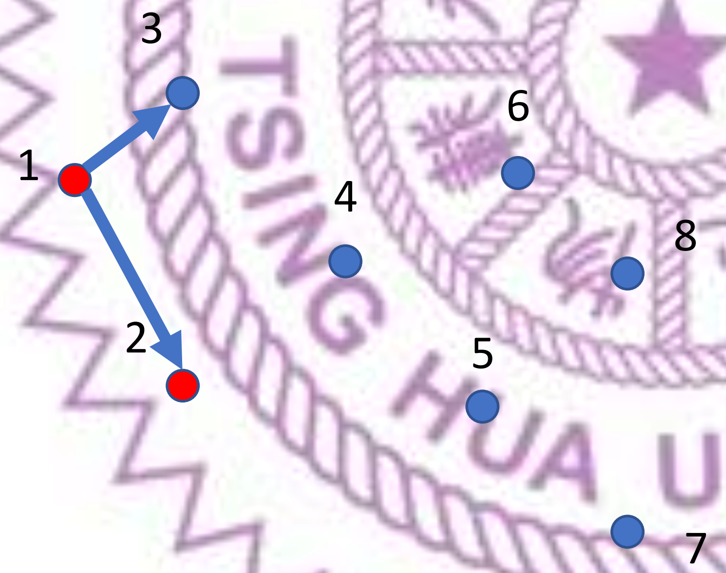
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



找出下側

- $\overrightarrow{12} \times \overrightarrow{13} > 0$不會造成凹陷
- 將第3個點加入凸包

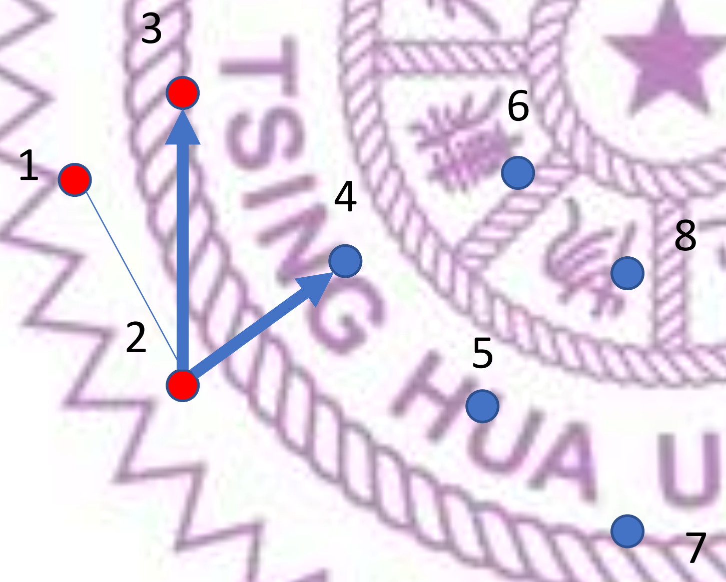


找出下側

- 將第3個點加入凸包

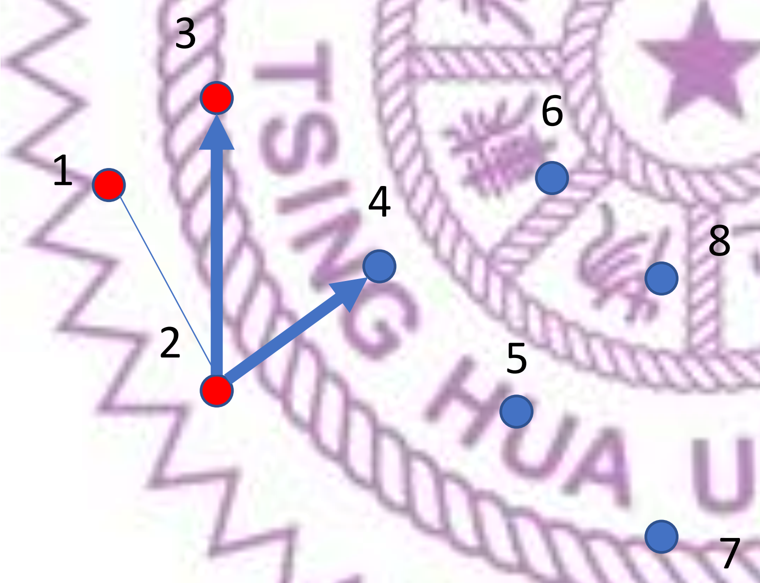
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



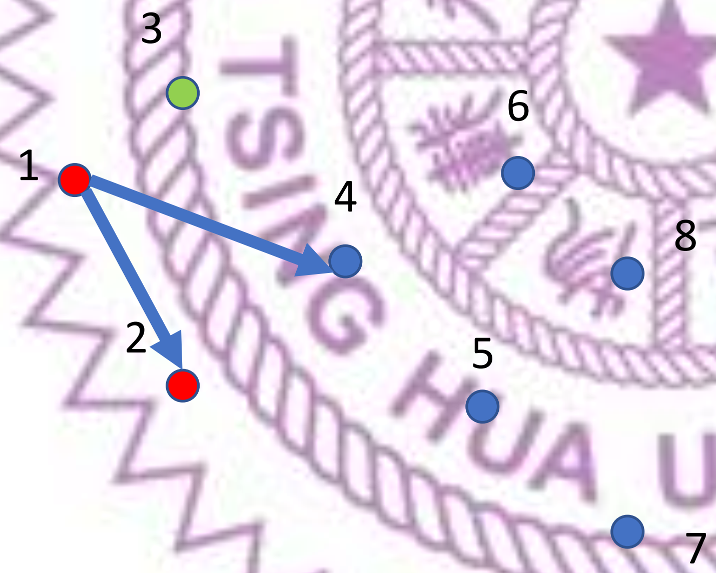
找出下側

- $\overrightarrow{23} \times \overrightarrow{24} \leq 0$會造成凹陷
- 因此要把第3個點移除



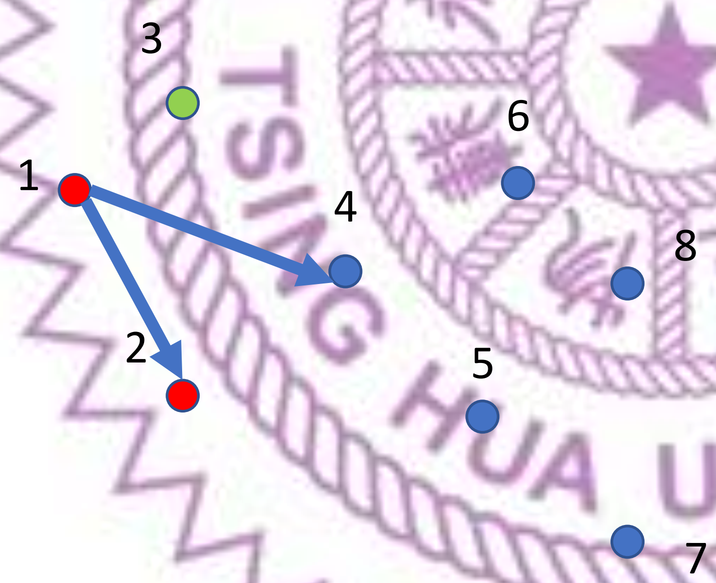
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



找出下側

- $\overrightarrow{12} \times \overrightarrow{14} > 0$不會造成凹陷
- 將第4個點加入凸包

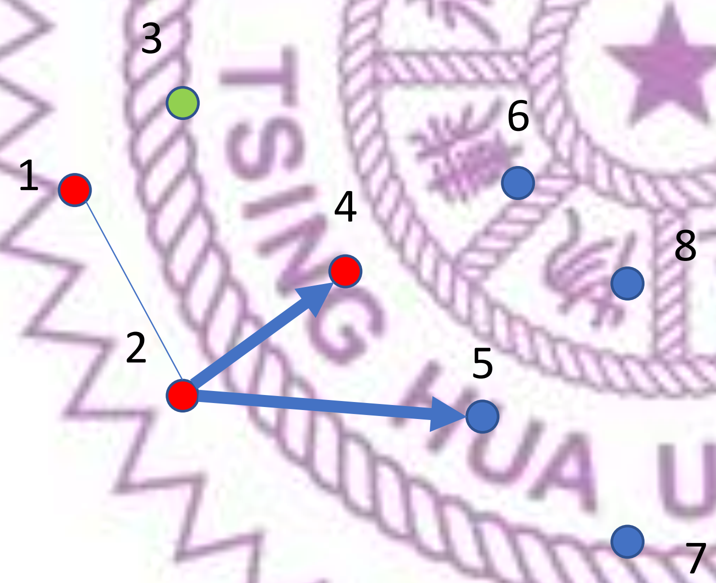


找出下側

- 將第4個點加入凸包

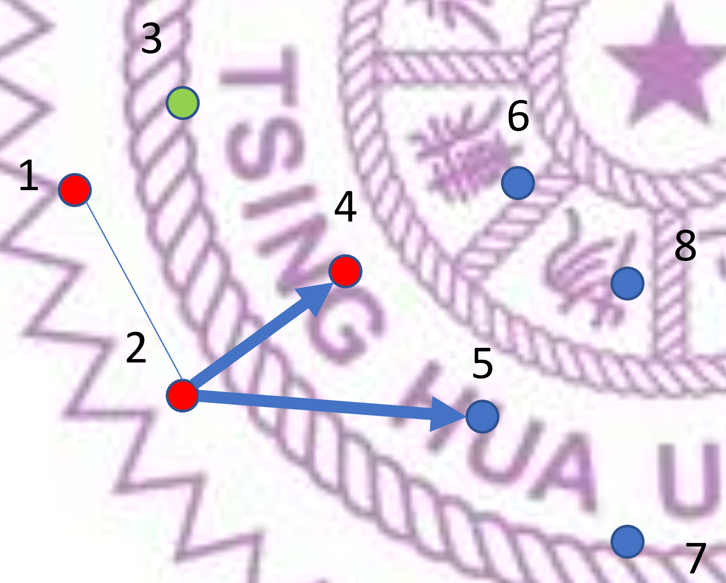
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



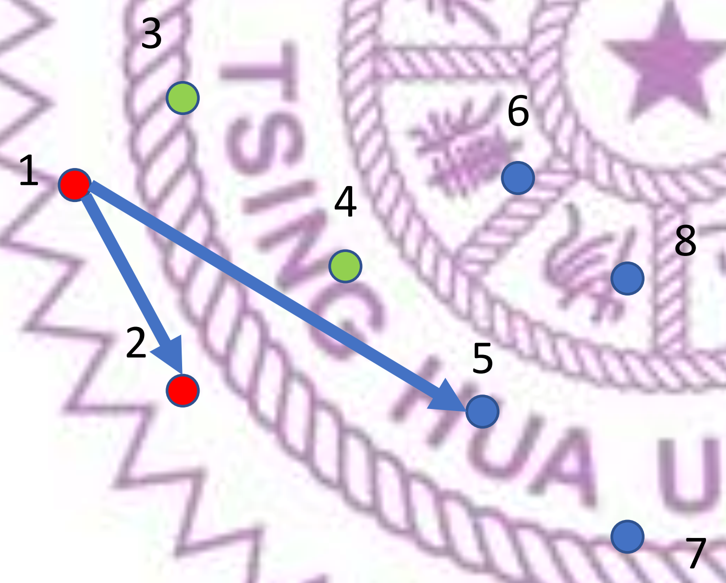
找出下側

- $\overrightarrow{24} \times \overrightarrow{25} \leq 0$會造成凹陷
- 因此要把第4個點移除



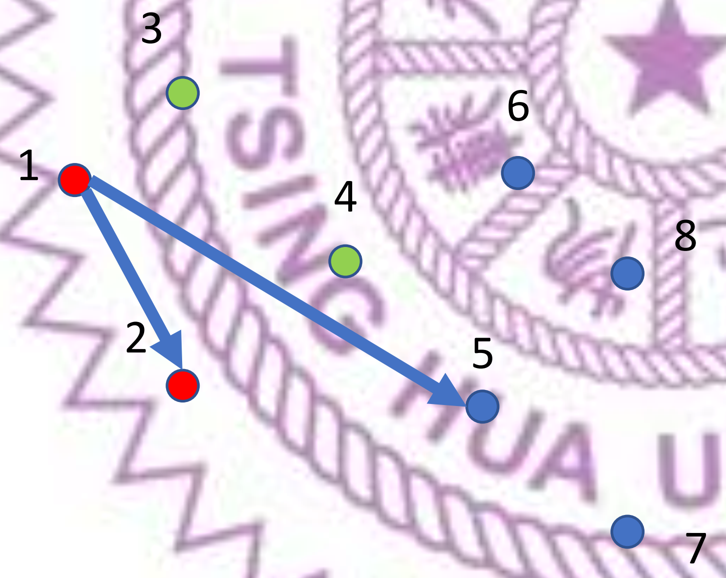
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



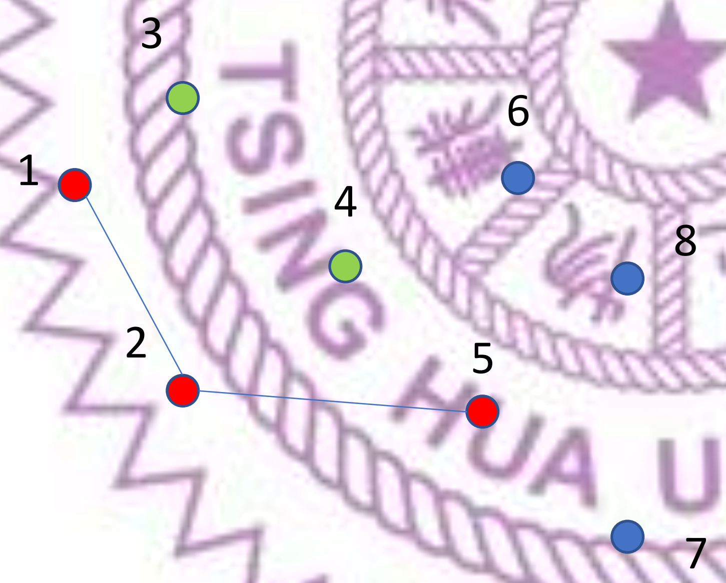
找出下側

- $\overrightarrow{12} \times \overrightarrow{15} > 0$不會造成凹陷
- 將第5個點加入凸包



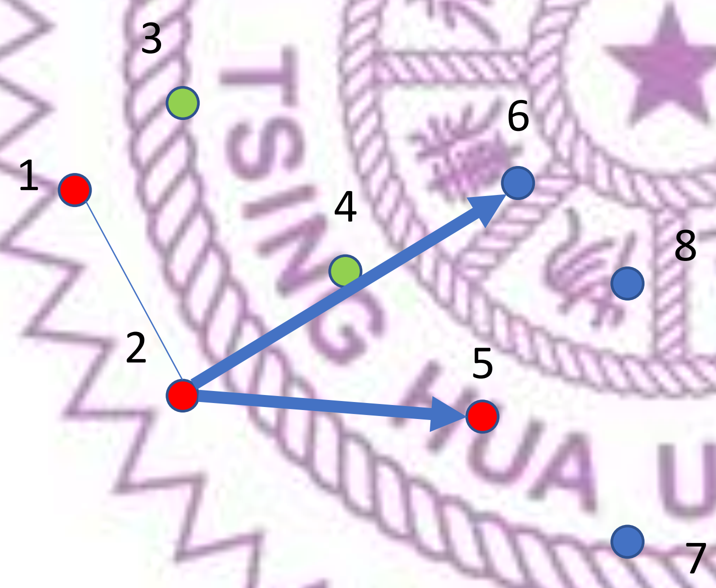
找出下側

- $\overrightarrow{12} \times \overrightarrow{15} > 0$不會造成凹陷
- 將第5個點加入凸包



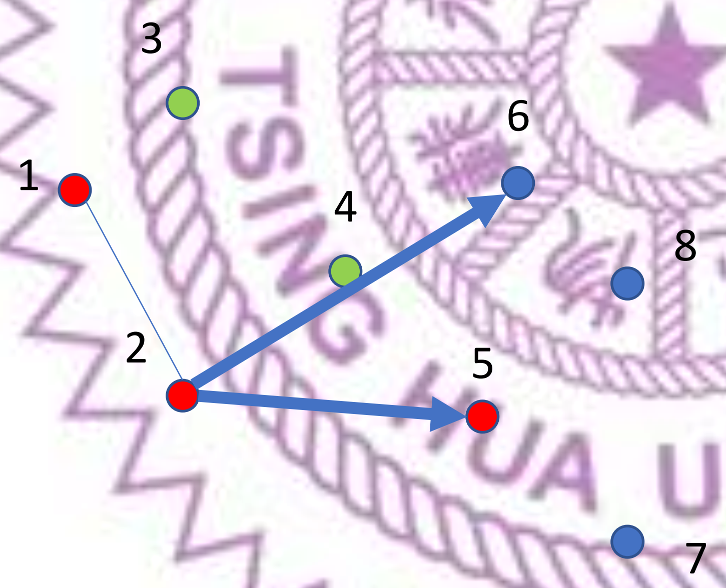
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



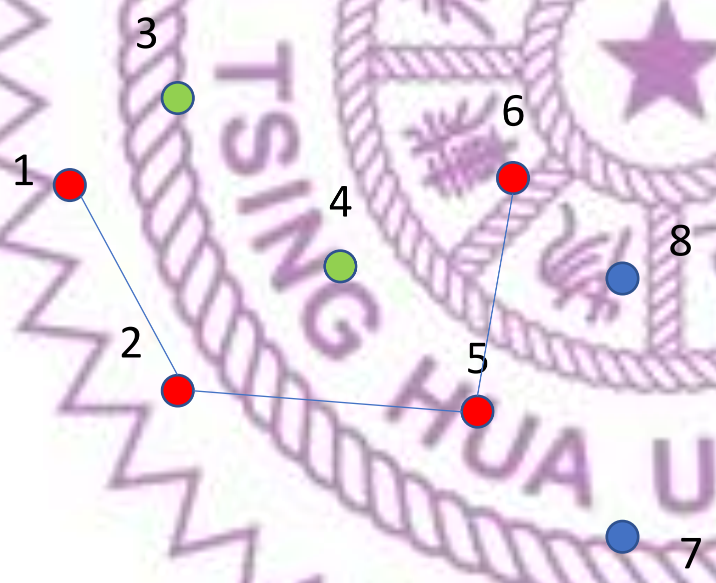
找出下側

- $\overrightarrow{25} \times \overrightarrow{26} > 0$不會造成凹陷
- 將第6個點加入凸包



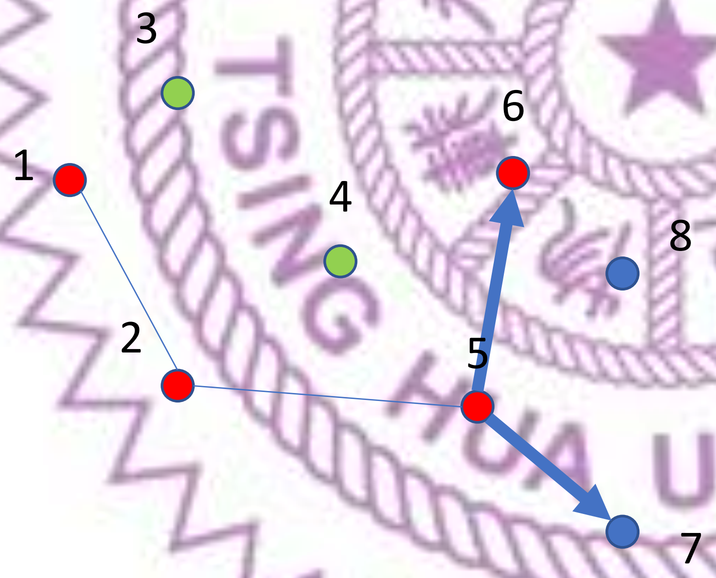
找出下側

- 將第6個點加入凸包



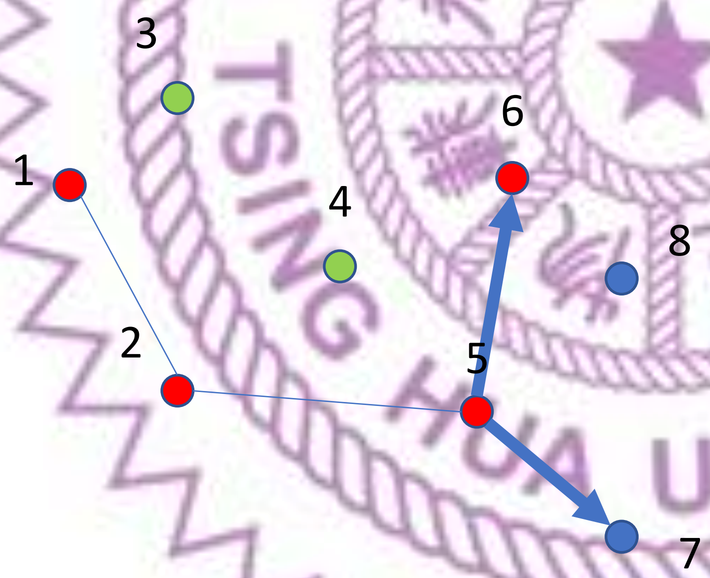
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



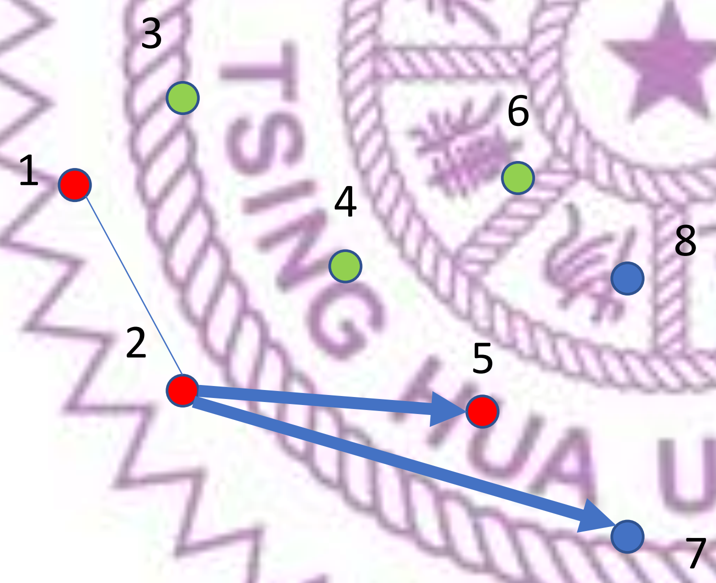
找出下側

- $\overrightarrow{56} \times \overrightarrow{57} \leq 0$會造成凹陷
- 因此要把第6個點移除



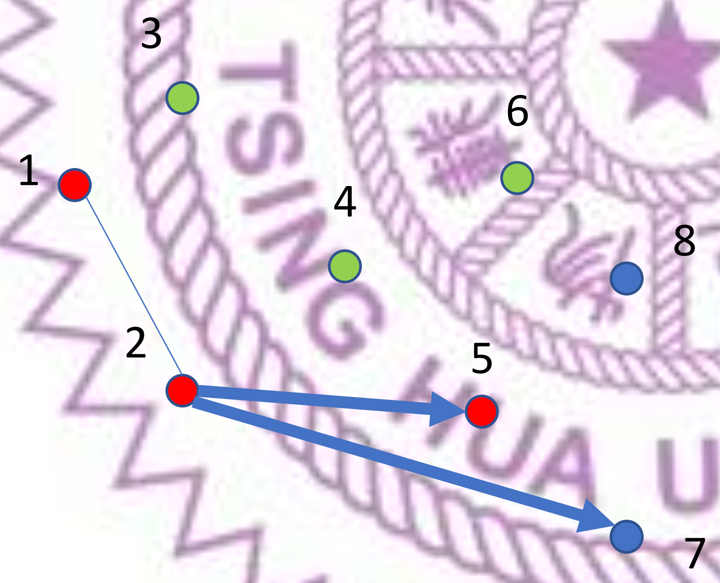
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



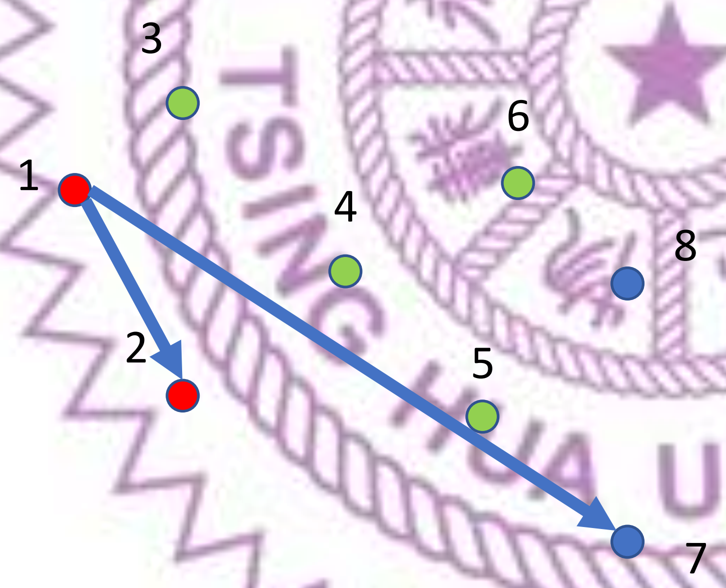
找出下側

- $\overrightarrow{25} \times \overrightarrow{27} \leq 0$會造成凹陷
- 因此要把第5個點移除



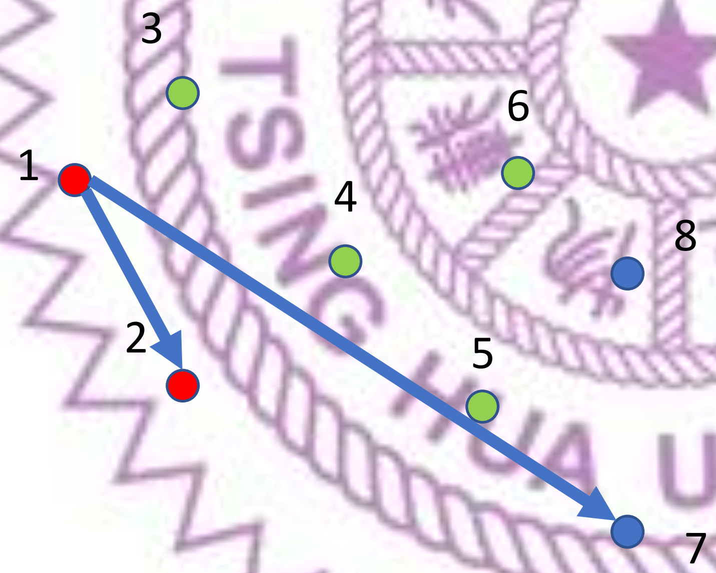
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



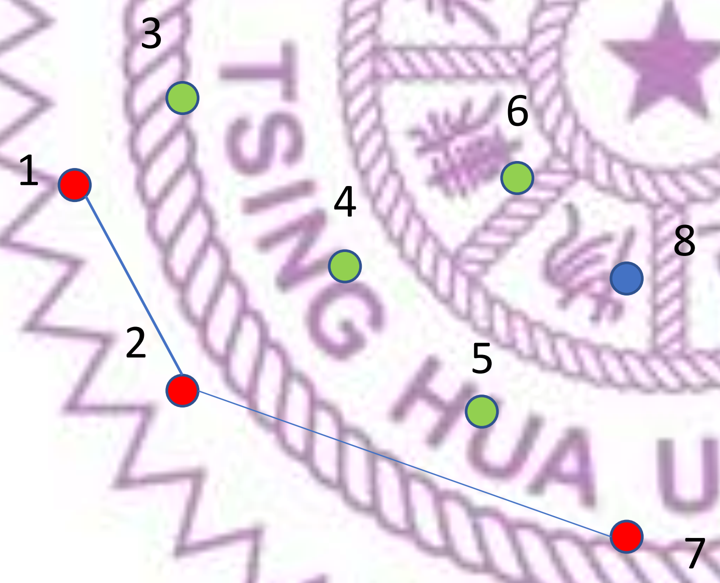
找出下側

- $\overrightarrow{12} \times \overrightarrow{17} > 0$不會造成凹陷
- 將第7個點加入凸包



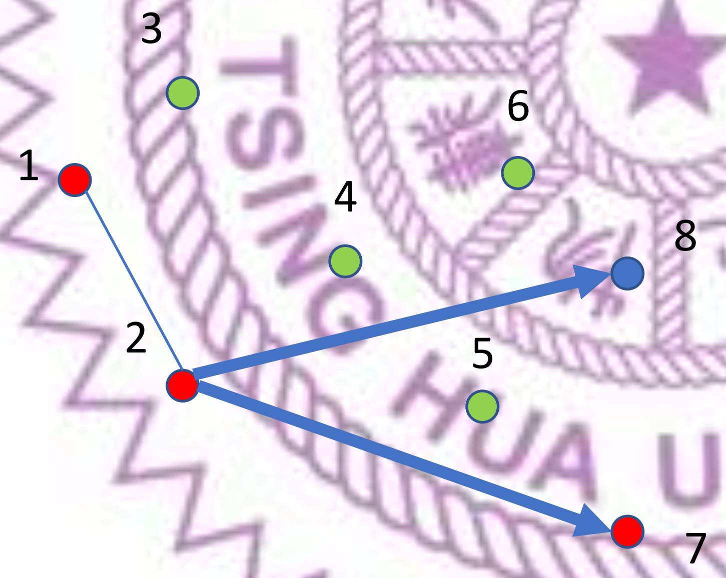
找出下側

- 將第7個點加入凸包



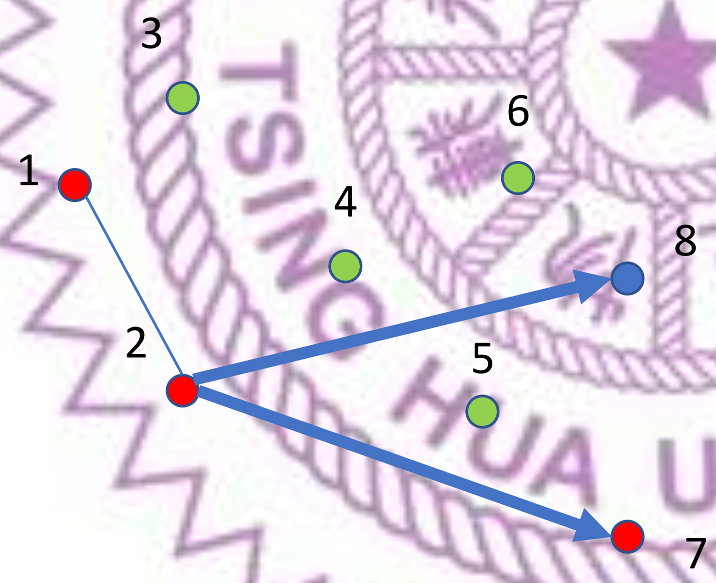
找出下側

- 因為已經有超過兩個點了
- 加入下個點之前要判斷會不會造成凹陷



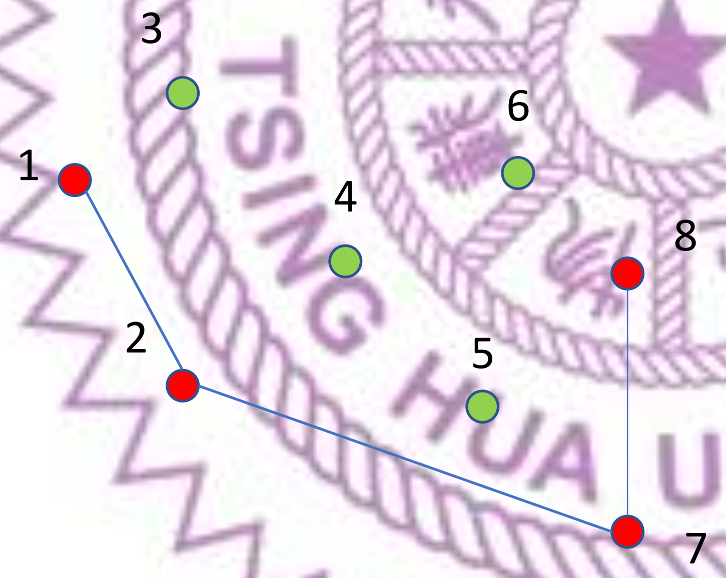
找出下側

- $\overrightarrow{27} \times \overrightarrow{28} > 0$不會造成凹陷
- 將第8個點加入凸包



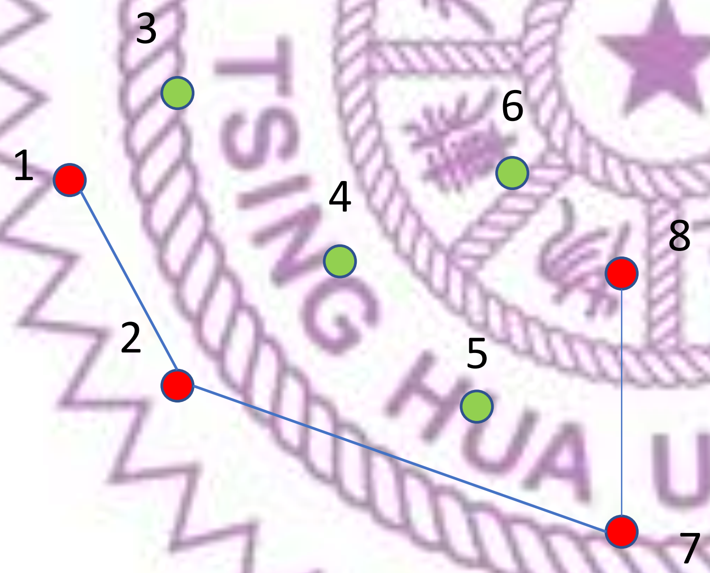
找出下側

- 將第8個點加入凸包



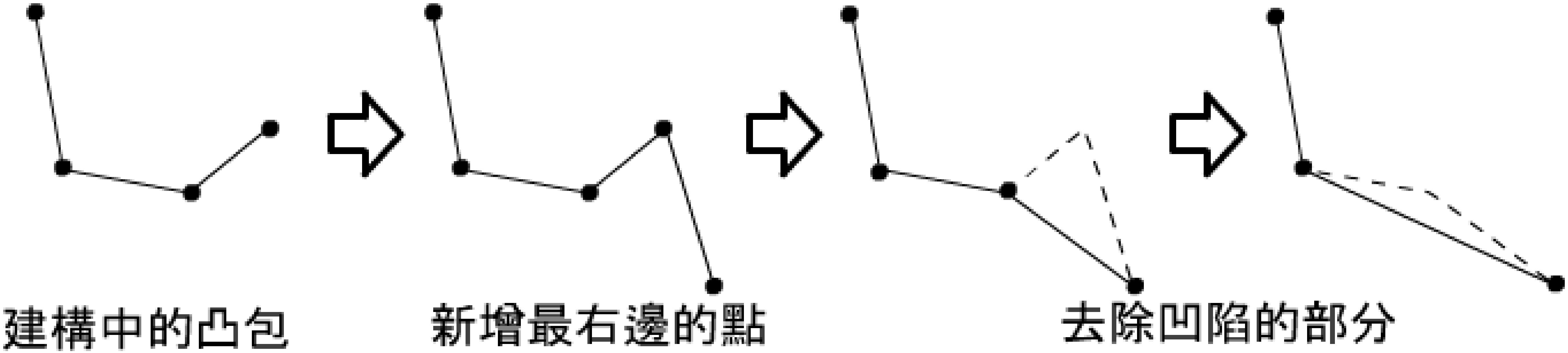
找出下側

- 下側凸包完成
- 上側唯一不同之處是從右邊做到左邊而已



重點步驟

凹陷的部分要去掉！



Andrew‘s Monotone Chain 重點程式碼

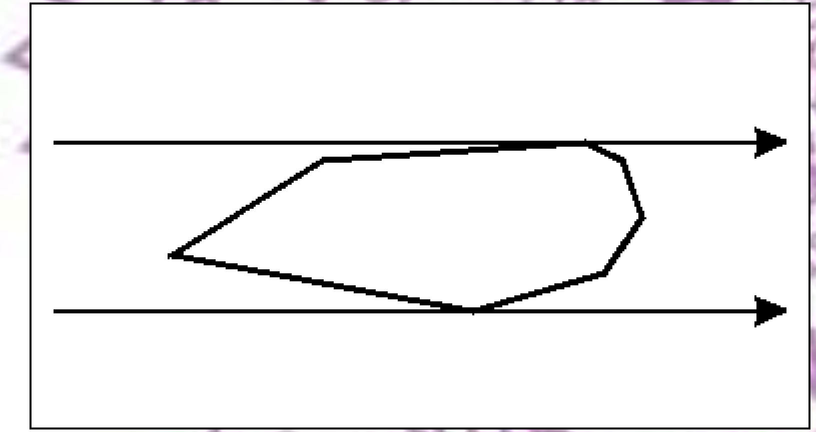
```cpp
vector<PT> getConvexHull(vector<PT> Points) {
  sort(Points.begin(), Points.end(), x_cmp);
  vector<PT> ans;
  int m = 0, t = 1;
  auto addPoint = [&](const PT &p) {
    while (m > t && (ans[m - 1] - ans[m - 2]).cross(p - ans[m - 2]) <= 0)
      ans.pop_back(), --m;
    ans.emplace_back(p);
    ++m;
  };
  for (size_t i = 0; i < Points.size(); ++i) addPoint(Points[i]);
  t = m;
  for (int i = int(Points.size()) - 2; i >= 0; --i) addPoint(Points[i]);
  if (Points.size() > 1) ans.pop_back();
  return ans;
}
```

旋轉卡尺

Rotating Calipers

旋轉卡尺

- 基本想法就是用兩條平行線卡住凸包然後繞一圈旋轉
- 程式要怎麼寫呢？

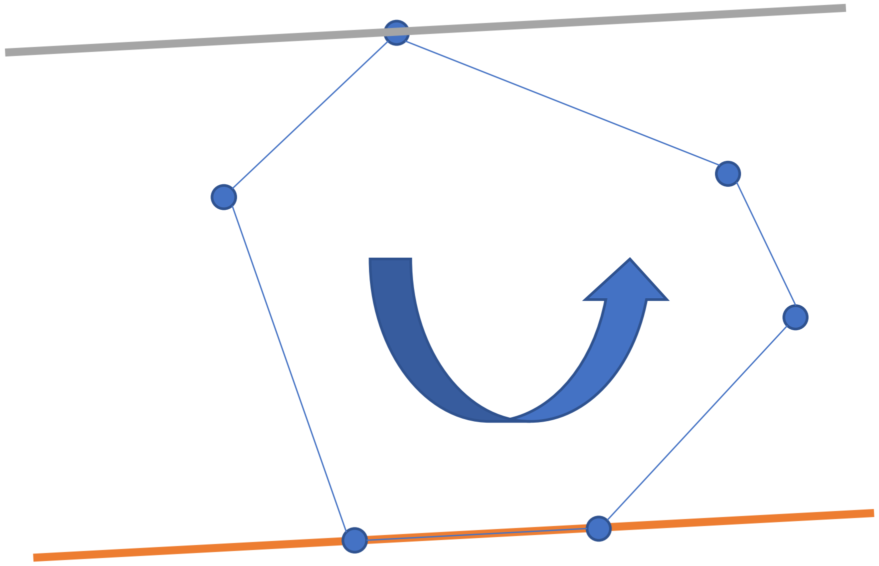


圖片 來自於網路

如何旋轉?

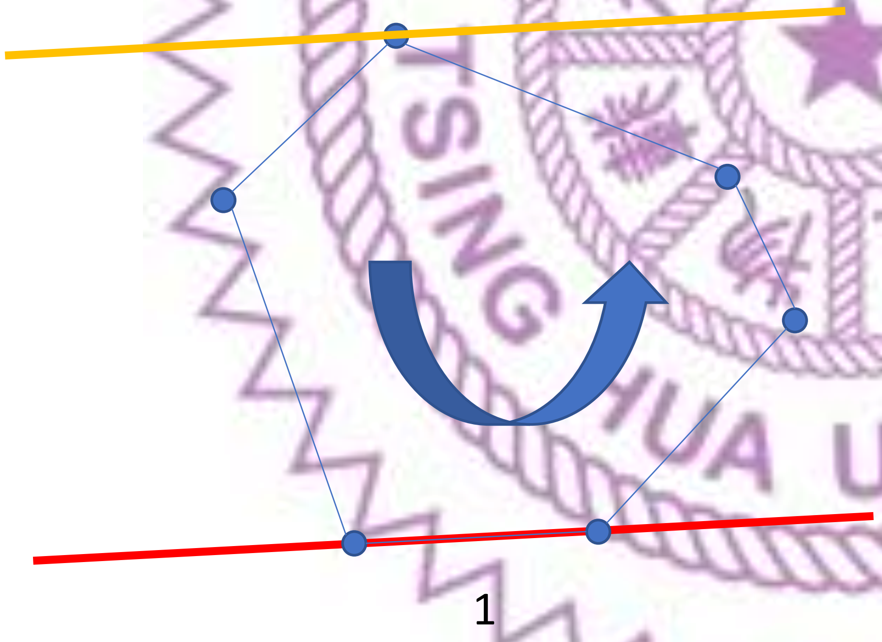
讓其中一條直線貼在凸包的邊上

另外一條直線則在距離該邊最遠的點上



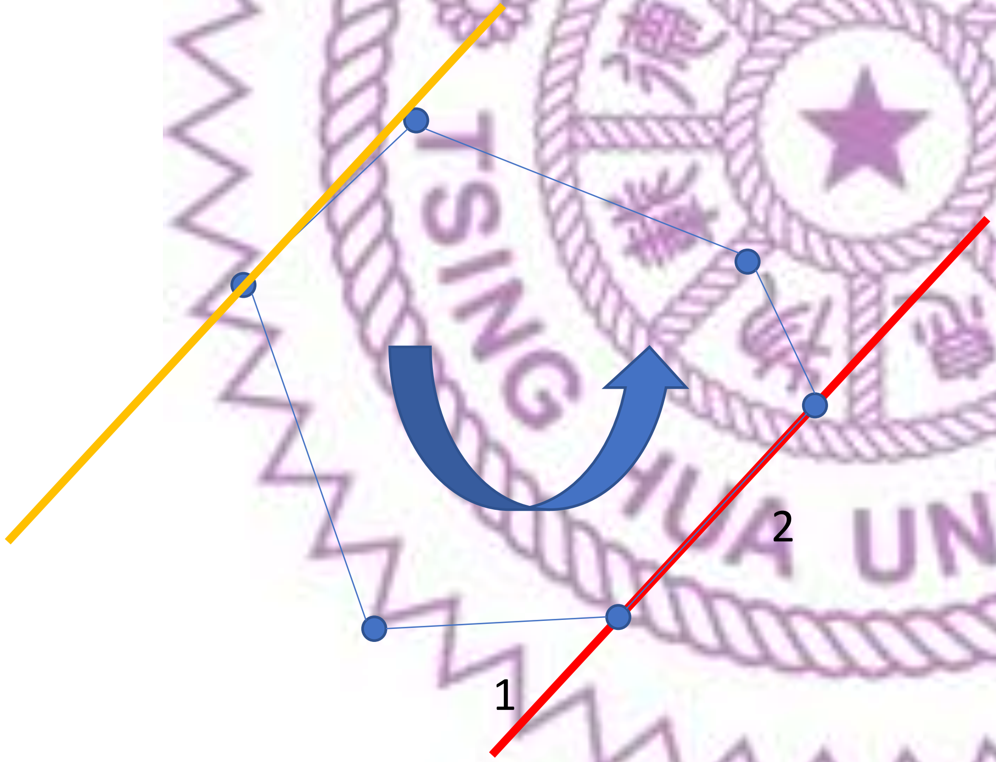
如何旋轉?

- 只要按逆時針順序枚舉所有邊
  對每條邊找出距離最遠的點就可以達到旋轉卡殼的目的了



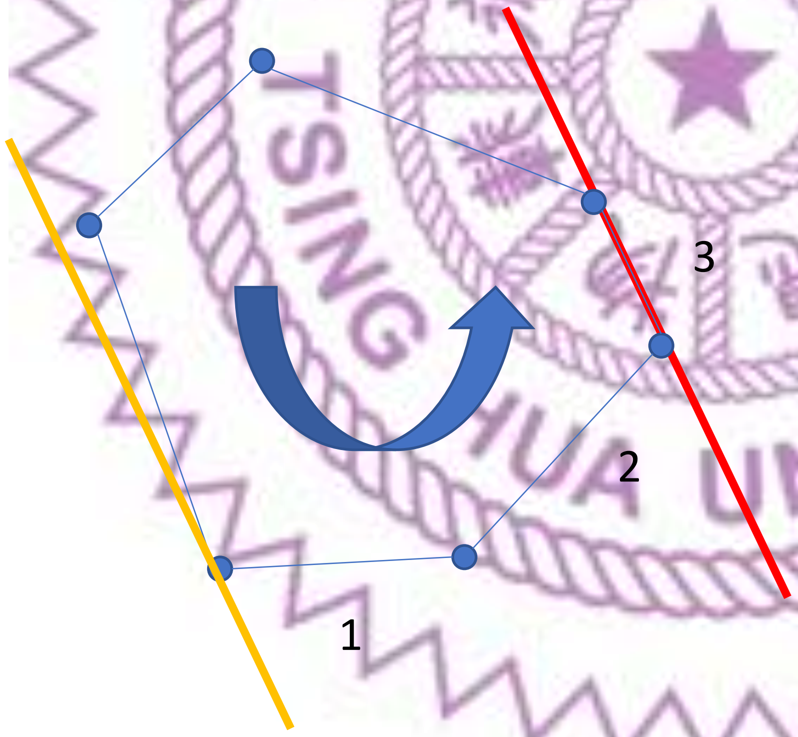
如何旋轉?

- 只要按逆時針順序枚舉所有邊
  對每條邊找出距離最遠的點就可以達到旋轉卡殼的目的了



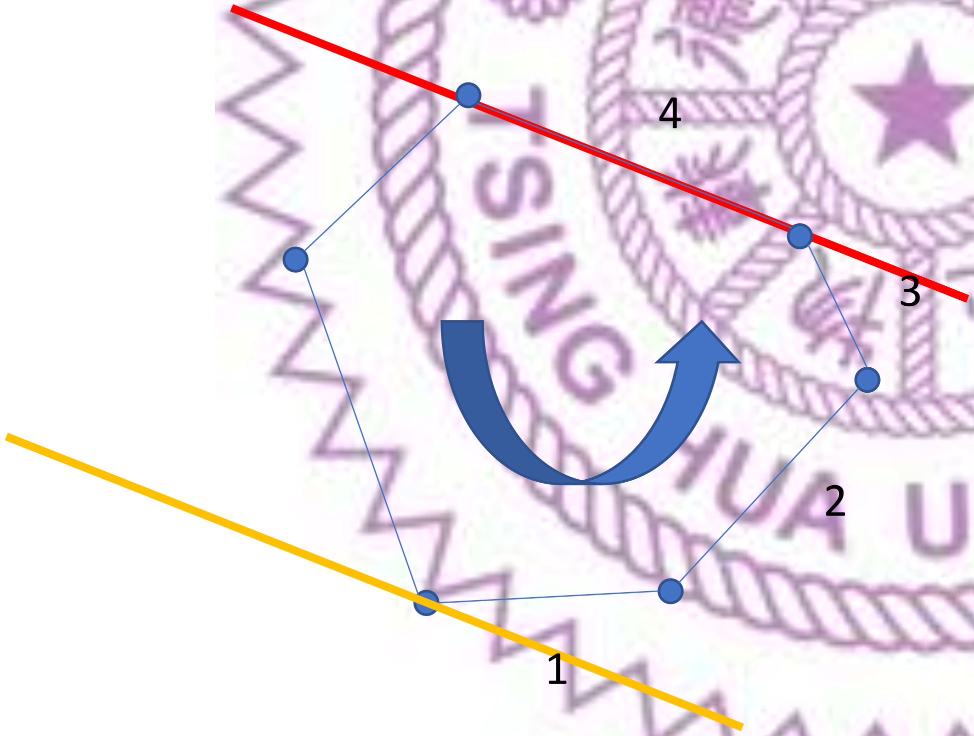
如何旋轉?

- 只要按逆時針順序枚舉所有邊
  對每條邊找出距離最遠的點就可以達到旋轉卡殼的目的了



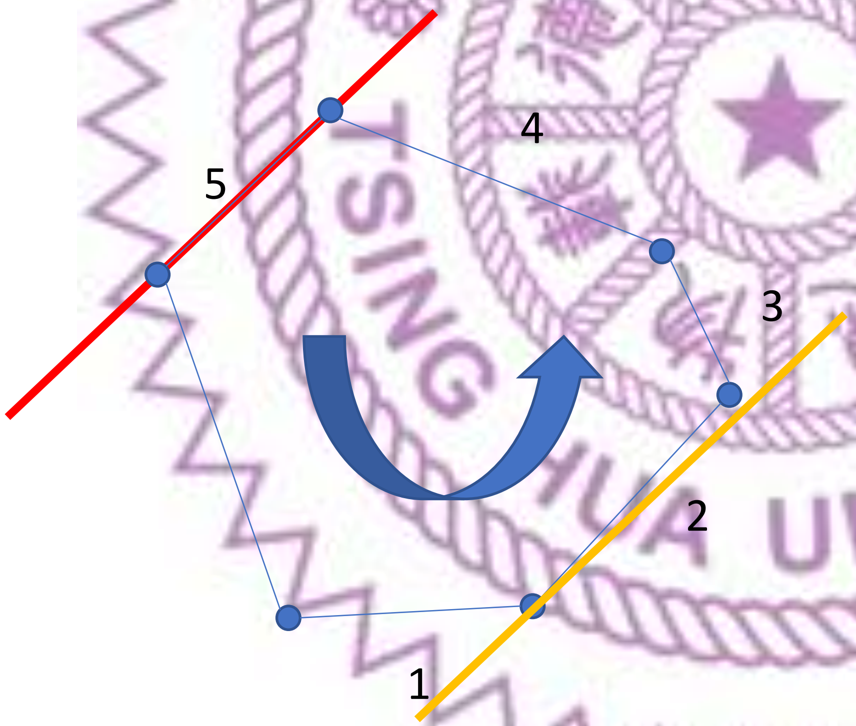
如何旋轉?

- 只要按逆時針順序枚舉所有邊
  對每條邊找出距離最遠的點就可以達到旋轉卡殼的目的了



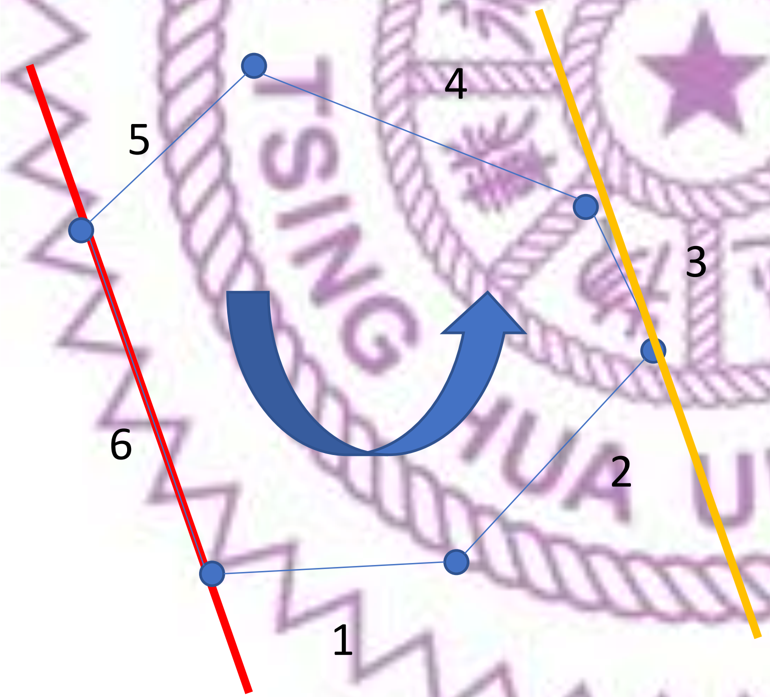
如何旋轉?

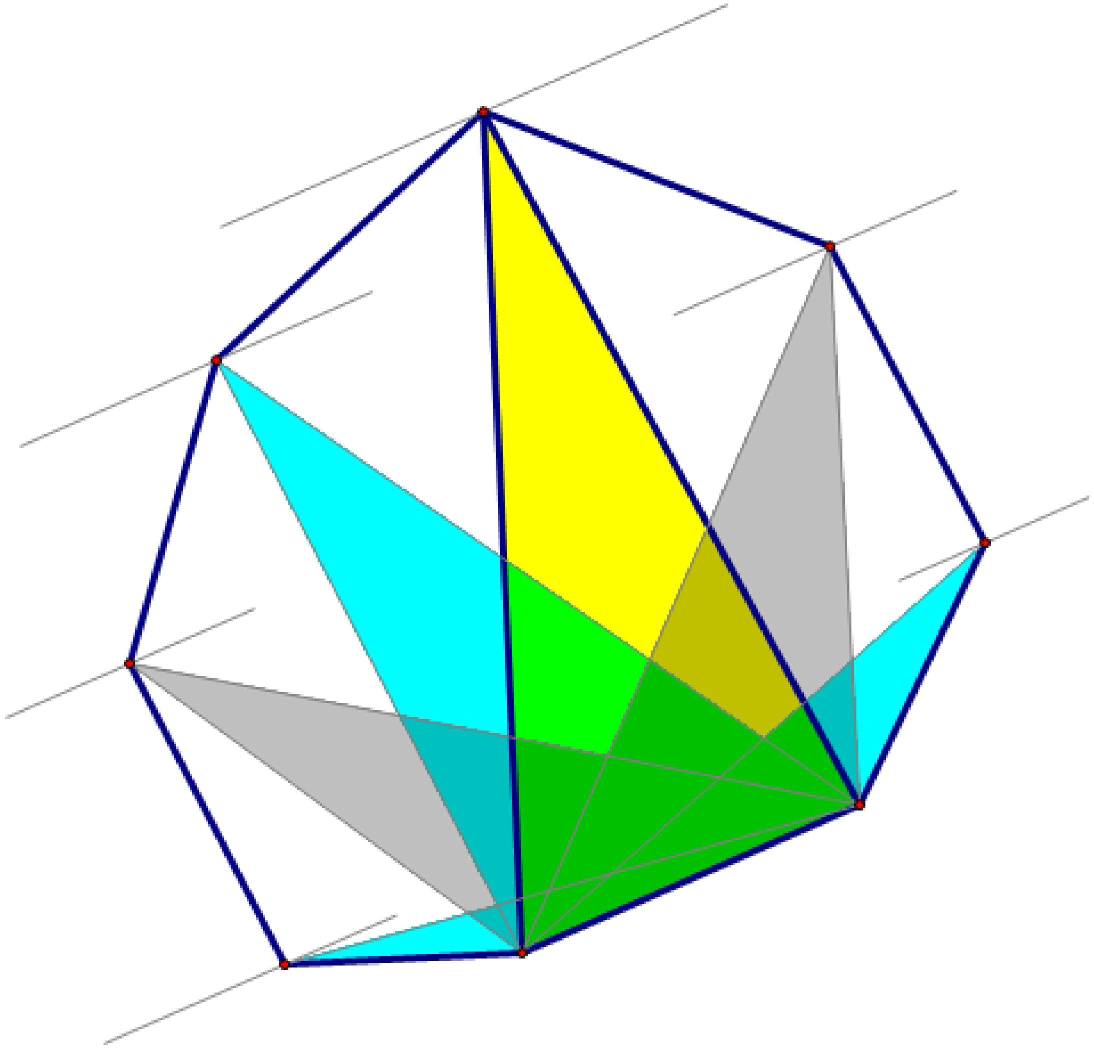
- 只要按逆時針順序枚舉所有邊
  對每條邊找出距離最遠的點就可以達到旋轉卡殼的目的了



如何旋轉?

- 只要按逆時針順序枚舉所有邊
  對每條邊找出距離最遠的點就可以達到旋轉卡殼的目的了





找出距離邊最遠的點？

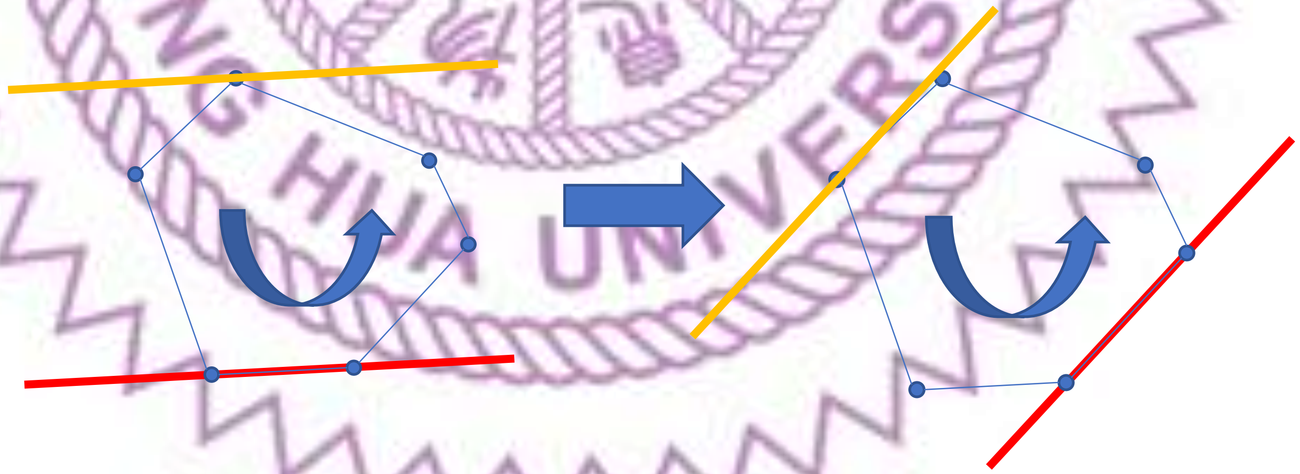
- 可以發現距離邊最遠的點和邊構成的三角形面積會是最大的
- 可以用cross輕鬆找出來

觀察旋轉過程

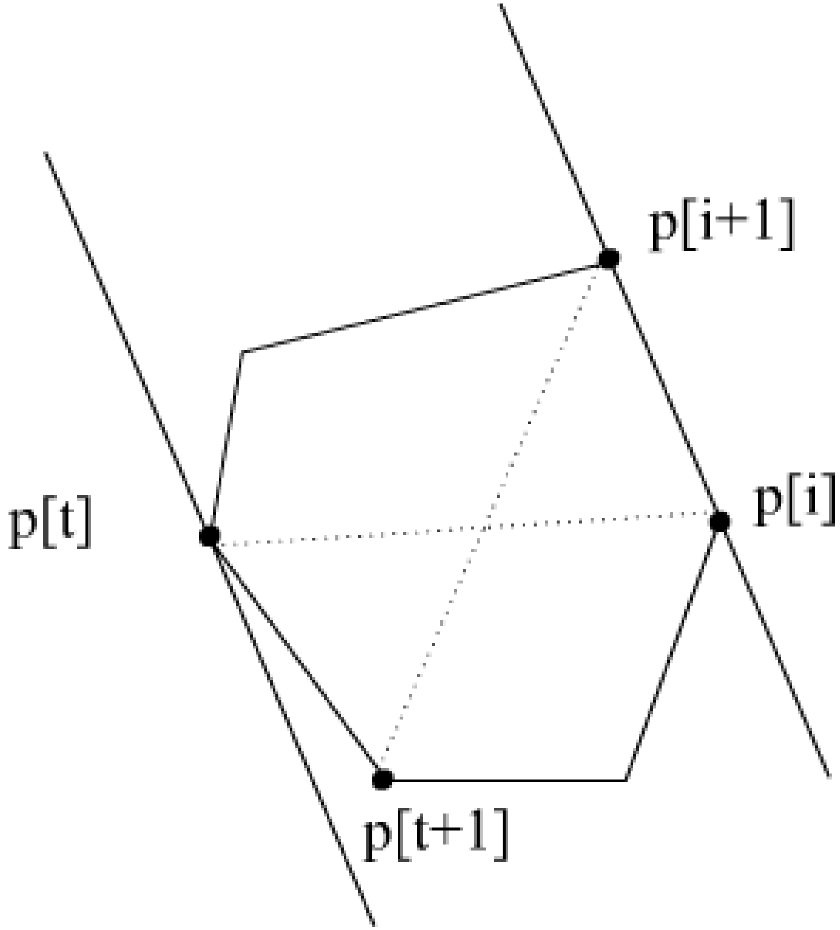
繞一圈只要 $O(n)$

- 由於枚舉邊的順序是逆時針旋轉的
  所以距離邊最遠的點也會跟著逆時針旋轉

- 因此只要從上一條邊的最遠點開始
  往逆時針的方向找就可以找到下一條邊的最遠點



最遠點對 $O(n)$ 作法



- 逆時針順序枚舉每一條邊
- 每次枚舉從上一條邊的最遠點開始，往逆時針方向找最遠點
- 計算最遠點和直線上兩點的距離，找出最大的就是答案

最遠點對 $O(n)$

```
double rotatingClaiper(vector<PT> ConvexHull) { // 計算最遠點對距離的平方
  int n = ConvexHull.size(), t = 1;
  double ans = 0;
  ConvexHull.emplace_back(ConvexHull[0]); // 起點放在後面可以省略特判
  for (int i = 0; i < n; i++) {
    PT now = ConvexHull[i + 1] - ConvexHull[i]; // 當前這條線的方向向量
    // 找出距離邊 (i,i+1) 最遠的點 t
    while (now.cross(ConvexHull[t + 1] - ConvexHull[i]) >
           now.cross(ConvexHull[t] - ConvexHull[i]))
      t = (t + 1) % n;
    ans = max(ans, max((ConvexHull[i] - ConvexHull[t]).abs2(),
                       (ConvexHull[i + 1] - ConvexHull[t + 1]).abs2()));
    // 其實可以簡化成： ans = max(ans, (ConvexHull[i]-ConvexHull[t]).abs2() );
    // 想一想為什麼?!
  }
  return ans;
}
```

可以用旋轉卡尺概念加速的東西

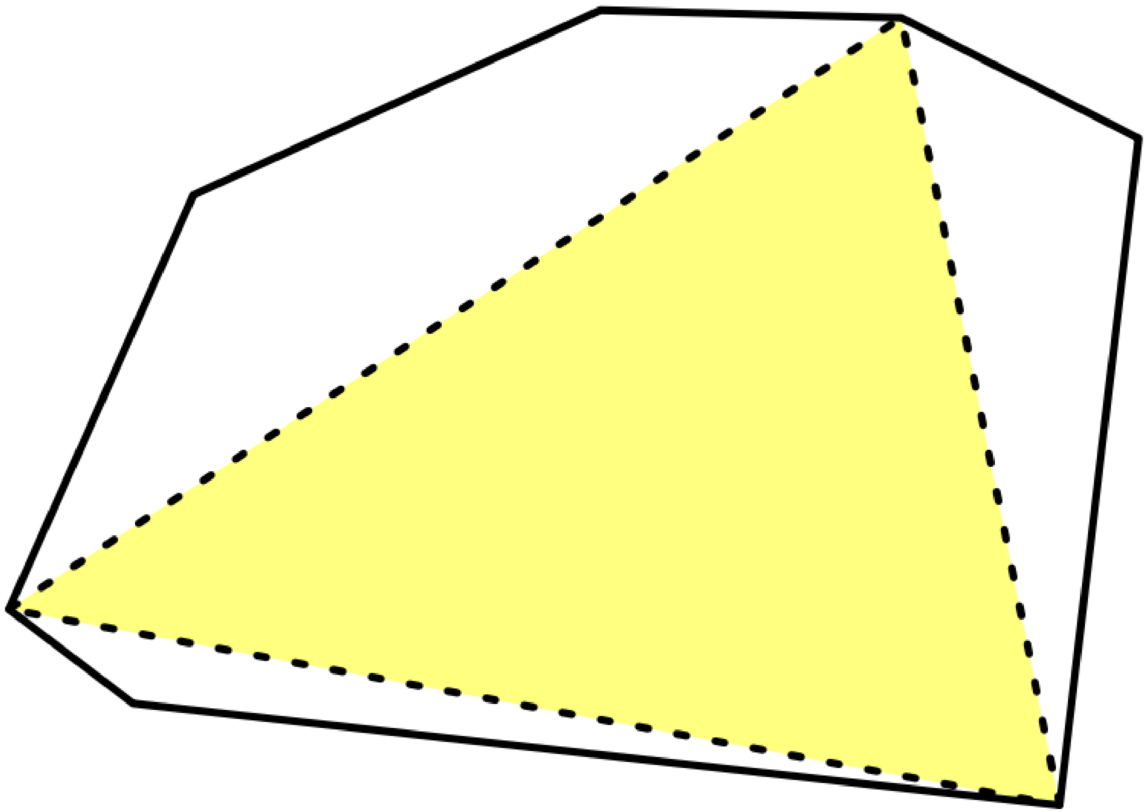
- 最遠點對距離 (直徑)
- 最小覆蓋矩形
- 兩凸包最近距離
- 凸包內接最大 K 邊形

凸多邊形
最大內接三角形

- [1705.11035] Maximum-Area Triangle in a Convex Polygon, Revisited

- 根據該篇 Paper ，目前大陸網站上的 $O(n)$ 旋轉卡殼算法都是錯的
甚至有附會出錯的測資

- 至少要會 $O(n^2)$ 的方法



为什么常见的 $O(n)$ 时间的找凸多边形内最大内接三角形的那个算法是错误的? - 知乎

凸多邊形
最大內接四邊形

- [1708.00681] Maximum-Area Quadrilateral in a Convex Polygon, Revisited

- 同理最大內接四邊形也不能 $O(n)$