Time complexity/ Space Complexity

在學習算法時，Time complexity 與 Space complexity 是兩個非常重要的概念，它們分別用來衡量一個演算法在輸入規模變大時，所需的執行時間與記憶體空間的成長幅度。

Time Complexity

以 time complexity 為例，一個 time complexity 為$O(n)$ 的算法，當輸入長度從k增加到2k時，執行時間大約也會增加為原來的兩倍。，相對地，如果一個算法 time complexity 為$O(n^2)$，當輸入長度從k增加到2k時，執行時間大約也會增加為原來的四倍。。因此縱使兩個同樣 time complexity 為 $O(n)$ 的算法，它們實際執行速度仍可能有所差異，只是他們成長幅度相同，因為time complexity只描述成長趨勢，並未考慮一些實作細節的常數因子。
下面舉幾個例子

`for(int i = 0 ; i < n ; i++) {`
`    sum += i;`
`}`
	`for(int i = 0 ; i < n ; i++) {`
	`    sum += i;`
	`}`
	`for(int i = 0 ; i < n ; i++) {`
	`    sum += i;`
	`}`
	`for(int i = 0 ; i < n ; i++) {`
	`    sum += i;`
	`}`

上面兩隻程式很明顯的右邊計算量為左邊三倍，因此在n足夠大時，時間上也約為左邊的三倍，但他們的time complexity 皆為O(n)

```
for(int i = 0 ; i < n ; i++) {
    for(int j = 0 ; j < n ; j++) {
        sum += i+j;
    }
}
```

而上方則為一個time complexity則為 O(n^2)的範例程式，可以發現當輸入 n 加倍時，程式所需執行時間會成長為原來的4倍

我們在學習演算法過程中，經常會遇到將資料/範圍分成兩塊或是更多塊處理，並持續做這個操作直到資料/範圍變成1為止，考慮一個簡單的簡單的情況為將資料分成兩塊，假設輸入長度為 n，每次操作後他長度都會剩 ½，問幾次操作後他長度會變成 1，也就是 n 要除幾次2 之後會變成 <=1 的數字，從下列式子中我們可以推導出這類演算法所需的操作次數 k:

$n/2^k = 1$
-> $n = 2^k$(等號兩邊乘以2^k)
-> $logn \ = \ k\, log\, 2$ (兩邊同時取log)

而 log2 是常數在計算複雜度的時候可以忽略掉，因此就可以得到，如果有演算法用到相同概念，他的複雜度就會是$O(logn)$。常見的例子包含我們後面會提到的 binary search。

Space Complexity

space complexity主要分為兩部分

1. 輸入的 space complexity
2. 一撇除輸入外的 space complexity

譬如說題目為輸入一個 array 請你輸出他的和

1. 輸入的space complexity為O(n) (array長度為 n)
2. 而撇除輸入外的space complexity，只需要宣告一個變數累加總和，因此額外的space complexity就是O(1)

在提出解決方案中除了討論 time complexity，討論 space complexity 並清楚區分這兩部分也非常重要。

○