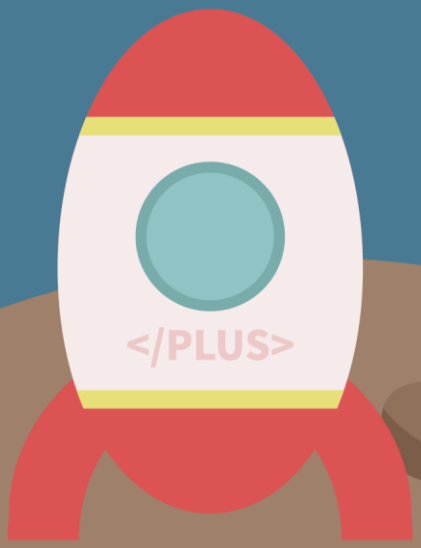
18 演算法



前言

本單元簡單介紹何為演算法、時間複雜度與搜尋排序。

18.1 何為演算法

什麼是演算法呢？指的是由一個一個步驟組合而成的解決問題的方式。

例如我們想要查字典之中的「Algorithm」，有幾種不同的方式可以找到這個字。例如我們從頭一頁一頁的往後翻，直到出現「Algorithm」的那頁為止。但我們通常不會這樣查字典，因為字典本身是有經過排序的，例如英文字典就是按照 A、B、C 這樣下去排列。所以我們隨意地翻開其中一頁，去比較我們要找的字的字首是在這頁的之前還是之後，若是在之後，我們就再度隨意翻開後面的某一頁，去重複這樣的步驟，直到找到目標為止。

以上兩種查字典的方式，都是為了找到「Algorithm」這個字為目標而產生的不同的處理方式。

18.1.1 演算法

在電腦程式中，問題相當於 Input（輸入），經過演算法後我們可以得到 Output（輸出）。

使用電腦語言編寫程式、執行

而我們必須將演算法改寫成電腦看得懂的電腦語言，這樣才能執行喔！因此我們可以得知演算法跟最後實作出的程式、電腦語言是有關聯但不同的東西。而解決的方式有千千百百種，有些好、有些壞，演算法也是如此，好的演算法易於瞭解、編寫程式，也易於運行。

要評估演算法的好壞有兩個面向，第一個是「空間」，好的演算法可能在記憶體中非常節省，第二個是「時間」，也就是演算法的效率問題。一般而言，我

們比較關注在演算法的執行時間，因為在生活上需要運行的程式中，常常都是要處理非常龐大的資料。當輸入的資料量龐大時，效率就非常的重要喔！

18.1.2 時間複雜度

剛剛提到的效率，指的是每一個演算法都有自己的「時間複雜度」。這邊會先簡單的講解時間複雜度，後面第二節就會再詳細介紹。

而若要衡量效率，我們有什麼辦法可以計算程式的運作時間呢？

首先，最直觀的就是在程式內加入計時器，如下圖：

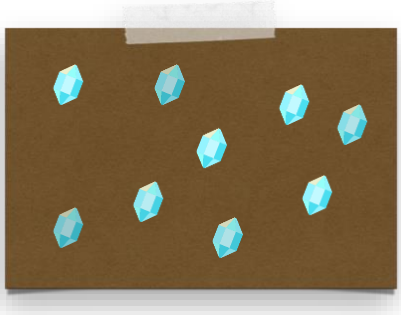
```
import time

def c_to_f(c):
    return c*9/5 + 32

t0 = time.clock()
c_to_f(100000)
t1 = time.clock() - t0
Print("t =", t, ":", t1, "s,”)
```

「timer」是加入計時器功能，「t0」為記錄起始時間，接著運作程式，最後「t1」則是現在時間減去起始時間後獲得的時間差，這樣就能知道運作時間有多長。但這運作時間與我們的電腦設備、資料輸入量有關，同樣的程式碼在不同設備的電腦下，執行的時間也不同；且資料量若很小的話，時間差也會小到幾乎沒有差異。另外這種紀錄方式，一次只能記錄一種 Input 的時間，若要測試不同的 Input 時間，就必須要讓程式執行很多次。

因此我們要來想想看是否有其他的衡量方法呢？若我們先撇除硬體設備的干擾，也就是先不看實際運作時間，那就是從程式碼下手了。

第二個方法我們去數程式內的步驟數，假設每一步驟都耗費一個時間單位，那步驟數越多，就代表耗費時間越長。

以下圖為例，我們該怎麼數圖片中的寶石呢？



首先我們用最常見的方法，也就是「數數」的方式。例如我們一顆一顆的數。

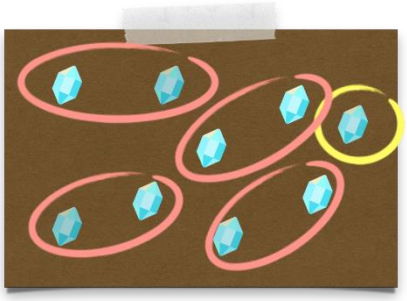
起始（ n ）為 0，當數到一顆時，就加 1 存回 n 內。

從零開始 n=0

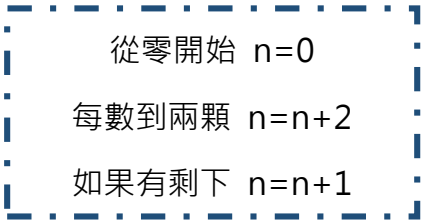
每數到一顆 n=n+1

最後出來的 n 就會是數量了。以上圖的方式數數的話，共執行了 9 次，也就是 9 個步驟。

若是兩顆兩顆數呢？



起始（ n ）為 0，當數到兩顆時，就加 2 存回 n 內。如果有剩餘就再加 1。



最後共跑了 5 個步驟。

由此可知，若圖片上的寶石數量更多的時候，越多個一起數，節省的步驟數越多，也就代表花的時間較少，效率變高了。但是別忘了，我們要比較的是演算法的效率，而不是實際作成程式後的效率喔！

我們需要比較的其實是在「數數」這個演算法的層面，因此算步驟的方向對了，因為步驟數越多，執行的時間也越長。但是我們必須在演算法的時候就要做比較。

18.1.3 時間複雜度的表示法

首先我們要評估的是演算法本身的時間複雜度，無關執行的電腦設備，也無關最後寫出的程式。以及必須瞭解當資料量變大時，演算法的表現、趨勢如何。這樣才能比較各種演算法在面對真正龐大問題時，哪一種才有比較好的解決效率。

這邊主要介紹 Big O 表示法。它是用來評估輸入值為 n 時，忽略不重要的部分後，執行時間在最差的狀況下的時間成長趨勢。而影響演算法表現的因素有那些呢？

分別是「輸入資料的狀態」與「輸入資料的大小」。首先先講解「輸入資料的狀態」，如下圖：

```
def search_for_elmt(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

如圖片上要在名為 L 的 List（串列）中搜尋 e 這個元素。使用的方式是利用 for 迴圈去一個一個比對 L 內的項目 i 是否等於 e。而最佳、平均及最差狀況分別會是什麼情形呢？

最佳狀況：e 剛好排在 L 中第一位
平均狀況：e 排在 L 的中央
最差狀況：e 沒有在 L 中

如果 L 內有 100 個元素好了，那迴圈就有可能運作 100 次都找不到 e 這個元素，這就是程式運作可能遇到的最差情況。所以相同大小的不同輸入值，可能造成演算法的執行時間不同。

通常要比較效率的時候，就要採用這種最差的情況來做衡量。

而另一種影響演算法的因素「輸入資料的大小」。剛剛提過 L 內有說少元素，最差情況就要運作多少次，從上圖內的程式來看，迴圈是影響時間最重要的一個環節，return 的步驟雖然也會花費時間，但卻沒有迴圈來的多。

也就是說，若有個 List 有數萬個元素，這樣裡面的迴圈運作的次數在最差狀況也會運作數萬次，那其他零碎的小步驟所花費的時間就相對小到我們能忽略不計。

因此，一個迴圈我們記錄為一個「n」，而這個演算法的時間複雜度我們便以「O(n)」表示。

實際上的操作會呈現怎麼樣呢？若得到輸入值 n 跟時間關係的多項式後，我們可以直接省略首項係數以及低階項，只留下影響最大的最高階項，就是我們時間複雜度 Big O 囉！

1. $n^2 + 2n + 2$ $O(n^2)$
2. $n^2 + 100000n + 3^{1000}$ $O(n^2)$
3. $\log(n) + n + 4$ $O(n)$
4. $0.0001*n*\log(n) + 300n$ $O(n \log n)$
5. $2n^{30} + 3^n$ $O(3^n)$

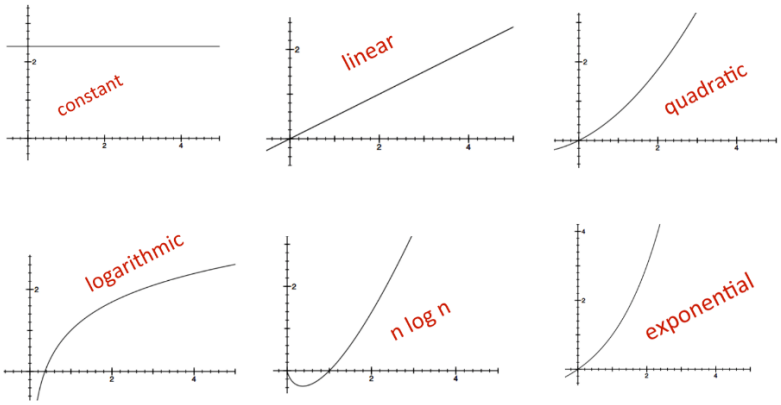
另外下表為常見的時間複雜度，接著就能比較時間複雜度，哪一種是比較理想的演算法。

Name	Time Complexity
常數時間	O(1)
對數時間	O(log n)
線性時間	O(n)
線性乘以對數時間	O(n log n)
次方時間	O(n^c)
指數時間	O(c^n)

下表為代入數字後的實際計算，不同的時間複雜度在 n 成長時，時間的成長比例為多少。

CLASS	n=10	= 100	= 1000	= 1000000
O(1)	1	1	1	1
O(log n)	1	2	3	6
O(n)	10	100	1000	1000000
O(n log n)	10	200	3000	6000000
O(n^2)	100	10000	1000000	1000000000000
O(2^n)	1024	1267650600228229401496703205376	1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695858127594672917553146825187145285692314043598457757469857480393456777482423098542107460506237114187795418215304647498358194126739876755916554394607706291457119647768654216766042983165262438683720566806937 6	Good luck!!

如第一個時間成長與 n 並沒有關係，無論輸入多少，運算時間都是相同的。而最後一個則是 n 在次方，也就是以指數關係成長的狀態時，當 n = 1000，花費時間便已經爆炸成一個天文數字了，可見這非常不適合處理龐大的資料。若將上表畫成趨勢圖該怎麼表示呢？



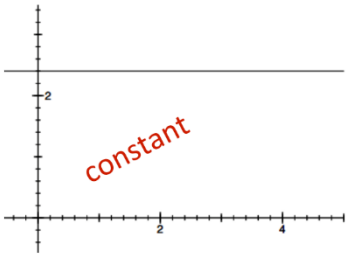
橫軸為資料量大小，縱軸可當作執行步驟，也就是代表耗費時間量的比例。根據圖片可以看到有些是穩定時間增加，有些則是在一定的量之後暴增。其中左下 log 的時間最後是趨緩的，這樣就能應付龐大的資料了。

18.2 時間複雜度

本節會更詳細的介紹一些常見的時間複雜度，帶大家一一的看下去。

18.2.1 常數型時間複雜度（Constant Complexity）

常數型時間複雜度，也就是執行時間不會隨著輸入的資料量變大而變長。



下圖為一個計算 List 中首項平方的一個程式，輸入 Items 可得到「Items 的第一（零）項乘以 Items 的第一（零）項」的結果。

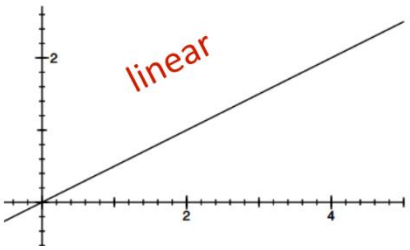
```
def constant_algo( items ):
    result = items[0] * items[0]
    print ( res )
```

所以永遠都是相乘，然後 print，這樣固定的兩個步驟，也就永遠花同一常數的時間。雖然常數時間很好，無論資料量多大都可以在相同的時間內算完，

但在實際生活中所實用的演算法，很少有單純常數時間這麼簡單就能解決的事情。因此這通常指是被包含在複雜演算法裡面的一小部分而已。

18.2.2 線性時間複雜度（Linear Complexity）

線性時間複雜度，也就是運算時間會隨著資料量增加而等比例上升，呈現正比狀態。



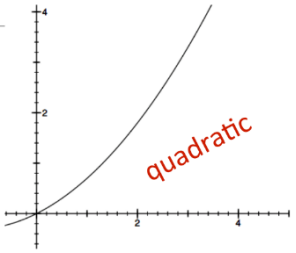
下圖為我們之前提過的在 List 中搜尋 e 元素的例子，這一個迴圈便是在一個一個比對 i 是否何 e 吻合，而最差的情況就是要全數比對，因此這一個 for 迴圈的時間複雜度就是線性的 O(n)。

```
def linear_search(L,e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

n

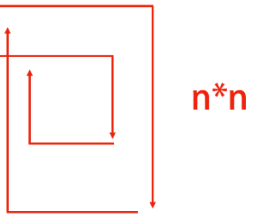
18.2.3 平方時間複雜度（Quadratic Complexity）

平方時間複雜度，同樣也是運算時間會隨著資料量增加而上升，但看下方圖片可清楚分辨和線性時間複雜度不同的上升方式。



從例子來看會發現它是多層迴圈的結構，也就是迴圈中還有迴圈，稱之為巢狀結構（巢狀迴圈），也因此得到 n 平方這樣的時間複雜度。

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

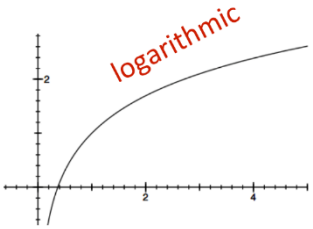


有 for 迴圈很好算，但這種情況我們實際要確認的是在 worst case（最差狀況）中是否 n 真的必須要跑完一次後再跑一次，若「是」則時間複雜度需兩個 n 相乘，以此類推而得到一個平方時間複雜度。

重新再看一次例子，這是在比對 L1 和 L2 是否有相同元素，最差狀況就是兩個 List 等長且沒有相同元素，因此要運作的次數就是兩者長度相乘，也就得到我們的時間複雜度 O(n^2)。

18.2.4 對數時間複雜度（Logarithmic Complexity）

對數時間複雜度，每增加一個輸入，所需要的額外時間會逐漸變小，這是很有效的一種演算法。



下方的例子是將一個輸入的數字轉換成字串，也就是用程式碼中的 digits，零到九，來拼出 i 這個數字。

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    res = ''
    while i > 0:
        res = digits[i%10] + res
        i = i//10
    return result
```

例如我們輸入「10」，就希望得到「1、0」，要怎麼做到呢？若輸入值 i 為零的話，就會 return 一個零的數字。而其他狀況呢？

如果 i 大於零就要進入 while 迴圈內，下一行的 Result（res）內原本是空的，而到這行每次都要用 i 除以十的餘數，去尋找對應的 digits，接著將 i 除以十，若大於零就繼續迴圈。這樣敘述可能比較難懂，我們實際舉例吧！

例如我們輸入「123」，進入 while 迴圈，123 除以 10 的餘數是 3，那我們就找到 digits 裡面找第三項，需注意是從第零項開始數喔！因此實際上會找到「3」這個數字再拼上 res，但因為 res 一開始是空的，因此不用拼上任何東西。接下來 123 除以 10 等於 12，12 大於零，再跑一次這個迴圈。

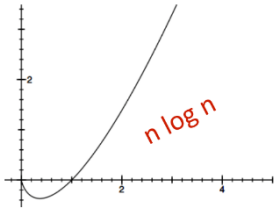
接著 12 除以 10，餘數為 2，digits 內的第二項是「2」，2 要拼上剛才的 res「3」，因此我們得到「2、3」，那 i 的 12 還要再除以 10，變成 1，還需要再跑一次迴圈。

1 除以 10 得到餘數 1，digits 內的第一項是「1」，再拼上剛剛的「2、3」因此我們得到「1、2、3」。

雖然程式碼中有個 while 迴圈在執行，但跑的過程並不是輸入多少便執行幾次，而是多了一個除法的動作。因此這裡的時間複雜度就會呈現「O(log n)」型。

18.2.5 線性對數時間複雜度（Log-linear Complexity）

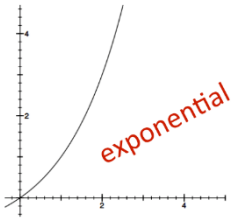
線性對數時間複雜度，顧名思義就是由線性（Linear）與對數（Logarithmic）組合而成的。它的增長比線性時間快，但比指數時間（指數 >1）增長的慢。



常見的例子是合併排序法（Merge Sort），此排序法的內容會在下一節介紹給大家喔！

18.2.6 指數時間複雜度（Exponential Complexity）

指數時間複雜度，隨著資料量的增加，執行時間也會很誇張的增長。



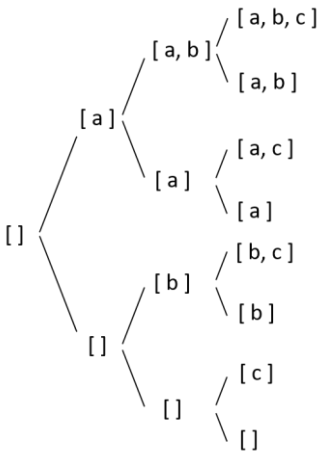
以下面例子來說明，此程式的目的是列出輸入 List 的子集合。是怎麼樣的子集合呢？如下方右圖所示。

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

{1,2,3,4} →
{ },
{1},{2},{3},{4},
{1,2},{1,3},{1,4},
{2,3},{2,4},
{3,4},
{1,2,3},{1,2,4},{1,3,4},
{2,3,4},
{1,2,3,4}

從圖上可以發現答案數列的順序會跟原本的數列相同，只是某些元素「有」，某些「沒有」。從「完全空的」到跟原本「一模一樣」共有 16 種組合。左圖的程式是怎麼跑右圖的數列呢？

其實就是將 List 內的元素分為「有」或「沒有」，再搭配成新的數列。我們用下圖來表示。

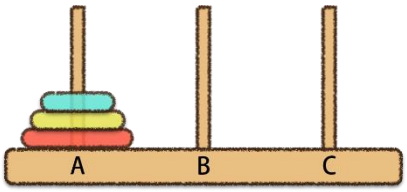


從最一開始的「什麼都沒有」，再來「是否有 a」、「是否有 b」，跟最後「是否有 c」，最後這排便是答案。而從圖中我們得知，若要得到最後一排的答案，我們必須先得到前一排的答案，以此類推。也因此若有 n 個數，便可得到2^n個答案。

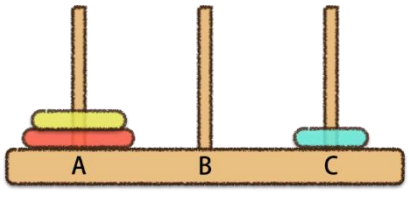
上圖例子中的程式碼，「smaller」就是在排除數列的最後一項，創造一個較小的數列以求答案，再將較小數列拆成更小的以求答案，而每次拆分剩下的元素就會存到「extra」內。這種在程式內又呼叫程式本身，在函式內呼叫函式的動作，稱為「遞迴結構」，它會將大問題一直拆分成小問題以作解決，而它終止於拆到剩下 0 個元素時。

利用「a, b, c」圖中最後兩層搭配程式碼來解說，「smaller」是排除最後一項「c」後的狀態，而「extra」內就是「c」。後 for 迴圈找出四個「smaller」的元素「i」，也就是「a, b」、「a」、「b」、「 」這層，接著在新開的「new」內添加「i」和「extra」，就會獲得「a, b, c」、「a, c」、「b, c」、「c」四項有 c 的組合，而最後的「smaller」與「new」則會獲得「a, b」「a」「b」「 」四項無 c 的組合，這樣最後一層的 8 項組合就出來啦！

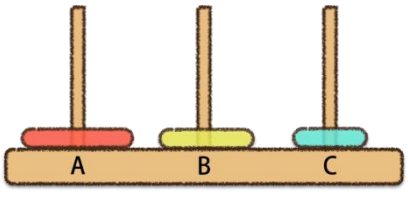
指數時間還有一些經典例子，如「河內塔」，河內塔是以三根塔和數個圓盤組成的遊戲（如下圖）。



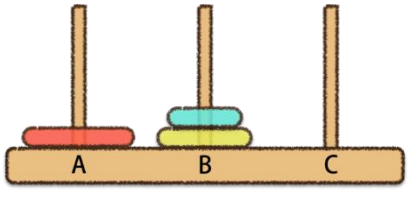
它的目標是要把所有的盤從 A 塔移到 C 塔，且一次只能動一個盤，大的盤不能疊在小的盤上面，該怎麼做呢？利用圖片來講解移動順序。



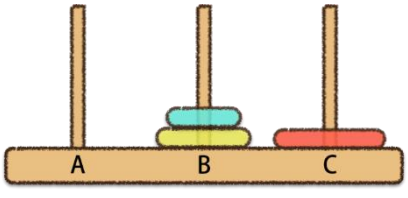
步驟 1



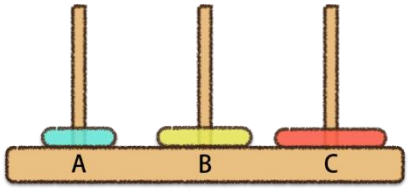
步驟 2



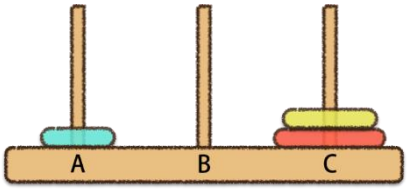
步驟 3



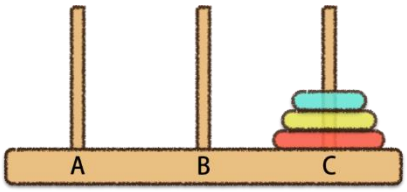
步驟 4



步驟 5



步驟 6



步驟 7

只有三個盤移動的時候很簡單，但假設是 n 個盤，目標都是把原塔 n-1 個盤都先移動到一個輔助塔上，在將最底部的盤移動到標塔，重複動作至完成。如此我們便得到時間（步數）的遞迴結構，也就是下圖所表示的。

$$
\begin{aligned}
t_n &= 2t_{n-1} + 1 \\
&= 2(2t_{n-2} + 1) + 1 \\
&= 4t_{n-2} + 2 + 1 \\
&= 4(2t_{n-3} + 1) + 2 + 1 \\
&= 8t_{n-3} + 4 + 2 + 1 \\
&= 2^k t_{n-k} + 2^{k-1} + \ldots + 4 + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 4 + 2 + 1 \\
&= 2^n - 1
\end{aligned}
$$

意思是移動 n 個盤的時間等於把 n-1 盤移到輔助塔的時間，加上把最下面第 n 盤移動到定位的時間，再加上把輔助塔 n-1 個盤移動到目標塔上的時間。這樣便是兩倍的「移動 n-1 盤的時間」加上「一步的時間」，把算式展開，計算到最後便會得到「$2^n - 1$」。因此 Big O 內就是「2^n」喔！

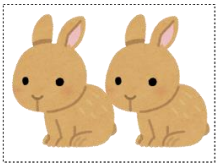
再來看另一個例子，例如費氏數列：0、1、1、2、3、5、8、13……

而這串數字是怎麼來的呢？其實是起源於一個想像中關於兔子數量的計算，首先有一些前提：

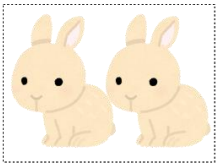
- 討論的兔子以「對」來計算
- 分別有大兔子與小兔子的分別

且每隔一段時間後會發生下列事情：

- 每對大兔子會生下新的一對小兔子
- 每對小兔子會成長為大兔子
- 不考慮兔子老死等其他因素

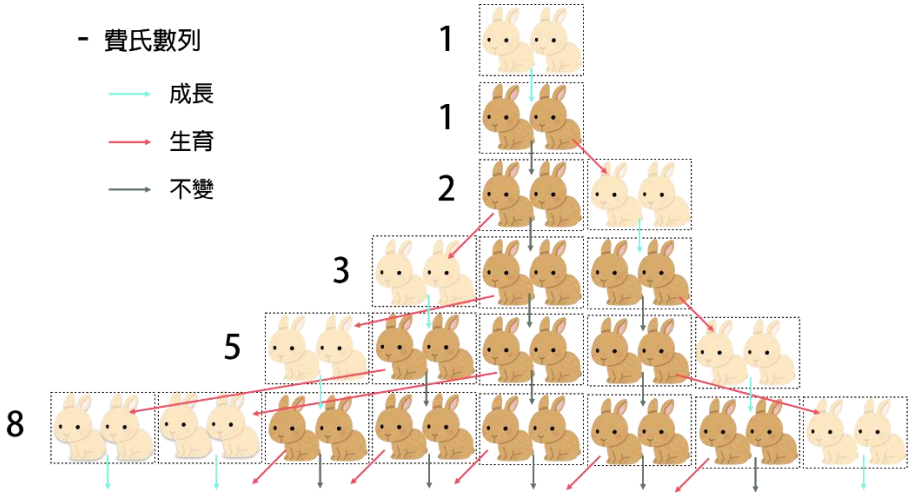


大兔子



小兔子

那麼這兔子對數的成長會呈現怎樣的數列呢？我們以圖片來表示，小兔子成長以藍線代表、大兔子生育以紅線代表，且因不考慮老死問題，除了生育外是一直存在以綠色代表。

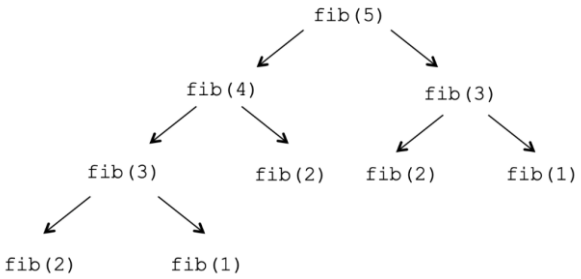


所呈現的數列就如一開始所說的：1、1、2、3、5、8……，經過整理後可以歸納成一個遞迴結構，將遞迴結構寫成程式就會獲得一個 Big O 是2^n的指數時間演算法喔！

F0 = 0
F1 = 1
Fn = Fn-1 + Fn-2 (n>=2)

```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

為什麼呢？若要獲得一個值，就必須先把它前兩個的值求出來，但若要得前兩個值，又分別需要它們自己的前兩個值，如下圖：



若需要找出上方第五項的值，就必須先找到第四項與第三項的值，但若需要第四項的值，又必須要先找到第三項與第二項的值，以此類推，以2^n成長，也因此時間複雜度是「2^n」。

但是大家還記得嗎？我們之前有提過指數時間不太理想，這樣是否有其他演算法可以解決這個問題呢？

其實是有的，線性時間複雜度就能解決費氏數列的問題囉！再度觀察上方的圖可發現，為了求出第五項，有不少項是被重複計算的，那我們只需透過一個變數來記錄計算過的值，這樣便能在需要時取用，而不是每次都需要重新計算。

```
def fib_inter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_i
        return fib_ii
```

O(n)

18.3 搜尋排序

這節會介紹一些程式常見的搜尋法以及排序法。

搜尋法是幫助我們在 List 中找到需要的項目，而排序法則是要將雜亂的 List 排列好。我們將會介紹兩個搜尋法以及三個排序法。

18.3.1 線性搜尋

首先是大家不陌生的線性搜尋，也就是之前提過在 List 中一個一個比對搜尋 e 元素的例子。

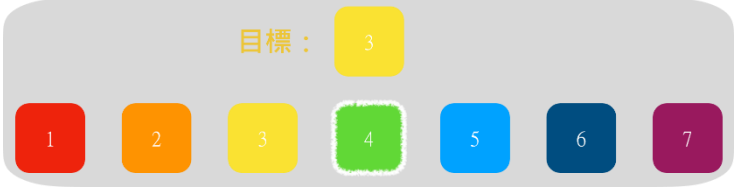
以圖片為例，它會從 List 的第一項開始搜尋至找到目標為止。


```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

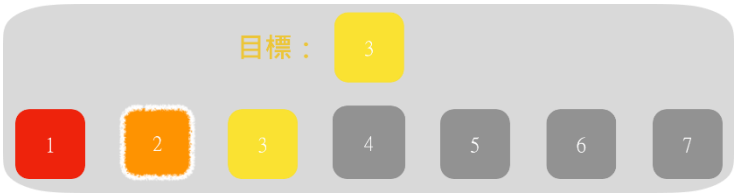
程式在前方也介紹過了，內有一個 for 迴圈，時間複雜度是隨著輸入量增加而等比增長的「O(n)」。

18.3.2 二元搜尋

此方法僅適用於數據已經排序好的狀況。此方法一開始會從 List 的中間項搜尋，接著去比對目標數字比中間項大或是小，這樣便能省去不重要的另外半邊，並重複動作直到找到目標為止。



步驟一



步驟二

步驟三

若轉換成程式的話，內容如下圖：

```
def bisect(L, e):
    def helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid:
                return False
            else:
                return helper(L, e, low, mid-1)
        else:
            return helper(L, e, mid+1, high)
    if len(L) == 0:
        return False
    else:
        return helper(L, e, 0, len(L)-1)
```

重複執行 helper 的動作，也就是切一半、比大小、排除不必要的元素這幾個步驟。同時可以發現這演算法並不需要像線性搜尋跑完整串 List 才能找到答案。而這演算法只需「O(log n)」的時間複雜度就能解決囉！

18.3.3 搜尋方式 - 小結

二元搜尋只能處理經過排序的資料，但線性搜尋並沒有這樣的限制，因此對於沒有排序的資料，二元搜尋就必須再多一個排序的成本。

雖然我們尚未提到排序，但是大家可以先想想看，「排序一個完整的數列」一定是整個數列都跑過一次，也就是每個數字都檢查一次吧！所以複雜度都是 O(n)以上。

也因此對於需要排序成本的二元搜尋並不一定比線性搜尋的效率高喔！

18.3.4 氣泡排序

一串數列由一個氣泡將數值兩兩包覆起來並比較大小，較大的數值會被往右邊移動，因此最大的數值會被移動到右邊的位置。後面重複步驟排序出倒數二個、倒數三個數值，至排序完成。如下圖：

步驟一

步驟二

步驟三

步驟四

步驟五

步驟六

步驟七

.....

若轉換成程式的話，如下圖：

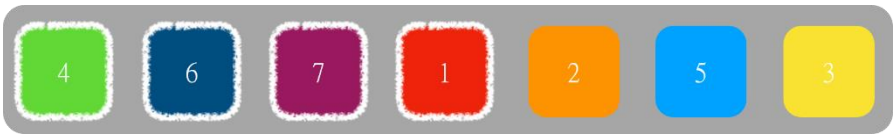
```
def bubble_sort(L):
    N = len(L)
    for i in range(N-1):
        for j in range(N-i-1):
            if L[j]>L[j+1]:
                temp = L[j]
                L[j] = L[j+1]
                L[j+1] = temp
```

程式內由 i 控制總共氣泡要跑幾趟，每次的目標都是比較並排列好最右邊的數值。因此長度為 N 的 List，最多就需要跑 N 趟，也就是 range(N-1)。而程式內的 j 控制的是每趟必須要做的「比較、交換」的動作次數，因為是兩個兩個比較，因此第一趟是會要跑 N-1 次，第二趟是 N-2 次，以此類推。紅色框內便是「比較、交換」的動作，先進行比較，再建立一個變數將較大的數值放入，之後再進行交換。

而氣泡排序的時間複雜度為雙層迴圈的「$O(2^n)$」。

18.3.5 選擇排序

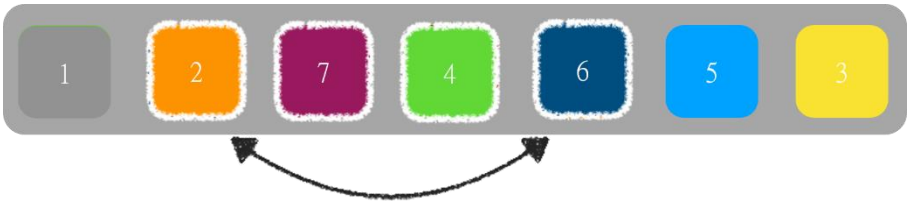
一串數列直接尋找最小的數值，再將它與最左邊的數值對調，以解決每回合最左邊的排序。如下圖：



步驟一

步驟二

步驟三



步驟四

.....

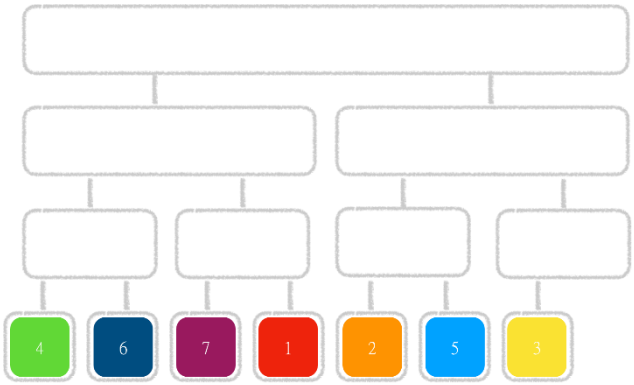
若轉換成程式的話，如下圖：

```
def select(L):
    suffix = 0
    while suffix != len(L):
        for i in range(suffix,len(L)):
            if L[i]<L[suffix]:
                L[suffix], L[i] = L[i], L[suffix]
        suffix += 1
```

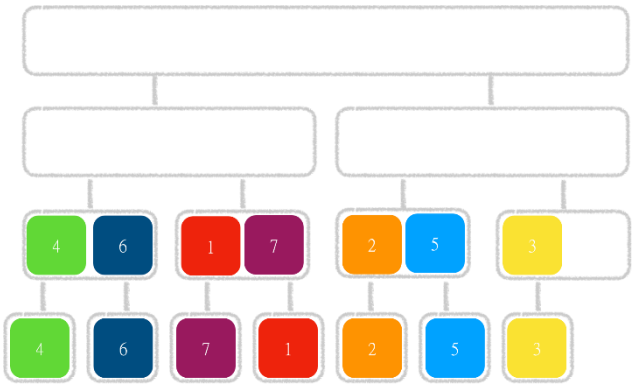
設定一個 suffix 循序漸進的跟 i 做比較，當項目 i 較小時就跟 suffix 對調。概念和氣泡排序有點類似，比較的總次數是相同的，時間複雜度也是「$O(2^n)$」。

18.3.6 合併排序

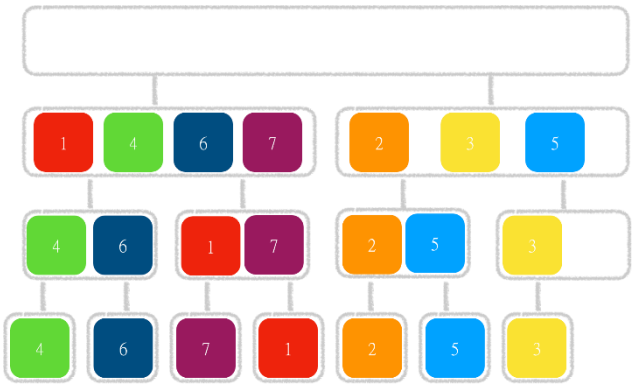
一串數列會持續一半一半的切分至最小項目，如一串數列有七個數字，最小就會被分成七塊，也就是一個大問題被分成七個小問題，小問題會向上排序，最後再處理完整個問題。如下圖：



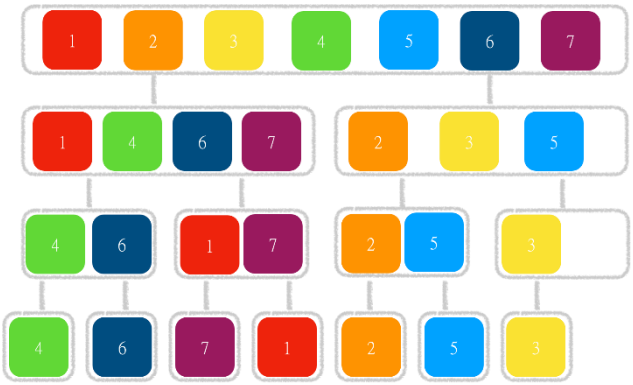
步驟一



步驟二



步驟三



步驟四

若轉換成程式的話，如下圖：

```
def merge(left, right):
    result = []
    i,j = 0,0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
      result.append(right[j])
      j += 1
    return result
```

合 併

O(n logn)

```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

拆 分

O (n)

程式碼分為兩個部分，分別為合併與拆分。首先先講解「拆分」的部分，此部分其實就是不停將 List 切分為左半邊與右半邊，直到剩下最後一個項目為止，切分完成後再將它們丟至上半部的「合併」處理。項目進至「合併」部份後，會先進行比較大小，再存入新的陣列，因此遞迴上去後，每個小項目皆會是排列好的狀態。因此進入下一層時，只需一直比較兩個陣列的第一項數值，將較小的存入新數列，再持續比較陣列內的較小數值。

而合併排序的時間複雜度，「拆分」的部分由於一串數列要拆成一個數值一個格子，因此會需要 n-1 個步驟，時間複雜度也就是 O(n)。「合併」的部分從

分出幾層可得知一共需要比較的回合數有 log n 次，而加上每回合內要比較的次數呢？便是 log n 回合乘上每回合 n 次，時間複雜度也就是 O(nlog n)。

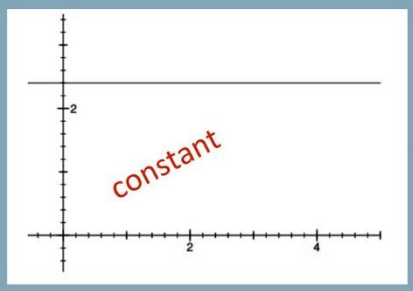
兩半部分綜合起來取影響值較大的時間複雜度，最後得到的即是「O(nlog n)」。

章節回顧

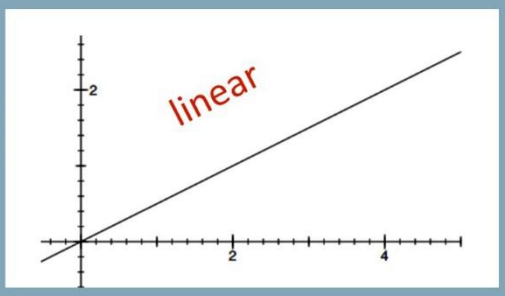
1. 演算法：

2. 時間複雜度

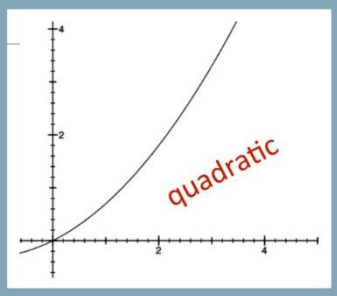
- 常數型時間複雜度　O(1)



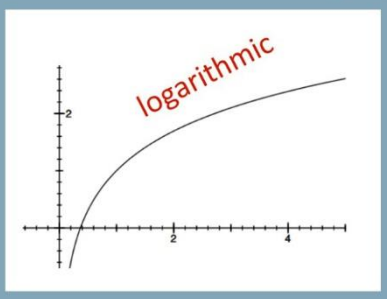
- 線性時間複雜度　O(n)



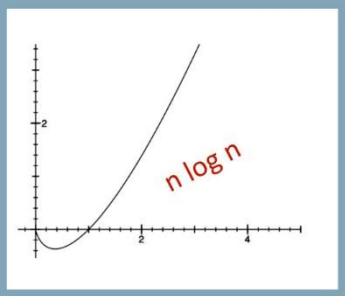
- 平方時間複雜度　O(n^2)



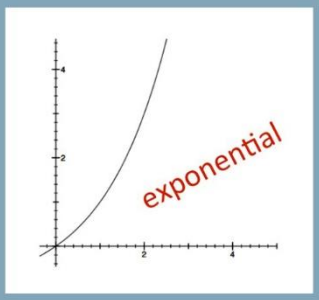
- 對數時間複雜度　O(log n)



- 線性對數時間複雜度　O(nlog n)



- 指數時間複雜度　O(2^n)



3. 搜尋排序

- 線性搜尋：O(n)
- 二元搜尋：O(log n)
- 氣泡排序：O(2^n)
- 選擇排序：O(2^n)
- 合併排序：O(nlog n)