# Ch2. 複雜度估算 Complexity

緊接著來看如何分析與運算特定程式碼的「複雜度」。

本章中，首先說明「為什麼要分析複雜度」以及「評估複雜度的方式」，再來，會介紹複雜度裡最重要的符號 Big-O。

之後，運用高中數學學過的「極限」來化簡 Big-O 的證明方式，並介紹除了 Big-O 以外，還有哪些估計複雜度的符號。最後，來看遞迴（式）的複雜度應該如何計算。

本章中用到一些數學工具，建議讀者可以動手實際運算一次。

## 第一節：複雜度簡介

1. 為什麼要評估複雜度

上一章中已經稍微提到此問題的答案：電腦並非無所不能。

雖然電腦運算比人腦運算快很多，但實際上還是有限制的。許多狀況下，即使是最先進的超級電腦也無法算出所有答案，這時如果採用更「好」的演算法，就能在單位時間內取得更好的解答。

再來，記憶體雖然很便宜，但絕非免費，特別是處理圖像或影片的時候，需要耗費許多記憶體空間，成本也可能很高昂。

當無法直接取得最佳解時，越有效率的算法可以找到越好的解答，正如同走迷宮時，通常只能看到眼前的路，但是如果從較高的位置望出去，就能夠看得更遠，也因此可以找到更好（更有效率）的路徑，以利更快走出迷宮。

(1) 如何評估複雜度

在評估複雜度之前，有一些前提條件 Criteria。

在檢視一個算法的複雜度前，要先看看它的「正確度」與「可讀性」：如果算法沒有正確度，即算法是「錯的」，無法被執行或達成設定好的目標，那自然沒有評估複雜度的意義，再來，程式碼要可以被閱讀，使其後續還可以被其他人維護與修改。

進入效能評估（Performance Analysis）的階段後，大致關注兩個方向：「時間複雜度」與「空間複雜度」。

* 時間複雜度代表一段程式執行所需的「運算次數與時間」
* 空間複雜度代表執行時會「佔用多少記憶體空間」。

2. 空間複雜度

(1) 空間複雜度的描述

：需要的總記憶體空間

：固定佔用的空間

：隨資料量變動的佔用空間

如上式所示，演算法需要的總記憶體空間 ，由 與 兩部分構成。

其中， 是一個常數，並不隨著輸入的資料量大小不同而改變，程式碼中的常數（constant）或全域變數（global variable）佔用的空間屬於這個部分。

則會隨著輸入資料量 的大小改變：如果輸入的資料量 變大，就會隨之變大。遞迴式用到的堆疊（recursive stack space）、位於函式中的局部變數（local variable）等所佔空間屬於這部分，因為隨著資料量增加，呼叫函式的次數越多，過程中就會產生越多的局部變數。

(2) 費波那契數列的空間複雜度

|  |  |
| --- | --- |
| 計算費波那契數 Fibonacci (n) | |
| 1  2  3  4  5  6  7 | int Fibonacci (int n)  {  if (n <= 2)  return 1;  else  return Fibonacci (n-1) + Fibonacci (n-2);  } |

第 n 個費波那契數 Fibonacci (n) 會被拆解成第 n-1 個費波那契數 Fibonacci (n-1) 和第 n-2 個費波那契數Fibonacci (n-2) 的和，所以用如上的遞迴式來運算費波那契數時，所需空間取決於總共拆解出的數字個數 的大小。

舉例來說，F(50) = F(49) + F(48)，等號右邊的前項成立 F(49) = F(48) + F(47)，後面一項成立 F(48) = F(47) + F(46)。每個費波那契數都被拆解成兩次對於函式的呼叫，簡單計算後，發現對函式的總呼叫次數接近 。

一個函式被呼叫 次，每次固定產生 個局部變數，則遞迴執行完畢前共產生 個局部變數（遞迴全部完成前，變數都不能被釋放掉），所以所需的總空間大小取決於 ，即 。

從空間複雜度的公式來看， 是一個常數，無論想得到的是第幾個費波那契數，此部分都為定值，但 則受資料量 的大小影響。

「大概」等於 ，加上一個係數 ，表示 和 成正比關係。

3. 時間複雜度

1. 時間複雜度的描述

：需要的總運算時間

：固定花費的時間

：花費時間中隨資料量變動的部分

時間複雜度也是一樣，所需的總時間會由兩個部分構成：第一部分是不會因為輸入資料大小而改變的時間 ，第二部分則是會因為輸入資料大小而改變的時間 。

|  |  |
| --- | --- |
| 兩種「1 到 N 的和」的算法 | |
| 算法1 | 算法2 |
| int sum = 0;  for (int i=1 ; i<=N ; i++)  sum += i; |  |
|  |  |

舉例而言，上表中左右兩個程式碼都可以得到整數「1 加到 N」的和。左邊是由 1 開始加 2、加 3、…，一路加到 N；右邊則利用公式解，即面積等於「（上底+下底）乘以高除以 2」。

左邊的程式碼中，N 越大，運算的次數就越多（迴圈共執行 N 次），因此運行的時間取決於 N；公式解需要的時間則（幾乎）不會因為輸入的 N 值大小而改變，是一個常數 C。

(2) 費波那契數的時間複雜度

|  |  |
| --- | --- |
| 計算費波那契數 Fibonacci(n) | |
| 1  2  3  4  5  6  7 | int Fibonacci (int n)  {  If (n<=2)  return 1;  else  return Fibonacci (n-1) + Fibonacci (n-2);  } |

費波那契數的「時間複雜度」方面，因為函式總共會被呼叫 次，所以總共需要的時間一樣是一個常數 ，加上正比於 的 ，即 。

## 第二節：複雜度的估計法

1. 估計複雜度的前提

首先，假設「包括但不限於下列的所有運算」都花費一樣的時間：

A. 加減乘除

B. 取餘數

C. 位運算、存取記憶體

D. 判斷、邏輯運算子

E. 賦值運算子

實際上，它們所需的時間當然不一樣，此假設只是為了方便運算與統計。

在如上假設下，只要統計出整段程式總共需要的「運算次數」，看「次數」的數量級大小，就可以得出複雜度，也就可以評估執行所需的時間。也就是說，在估計複雜度時，一般假定「做 10 次運算」正好需要「做 1 次運算」的 10 倍時間，實際上這當然並不完全精確。

1. Step Count Table

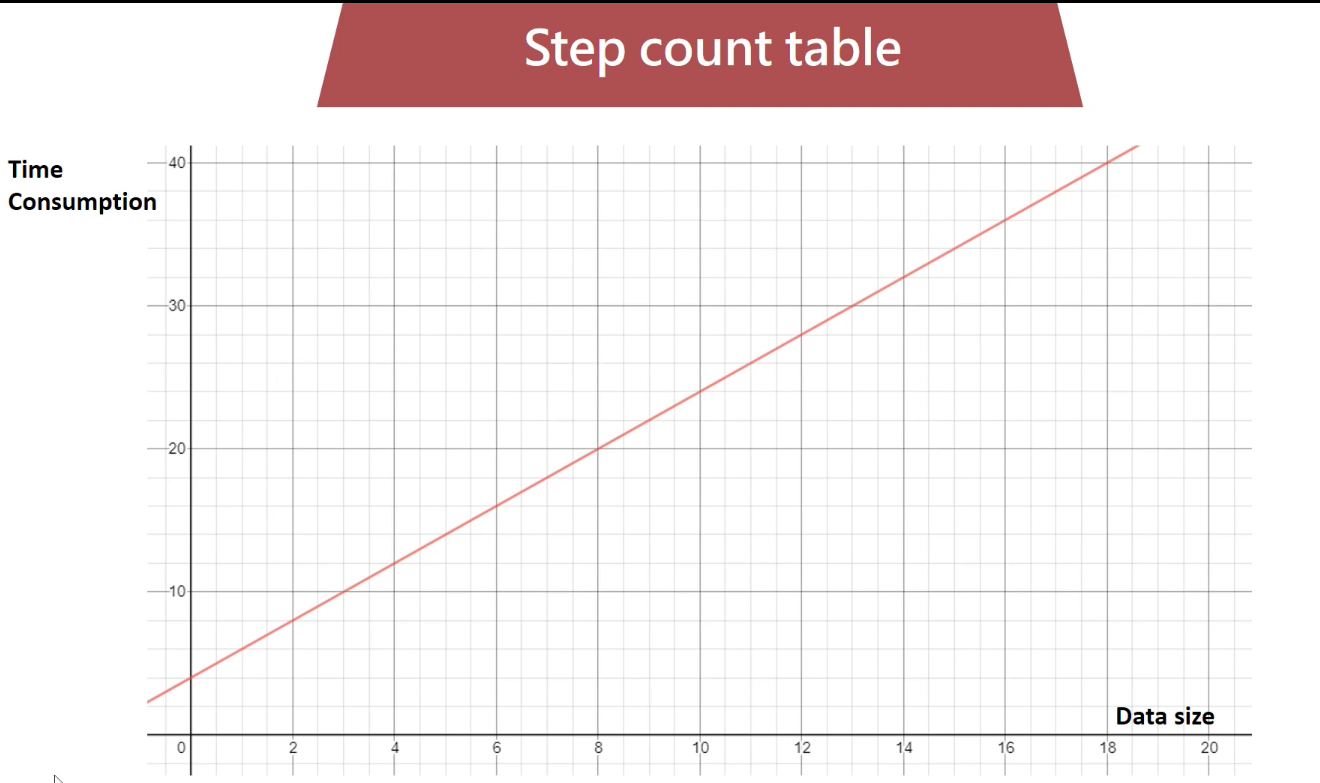
設計一個如下的函式，它的功能是把一個陣列裡的值全部加起來。傳入值是陣列開頭的指標 \*p 和一個整數長度 len，接著，用一個 for 迴圈把每一筆資料加到 sum 裡。

|  |  |  |
| --- | --- | --- |
|  |  | Steps Frequency Sum of steps |
| 1  2  3  4  5  6  7  8  9 | int sum (int \*p, int len)  {  int sum = 0;  if (len > 0){  for (int i=0 ; i<len ; i++)  sum += \*(p+i);  }  return sum;  } | 0 1 0  0 1 0  1 1 1  1 1 1  1 len+1 len+1  1 len len  0 1 0  1 1 1  0 1 0 |
|  |  | Total steps：2len+4 |

怎麼計算複雜度呢？首先，可以看每一行程式碼需要的「運算步數 Steps」，以及這行程式碼執行的「總次數 Frequency」，將這兩項相乘，就是這段程式碼的總運算步數（Sum of steps）。

以第三行為例，int sum = 0需要的步數 Step 是 1，且這行總共執行的次數Frequency 是 1 次，所以這行得到的總步數是 1；第六行的 sum += \*(p+i) 一樣需要一步，總執行次數是 len 次（ 從 0 遞增到 len-1），因此所需總步數為 步。

注意到第 5 行總共執行 次，而非 次，因為 正好與 相等時，仍需要經過「比較 和 」的動作後，才能決定應跳出迴圈往下執行。



將每一行需要的步數加總，發現整段程式碼需要的總時間是 次。把 畫在一個圖表上，橫軸是資料大小 ，縱軸是所需時間， 是一個二元一次方程式。

圖中的 y 截距 4 即剛剛講的 ，並不會跟著資料量改變； 的部分則隨著資料量增加而變大，因此所需總時間與資料量大致呈正比關係。

1. 「小時候胖不是胖」

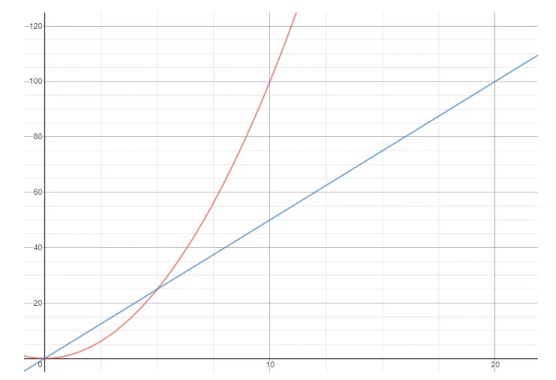
對於一個特定的演算法而言，資料量小和資料量大時的優劣不一定相同，通常需要在意的是「資料量大」的時候演算法的表現，因為資料小的時候所耗費的時間原本就不多，所以可以不予考慮。

|  |  |
| --- | --- |
| 算法1 | 算法2 |
| int sum = 0;  for (int i=1 ; i<=N ; i++)  sum += i; |  |

以剛剛看過的程式碼而言，左邊是由 1 加到 N，右邊則是利用公式解。

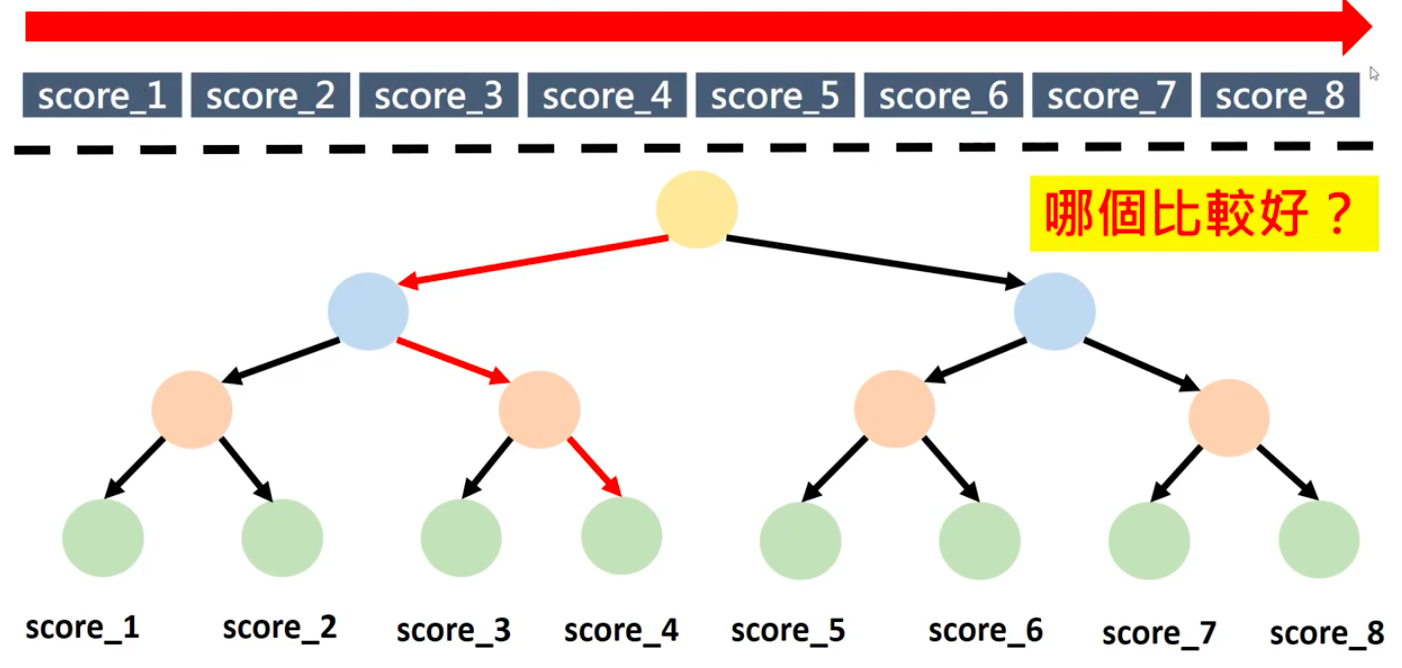
兩種方法在資料量小與資料量大時有固定的優劣關係嗎？極端的情況，如 n 是 1 的時候，其實右邊要比左邊花費更久的時間：因為右邊的算法需要進行乘法和除法，而左邊則只有加法而已。

也就是說，資料量小的時候，左邊比較快，右邊比較慢，但是資料量一大，很明顯的右邊的算法會比較快。



上圖（橫軸為資料量、縱軸為所需時間）可以做一個類比，當資料量小的時候，紅線表現得比較好（速度快、需要的時間短），但是資料量一大，藍線就表現得比較好（速度快、需要的時間短）。因為在意的通常是資料量大的時候，所以藍線對應的演算法較佳。

1. 不同的搜尋演算法



上圖比較的是搜尋陣列時的「循序搜尋」和「二分搜尋」兩種方法。

從第一個搜尋到最後一個就叫「循序搜尋」，而把資料用二元樹的形式儲存，形成二元搜尋樹，每次搜尋皆可去除一半可能位置的方法，叫做「二分搜尋」。

循序搜尋中，最少需要搜尋 1 次（目標在資料開頭），最多則需要搜尋 8 次（目標在資料結尾）；相對的，二分搜尋不管目標資料位在何處，都必須進行 3 次搜尋（n 次搜尋可以找遍 2n，23 = 8）。也就是說，對於不同搜尋目標而言，哪一種方法比較好，可以更快找到，其實並非一定。

1. 什麼才是「好」？

究竟是時間花的少，還是記憶體空間花的少更重要？亦或精準度或正確率較高才是需要關注的？

進行某些工程運算時，可以藉由犧牲部分精準度來節省運算時間，比如使用「二分搜尋法」進行根號運算的時候，如果能容忍的誤差越大，所花的時間也就可以越少。

就像這樣，「速度快」、「節省空間」、「精準度高」和「開發成本低」等考量之間，其實並沒有哪個一定更重要，隨時需要進行「平衡」。之前舉過的例子是以抽樣代替普查，雖然抽樣的精準度比普查差，但是時間在此例中，顯然比精準度來得更重要，畢竟 7 年才做出的普查意義不大。

只是再次重申，由於 CPU 運算資源通常更珍貴，所以大多時候更關注如何改善「時間複雜度」。

1. 時間換取空間 v.s. 空間換取時間

「時間換取空間」的策略是指用 CPU 的運算時間來節省記憶體空間的使用，具體來說，就是每次運算完後，都不儲存結果，每次要用到相同結果都重新再算一次。

「空間換取時間」的策略則每次運算完都將得到的結果儲存起來，下次要用到時查表就好，表上沒有所需結果時，也可使用內插法。這種做法會產生一個很大的表，佔用許多記憶體空間，但是一旦建出表格後，未來就能節省許多運算時間。

兩種策略中，較常使用的是以「空間換取時間」。

1. 時間 = 資料量 x 複雜度

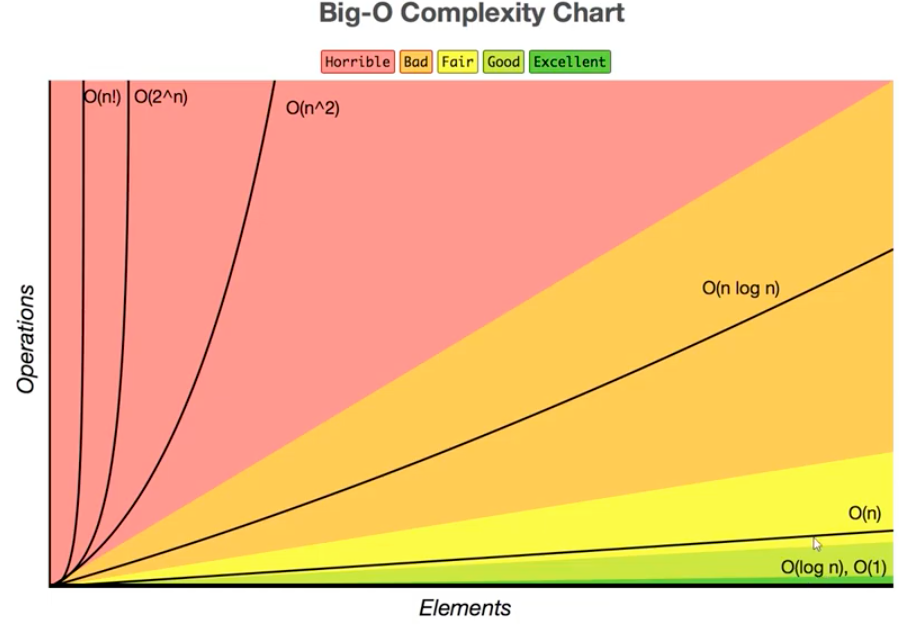
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 複雜度 -> | 1 | N | N2 | N3 | 2N |
| 資料量 |
| 1 | 1毫秒(ms) | 1 | 1 | 1 | 1 |
| 10 | 1 | 10 | 102 | 103 | 1024 ~ 103 |
| 100 | 1 | 102 | 104 | 106 | ~1030 |
| 1,000 | 1 | 103 | 106  (~15分鐘) | 109  (~12 天) | ~10300 |
| 10,000 | 1 | 104 | 108  (~25 小時) | 1012  (~30 年) | ~103000 |

從上面的表格裡可以看出，當資料量從 1 變成 1,000，成長 103 倍時，如果演算法的複雜度是 ，所需時間變成 106 倍；如果複雜度是 ，所需時間則變成 109 倍，這代表運算總次數會隨著 改變而有巨幅變動。

假設一次運算需要耗費 1 毫秒（ 秒），那麼若有 10,000 筆資料，而複雜度是 ，總共需要花費 10 秒（）；複雜度為 時，大約需要 25 小時； 約需 30 年的時間； 時更是根本算不完。

這也是描述複雜度時習慣以「 的次方數」表示的原因， 的次方數一增加，只要資料量變大一個數量級，所需的時間就會呈指數成長，變動非常大。

1. 複雜度的優劣之分



討論複雜度的時候，可以用幾種不同的等級來表達。

如果複雜度是 O(1) 或 O(logn)，就是相當好的演算法，如果是 O(n)，即需要的時間大致上與資料量 n 成正比，那可以算是「普普通通」。

O(nlogn) 大概位於可接受的邊緣，如果複雜度到達 O(n2)、O(2n)，甚至 O(n!)，那麼在資料量大時，幾乎不可能使用現在的電腦將結果算出來。

也就是說，一經評估複雜度，就可以大致瞭解一個演算法的複雜度落在較好或較差的區間，也決定了資料量大時，該演算法還具不具備實用性。

在做競賽題目時，通常複雜度會落在 O(nlogn)，這是因為許多常見的演算法如排序、插入、刪除等，複雜度都為 O(nlogn) 。在寫題目時，可以計算一下自己的答案是不是落在 O(nlogn) 或更好的範圍，如果超過了，比如複雜度是 O(n2) 或 O(2n) ，很可能代表有問題，少數狀況 O(n2) 可以接受，但只要到達 O(n3) 以上，通常就無法接受了。

1. 範例與練習

|  |  |
| --- | --- |
| 計算下列程式碼「執行輸出」的次數 | |
| 1  2  3  4 | for (int i=0 ; i<N ; i++)  for (int j=0 ; j<N ; j++)  for (int k=0 ; k<M ; k++)  cout << i \* j \* k << " "; |

最內層的迴圈會優先執行， 從 0 增加到 M-1，總共執行了 M 次；中間這層的 從 0 遞增到 N-1，總共執行了 N 次；最外層的 從 0 遞增到 N-1，也是總共執行 N 次。

綜合起來，這個巢狀迴圈總共會執行 次。

|  |  |
| --- | --- |
| 計算下列程式碼「執行輸出」的次數 | |
| 1  2 | for (int j=1 ; j<N ; j+=2)  cout << i << " " << j; |

第一次執行時 j 是1，第二次執行時 j = 3，第三次執行時 j = 5，以此類推。若將迴圈目前執行的次數以 表示，那麼 j 的值可以寫成 ，只有 時，才會滿足迴圈的條件而繼續執行。

移項一下，可以得到 。

|  |  |
| --- | --- |
| 計算下列程式碼的迴圈執行次數 | |
| 1 2  3 | for (i=M ; i>1 ; i/=2){  cout << i << endl;  } |

迴圈從 時開始執行， 必須大於 1，且每次執行完 會除以 2。

所以第一次執行時 ，第二次執行 ，第三次執行時 ，第 次執行時 。

因為 要滿足大於 1 的要求，所以：

。

|  |  |
| --- | --- |
| 1. 計算下列程式碼的時間與空間用量 | |
| 1  2  3  4  5  6 | for (j=0 ; j<M ; j++){  cout << j << endl;  }  for (i=0 ; i<N ; i++){  cout << i << endl;  } |

1. 空間複雜度：不管輸入的 M 和 N 是多少，都會使用同樣的記憶體空間。
2. 時間複雜度：第一個迴圈中， 從 0 到 M-1，總共跑 M 次；第二個迴圈中， 從 0 到 N-1，總共跑 N 次，上下加起來，迴圈執行總次數是 M+N。

|  |  |
| --- | --- |
| 計算下列程式碼的迴圈執行次數 | |
| 1 2 | for (int j=1 ; j<N ; j\*=2)  cout << j; |

從 1 開始執行， 需要小於 N 才會繼續執行，而 每次執行都會乘以2。假設執行第 n 次，此時 是 2n-1。

由於 時迴圈才會執行，兩邊取 ，可以得到 。

|  |  |
| --- | --- |
| 計算下列程式碼的迴圈執行次數 | |
| 1 2  3 | for (int i=1 ; i<N ; i++)  for (int j=1 ; j<N ; j\*=2)  cout << i << " " << j; |

內圈與剛才的例題完全相同，會執行 次。外圈的 則從 1 跑到 N-1，總共執行 N-1 次。

內外圈結合起來，總共執行 次。

## 第三節：Big-O 的運算證明

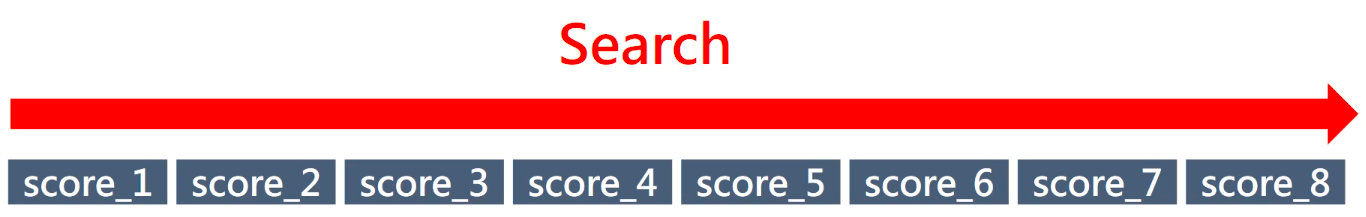
本節來看看如何用 Big-O 符號來描述演算法的複雜度，複雜度 Complexity就是描述演算法「工作效率」的函數。

1. 上界與下界

同一個演算法依資料散布情形的不同，處理次數也會有不同：

1. 最壞的情況 Worst-case 下需要的處理次數叫做「上界 upper bound」，因為最壞的狀況下需要的時間是最多的，所以實際執行需要的時間一定不會超過這個「上界」。
2. 最好的情況下需要的處理次數是「下界 lower bound」，「下界」對應的時間是最少需要的時間。

另外也會考慮平均的情況 Average-case。

1. 循序搜尋的上界與下界

如果要在一個長度為 Len 的陣列裡搜尋資料，可以使用「循序搜尋」，從陣列開頭一筆一筆資料確認。

最壞的狀況下，搜尋到最後一筆才找到所需的資料，因此共訪問了 Len 筆資料，這個 Len 就是「上界 upper bound」；最好的情況則是第一筆資料就是所需的資料，這時只要搜尋一筆資料，1 就是「下界 lower bound」。

平均來說（Average-case），需要搜尋 次。

1. Big-O 的理想條件

通常想知道的是某一算法在「最壞情形」下的表現，也就是最差的狀況下需要多少時間，這樣才能事先留下足夠時間供其執行。也就是說，在意的是 Worst-case，或者資料增加時「所需時間的成長幅度」，即 Growth rate。

另外，也希望複雜度不會受到單位的影響，不管是用 Byte 還是 Bit 為單位，或者時間上以秒、分、小時為單位，都可以得到相同的複雜度。

最後，不在乎小資料時的狀況（也就是「小時候胖不是胖」），畢竟資料量小時僅差幾毫秒，沒有實際影響。

1. Big-O 的數學定義

將上面提到的要求結合起來，就能構造出 Big-O 的數學定義：

[ 存在某個 和 ，使得對於所有 ，都成立 ]

其中：

* 是一個常數，加上這個係數後，Big-O 就不受記憶體 / 時間單位影響
* 指的是資料量 大於某個數

如果 ，那麼某個「所需時間可用 來表示」的演算法，它的複雜度就是 。

要證明 ，需要試著找到一組 跟 ，使得資料量 夠大（超過 ）的時候， 總是成立。

若覺得此處的數學式不太直觀，不妨直接往下看看實例。

1. 實際使用 Big-O

如果想知道 能不能寫成 ，就要試著找到一組 和 ，使得當資料量 超過某個門檻（）時，都有 ，這裡 可以取任意值。

可以按照如下步驟進行：

A. 把等號左右的 約掉（因為 代表資料量，必定是正數），得到

B. 取 （或任意取其它值），同時取

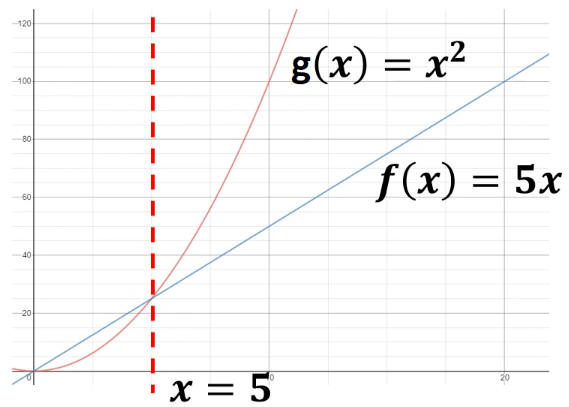
D. 根據前兩步驟，當 時，總是成立

E. 得證

用數學方式表達：給定 、，欲證有一組 、，使得對所有 時，總是滿足 。

1. 取 ,
2. ,

因此 ，此例中即 。



把 和 畫在上圖中，紅線是 對應的時間，藍線是 對應的時間，當 ，即資料量超過 5 筆時，藍線都低於紅線，代表 。

Big-O 相當於「上界」， 也可以說成「 是 的上界」。上面的例子裡， 是 的上界，在資料量超過 5 筆的時候，不管資料量再怎麼大，一個所需時間為 的演算法，需要的時間都小於 。

1. 範例與練習
2. 證明或否證

證明過程：

A. 取 ,

B. , (這個式子就是 ∀x > 0, )

因為找到一組 ， 滿足 Big-O 的條件，所以 。

1. 證明或否證

證明過程：

A. 目標是找到一組解滿足 (也就是 )

B. 等號左右同消掉 ，

C.

這代表等式「」只有在 的時候才會成立，所以無論 取多少，都不能找到對應的 （因為反過來說， 的時候一定不會成立），所以 。

1. 證明或否證

證明過程：

A. 要找到一組解滿足

B. 取 ，

C.

D.

E. ，

在 的情況下， 取 100 就一定會使條件 成立，這樣我們就找到一組 和 ，也就證明了 。

使用 Big-O 定義來求複雜度似乎有點麻煩，等一下會介紹一種更簡便的方式。

### 證明或否證

A. 找一組、，使

B. 取 ,

C.

D. 得證

### 證明或否證

A.

B. 兩邊平方，

C.

D. 反過來說，當 時，一定不成立

E. 由上可知，不管 取多少，都找不到 使得定義成立

no matter what c is, if , then

F. 否證

## 第四節：極限的表達方式

1. 用極限方法證明 Big-O

上一節中，介紹了從定義上證明 Big-O 的方式：找到一組 、，使得資料量大於 的情況下，都有 ，如此一來，可知 是 的上界。

--- (1)

--- (2)

但要找到這樣的 和 不一定總是很容易，所以會想另尋一種更簡便的方式。觀察 (2) 式，若將不等號左右同除以 ，可以得到下面的 (3) 式。

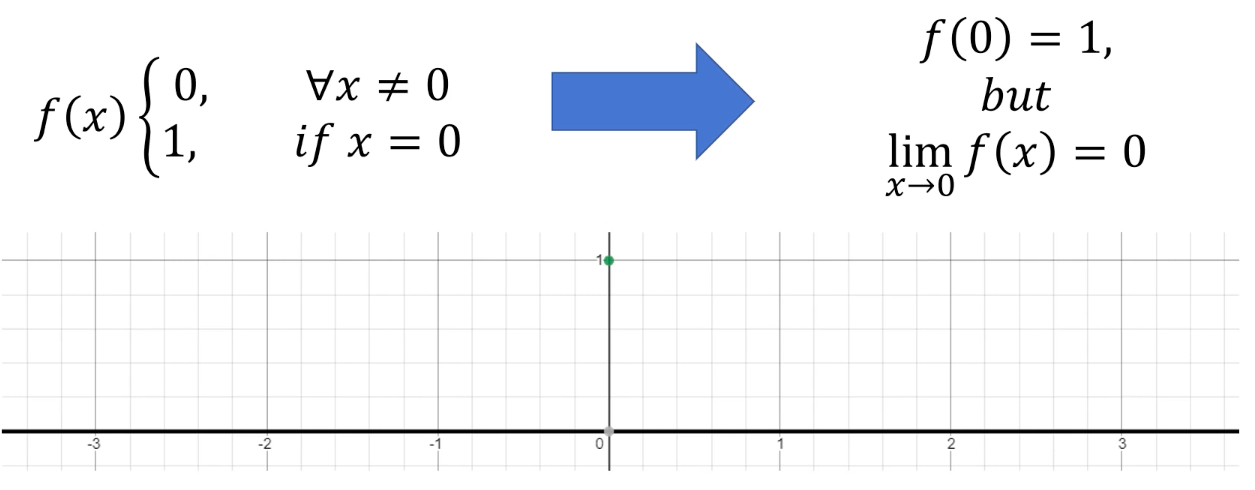
--- (3)

因為 (1) 的條件中只要求 (2) / (3) 的不等式在「 大於某個 時」成立，所以可以直接將 推到無限大（然後 取一個小於無限大的值）。如果下面的 (4) 式成立，即「 接近無限大時， 收斂到一個常數 」，那麼就知道 。

--- (4)

2. 極限的複習

往下進行之前，很快複習一下高中數學裡教過的「極限」。如果 在 逼近 時的極限為 ，可以寫成 ，這就是極限的定義。



舉一個例子，有一個 的函數 ，當 時， 的值是 0，而當 時，，因此 (0,1) 在 上。

雖然 ，但是 「趨近於」0 時，並不會真的「到達」 0，所以 在 x 趨近於 0 時，值仍然是 0，即 。

3. 極限的運算規則

給定下面三式：

根據極限的定義，下面的 A 到 E 式成立。也就是說，要得到「 和 的運算」的極限，可以先將 與 個別的極限值 L、K 取出之後，再進行相應的運算。

A.

B.

C.

D.

E.

特別注意 E 式， 會使得 ，也就是「無限大分之一」= 0。

4. 範例與練習

1. 計算

上下同除以 n，

因為其中所有形如 （ 是常數）的值都是 0，所以，

只要計算出 的極限值，看它是不是無限大，就可以決定是否有 。比如令 、，因為 ，所以 。

1. 證明或否證

因為 的值收斂到 5，而非無限大，得證 。

1. 證明或否證

上下同除以 x3，

因為 的值收斂到 0，而非無限大，得證 。

1. 證明或否證

因為 等於無限大，並不會收斂，所以。

1. 證明或否證

得證 。

### 證明或否證

(⇒)

假設左邊成立，，即 。

此時右邊是否成立？

因此右邊也成立。

(⇐)

假設右邊成立，，即 。

此時左邊是否成立？

因此左邊也成立，兩個寫法等價。

## 第五節：複雜度的其他符號

來看看除了 Big-O 以外，還有哪些符號可以被用來表示複雜度。

1. Big-O 存在的問題

Big-O 是上界，也就是「最壞的情況」，這代表 Big-O 裡面任意放一個非常大的數字時， 都會成立：

…

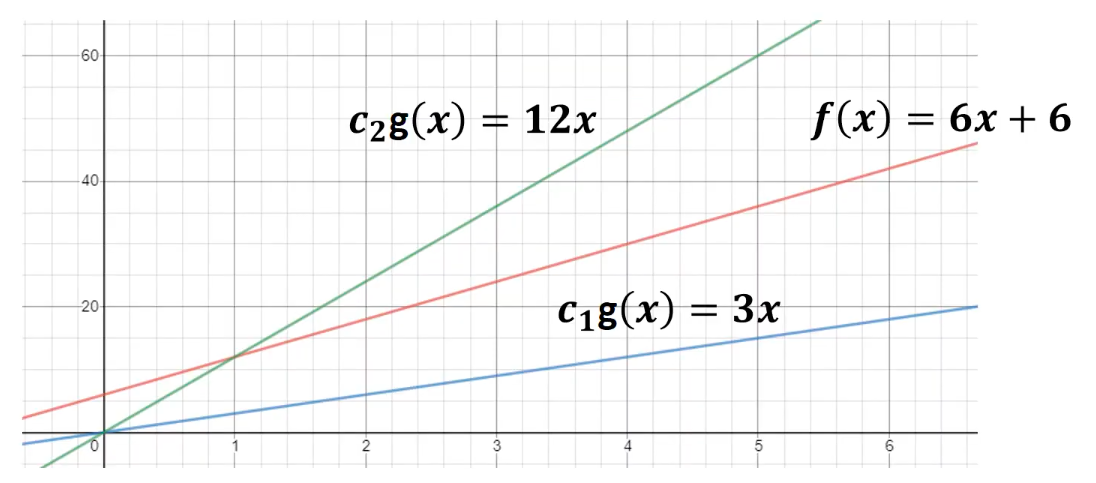
這就像如果媽媽和你說：「只要你比任一個學生成績好，我就給你獎金」，那麼只要去和全班、甚至全學年成績最差的同學比，很容易就可以達成。

只要 ， 就可以寫成 、、、…，且繼續把 Big-O 中 的指數任意增加後都仍會成立。

2. Big-Theta Θ

(1) Big-Theta Θ 的定義

Big-Theta Θ 代表 會被 和 上下包夾在一起，另一種說法是 在乘以兩個相異常數後，可以使 介在其間。



是否成立？

取 、兩常數 與 分別為 3 與 12，由上圖可看出紅線 在 時，被包夾在藍線 和綠線 之間，因此 成立。

(2) Big-O 和 Big-Theta 的差異

Big-O 的 x 次方數可以不斷往上寫，Big-Theta 則要求特定 x 的最大次方數，如下表所示：

|  |  |
| --- | --- |
| Big-O | Big-Theta |
| ... | ... |

(3) 使用極限方法證明 Big-Theta Θ

---(1)

---(2)

上面 (2) 式各項同除以 g(n)：

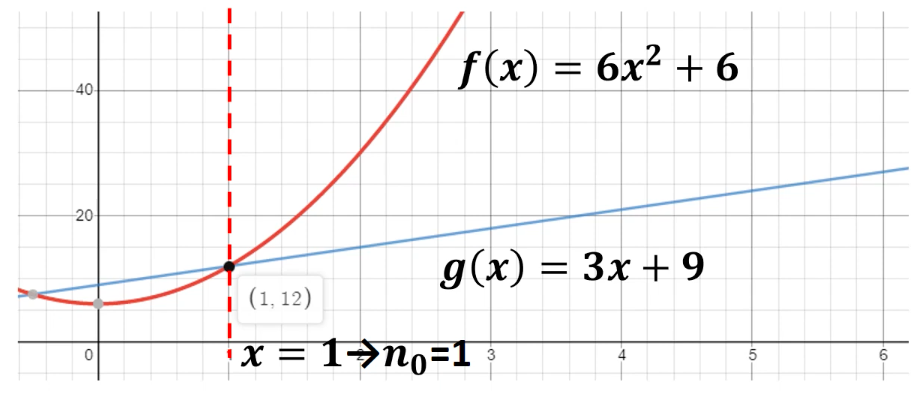
只要 的極限值「不是無限大」而且「大於 0」，則 。

注意： 只跟 中的 的最大次方有關。

3. Big-Omega Ω

(1) Big-Omega Ω 的定義

即 代表 是 的下界 (Big-O 則是上界)。



上圖中， 是否成立？

取 ，紅線 在 時都大於藍線 ，因此 成立。

(2) 使用極限方法證明 Big-Omega Ω

---(1)

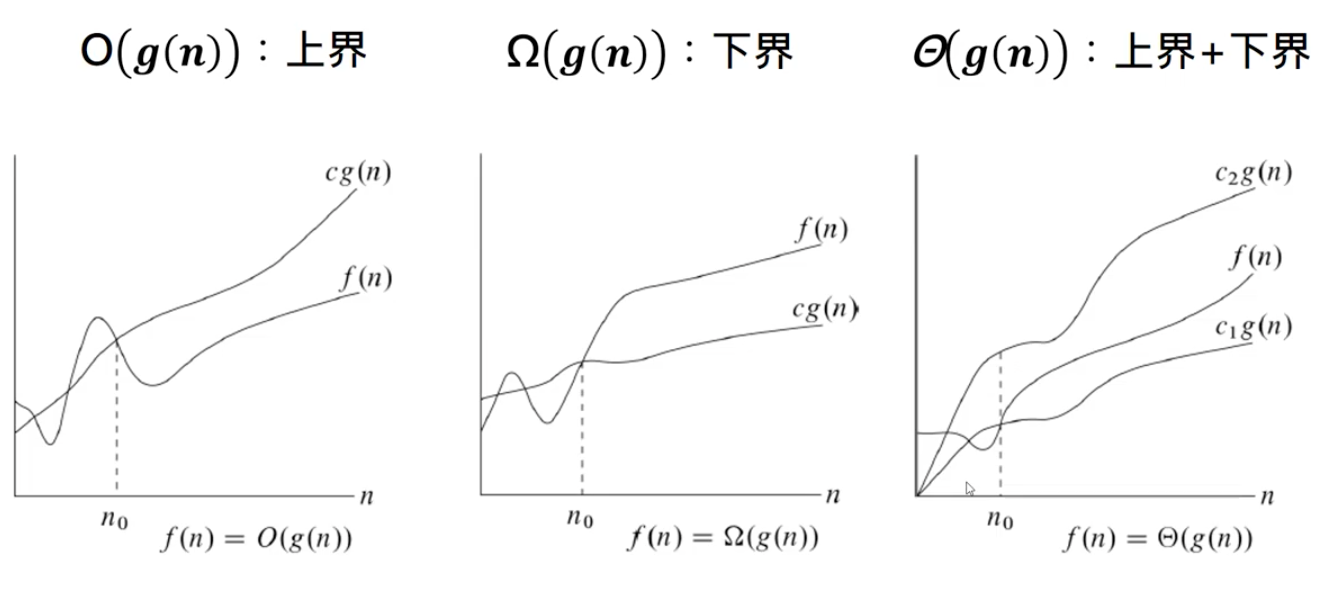
---(2)

上面 (2) 式中各項同除以 g(n)，

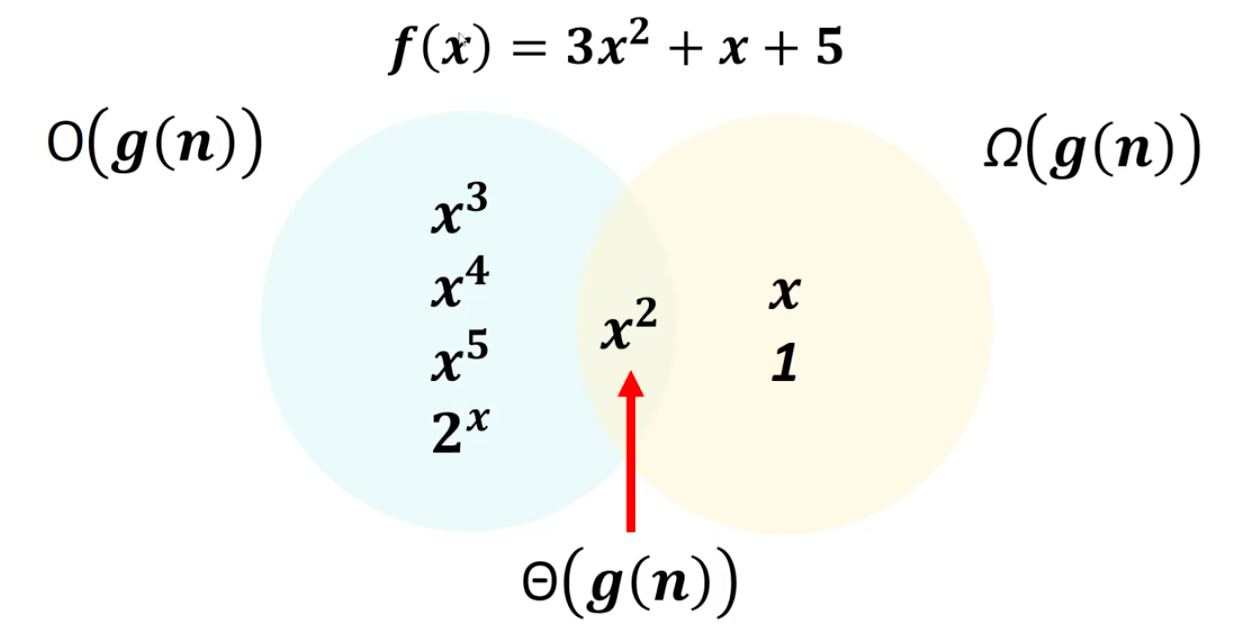
只要 的極限值大於0（ 可取任意大於 0 的小值），則

。

4. 各個符號的比較



舉例來說，如果時間複雜度 ，Big-O 可以取 、、、、、，因為這些函數都是 的上界；Big-Omega 則是 的下界，可以取 、、1 等；Big-Theta 則是上界與下界的交集，最大次方項必須與 中 的最大次方項 的次方相同。



## 第六節：遞迴的複雜度計算

如何計算遞迴式的時間複雜度？

1. 三種計算方法

這裡只介紹三種方法，第四種方法「支配理論」與演算法中的「分治法」有關。

A. 數學解法 Mathematics-based Method

直接以遞迴的觀念算出複雜度

B. 代換法 Substitution Method

猜一個數字後代入看是否成立

C. 遞迴樹法 Recurrence Tree Method

畫出遞迴樹後加總

2. 計算遞迴式的複雜度

計算下列程式碼的時間複雜度：

(1) 數學解法：直接以遞迴的觀念算出複雜度

// 共有 n-1 個 3，且

(2) 代換法：猜一個數字後代入

猜 ，即 ，

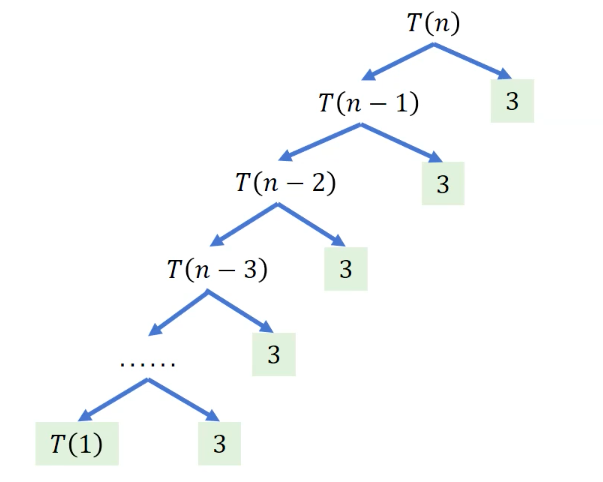
取 ，。

代入檢驗：

// 等號成立

// 等號成立

先猜一個複雜度之後，把數字代進去檢驗，看是否產生矛盾，如果沒有矛盾則得證。

(3) 遞迴樹法：畫出遞迴樹後加總之

把遞迴樹畫出來後，發現每個 都可以拆成兩個部分： 和 3，又 。

得證 。

3. 使用任一方法計算下列程式碼的時間複雜度

推薦：數學解法/遞迴樹

可以被寫成 ， 又可以換成 ，一直換下去，可以換成 ，又 ，所以 ，時間複雜度是 。