# Ch2. 複雜度估算 Complexity

緊接著我們來看我們要如何分析或運算複雜度。

首先我們會說明我們為什麼要分析複雜度以及評估複雜度的方式，再來我們會介紹複雜度裡最重要的符號 Big-O 要怎麼運算與證明，之後，我們會以高中數學學過的「極限」來化簡 Big-O 的證明方式，再來我們會介紹除了 Big-O 以外，還有哪些估計複雜度的符號，最後，我們會來看遞迴（或遞迴式）的複雜度應該如何計算。

在這章中，我們會用到一些數學工具，建議同學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

在上一章中，我們已經有稍微提過為什麼要評估複雜度，因為電腦並非無所不能，雖然電腦運算比人腦運算快很多，但實際上它還是有限制的，在特定的狀況下，即使是超級電腦也無法算出所有答案，這時如果我們有更好的演算法，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，但也不是免費的，特別是在處理圖像或影片的時候，需要耗費許多記憶體空間。

當我們無法直接取得最佳解時，越有效率的算法可以帶我們找到越好的解答。

這就像是在走迷宮時，通常只能看到眼前的路，但是我們如果能夠站得位置越高，就能夠看得越遠，也因此可以找到越好（最有效率）的路徑，讓我們更快走出迷宮。

(1) 如何評估複雜度

在評估複雜度之前，有一些前提條件（Criteria）。我們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，也就是算法是「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，程式碼至少要可以被閱讀，使其之後可以被其他人維護與修改。

進入效能評估（Performance Analysis）的階段後，大抵而言會看兩個方向：「空間複雜度」與「時間複雜度」。空間複雜度通常代表的是一段程式執行時會佔用多少記憶體空間；時間複雜度則代表了它的運算次數與時間。

2. 空間複雜度

(1) 空間複雜度的描述

S(I) = C +

S(I)：需要的總記憶體空間

C：需要的固定空間

：需要的變動空間

如上式所示，S(I)，也就是演算法需要的總記憶體空間，是由兩部分構成的： C 與 。

C 是一個常數，並不會因為輸入的資料量大小而改變，這部分包含了程式碼中的常數（constant）或全域變數（global variable）等； 則會根據輸入資料量 I 的大小而改變，如果輸入的資料量 I 大，就會花費更大的空間，這部分包含了遞迴式的堆疊（recursive stack space）、局部變數（local variable）等，因為呼叫的函式越多，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，所以如果用遞迴式來運算費波那契數，需要的空間大概是取決於 2n。

舉例來說，F(50) = F(49) + F(48)，後面這一項又成立 F(48) = F(47) + F(46)。每個費波那契數都被拆解成兩次對於這個同樣的函式的呼叫，也因此對這個函式的總呼叫次數大約是 2n。

一個函式被呼叫 2n 次，就會產生 2n 個局部變數（在遞迴全部完成前這些變數佔用的空間是不能被釋放掉的），所以所需的總空間大小，就會取決於 2n 這個值的大小。

S(I) = C +

= C + k2n

用空間複雜度的公式來看，我們知道 C 仍然是一個常數，無論我們想得到的是第幾個費波那契數，都必定需要花費這個空間大小，但是 則會被資料量 I 的大小影響，由於我們知道所需空間「大概」等於 2n，所以我們可以加上一個係數 k，讓 和 2n 成正比關係。

3. 時間複雜度

(1) 時間複雜度的描述

T(I) = C +

T(I)：需要的運算總時間

C：需要的固定時間

：需要的變動時間

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C，第二部分則是會因為輸入資料大小而改變的時間 。

|  |  |
| --- | --- |
| 兩種數字 1 到 N 的和的算法 | |
| int sum = 0;  for (int i=1 ; i<=N ; i++)  sum += i; |  |
| ∝ N  T(I) = C +  = C + kN |  |

舉例而言，我們有左右這兩個程式碼，都可以得到 1 加到 N 的整數和。左邊是由 1 開始加 2、加 3、…，一路加到 N；右邊則是公式解，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（迴圈執行 N 次），因此運行的時間取決於 N；公式解需要的時間則不會因為輸入的 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);  } |

至於費波那契數的時間複雜度，因為函式總共會被呼叫 2n 次，所以總共需要的時間一樣是由一個常數 C，加上正比於 2n 的 。

T(I) = C +

= C + k2n

## 第二節：複雜度的估計法

1. 估計複雜度

(1) 估計複雜度的前提

首先，我們會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，這是為了方便我們運算跟統計。在上面的假設下，我們只要統計出總共需要的運算「次數」，看「次數」的數量級的大小，就可以得到複雜度，也就可以評估執行需要的時間。

在我們的假設下，做 10 次運算就會需要做 1 次運算的 10 倍時間，不過實際上當然會有誤差存在。

(2) Step Count Table

|  |  |  |
| --- | --- | --- |
| 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;  } | Steps Frequency Sum of steps  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 |

我們如果設計一個如下的函式，它的功能是可以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 迴圈幫我們把每一筆資料都加到 sum 裡。

這時我們要怎麼計算複雜度呢？首先，我們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頻率，將這兩項相乘，就是這段程式碼運算的總步數（Sum of steps）。

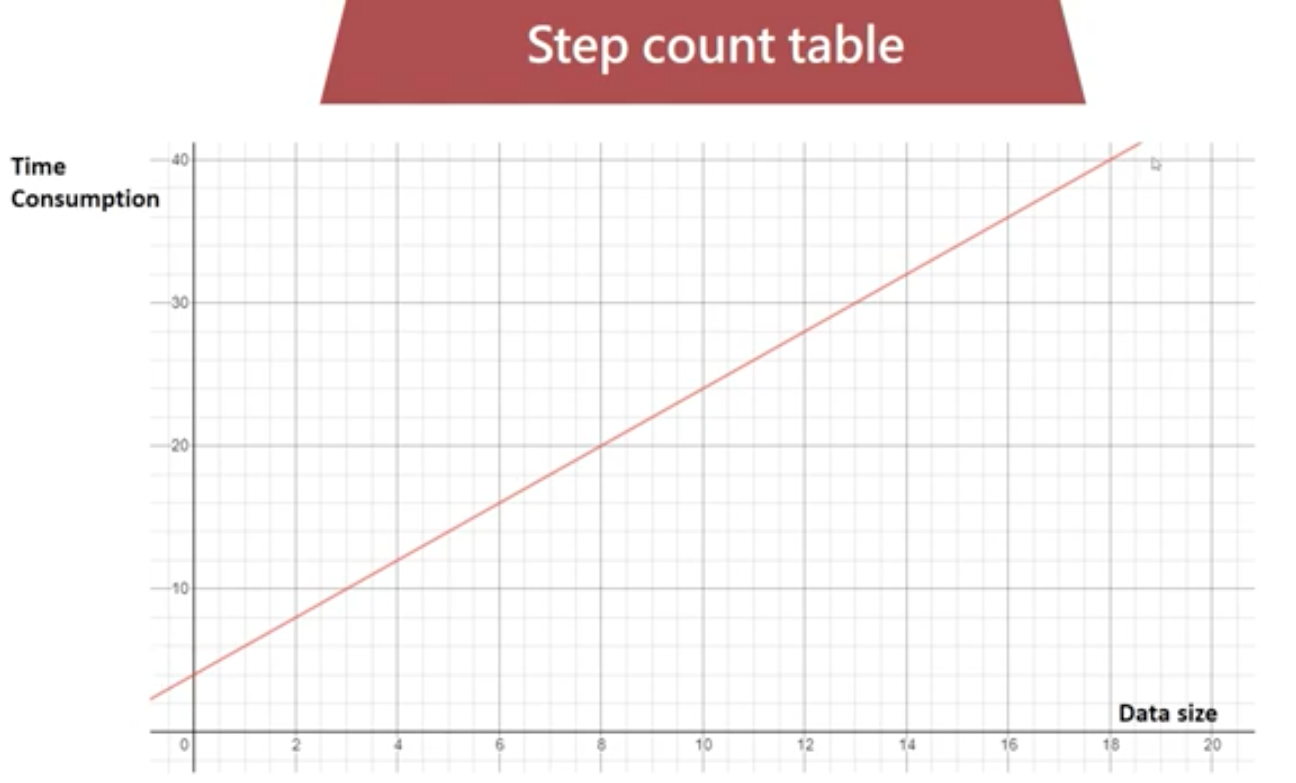
以第三行為例，int sum = 0需要的步數（Step）是 1，且這行總共執行的次數是 1 次，所以這行得到的總步數是 1；第六行的 sum += \*(p+i) 一樣需要一步，總執行次數是 len 次（i 從 0 遞增到 len-1），因此所需總步數為 1 x len = len 步。

其中，注意到第 5 行總共會執行 len+1 次，而非 len 次，因為當 i = len 時，它仍然需要把 i 和 len 比較後，才能決定跳出迴圈往下執行。

將每一行需要的步數加總，我們發現整段程式碼需要的總時間是 2len + 4 次。

我們可以把 2len+4 畫在一個圖表上，橫軸是資料大小，也就是 len，縱軸指的則是所需要的時間，數學是 T(len) = 2len+4 是一個二元一次方程式。

圖中的 y 截距 4 是剛剛講的 C，不會跟著資料量改變； 的部分則會隨著資料量增加而變大，因此需要的總時間與資料量大致呈正比關係。



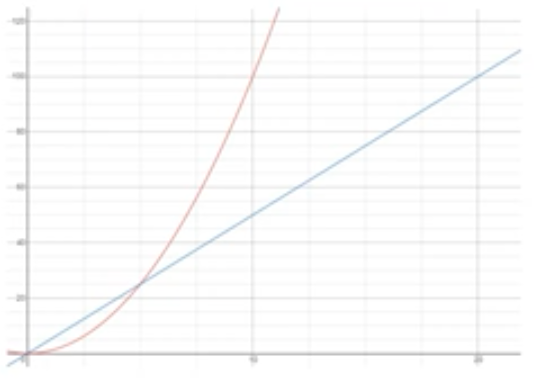
(3) 「小時候胖不是胖」

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，通常我們比較在意資料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，所以我們不做考慮。

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

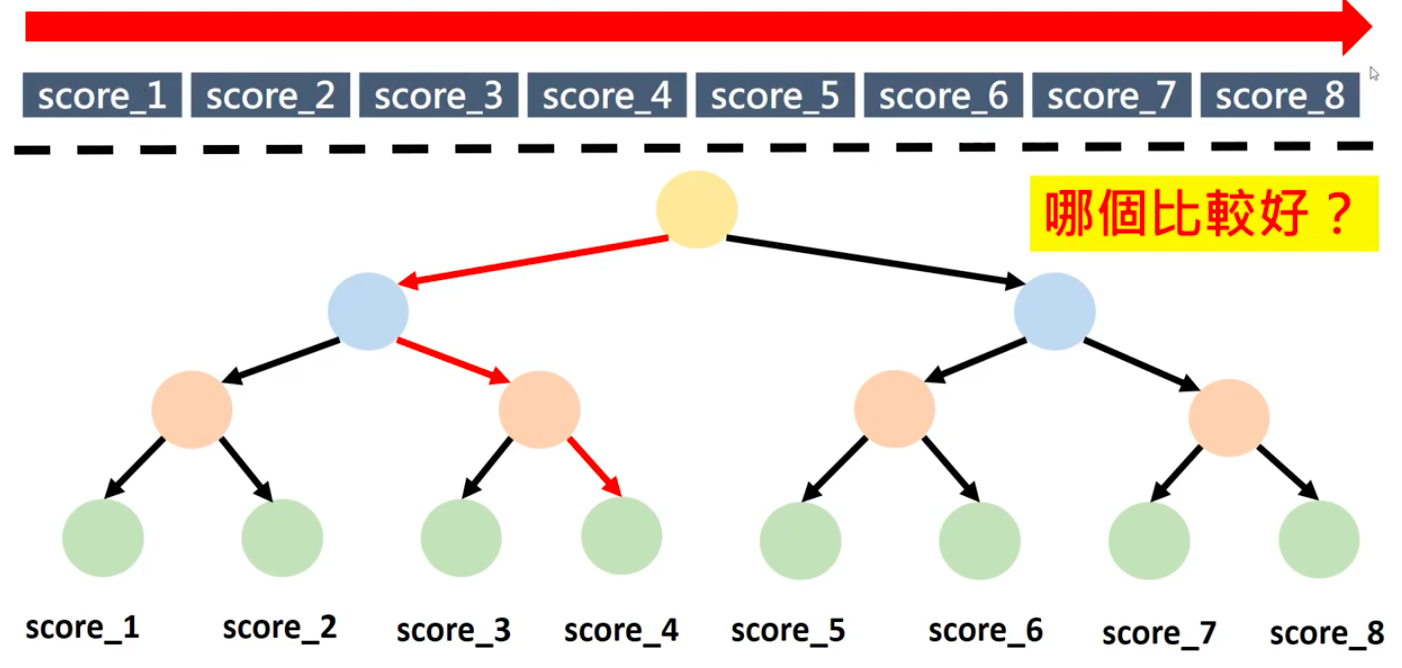
以剛剛看過的程式碼而言，左邊是由 1 加到 N，右邊則是利用公式解。

這兩種方法在資料量小與資料量大時有固定的優劣關係嗎？在極端的情況，n 是 1 的時候，其實右邊比左邊要花更久的時間，因為右邊的算法需要進行乘法和除法，而左邊只有加法而已，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就比較快。



上圖可以做一個類比，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）。因為我們在意的通常是資料量大的時候，所以藍線對應的就是較佳的演算法。

(4) 比較兩種狀況



上面的陣列搜尋裡，我們需要從第一個搜尋到最後一個，這叫「循序搜尋」，下面的方法中，我們則把資料用二元樹的形式儲存，叫做二元搜尋樹，這種方法每次搜尋可以去除一半的可能位置。

上面的方法裡，我們最少需要搜尋 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）；下面的方法裡，則不管怎樣都需要 3 次搜尋（n 次搜尋可以找遍 2n，23 = 8）。哪一種方法比較好，其實沒有一定。

(5) 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？又或是精準度或正確率較高才是我們所關注的？

進行某些工程運算時，我們可以藉由犧牲部分精準度來節省運算時間，我們之後會提到利用「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。因此，「速度快」、「節省空間」、「精準度高」和「開發成本低」之間並沒有哪個一定更重要，我們隨時需要進行「平衡」。

之前我們舉的例子是以抽樣代替普查，雖然抽樣的精準度比普查差，但是這個例子裡時間比精準度來得更重要，7 年才做出的普查已經沒有意義了。在我們的課程裡，由於 CPU 運算資源通常更珍貴，所以大多時候我們看的都是時間複雜度。

(6) 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，具體而言，就是每次運算完後，不要把結果存下來，下次要用到就重新再算一次。

「空間換取時間」的策略則是每次運算完都將得到的結果存起來，下次要用到時直接查表就好，如果表上沒有，則使用內插法。這種做法會產生一個很大的表，佔用許多記憶體空間，但是把表格建出來後，未來就能節省許多運算時間。

因為 CPU 運算資源比記憶體空間珍貴，我們更常使用「空間換取時間」。

(7) 時間 = 資料量 x 複雜度

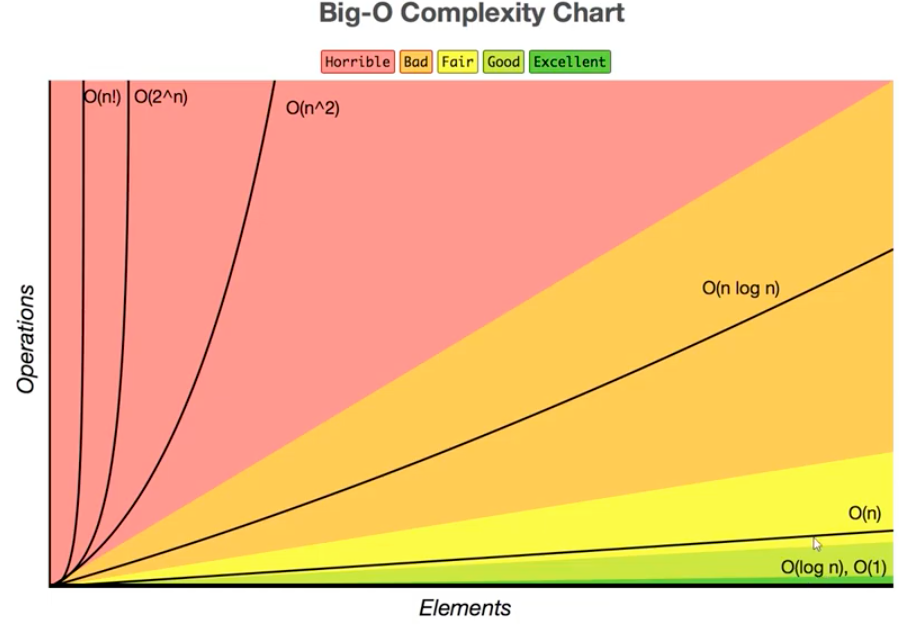
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 資料量 | 1 | N | N2 | N3 | 2N |
| 1 | 1毫秒(ms) | 1 | 1 | 1 | 1 |
| 10 | 1 | 10 | 102 | 103 | 1024 ~ 103 |
| 100 | 1 | 102 | 104 | 106 | ~1030 |
| 1000 | 1 | 103 | 106  (~15分鐘) | 109  (~12 天) | ~10300 |
| 10000 | 1 | 104 | 108  (~25 小時) | 1012  (~30 年) | ~103000 |

從上面的表格裡我們可以看出，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 N2，那麼所需時間就變成 106 倍，如果複雜度是 N3，所需時間則變成 109 倍，這代表運算總次數會隨著 N 改變而有巨幅波動。

假設一次運算需要耗費 1 毫秒（10-3秒），那麼如果有 10,000 筆資料，而複雜度是 N，我們總共就需要花費 10 秒（10 = 10-3 x 104），如果複雜度是 N2，會需要大約 25 小時，N3 需要 30 年的時間，2N 更是根本算不完。

這也是我們描述複雜度時習慣以 N 的次方數來表示的原因。當 N 的次方數增加，只要資料量變大一個數量級，所需的時間就會呈指數成長，差距非常大。

(8) 複雜度的優劣之分



討論複雜度的時候，我們可以用幾種不同的等級來表達。如果複雜度是 O(1) 或 O(log n)，就是相當好的演算法，如果是 O(n)，也就是需要的時間大致上與資料量 n 成正比，那我們認為是「普普通通」。

O(nlogn) 大概是可接受的邊緣，如果複雜度到達 O(n2)、O(2n)，甚至 O(n!)，那麼在資料量大時，幾乎不可能用電腦將結果算出來。評估複雜度就可以大致瞭解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，還有沒有可能用電腦來解決問題。

當我們在做競賽題目時，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，都屬於 O(nlogn) 的複雜度。在寫題目時，我們可以計算一下，檢查自己的答案是不是落在 O(nlogn) 以下，如果超過了，比如複雜度是 O(n2) 或 O(2n) ，可能就代表有問題，少數狀況 O(n2) 可以接受，到了 O(n3) 以上，基本上就無法接受了。

### Example 1：計算下列程式碼的迴圈執行次數

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 << " ";

我們知道最內層的迴圈會優先執行，k 從 0 增加到 M-1，總共執行了 M 次，中間這層的 j 從 0 到 N-1，總共執行了 N 次，最外層的 i 也從 0 到 N-1，總共執行了 N 次。

綜合起來，這個巢狀迴圈總共會執行 M x N x N = M x N2 次。

### Example 2：計算下列程式碼的迴圈執行次數

for (int j=1 ; j<N ; j+=2)

cout << i << " " << j;

我們知道 j 會從 1 開始執行，每次執行迴圈後 j 就會增加 2。如果限制最大值是 N，代表最後 j 會停留在某個 1+2(n-1)，其中 n 是正整數。

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，因此若我們將迴圈目前執行的次數以 n 表示，那麼 j 的值就可以寫成 1+2(n-1)，當然這個 1+2(n-1) 也需要小於 N，才會滿足迴圈的條件而繼續執行。

移項一下，我們就可以得到 。

### Example 3：計算下列程式碼的迴圈執行次數

for (i=M ; i>1 ; i/=2){

cout << i << endl;

}

迴圈從 i = M 時開始執行，i 必須大於 1，每次執行完 i 會除以 2。所以第一次執行時 i = M，第二次執行 i = ，第三次執行時 i = ，第 n 次執行時 i = 。

因為 要滿足大於 1 的要求，所以：

M > 2n-1

。

### Practice 1：計算下列程式碼的時間與空間用量

for (j=0 ; j<M ; j++){

cout << j << endl;

}

for (i=0 ; i<N ; i++){

cout << i << endl;

}

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。

時間複雜度：上面的迴圈中，j 會從 0 到 M-1，總共跑 M 次；下面的迴圈中，i 會從 0 到 N-1，總共跑 N 次，上下加起來，總時間需求是 M+N。

### Practice 2：計算下列程式碼的迴圈執行次數

for (int j=1 ; j<N ; j\*=2)

cout << j;

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

由於 2n-1 < N 才會執行，兩邊取 ，可以得到n < 。

### Practice 3：計算下列程式碼的迴圈執行次數

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 i 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

1. Big-O 的運算與證明

(1) 用 Big-O 來描述演算法的複雜度

我們來看看要如何用 Big-O 來描述演算法的複雜度。描述演算法「工作效率」的函數，稱為複雜度(Complexity)。

演算法會依資料散布情形的不同而有不同的處理次數：

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」。

另外還有平均的情況 (Average-case) 和最好的情況 (Best-case)。最好的情況下需要的處理次數是「下界 (lower bound)」，「下界」對應的時間也就是最少需要的時間。

(2) 搜尋的複雜度

如果要在一個長度為 Len 的陣列裡搜尋我們要的資料，通常會使用「循序搜尋」，從陣列開頭一筆一筆資料確認。

在最壞的狀況下，我們搜尋到最後一筆才找到所需的資料，因此搜尋了 Len 次，這個 Len 就是我們的「上界 (upper bound)」；最好的情況則是第一筆資料就是我們所要的資料，這時只要搜尋一筆資料，1 就是我們的「下界 (lower bound)」。

平均 (Average-case) 來說，我們需要搜尋 次。

(3) Big-O 的理想條件

通常我們會想知道的是某一算法在最壞情形下會跑多久，也就是最差的狀況下需要多少時間，這樣我們才能事先把時間準備好。因此我們通常在意的是 Worst-case，或是資料增加時，所需時間的成長幅度，也就是Growth rate。

另外我們希望複雜度不會受到單位的影響，不管我是用 Byte 還是 Bit 為單位，或者時間上我用秒、分、小時為單位都相同。

最後，我們不在乎小資料時的狀況（也就是「小時候胖不是胖」），小資料差了幾毫秒我們並不在意。

(4) Big-O 的數學定義

將上面提到的要求結合起來，我們就能構造出 Big-O 的數學定義：

[ 存在 c>0、，使得所有 都滿足 ]

其中：

c 是一個常數，以它作為係數可以使我們可以不受單位影響

代表資料量大於 的時候

如果 ，那麼某個「所需時間可用 f(n) 來表示」的演算法，它的複雜度就是 O(g(n))。也就是說，存在一個 c > 0 和 ，使得對於所有的 來說，都滿足 。

要確保 ，我們要找到一組 c 跟 ，使得資料量 n 夠大（超過 ）的時候， 總是成立。

(5) 實際使用 Big-O

如果我想知道 5x 能不能寫成 O(x2)，我就要試著找到一組 c 和 ，使得當資料超過某個量， 時，都有 ，c 可以取我們想要的值。

我們的目標是找到某個情況下（某個 c 和 x > 時）， 總是成立：

A. 把等號左右的 x 約掉（因為 x 是資料量，必定是正數），

B. 取 c = 1（不是一定，c 可以任意求解），同時取 = 5

D. 當 x > 時， 總是成立

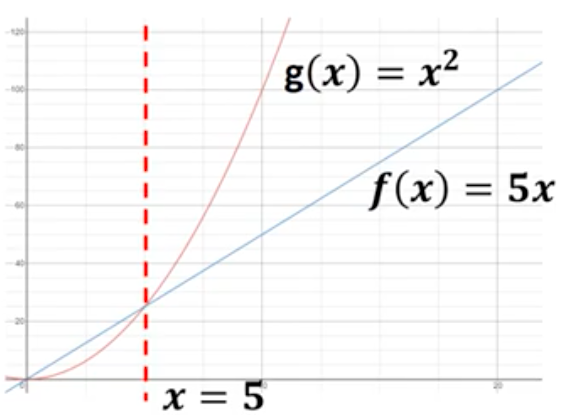
E. 得證

用數學方式表達：給定f(x) = 5x、g(x) = x2，欲證有一組 c>0、，使得對所有 n > 時滿足 。

取 c = 1,

∀x > 5,

因此 ∈ O(g(n))，在這個例子裡指的就是，x ∈ O(x2)。



我們把 f(x) = 5x 和 g(x) = x2 畫在上圖中，紅線是 x2 需要的時間，藍線是 5x 需要的時間。

當 x > 5，意即資料量超過 5 筆時，藍線都低於紅線，5x < x2。

Big-O 相當於「上界」， ∈ O(g(n)) 也可以說成 g(n) 是 f(n) 的上界。在這個例子當中，x2 是 5x 的上界，在資料量超過 5 筆的時候，不管資料量再怎麼大，5x 需要的時間都小於 x2。

Example 1：證明或否證 5x ∈ O(x)

證明過程：

A. 取 , c = 6

B. ∀x > 0, (這個式子就是∀x > 0, )

因為我們找到一組 ， 滿足 Big-O 的條件，所以 。

Example 2：證明或否證

證明過程：

A. 我們要找到一組解滿足 (也就是 )

B. 等號左右同消掉 x，

C.

只有在 的時候才會成立，所以無論 c 取多少，都不能找到對應的 （ 的時候就一定不會成立），所以 。

Example 3：證明或否證

證明過程：

A. 我們要找到一組解滿足

B. 取 c = 100，

C.

D.

E. ，

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

這裡我們是用 Big-O 定義來證，似乎有點麻煩，等一下我們會介紹另一種更簡便的方式。

### Practice 1：證明或否證

A. 找一組c、，使

B. 取 c = 3,

C.

D. 得證

### Practice 2：證明或否證

A.

B. 兩邊平方，

C.

D. 時一定不成立

E. 不管 c 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

(1) 另一種觀點

從定義上證明 big-O，就是找到一組 c>0、，使得資料量大於 的情況下，都一定有 ，也就是 是 的上界。

--- (1)

--- (2)

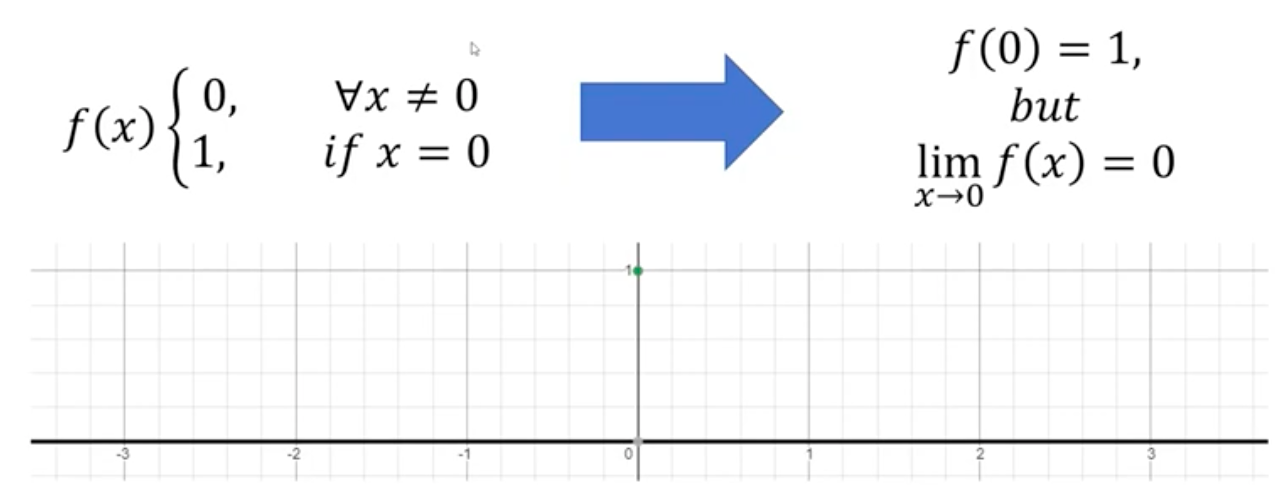
但要找到這樣的 c>0、 不一定很容易。觀察 (2) 式，將不等號左右同除以 ，就可以得到下面的 (3) 式。

--- (3)

因為 (1) 的條件中要求對「所有」大於 的 n 而言，(2) 與 (3) 的不等式都應該成立，我們可以直接將 n 推到無限大來進行確認。如果下面的 (4) 式成立，也就是 n 接近無限大時， 收斂到一個常數 c，那麼我們就知道 。

(2) 極限的複習

我們很快複習一下高中數學中的「極限」。如果 f(x) 在 x 逼近 a 時的極限為 L，可以寫成 ，這就是極限的定義。



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

這告訴我們雖然 f(0) = 1，但是 x「趨近於」0 的情況下，因為再怎麼接近 0 也不會真的到達 0，所以 f(x) 在 x 趨近於 0 時的值仍然是 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。意即 f(x) 和 g(x) 之間運算的極限，可以先將 f(x) 與 g(x) 個別的極限值 L、K 取出之後進行相應的運算。

A.

B.

C.

D.

E.

特別注意 E 式， 會使得 ，講白話一點就是「無限大分之一 = 0」。

Example 1：請計算

上下同除以 n，

因為其中所有形如 (c 是常數) 的值都是 0，所以，

透過這種方法，只要我們計算出 的極限值，看它是不是無限大，就可以決定是否 f(n) ∈ O(g(n))。比如令 f(n) = 3n+2，g(n) = 2n+5，因為 ，所以 f(n) ∉ O(g(n))。

Example 2：證明或否證

因為 的值收斂到 5，而非無限大，得證 。

Example 3：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 。

Example 4：證明或否證

∃c > 0,

因為 ，即 等於無限大，並不會收斂，所以。

**Practice 1：證明或否證**

得證 。

### Practice 2：證明或否證

(⇒)

假設左邊成立，，即 x2。

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

我們來看看除了 Big-O 以外，還有哪些符號可以拿來表示複雜度。

1. Big-O 存在的問題

Big-O 是上界，也就是最壞的情況，這代表把 Big-O 裡面任意放一個非常大的數字，f(x) ∈ O(g(x)) 都會成立：

f(x) ∈ O(x2)

f(x) ∈ O(x2+x)

f(x) ∈ O(3x2+2x)

f(x) ∈ O(x3+1)

f(x) ∈ O(2x4+x)

f(x) ∈ O(x5+x2+2)

...

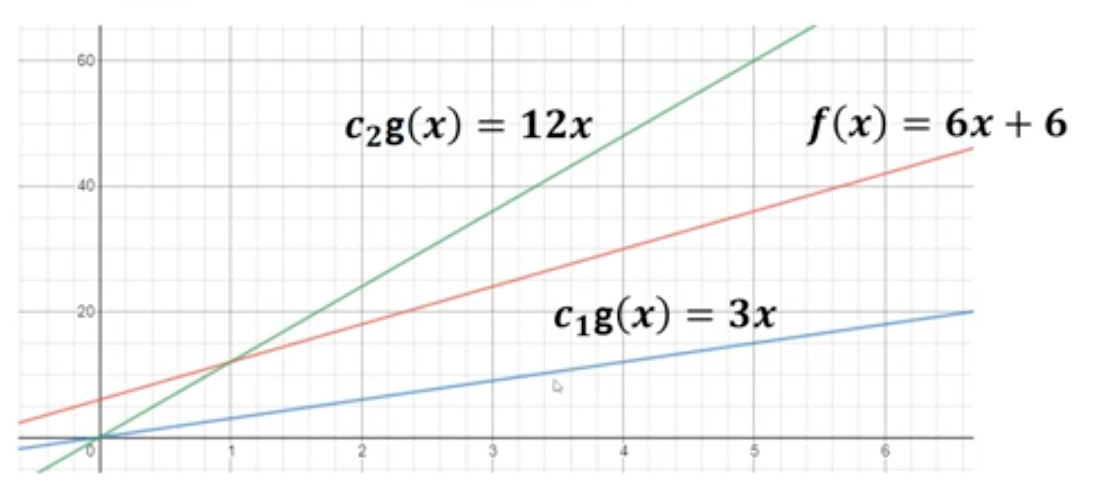
這就像如果媽媽和你說：「只要你比任一個學生成績好，我就給你獎金」，那麼我就去和全班甚至全學年成績最差的同學比，很容易就可以達成。

只要 f(x) ∈ O(x2)，f(x) 也就可以寫成 O(x3)、O(x4)、O(x5)、…，繼續把 Big-O 中 x 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f(n) 會被 和 上下包夾在一起，g(x) 在乘以兩個相異常數後，會使 f(x) 介在其間。



是否成立？

當我們取 g(x) = x、兩個常數 與 分別為 3 與 12，由上圖可以看出紅線 f(x) 在 x>1 時被包夾在藍線 和綠線 之間，因此 明顯成立。

(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)：

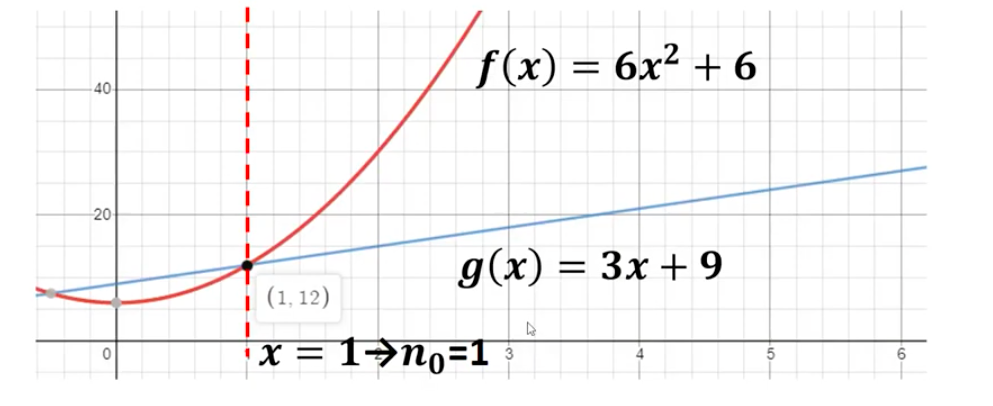
只要 的極限值不是無限大而且大於 0，則 。

注意：g(n) 只跟 f(n) 的最大次方有關。

3. Big-Omega Ω

(1) Big-Omega Ω 的定義

即 代表 g(x) 是 f(x) 的下界 (Big-O 則是上界)。



是否成立？

取 ，藍線 g(x) = 3x+9 都小於紅線 f(x) = ，因此 成立。

(2) 使用極限證明 Big-Omega Ω

---(1)

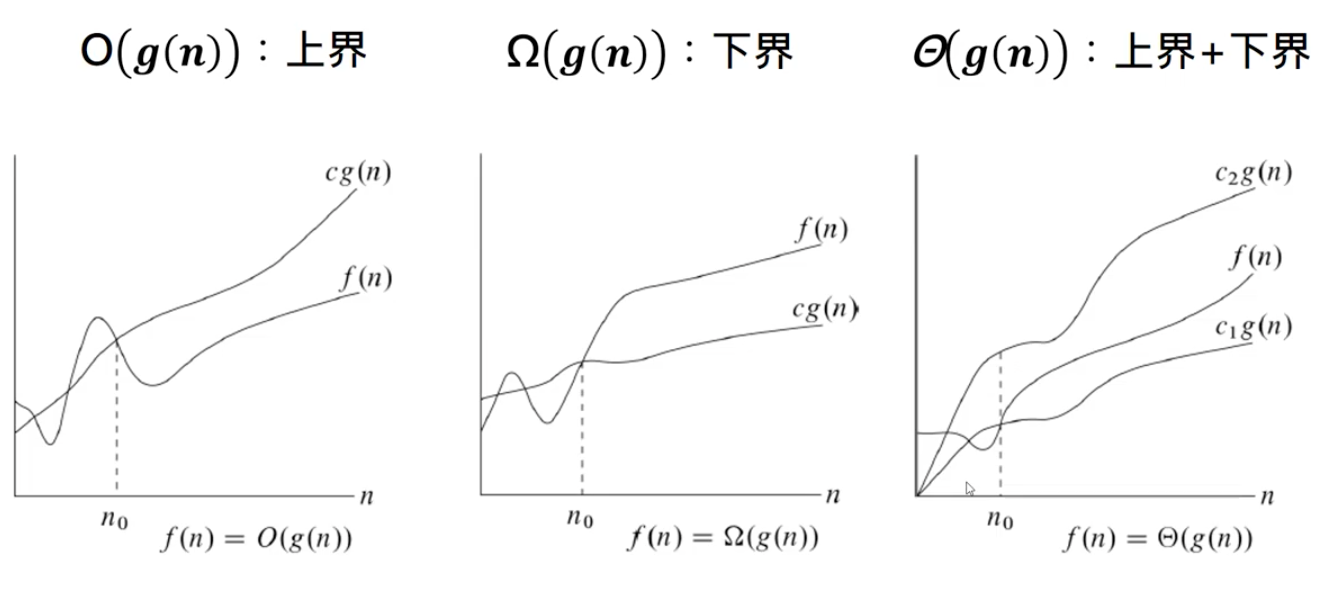
---(2)

上面 (2) 式中各項同除以 g(n)，

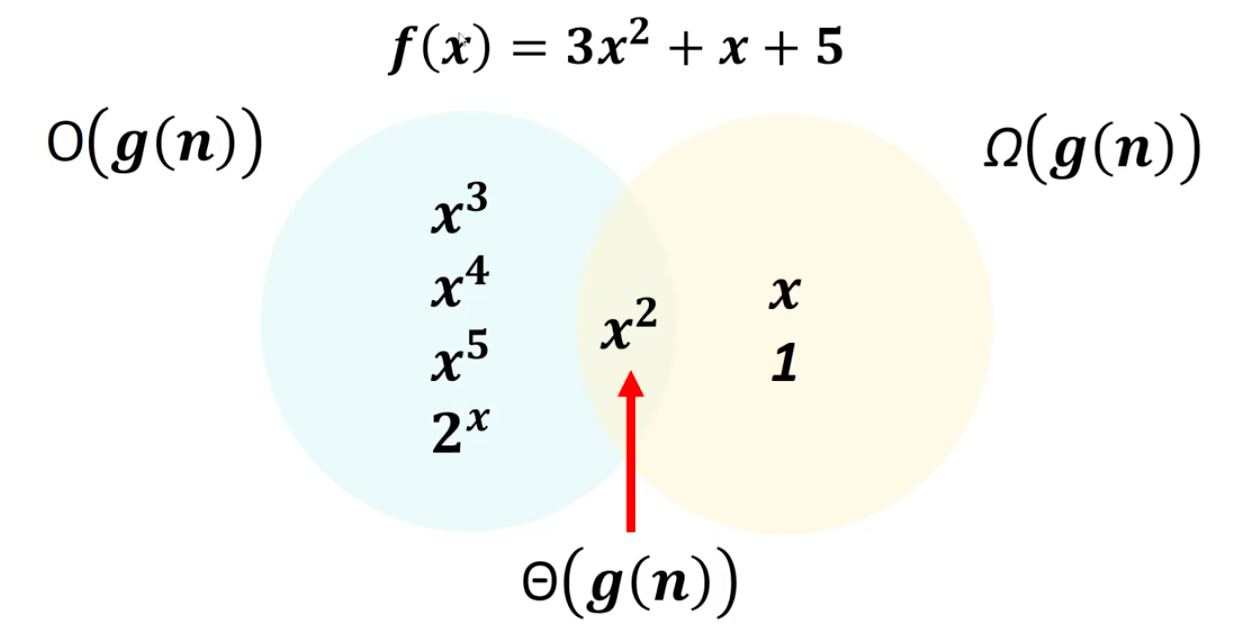
只要 的極限值大於0（c 可以取任意大於 0 的小值），則

。

4. 各個符號的比較



舉例來說，如果時間複雜度 f(x) = 3x2 + x + 5，Big-O 可以取 x3、x4、x5、2x 等等，因為這些函數都是 f(x) 的上界，Big-Omega 則是 f(x) 的下界，可以取 x、1 等，但 Big-Theta 則是上界與下界的交集，最大次方項必須與 f(x) 中 x 的最大次方項 x2 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

這裡先只介紹三種方法，之後介紹分治法後會再介紹第四種方法。

A. 數學解法 Mathematics-based Method

直接以遞迴的觀念算出複雜度

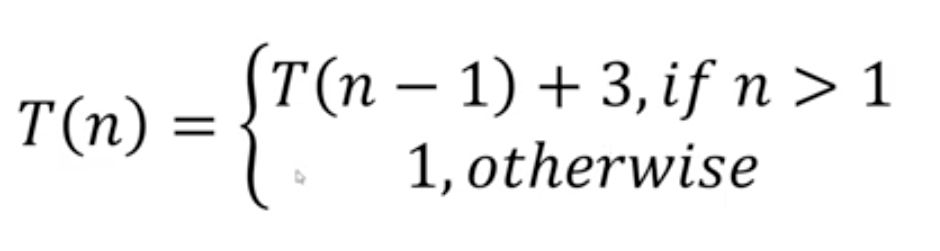
B. 代換法 Substitution Method

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

T(n)

= T(n-1) + 3

= T(n-2) + 3 + 3

= T(n-3) + 3 + 3 + 3

= ...

= T(1) + 3(n-1) // 共有 n-1 個 3，且 T(1) = 1

= 3n - 2

T(n) = 3n-2 ∈ O(n)

(2) 代換法：猜一個數字後代入

猜 ，即 T(n) = cn，

取 c = 3，T(n) = 3n。

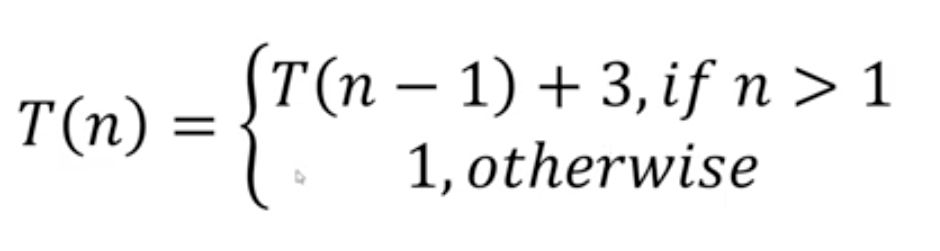
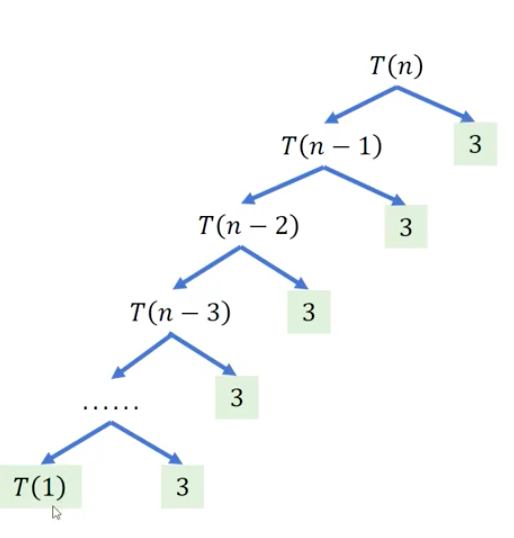
代入檢驗：

// 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 T(n) 都可以拆成兩個部分：T(n-1) 和 3，又 T(1) = 1。

T(n)

得證 。

Practice 1：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

T(n)

= 2 T(n-1)

= 22 T(n-2)

= 23 T(n-3)

= ...

= 2n T(0)

= 2n ∈ O(2n)

T(n) 可以被寫成 2 x T(n-1)，T(n-1) 又可以換成 2 x T(n-2)，一直換下去，可以換成 2n x T(0)，又 T(0) = 1，所以 T(n) = 2n，時間複雜度是 O(2n)。