# 1

# 基本概念

簡單地說，電腦之所以能解決許多人類生活中的許多問題，無非是其能儲存大量資料與其能快速處理資料的緣故。然而對資訊相關科系的同學而言，這樣的說法需要更深入的探討—資料如何儲存在電腦中？而電腦又如何快速處理這些資料？不同形式或需求的資料，他們儲存的方式是否會不一樣？處理這些資料的方式是否也會不一樣？該如何判定處理資料的方法夠不夠好？這些問題其實就是資料結構和演算法課題的濫觴。電腦學家透過程式語言，驅使電腦硬體將資料做最妥善的呈現，進而以最有效率的方法予以處理，靠的就是資料結構和演算法的訓練。

## 1.1 資料結構、演算法

隨著文明與科技的進展，人類和資料之間的關係與互動也愈加地頻繁。在資料量少的時候，我們可以在腦中記憶並計算、存取這些資料；但是當資料量大時，人腦的容量和反應是很不穩定的機制。聰明的人類即利用各種不同的工具來存放資料，譬如說筆記本、手機、PDA (personal digital assistant)、錄音筆、…等等，或者有像算盤、計算器之類的工具，來協助人們解決在計算求解上的需求。這些工具的使用無非是希望能夠輔助或代替人腦，取得更大的儲存空間和更快的處理速度。當這樣的需求持續膨脹時，也就是說資料持續的增加，或是想要解決的問題需要更大量的運算，甚至兩者皆然，電腦大概就是最好選擇了。電腦要如何存放這些資料？有什麼呈現方式 (data representation)？使得這些資料在引用時可以很快地取得；要如何運算這些資料？使得求解的過程可以正確迅速，這是電腦學家該下的功夫。「資料結構」(data structurs) 所探討的，就是在電腦中有效率地存放資料，使其方便處理的學問；而探討解決問題的策略，就開展了「演算法」(algorithm) 這樣的一門學問。

## 1.2 資料結構與演算法

在這裏我們將電腦處理的對象稱爲「資料」(data)，也就是指所有輸入至電腦中，即將、正在或已經被電腦程式處理的符號總稱；包括了數值 (numerical) 資料、字串 (string) 資料等等，也包括多媒體 (multimedia) 軟體所處理的影像、聲音、視訊等多媒體資料。

當這些資料集合在一起時，會因處理需求的不同而存在一種或多種彼此之間的特定關係，這些資料之間的邏輯關係，就稱爲結構 (structure)。研究資料的邏輯關係，探討這些邏輯關係在電腦中的表現 (representation) 和處理 (manipulation) 方式，即爲資料結構。在計算機概論或電腦概論，甚至在數位邏輯 (digital logic) 的課中，我們知道資料在電腦中是以 0 和 1 的組合來表示；不同的資料型態 (data type)，其組合表示的形式會因而不同；使用不同的電腦硬體，其組合表示的形式也會有所不同；這個層次在探索資料本身與電腦硬體之間的諸多關係。然而爲解決實際生活中的許多問題，資料之間的邏輯關係會因問題需求的不同、或處理效率的考量而可能重新組織，於是資料結構著重的是問題需求和資料之間關係的各種課題。

在此我們提及「資料結構」時，事實上包含著「資料結構與演算法」—在透過資料結構解決問題時，與演算法之間確實有緊密結合的關係。我們以資料結構的探討爲主軸，輔以因需求而產生的必要演算法（在資訊相關學系中，解決問題的方法重要到有演算法的完整課程加以探討，屆時問題與演算法的關係即爲主軸）。至於資料的邏輯關係以及與問題之間求解之道，我們靠程式語言 (programming language) 做爲中間的橋樑，事實上問題要在電腦上解決，少不了用程式語言將腦中解題的演算方法，撰寫成軟體程式，予以實作 (implement) 執行。而電腦硬體的特性，則不是本書想涵蓋的內容。書中討論的各種資料結構和演算法，在所有類型的電腦上皆能透過適當的程式語言，實作出來並正常運作。

本書選用的程式語言爲 C 語言，我們認爲 C 語言的一般性 (generality)、可攜性 (portability)、提供足夠的資料型態[1]…等特色，十分符合資料結構使用上之所需。其對資料結構的掌握與運用，恰如其分，不會太少而不敷使用；也不致龐大到模糊了學習的焦點。在說明演算法時也用類似 C 語法的型式來表示。程式或演算法的表示以精簡爲原則。若要將書中討論的程式，以其他程式語言改寫，也不至於太過困難[2]。

## 1.3 簡單的資料結構

我們可以先看看資料之間常見的可能邏輯關係：

**範例 1-1**

(1) 校園中有一群人；

(2) 每個中華民國國民皆有唯一的身分證字號；

(3) 每位同學在班上皆有唯一的座號、在學校皆有唯一的學號；

1 C 語言提供了基本資料型態 (fundamental data type)：字元 (character)、整數 (integer)和不同大小的浮點數 (floating-point numbers)；和結構化資料型態 (structured data type)：陣列 (array)、結構 (structure)、聯集 (union) 和指標 (pointer)。在變數存活區間 (scope)、函數 (function) 和程序 (procedure)、參數 (parameter) 的傳遞包括傳值 (pass by value) 和傳位 (pass by address)、遞迴 (recursion) 等等技巧的提供，正符合資料結構使用上之所需。

2 以物件導向 (object-oriented) 的概念撰寫程式，是頗符合當代程式設計理念的做法；以視窗化程式語言來撰寫程式，也滿足個人電腦上程式寫作的訓練；所以作者認為 C++或 Java (C++ Builder 或 J Builder) 也是不錯的選擇。然而在課程安排的考量上，我們希望將內容重心放在資料結構本身上，降低因物件導向設計或視窗程式設計的引入，而模糊學習焦點的疑慮。

(4) 在大學中的科系有系別、年級別、甚至於班別；

(5) 排隊的時候，我們可能只關心排在前一位的是誰？有時可能還在意排在自己前面的共有多少人？

(6) 出國旅行時，我們會想知道各景點間是否有班機直飛？

(7) 上網尋找資料，網頁間的鏈結，將網頁連結成複雜的全球性的網絡 (world-wide web)。

範例 1-1 列出了幾種生活中常見的資料相互關係，各位應該不會陌生：面對範例 1-1 (1) 的描述，各位應可知道在該描述時段有些人在校園裏。範例 1-1 (2) 的資料利用編號 (身份證字號)，將個別的各式資料以單純的數字予以對應，可方便識別 (identification)，也賦予順序性 (sequential order)，方便排其次序、搜尋比對。由範例 1-1 (3) 可見相同的資料可能因不同的需求、場合或應用，有不同的編號方式。範例 1-1 (4) 強調了資料間有階層性 (hierarchical structure) 存在，像家譜、公司組織架構、成品與零件的構成關係等資料都有類似的性質。範例 1-1 (5) 中的排隊例子各位應也都有經驗，在不趕時間排隊時可能還會在意共有多少人等候買票。要回答範例 1-1 (6) 的問題，可能就得查對各航空公司的飛機航線，如果畫出手邊飛機的航線，那麼應會畫出如範例 1-1 (7) 的網絡圖形；範例 1-1 (7) 顯示了資料的關係即令單純，也可能因其是多對多的關係而相當複雜，甭論要同時考量多種關係。

這些生活中的實際應用和處理經驗，該如何在電腦中表達呈現？正是本書想傳達的核心資訊。各位曾經有的學習經驗，包括數學、代數、計算機概論、程式語言、邏輯等學問，都對資料結構與演算法學習有幫助。在後面的章節中各位將會瞭解，範例 1-1 (1) 的資料可能用集合的概念，即可回答相關延伸的問題 ( 如甲：在不在校園中？共有多少人？ )；範例 1-1 (2) 和 (3) 的資料可以用陣列來組織，是典型的線性結構 (linear structure)；範例 1-1 (4) 的資料可以用樹狀結構 (tree) 來表示，而該樹可能用陣列形成的鏈結串列

(linked list)、或以動態指標構成的串列來組成；範例 1-1 (5) 的前項需求可以透過串列而成的佇列 (queue) 來存放資料而達成；而後項需求可加入額外的計數資訊來回答；範例 1-1 (6) 和 (7) 的資料可以利用圖形結構 (graph) 加以組織。這些不同的資料結構皆有可能的應用以及限制，所以它們的適用時機、適用範疇、搭配的演算法、效率如何等主題，都是研究的重點。在此提醒各位，範例 1-1 裏的資料都已存在多年，除了上網是廿世紀方出現的生活型式外，其他的資料都已深植現代人的生活中，資料結構與演算法的學問無非在將人類解決問題的智慧與經驗，透過電腦發揚光大，使人類追求更多、更快的理想得以更上層樓。

## 1.4 演算法初探

簡單地說「演算法」(algorithms) 就是解決問題的方法，然而這樣的解釋太過抽象，在韋氏詞典 (Dictionary by Merriam-Webster) 中，algorithm 的解釋爲：『在有限步驟，解決數學問題的程序』 (a procedure for solving a mathematical problem in a finite number of steps)。在資訊科學的領域中，演算法則強調透過電腦來執行解決問題，而且電腦的應用早已脫離解決數學問題的範疇，而所謂的有限步驟也有更具體的衡量方式；在此我們已提到幾個詮釋演算法必要的關鍵詞：電腦、有限步驟、解決問題；至於其正式的定義，在此我們沿用 Horowitz, Sahni 和 Mehta 在其資料結構的經典鉅作[3]中，給予演算法的明確定義：

3 Ellis Horowitz, Sartaj Sahni and Dinesh Mehta, *Fundamental of Data Structures* in C++, 1995, W. H. Freeman and Company, NY.

**定義**：演算法是一組可完成特定工作的指令集合，並且所有的演算法都需滿足下列條件：

(1) 輸入 (input)：可以有多個或甚至是沒有輸入；

(2) 輸出 (output)：至少產生一個輸出；

(3) 明確 (definiteness)：每個指令都是清楚而明確的；

(4) 有限 (finiteness)：在任何情況下，如果逐步追蹤演算法的每個指令，演算法會在有限的步驟內結束；

(5) 有效 (effectiveness)：原則上每個指令都需基本到只需紙和筆即可實踐之，並且每個指令的運算不止得如條件 (3) 般明確而已，還必須是可以正確求解的 (feasible)。

在計算理論 (computation theory) 的領域中，對演算法即有如上的嚴格要求，才方便深入探討電腦在計算能力上的強度與限制。而在資料結構中，也有分析演算法效率的需求，所以我們引用這個嚴格又有彈性的定義。

輸入和輸出除了給予電腦指令集合更明確的組成必要條件之外，演算法的完整還包括須對所有的或刻意限定的輸入皆有適當的輸出。條件 (4) 的有限要求，也提供了演算法和程式 (program) 之間的分野—程式並不要求必須在有限步驟內結束（你可能不小心就寫出個不會結束的迴圈，使得程式不會停）；像作業系統 (operating system) 就是一個不停止的系統程式，我們不會用演算法來描述整個作業系統（當然作業系統會包括許多演算法）。在實體呈現上，程式一定是利用程式語言撰寫而成，可以在編譯後執行 (execution)；而在上面的演算法定義中，並沒有要求演算法必須可在電腦中執行，但是一定得明確、有限而且有效可正確求解。

除了演算法的定義外，我們還得想想兩個問題：

(1) 該如何表示演算法？

(2) 什麼是好的演算法？

演算法既然強調解決問題指令的明確、有限和效率，其實可以用自然語言或「虛擬碼」(pseudo code) 來表示，只要明確、在有限的時間內會停止、可用紙筆模擬，確實可行、對應的輸入皆找得到正確解答即可。然資料結構與程式語言的關係實在太過密切，所以如 1.2 節所述，本書的演算法大多已寫出對應 C 語言的程式碼—以程序 (procedure) 或函數 (function) （在本書中這兩個詞是同義的）型式呈現，若干需要集中思緒在解決問題邏輯上的演算法，則用類似 C 語言語法的虛擬碼來表示。至於演算法的好壞程度，則透過分析演算法的技巧來予以衡量，我們在 1.5 節中會討論如何分析演算法。

## 1.4.1 程式撰寫的流程

思考問題的解決方法是個抽象的思維過程，電腦程式的輸出則是具象的結果呈現；兩者之間的所有過程，都是成爲程式設計師必須經歷的磨練。資料結構與演算法的學習，正是必要的磨練之一。我們把上述的解題流程，用下圖來簡化描述之：

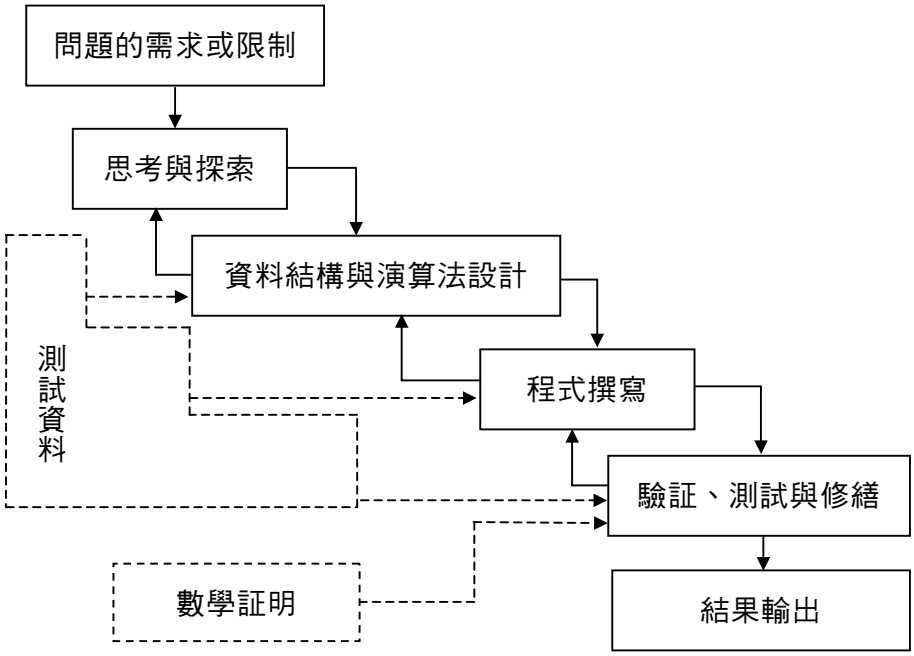



圖 1-1 撰寫程式解決問題流程圖

圖 1-1 是撰寫程式以解決問題的簡化流程，實線方塊表示行動，向下的流程實線表示向下一階段的行動演進，向上的流程實線表示修正上階段考慮欠周全的地方。橫向的流程虛線表示測試資料的流向，上小下大的階梯狀虛線方塊，其面積大小與同階層的行動所需的測試資料量呈正相關。在上下往來流程中，可能導致程式寫作的過程變成迴圈而無法順利完成。在思考與探索、資料結構與演算法設計甚至於程式撰寫的行動階段，可能只是紙筆作業；測試的資料也許只是若干具代表性的資料，但這些階段的考量若能儘量完善，將對程式的正確有極大的保障。「將目前階段的行動考慮周全，方進入下一階段」是加速程式完成的不二法門！驟然上機，邊想邊改的程式寫作行爲，實爲初學者之大忌。

下面我們以將 *n* 個整數由小到大排序的問題爲例，說明撰寫程式來解決此問題的過程。

## 範例 1-2

**思考與探索：**

欲將整數由小至大排序，可把數字小者放在左邊，數字大者放在右邊，…。

- 可以挑出所有資料中最小者，做爲左邊第一筆資料，接著再挑出剩下資料中最小者，放在左邊做爲第二筆資料，依此類推，直至全部資料都排列完成。
- 若所有資料共計 *n* 筆，則會執行 *n* 次「挑出最小」的運算，其第 *i* 次的運算，即爲挑出未排序資料中最小者，其結果則做爲第 *i* 筆資料。

**資料結構與演算法設計：**

輸入的 *n* 個整數可考慮用陣列 `data` 予以儲存，上述解決排序問題的思維，可整理成下面的演算法；因基本運算爲挑出最小者，命之名爲「挑選排序法」(selection sort)。演算法的描述用的是類似 C 語言的語法（虛擬碼）。

**演算法 1-1 挑選排序法：**

```
輸入：data[0], data[1], data[2], ..., data[n-1],共 n 筆整數資料
輸出：data[0],data[1],...,data[n-1]; 其中
     若 i<j,則 data[i]≤data[j], 0≤i,j≤n-1
1  for (i=0; i<n; i++)
2  {   data[j] = 挑出 data[i]至 data[n-1]中最小者;
3      swap(data[i], data[j]); // 將 data[i]和 data[j]對調
4  }
```

上述演算法可用紙筆，代入若干測試數字，檢驗其正確性；俟正確無誤，方進入程式撰寫階段。注意：這以虛擬碼表示的演算法旨在表達解題思維，尚無法通過 C 編譯器的語法檢查。

**程式撰寫：**

我們將上面的演算法，以C程式語言改寫成下面可正確執行的程序或函數。

**程式 1-1 挑選排序法**

```
1  # define SWAP(x, y, t) (t=x, x=y, y=t)
2  void SelectionSort(int data[], int n)
3  {   int i, j;
4      int min, temp;
5      for (i=0; i<n; i++)
6      {    min = i;
7           for (j=i+1; j<n; j++)
8                if (data[j]<data[min]) min = j;
9           SWAP(data[i], data[j], temp);
10     }
11 }
```

程式 1-1 第 1 行定義了一個巨集 (macro) —其爲一連串指令的集合，被呼叫時其巨集變數會由所傳入的變數取代，而巨集指令隨而執行[4]—於是第 8 行 `SWAP(data[i], data[j], temp);` 被呼叫時，下面三行即會取而代之：

```
temp = data[i];      // t=x
data[i] = data[j];   // x=y
data[j] = temp;      // y=t
```

所做的事正是交換 `data[i]`與 `data[j]`的內容。第 2 行參數中的 `data` 爲欲排序元素所在的整數陣列，而 `n` 爲其欲排序元素之個數。

再次提醒：在上機實作之前，用紙筆代入若干測試數字，檢驗其正確性。俟所測資料皆正確無誤，再上機編譯執行之，並記得測試足量的資料。

**驗証、測試與修繕：**

爲求程式正確無誤，以數學的方法予以証明是最佳的驗証[5]。然而當程式龐大到某個程度時，數學的証明會相當困難；此時使用大量資料測試之，是絕對必要的。若此爲提供他人使用的程式，還應考慮輸入正確性 (input validation)；若資料量非常大，則資料儲存用的陣列，宜用動態配置的技巧宣告使用；若資料對調在此以 `SWAP` 巨集定義並呼叫執行以簡化程式，另可考量寫成一程序（命名爲 `swap`，欲對調的兩項資料以指標參數傳入，在 `swap` 程序中進行對調，後文程式 2-3 會介紹）。這個驗証、測試與修繕的階段，是軟體開發的重

4 巨集所定義的變數不必宣告（常以大寫字母命名，指令以′,′相隔…請留意其定義方式—不同編譯器可能會不同），巨集的呼叫可視為「將該連串指令複製進入程式中」（其變數以傳入的變數取代）；於是巨集的變數不受資料型態的限制。以此 `SWAP` 而言，整數、浮點數、字串…變數皆可呼叫 `SWAP`。

5 驗証的階段可以獨立出來，並上移至資料結構與演算法設計與程式撰寫階段之間。

要過程，在系統分析、程式設計、軟體工程等課程中皆會專文探討。在本書中我們會針對若干演算法或程式，討論驗証的技巧；至於測試與修繕，則不在本書所討論的範圍。

在系統分析或軟體工程的課程中，類似圖 1-1 的流程關係還會出現，因其探討的範疇不同，而有不同的行動內容。各位應掌握學習資料結構與演算法的心得和經驗，這些絕對是學好其他資訊課程的基礎。

## 1.4.2 遞迴演算法

以「結構化程式設計」(structured programming) 或「物件導向程式設計」(object-oriented programming) 的觀點來看，使用程序[6] (procedure) 可以增加程式的可讀性和正確性，對程式碼再利用 (reuse) 和除錯 (debug) 時的輔助也有相當的貢獻。一段程式邏輯一旦形成程序或函數，此程序／函數名稱及其參數即可視爲新的指令，可在程式其他地方引用。程式的流程將在呼叫程序 (procedure call) 的同時，移至該程序處，俟該程序執行完畢，則回到呼叫處的下一行—此流程轉換的正確性由系統堆疊 (system stack) 存放必要資訊而達成[7]—如圖 1-2 所示。

6 在 C 語言中，「函數」(function) 即是最常用的程序，在本書中兩者是同義的。

7 呼叫程序時，呼叫者的呼叫參數、返回地址、區域變數…等必要資訊會存入系統堆疊中；俟所呼叫的程序執行完畢，系統會利用堆疊中的資訊，回復呼叫者當時的狀態，繼續執行。有關堆疊在程序呼叫時流程轉換所扮演的角色，第 3 章會有更詳細的介紹。

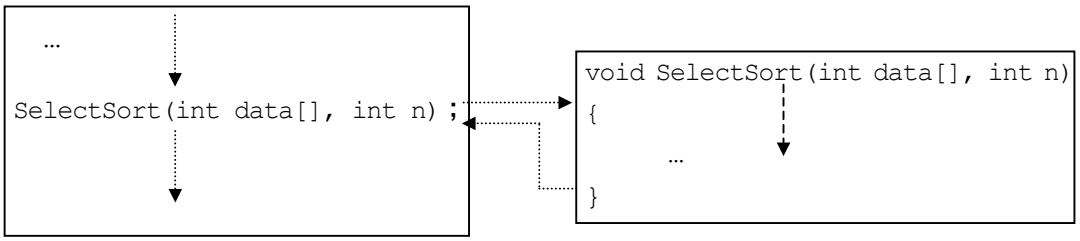



圖 1-2　程序呼叫時流程的轉換

在程序執行完成前呼叫了自己這個程序，即形成了「遞迴」(recursion)；這是「直接遞迴」(direct recursion)，或者在程序執行完成前呼叫了其它會再度引用到自己的程序，這稱爲「間接遞迴」(indirect recursion)。這種遞迴的概念可以引發許多的應用，在此強調的是遞迴可以協助我們整理腦中的思緒，把原本複雜的程式更簡潔的表示出來，對於解決相同狀況不斷重複的問題很有幫助。在設計遞迴程式時，切記「終結條件」(termination condition) 的設立，否則不斷地呼叫自己，會形成無窮迴圈。下面提供幾個直接遞迴的範例。

## 範例 1-3

階乘的計算即可用遞迴的概念詮釋，請看：

$$n! = n \times (n-1)!$$

亦即階乘的計算可利用階乘來回應—直接遞迴！於是可以寫一個計算階乘的程序 `X(int n)`，它接受傳入的整數 `n`，傳回 `n*X(n-1)`。請看程式 1-2。在撰寫程式之前，該注意終結條件的設立：當傳入的整數 `n` 已爲 1 時，應可傳回 1（繼續呼叫 `X(0)` 是沒有意義、不正確的）。

**程式 1-2 計算階乘**

```
1  int X(int n)
2  {   if (n == 1) return 1;
3      return n*X(n-1);
4  }
```

## 範例 1-4

0, 1, 1, 2, 3, 5, 8, 13, … 是著名的費氏數列 (Fibonacci sequence)[8]，其第 $n$ (≥2) 項爲其 $n$–1 與 $n$–2 項的和，而第 0 項爲 0 且第 1 項爲 1。正式定義如下：令 $F_n$ 爲費氏數列，則

$$F_n = \begin{cases} 0 & n = 0; \\ 1 & n = 1; \\ F_{n-1} + F_{n-2} & n \geq 2. \end{cases}$$

以遞迴的觀點來看，$F_n$ 確實可由 $F_{n-1}$ 與 $F_{n-2}$ 的和求算；而且終結條件有兩項。程式 1-3 實踐了對應的遞迴演算法。

**程式 1-3 計算費氏數列的第 n 項**

```
1  int Fib(int n)
2  {   if (n == 0) return 0;
3      if (n == 1) return 1;
3      return Fib(n-1)+Fib(n-2);
4  }
```

讀者大約也感覺到了，上面兩個範例十分簡單，不必用遞迴的概念，而採用 for 迴圈即可順利完成程式，效率也會提昇[9]。然而接下來的範例，用遞迴的思緒來考慮來清楚易懂，反之則不容易。

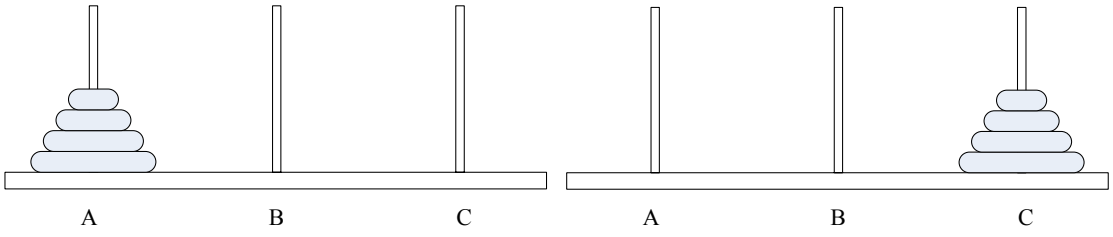
8 費氏數列出現於 1202 的著作 *Liber Abaci*；其中介紹有個假設的費氏兔子問題，可與費氏數列對應。假設一對費氏兔子出生後一個月可長為成兔，並生出一對費氏兔子，且費氏兔子不會死去。若第一個月只有一對費氏兔子，請問第 $n$ 個月會有多少對費氏兔子？此即為第 $n$ 項費氏數列！

9 遞迴呼叫頗消耗系統空間資源—每次呼叫系統堆疊即有對應呼叫參數、返回地址、區域變數…的存入，未設立終結條件的遞迴呼叫，極易耗盡系統資源，造成系統停滯。

## 範例 1-5

河內塔 (towers of Hanoi) 的搬運問題是個有趣的數學遊戲，它包含三根柱子和 $n$ 個大小互異的圓盤，令這三個柱子分別命名爲 A、B 和 C，一開始 $n$ 個圓盤由小至大依序套在圓柱 A 上（如圖 1-3 (a)），遊戲的目的希望搬運這 $n$ 個圓盤依至圓柱 C 上—依然是由小至大順序（如圖 1-3 (b)），圓柱 B 可做爲中介存放區；搬運的規則如下：

- 一次只能移動一個圓盤；
- 每次的移動只能取某柱頂端的圓盤，移至另一柱的頂端；
- 小圓盤不得在大圓盤之下；




(a) 初始狀態　　(b) 結束狀態

圖 1-3　4 個圓盤的河內塔搬運問題

讀者可利用手邊的書本、盒子試試搬搬看，思索解題的線索。下面提供一個解決此問題的方法：整個問題欲將 A 上的 $n$ 個圓盤搬移至 C，我們可以

- ✧ 將 A 頂端前 $n$–1 個圓盤搬移至 B 暫存；
- ✧ 將 A 頂端所剩的 1 個圓盤搬移至 C；
- ✧ 將 B 頂端的 $n$–1 個圓盤搬移至 C。

請先想到是否有遞迴的現象（搬移 $n$ 個圓盤可靠搬移 $n$–1 個圓盤…完成）？想想終止條件該如何設想（搬移 0 個圓盤該如何因應）？想想遞迴呼叫該傳回什麼參數（搬移的起點、終點、暫存區似乎依當時圓盤所在情境而定）？再想想輸入、輸出如何呈現！

演算法 1-2 將上述構思嚴謹地陳述之；其中圓盤個數和柱子 A、B、C 顯然是遞迴呼叫的必要參數，我們利用變數 n 和 A、B、C 記錄當時的情境；輸出就以「搬移圓盤 n 自 A 至 C」的字串型式表示—各變數以其當時的值轉爲字元置入也。

**演算法 1-2 搬運河內塔**

輸入：柱子 A, B, C 和圓盤個數 n(n 個圓盤依大小順序皆在 A 上)
輸出：搬移圓盤的步驟，使得所有圓盤依大小順序皆在 C 上

```
1  towerHanoi(n, A, B, C)
2  {   if (n == 0) return;
2      towerHanoi(n-1, A, C, B)
3      印出 "Move disk "+n+" from "+A+" to "+C
4      towerHanoi(n-1, B, C, A)
5  }
```

依據演算法 1-2，我們可以撰寫出程式 1-4。

**程式 1-4 搬運河內塔**

```
1  void towerHanoi(int n, char A, char B, char C)
2  {   if (n == 0) return;
3      towerHanoi(n-1, A, C, B);
4      cout << "Move disk "<< n <<" from "<< A <<" to "<< C
5      towerHanoi(n-1, B, C, A);
4  }
```

至於主程式可呼叫程序 `towerHanoi` 如下：

```
towerHanoi(n, 'A', 'B', 'C');
```

圖 1-4 以 towerHanoi(3, 'A', 'B', 'C')爲例，將遞迴呼叫時各程序參數的傳遞和程序之間流程轉換的關係，以具有方向的虛線連接（起點處附帶編號），方便讀者追蹤。簡單的遞迴程式竟有如此豐富的執行內容，其威力不容小覷。

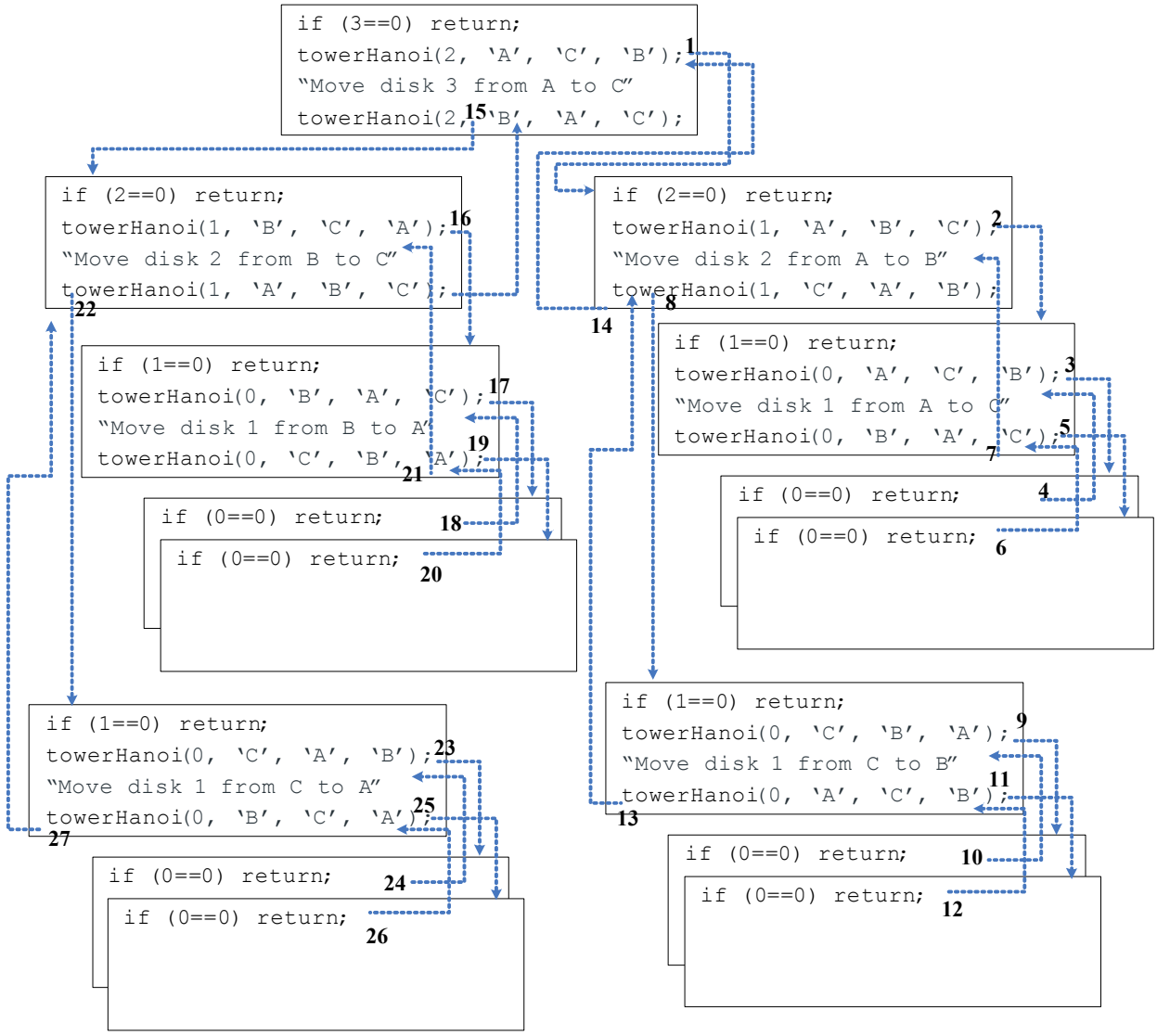



圖 1-4　3 個圓盤河內塔遞迴程式的流程轉換

至於 towerHanoi(3, 'A', 'B', 'C')執行後的輸出如下：

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

前三行輸出步驟實爲遞迴呼叫 towerHanoi(2, 'A', 'C', 'B')的執行結果；第四行即爲程式第 4 行列印指令的結果；而後三行則爲遞迴呼叫 towerHanoi(2, 'B', 'A', 'C')的執行結果。

此時不妨想想，若不以遞迴的方式來解這個問題，是否不太容易？透過遞迴確實可以讓思緒較爲簡潔清楚。

## 範例 1-6

排列 (permutation) 是數學上常用到的運算。以集合{A, B, C}爲例，其排列所成的集合爲 {(A, B, C), (A, C, B), (B, A, C), (B, C, A), (C, A, B), (C, B, A)}。 $n$ 個元素的排列共有 $n!$ 種。

再仔細觀察這個 {A, B, C} 排列的例子，我們可以這樣想：倘若寫出一個遞迴程序 perm，則在傳入{A, B, C}時，應可靠

```
A, perm({B, C})      // A 放在 {B, C} 的所有排列之前
B, perm({A, C})      // B 放在 {A, C} 的所有排列之前
C, perm({A, B})      // C 放在 {A, B} 的所有排列之前
```

產生出所有的排列－每個{A,B,C}的元素皆可擺放在(當時的)第一個位置—可利用一 for 迴圈讓(當時的)所有元素依次擺放於(當時的)第一個位置—

然後自下一個位置起擺上其他元素的排列—遞迴呼叫。

我們可用字元陣列 c[] 來存放需要排列的輸入字元；perm 程序的參數可以是 perm(char c[], int k, int n)，表示欲得到 c[k]~c[n-1] 的排列，0≤k≤n-1（位置 k 即爲當時的第一個位置，是遞迴必要傳遞的參數）；終結條件應爲傳入的 k 已等於 n-1，屆時應輸出當時該項排列。那麼產生排列的程式可撰寫如下：

**程式 1-5 產生排列**

```
1  # define SWAP(x, y, t) (t=x, x=y, y=t)
2  void perm (char c[], int k, int n)
       // 產生 c[k],...,c[n-1] 的所有排列
3  {   if (k == n-1)          //終結條件成立時輸出此項排列
4      {   for (int i = 0; i < n; i++)
5              cout << c[i] <<" ";
6          cout << endl;
7      }
8      else    // 讓 c[k]固定不動，求 perm(c[], k+1, n)
9      {   char temp;
10         for (int i=k; i<n; i++)
11         {   SWAP(c[k], c[i], temp); //讓 c[k]~c[n-1]輪流當 c[k]
12             perm(c,k+1,n);  //產生 a[k+1],...,a[n-1]的所有排列
13             SWAP(c[k], c[i], temp); //還原原字元順序
14         }
15     }
16 }
```

第 3~7 負責在終止條件(k==n-1)成立時，輸出當時的排序；亦即：當排列的起點 k 爲 n-1 時，位置 n-1 的字元自身即爲自己的排列，可列印 c[0]~c[n-1]，其即爲當時的排列）。第 10 行的 for 迴圈中第 10, 11 行讓當時各個元素皆有機會擺放至位置 k，第 11 行 perm(c,k+1,n)則進行位

置 k+1 起的排列遞迴呼叫；而第 12 行必須執行－才能回復「各個元素皆有機會擺放至位置 k」的「當時情境」（若不執行之，當時情境不會重現，所產生的排列自然不會如預期呈現）！

假設 c[0~3]={'A', 'B', 'C', 'D'}；則呼叫 perm(c, 0, 4)可印出如下 24 個排列：

```
A B C D
A B D C
A C B D
A C D B
A D C B
A D B C
B A C D
B A D C
B C A D
B C D A
B D C A
B D A C
C B A D
C B D A
C A B D
C A D B
C D A B
C D B A
D B C A
D B A C
D C B A
D C A B
D A C B
D A B C
```

各位可以檢查遞迴的概念，前六個排列皆以 A 爲首，後面是{B, C, D}的所有排列；再把焦點集中在{B, C, D}的所有排列中，可發現前兩項以 B 爲首，後面是{C, D}的所有排列；再把焦點集中在{C, D}的所有排列，可發現前一項以 C 爲首，後面是{D}的所有排列。接下來六個排列皆以 B 爲首，後面是{A, C, D}的所有排列…以此類推。

由範例 1-3 至 1-6，各位可體會遞迴演算法在整理思緒、簡化程式上的貢獻。一般而言，遞迴演算法皆可改寫成以迴圈和堆疊爲基礎的程式（本書會在第三章中介紹這種技巧），以減少系統資源的消耗，提高執行的效率。

## 1.5 演算法的效率分析

什麼是有效率的演算法？電腦學家爲此衡量準則，提供了客觀的標準—分析演算法的執行時間和記憶體需求；以時間複雜度或空間複雜度來討論演算法的效率。請看下面的定義：

**定義：一個程式或演算法的時間複雜度 (time complexity) 為其所需的執行時間；一個程式或演算法的空間複雜度 (space complexity) 為其所需的記憶體空間。**

不同的輸入資料，可能使演算法或程式的執行時間不盡相同；有些輸入資料可能使演算法很快就求解完成，它們是求解此問題演算法的「較佳狀況」；有些輸入資料則可能使演算法竭盡其力方求出解答，它們是求解此問題演算法的「較差狀況」。本書中所有複雜度的討論，皆以「最差狀況」(the worst case) 爲主—要能把最差狀況下的輸入資料也妥善解決，才稱得上是解該問題的演算法。解決相同的問題，演算法所用的時間複雜度和空間複雜度愈少愈好。一般

而言時間複雜度又比空間複雜度來得緊要[10]，本書的分析討論也以時間複雜度爲主。

## 1.5.1 演算法的執行次數

演算法或程式執行時間的取得，有不夠客觀的爭議，因爲在不同電腦上執行相同程式，其執行時間不會相同。所以我們把焦點放在演算法或程式的執行次數。範例 1-7 描述計算演算法或程式執行次數的方法。

**範例 1-7**

程式 1-6 將傳入整數陣列 data 中的 n 個元素( data[0]~data[n-1] )，加總至整數變數 summation 中傳出。

**程式 1-6 陣列元素加總**

```
1  int Sum(int data[],int n)
2  {   int summation=0;
3      for (int i=0; i<n; i++) summation += data[i];
4      return summation;
5  }
```

我們在程式 1-6 中加入一全域變數 count，來加總所有指令執行的次數；新的程式將如程式 1-7 所示。

10 從硬體成本的眼光來看，CPU 的成本一直高於記憶體的成本；遂電腦學家比較在乎計算時間的成本。

**程式 1-7** 陣列元素加總並計算所有指令執行的次數

```
1  int count = 0;                  // 全域變數宣告
2  count++;                        // 計算宣告 int 指令的執行
3  int Sum(int data[],int n)
4  {   int summation=0;
5      count++;                    // 計算宣告 int 指令的執行
6      for (int i=0; i<n; i++)
7      {   count++;                // 計算 for 指令的執行次數
8          summation += data[i];
9          count++;                // 計算 = 指令的執行次數;
10     }
11     count++;                    // 計算最後一次 for 指令的執行
12     count++;                    // 計算 return 指令的執行
13     return summation;
14 }
```

程式 1-8 精簡了程式 1-7，它只保留程式的主體架構和加總 count 的指令。

**程式 1-8** 計算陣列元素加總所有指令執行的次數

```
1  count++;
2  int Sum(int data[],int n)
3  {   count++;                    // 計算宣告 int 指令的執行
4      for (int i=0; i<n; i++)
5          count += 2;
6      count += 2;
7  }
```

由程式 1-8 的結果可知，此程式的執行總次數爲 $2n+4$，其中 for 迴圈每

執行一次，count 會計數兩次；而迴圈共計有 $n$ 次，所以 for 迴圈內即執行 $2n$ 次。

範例 1-7 描述了計算程式執行次數的方法。這裏其實已引用了這個假設：每行指令的執行時間都一樣；不論是宣告、for 迴圈（包含比較大小(i<n)、累加(i++)）、或 += 指令…，都一視同仁，皆計數一次。這個假設似乎不合理，但在後面的 $O$ 表示法中，我們會瞭解對演算法的時間複雜度而言，這個假設是否合理是不重要的。總之範例 1-7 至少說明了：若要知道演算法的執行次數是的確可行的。

## 1.5.2 演算法複雜度的表示方法：$O$、$\Omega$ 和 $\Theta$

上一節的內容告訴我們，精確地計算一個演算法的執行步驟或時間是瑣碎的工作；事實上我們對演算法效率的要求，不應拘泥在太瑣碎的細節上；試想需要電腦程式來處理的問題，其資料量應該會大到某種程度；我們比較在乎當資料量大時，演算法的表現是否夠好。範例 1-8 先針對資料量大小對演算法效率的影響，給各位一個直覺的判斷。

### 範例 1-8

假設有兩個演算法都可解決問題 $P$，其輸入資料量爲 $n$；演算法 $A$ 的執行次數爲 $4n^2+174$，演算法 $B$ 的執行次數爲 $n^3+5n+6$。圖 1-5 描繪了兩個演算法其執行次數與輸入資料量大小的關係。

當 $n$ 小於或等於 7 時，演算法 $B$ 的執行次數會少於演算法 $A$；但 $n$ 一旦大於 7，演算法 $A$ 的執行次數會少於演算法 $B$ 者；而且當 $n$ 愈大，兩者的差距會愈大。各位是否同意：演算法 $A$ 要比演算法 $B$ 更有效率？即令當 $n$ 小於或等於 7 時，演算法 $A$ 表現的不好。（用電腦程式執行演算法 $B$ 來解決 $n \le 7$ 的問題，在大多數的情況下是殺雞用牛刀了。）

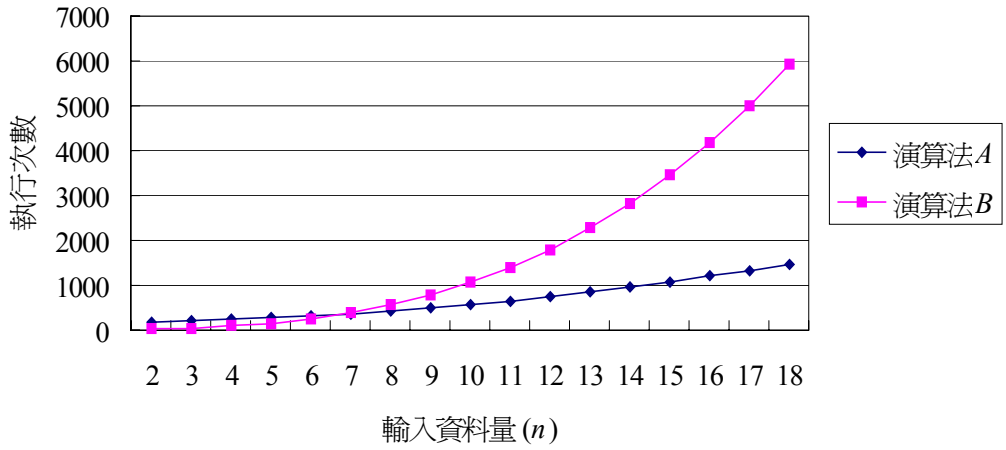



圖 1-5　演算法 $A$ 和 $B$ 執行次數與輸入資料量大小的關係

基於以上的討論，我們知道對演算法的複雜度分析，應需要更精簡的表示法（不必拘泥在 $4n^2+174$ 和 $n^3+5n+6$ 具體執行次數的細節）。電腦學家遂用 $O$、$\Omega$、$\Theta$ 的符號來簡化演算法在時間（執行步驟）複雜度上的表示。我們先看看這些符號的定義，假設 $f, g \geq 0$：

**定義：$f(n) = O(g(n))$ 若且唯若存在兩個正數 $c$ 和 $n_0$，當 $n \geq n_0$ 時，$f(n) \leq c \times g(n)$。**

我們用「大 $O$」(big-O) 稱呼上面定義中 $O$ 的符號；這個定義在數學上賦予了大 $O$ 生命，但在演算法複雜度上，該如何解讀它呢？$f(n)$ 指的是演算法的執行時間（步驟），我們希望能找到 $g(n)$，只要在 $n \geq n_0$ 後，$c \times g(n)$ 一定會大於或等於 $f(n)$，那麼就可以用 $O(g(n))$ 來表示 $f(n)$。「$n \geq n_0$」強調了我們比較在乎在 $n$ 大時，演算法執行隨 $n$ 變化的趨勢 (order of magnitude)；而「常數 $c$」的存在則希望消弭了因程式語言、程式設計師寫作技巧、硬體環境、作業系統、… 所可能造成的執行效能差異（這些不同的主客觀因素可用一個選定合乎定義的常數 $c$ 概括忽略之）。這些數學上的符號與限制，給演算法複雜度的分析可以客觀、便利、有彈性又不失精確。

請與範例 1-8 對照比較，$f(n)$ 是演算法的執行步驟，$f(n) = 4n^2+174$，我們

希望用精簡的 $g(n)$ 來表示 $f(n)$；可以選 $g(n) = n^2$，因爲存在 $c = 5$，$n_0 = 14$，使得 $n \geq 14$ 後，$5n^2 \geq 4n^2+174$；於是我們用 $O(n^2)$ 來表示 $4n^2+174$。我們畫出圖 1-6 來圖解 $f(n)$ 和 $g(n)$ 的關係：

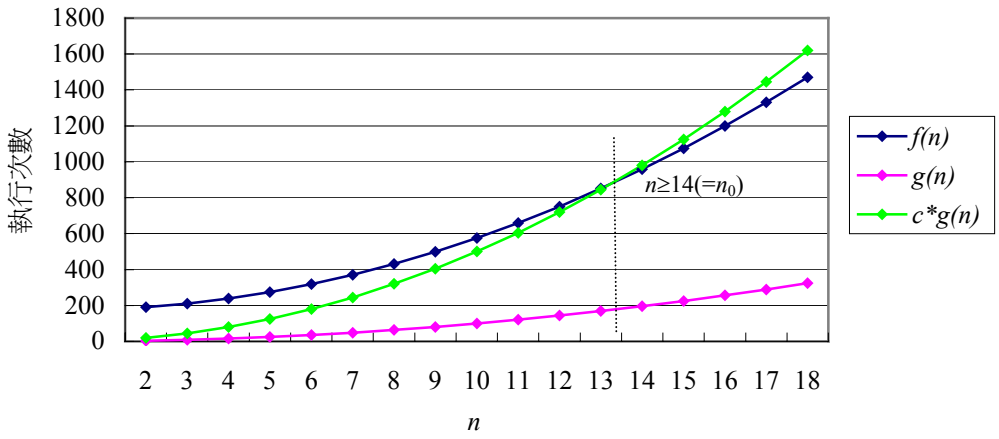



圖 1-6 $f(n)$、$g(n)$ 和 $c \times g(n)$ 的關係

請注意在圖 1-6 中，當 $n \geq 14$ $(=n_0)$ 之後，$c \times g(n)$ 一定會大於或等於 $f(n)$ $(5n^2 \geq 4n^2+174)$，所以 $O(n^2)$ 足以代表 $4n^2+174$（$n^2$ 與 $4n^2+174$ 在 $n$ 夠大時，只差一個常數倍，在分析複雜度時，這個常數倍是可以忽略的（可能因程式語言、程式寫作技巧、硬體環境、作業系統…的不同，使得此常數倍不再顯著）；我們可以想成這兩個演算法在 $n$ 增大的時候，執行時間的增加趨勢是相當的，也就是在 $O$ 的考量標準下，這兩個演算法是一樣好的）。所以上面的定義旨在告訴我們：只要找到符合條件的 $g(n)$，即可用 $O(g(n))$ 來表示 $f(n)$。

## 範例 1-9

(a) $4n+12 = O(n)$，因爲存在 $c = 5$，$n_0 = 12$，使得 $n \geq 12$ 後，$4n+12 \leq 5n$（或 $c = 6$，$n_0 = 6$，使得 $n \geq 6$ 後，$4n+12 \leq 6n$）；

(b) $10n+25 = O(n)$，因爲存在 $c = 11$，$n_0 = 25$，使得 $n \geq 25$ 後，$10n+25 \leq 11n$；

(c) $8n^2+11n+18 = O(n^2)$，因爲存在 $c$ =9，$n_0 = 13$，使得 $n \geq 13$ 後，$8n^2+11n+18 \leq 9n^2$；

(d) $6\times2^n+n^2 = O(2^n)$，因爲存在 $c = 7$，$n_0 = 4$，使得 $n \geq 4$ 後，$6\times2^n+n^2 \leq 7\times2^n$；

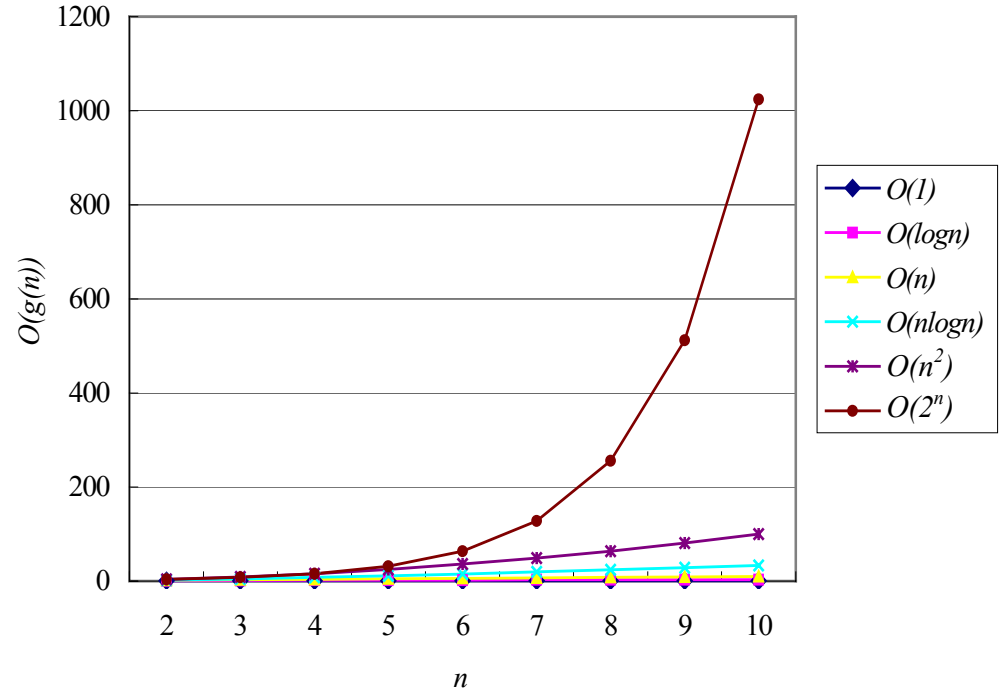
(e) $326 = O(1)$，因爲存在 $c = 327$，$n_0$ 可任取，使得 $n \geq n_0$ 後，$326 \leq 327\times1$；

(f) $9n^2+n+11 \neq O(n)$，因爲找不到適當的 $c$ 和 $n_0$，使得 $n \geq n_0$ 後，$9n^2+11 \leq cn$；

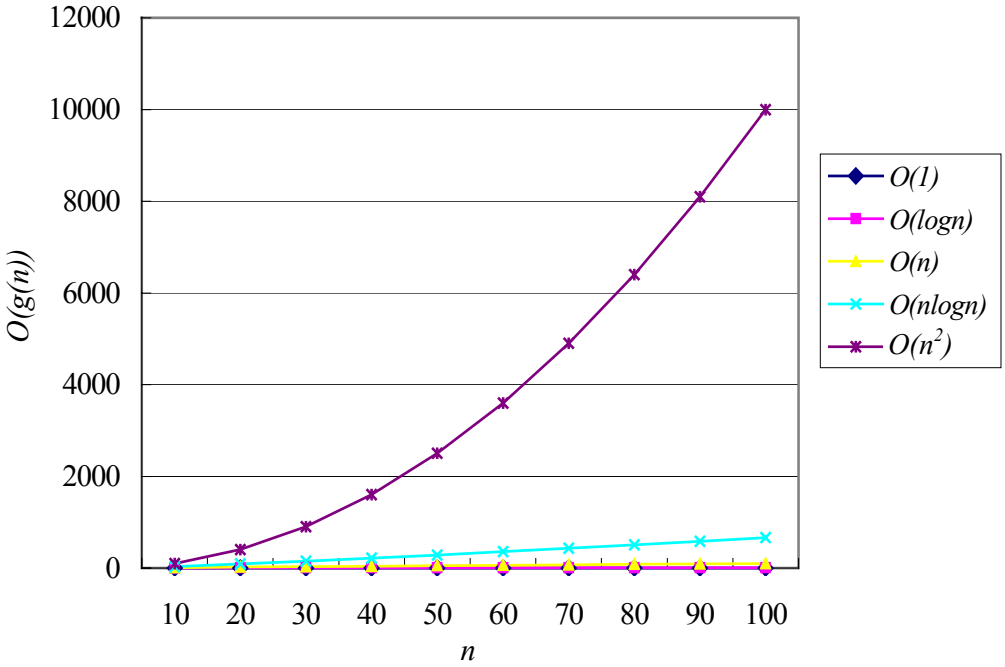
(g) $100n^3 = O(n^4)$，因爲存在 $c = 16$，$n_0 = 8$，使得 $n \geq 8$ 後，$100n^3 \leq 16n^4$。

請注意：範例 1-9 (f) 的例子告訴我們，複雜度所用的大 $O$ 表示法旨在找到在 $n$ 夠大的時候，$f(n)$ 的代表函數 $g(n)$，而 $g(n)$ 的某個常數倍會大於 $f(n)$；所以 $g(n)$ 中 $n$ 的最高乘冪，不可能比 $f(n)$ 中 $n$ 的最高乘冪小。而範例 1-7 (g) 的例子則告訴我們只要任何 $g(n)$，其 $n$ 的最高乘冪，比 $f(n)$ 中 $n$ 的最高乘冪大，均可成爲 $f$ 的大 $O$ 代表函數。然而即令定義上允許，在比較複雜度優劣時，我們依然希望 $f$ 的大 $O$ 代表函數 $g$ 應愈貼近 $f$ 愈好；以範例 1-7 (g) 爲例，取 $g(n) = n^3$，而 $100n^3 = O(n^3)$，是比較合理的。由此可知，若 $f(n)$ 中 $n$ 的最高乘冪爲 $k$，則 $f(n) = O(n^k)$。

有了大 $O$ 的表示法，演算法間的執行時間 ( 時間複雜度 ) 的優劣程度，就可直接用其大 $O$ 代表函數來比較。範例 1-9 (e) 中的 $O(1)$ 稱爲「常數時間」(constant time)，即不論演算法的步驟須需要多少指令，只要沒有迴圈重複執行，皆視爲常數時間；$O(n)$ 稱爲「線性時間」(linear time)，取其執行步驟的增加趨勢與 $n$ 的增加趨勢爲線性關係之意；$O(n^2)$ 爲「平方時間」；依此類推，而 $O(2^n)$ 則稱爲「指數時間」(exponential time)。如此一來在解決相同問題時，我們會說 $O(\log n)$ 的演算法比 $O(n)$ 來得有效率，$O(n)$ 比 $O(n^2)$ 來得有效率…。圖 1-5 描繪了幾個常見大函數其函數值與 $n$ 值的關係。




(a) $n \leq 10$ 時，大 $O$ 函數值的變化




(b) $10 \leq n \leq 100$ 時，大 $O$ 函數值的變化

圖 1-7 幾個常見大 $O$ 函數其函數值與 $n$ 值的關係

由圖 1-7 (a) 可知，演算法的時間複雜度若爲指數時間，則其時間成長會隨 $n$ 增加而極劇成長；這種指數暴增 (exponential explosion) 的演算法是我們儘量避免使用的。在圖 1-7 (b) 中捨去了 $O(2^n)$ 的函數，將其它多項式時間函數做一比較。綜合以上的討論，可知大 $O$ 表示函數的優劣順序爲：

$$O(1) \rangle O(\log n) \rangle O(n) \rangle O(n\log n) \rangle O(n^2) \rangle O(n^3) \rangle O(2^n)$$

其中符號 〉表示「優於」。

請注意在後面討論演算法的時間複雜度時，我們在乎的是在 $n$ 增大的時候，其大 $O$ 表示函數的成長趨勢；以此成長趨勢論及演算法的優劣效率。

大 $O$ 的符號可以方便討論「演算法的複雜度」；而 $\Omega$ 可以方便討論「問題的難易程度」。其定義如下，假設 $f, g \geq 0$：

**定義：$f(n) = \Omega(g(n))$ 若且唯若存在兩個正數 $c$ 和 $n_0$，當 $n \geq n_0$ 時，$f(n) \geq c \times g(n)$。**

**範例 1-10**

(a) $4n+12 = \Omega(n)$，因爲存在 $c = 1$，$n_0 = 1$，使得 $n \geq 1$ 後，$4n+12 \geq n$；

(b) $10n+25 = \Omega(n)$，因爲存在 $c = 10$，$n_0 = 1$，使得 $n \geq 1$ 後，$10n+25 \geq 10n$；

(c) $8n^2+11n+18 = \Omega(n^2)$，因爲存在 $c = 1$，$n_0 = 1$，使得 $n \geq 1$ 後，$8n^2+11n+18 \geq n^2$；

(d) $6\times 2^n+n^2 = \Omega(2^n)$，因爲存在 $c = 1$，$n_0 = 1$，使得 $n \geq 1$ 後，$6\times 2^n+n^2 \geq 1\times 2^n$；

(e) $326 = \Omega(1)$，因爲存在 $c = 1$，$n_0$ 可任取，使得 $n \geq n_0$ 後，$326 \geq 1\times 1$ 或 1；

(f) $9n^2+n+11 \neq \Omega(n^3)$，因爲找不到適當的 $c$ 和 $n_0$，使得 $n \geq n_0$ 後，$9n^2+11 \geq cn^3$；

(g) $100n^3 = \Omega(n^2) = \Omega(n) = \Omega(1)$，因爲存在 $c = 1$，$n_0 = 1$，使得 $n \geq 1$ 後，$100n^3 \geq n^2 \geq n \geq 1$。

請注意：範例 1-10 (f) 的例子告訴我們 $g(n)$ 中 $n$ 的最高乘冪，不可能比 $f(n)$ 中 $n$ 的最高乘冪大。而範例 1-10 (g) 的例子則告訴我們：只要任何 $g(n)$，其 $n$ 的最高乘冪，比 $f(n)$ 中 $n$ 的最高乘冪小，均可成爲 $f$ 的 $\Omega$ 代表函數。然而即令定義上允許，在問題的難易程度時，我們希望 $f$ 的 $\Omega$ 代表函數 $g$ 應愈貼近 $f$ 愈好；以範例 1-10 (g) 爲例，取 $g(n) = n^3$，而 $100n^3 = \Omega(n^3)$，是比較合理的。由此可知，若 $f(n)$ 中 $n$ 的最高乘冪爲 $k$，則 $f(n) = \Omega(n^k)$。

在討論演算法的複雜度時，$f$ 通常指的是演算法的執行步驟；在討論問題的難易程度時，$f$ 通常指的是証明問題難易程度時，推導出欲解決問題所要花的最小代價之對應函數；而其 $\Omega$ 代表函數，方便我們用來比較不同問題間花費之最小代價。若解決問題甲的最小代價爲 $\Omega(n\log n)$，解決問題乙的最小代價爲 $\Omega(n^2)$，則我們可說問題甲比問題乙容易。

演算法時間複雜度 $f$ 的大 $O$ 表示，爲其找到上界 (upper bound) 函數 $(c\times g(n) \geq f(n))$，此上界即爲解決該問題至多應付出的代價（至多也不過 $O(g(n))$，即執行此演算法的時間）；經証明後導出之問題難易程度 $f'$ 的 $\Omega$ 表示，則爲該問題找到了下界 (lower bound) 函數 $(f'(n) \geq c\times g'(n))$，此下界爲解決該問題至少應付出的代價（至少要 $\Omega(g'(n))$ 的時間）。請注意：大 $O$ 表示乃用在解決問題 $P$ 的演算法複雜度上，有演算法即可找其對應的大 $O$ 時間複雜度；此演算法確能解決問題 $P$，所以欲解決問題 $P$ 至多需要執行該演算法的時間，遂此大 $O$ 時間複雜度爲解決問題 $P$ 提供了上界。$\Omega$ 表示是用在証明解決問題 $P$ 所需要的最少代價上，必須透過數學邏輯的推演，方可找到；既然該 $\Omega$ 表示是解決問題 $P$ 所需要的最少代價，自然是解決問題 $P$ 所需時間的下界。

如果我們既可找到演算法 $A$ 來解決問題 $P$，時間複雜度爲 $O(g(n))$，且又能証明解決問題 $P$ 的最少代價亦爲 $\Omega(g(n))$；亦即欲解決 $P$ 的時間，至多要 $O(g(n))$，至少爲 $\Omega(g(n))$；不可能多於 $O(g(n))$，也不可能少於 $\Omega(g(n))$（最多或最少都只是 $g(n)$ 的常數倍），則演算法 $A$ 是解決問題 $P$ 的最佳 (optimal) 演算法。下面的定義恰可清楚敘述最佳演算法的含意：

**定義：** $f(n) = \Theta(g(n))$ 若且唯若存在參個正數 $c_1$、$c_2$ 和 $n_0$，當 $n \geq n_0$ 時，
$c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$。

**範例 1-11**

(a) $4n+12 = \Theta(n)$，因爲存在 $c_1 = 1$、$c_2 = 5$，$n_0 = 12$，使得 $n \geq 12$ 後，$1 \times n \leq 4n+12 \leq 5 \times n$。

(b) $8n^2+11n+18 = \Theta(n^2)$，因爲存在 $c_1 = 1$、$c_2 = 9$，$n_0 = 13$，使得 $n \geq 13$ 後，$1 \times n^2 \leq 8n^2+11n+18 \leq 9 \times n^2$。

範例 1-11 的例子，皆滿足 $\Theta$ 的定義；然而在演算法複雜度分析上，$\Theta$ 符號的使用強調了解決問題 $P$ 的最佳演算法 $A$ 已經找到。如上所述，必須找到解決 $P$ 的方法 $A$，求得其時間複雜度爲 $O(g(n))$，且還能証明解決 $P$ 的最少時間亦爲 $\Omega(g(n))$，我們不可能找到比 $\Omega(g(n))$ 更快的方法，$O(g(n))$ 已無法再改進，亦即 $A$ 已是解決問題 $P$ 的最佳演算法，其時間複雜度正是 $\Theta(g(n))$。

下面舉幾個簡單的例子，說明實際問題與其演算法之間的關係。

**範例 1-12**

程式 1-4 提供了 $n$ 個陣列元素加總的演算法，其時間複雜度爲 $O(n)$ —爲解此問題的演算法提供了所需時間的上界。我們當然可以用更高的時間複雜度解決這個加總問題（例如：另加一迴圈使時間提高爲 $O(n^2)\cdots$），但 $O(n)$ 既然可以解決問題，任何高於 $O(n)$ 的演算法已不切實際，對解決問題的成本上界沒有具體貢獻—只提供無聊 (trivial) 的上界罷了。我們再想想這個問題至少需要的解決時間（謂之下界）：$\Omega(1)$ 如何？（的確至少需要讀入 $O(1)$ 的資料、花 $O(1)$ 的時間處理）但這也有點無聊，只處理 $O(1)$ 的資料無法正確求出 $n$ 個陣列元素的加總！$\Omega(n)$ 如何？$\Omega(n)$ 就明確多了：至少需要讀入 $n$ 個元素資料、花 $\Omega(n)$ 的時間處理。綜合以上說明：解此問題的演算法上界

爲 $O(n)$，而且問題本身的處理下界爲 $\Omega(n)$，根據定義可知：程式 1-4 的演算法是求解陣列元素加總問題的最佳演算法。

### 範例 1-13

第七章裏我們會証明排序 (sorting) 問題所需要的最少時間爲 $\Omega(n\log n)$（至少要 $\Omega(n\log n)$）—不是 $\Omega(1)$、$\Omega(n)$ 等無聊的下界；而且會介紹堆積排序法 (heap sort) 或合併排序法 (merge sort) 的時間複雜度爲 $O(n\log n)$（至多要 $O(n\log n)$）—不是挑選排序演算法 $O(n^2)$ 之類無聊的上界；所以得知堆積排序或合併排序法可謂解決排序問題的最佳演算法。

### 範例 1-14

考慮問題「找出 $n$ 個整數中最小者」。可以先想想問題的下界：$\Omega(1)$—至少得看常數個輸入的整數—如何？話是沒錯，但顯然是個無聊的下界；至少得看「所有 $n$ 個輸入的整數」才貼合問題，於是下界爲 $O(n)$。此問題可引用排序演算法解決，即上界爲 $O(n\log n)$；但仔細思考，排序似乎提供了比問題所需更多的資訊—所有數字間的大小關係皆已決定—所求問題不需要如此多的資訊！事實上程式 1-1 第 5~8 行，即可決定 $n$ 個數字中最小者；據之重新寫出程式如下：

**程式 1-9 找出整數陣列中最小元素**

```
1  int findMin(int data[], int n)
2  {   int min = 0, j;
3      for (j=1; j<n; j++)
4          if (data[j] < data[min]) min = j;
5      return data[min];
6  }
```

其中令 data 陣列已存放所考慮的 $n$ 個數字；程式 1-9 的時間複雜度爲 $O(n)$。綜合以上而言，解此問題的演算法上界爲 $O(n)$（至多需要的成本），而且問題

本身的下界爲 $\Omega(n)$（至少需要的代價），根據定義可知：程式 1-7 的演算法是求解問題的最佳演算法。

在解決問題的過程中，自然希望演算法的時間複雜度越低越好，或說解決問題的上界（至多成本）越低越好（用一個其慢無比的方法來解題當然無聊）；在証明問題所需的最小代價時，無非盼望証明最小代價越高越好，或說解決問題的下界（至少成本）越高越好（「解決問題至少要 $O(1)$ 的時間」幾乎是所有問題的下界，但它太不具代表性了，俗稱「無聊的下界」(trivial lower bound)；只要証明是正確的，當然盼望証出來的下界儘量高，越逼近眞實的最小成本越好）。

在更快演算法的尋覓研究中，証明問題的 $\Omega$ 下界，正提出了往下尋覓的疆界；而証明問題到底最少所需代價時，找到演算法，貢獻出 $O$ 上界，也爲數學向上逼近的最小代價証明，提出了實際解法及時間上界。兩者的關係，實值得各位玩味。

# 本章習題

1. 寫一個程序，將傳入的整數參數 $x$、$y$ 和 $z$ 由小到大印出。此程序的計算時間爲多少？

2. 二項式係數 (binomial coefficient) 的定義如下：

$$\binom{n}{m}=\frac{n!}{m!(n-m)!}\ ;$$

請用迴圈撰寫程式，計算二項式係數（輸入 $n$，求 $\binom{n}{m}$，$m$=0, 1, 2, …, $n$）。

而該式可用下面的遞迴式表示：

$$\binom{n}{m}=\binom{n-1}{m}+\binom{n-1}{m-1}。$$

請用遞迴的技巧撰寫程式，計算二項式係數。試比較兩者的優劣。

3. 費氏 (Fibonacci) 數列被定義成：

(1) $F_0 = 0, F_1 = 1;$

(2) $F_n = F_{n-1} + F_{n-2},$ 當 $n \geq 2$。

寫出遞迴和迴圈版本（非遞迴）的程序來計算費氏數列（輸入 $n$，求 $F_1, F_2, \ldots, F_n$）。

4. Lucas 數列被定義成：

(1) $L_0 = 2, L_1 = 1;$

(2) $L_n = L_{n-1} + L_{n-2},$ 當 $n \geq 2$。

寫出遞迴和非遞迴版本的程序來計算 Lucas 數列。

5. Ackerman's 函式 $A(m, n)$ 的定義如下：

$$A(m,n)=\begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } n=0 \\ A(m-1, A(m,n-1)) & \text{otherwise} \end{cases}$$

這個函式在 $m$ 和 $n$ 的值還不是很大的時候，就已成長的非常快。寫一個遞迴程序和非遞迴程序計算它。

6. 給一個正整數 $n$，寫一個程式來判斷 $n$ 是不是其所有因數的總和。亦即是否 $n$ 是所有 $t$ 的總和，其 $1 \le t < n$，且 $t$ 整除 $n$。

7. 假如 $S$ 是一個含有 $n$ 個元素的集合，則 $S$ 的 power set 就是「所有 $S$ 可能的子集合的集合」，例如：假如 $S = \{a, b, c\}$ ，則 $powerset(S) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$，寫一個遞迴程式來計算 $powerset(S)$ 。

8. [Towers of Hanoi] 請參考範例 1-5、程式 1-4，輸入爲 $n$ 個圓盤：

   (a) 實作出有三根支柱，搬運河內塔的遞迴程式。

   (b) 請想想倘若有四根支柱，$n$ 個圓盤該如何搬運？

9. 比較 $n^3$ 和 $2^n$ 這兩個函式，計算出哪個 $n$ 值會使後者大於前者。

10. 使用數學歸納法證明：

   (a) $\sum_{1 \le i \le n} i = n(n+1)/2, n \ge 1$ ；

   (b) $\sum_{1 \le i \le n} i^2 = n(n+1)(2n+1)/6, n \ge 1$ ；

   (c) $\sum_{0 \le i \le n} x^i = (x^{n+1}-1)/(x-1), x \ne 1, n \ge 0$ ；

(d) $\sum_{1\le i\le n}(2i-1)=n^2$ 。

11. 分析下列程式的時間複雜度：

```
1   for (i=1; i<=n; i++)
2       for (j=1; j<=i; j++)
3           for (k=1; k<=j; k++)
4               x++;
```

(a)

```
1   i=1;
2   while (i<=n)
3   {   x++;
4       i++;
5   }
```

(b)

```
1   for (i=1; i<=n; i++)
2       for (j=1; j<=i; j*=2)
3           x++;
```

(c)

```
1   for (i=1; i<=n; i*=2)
2       for (j=1; j<=i; j++)
3           x++;
```

(d)

12. 證明下列等式是正確的：

(a) $5n^2\text{-}6n = \Theta(n^2)$

(b) $n! = O(n^n)$

(c) $2n^2 2^n + n\log n = \Theta(n^2 2^n)$

(d) $\sum_{i=0}^{n} i^2 = \Theta(n^3)$

(e) $\sum_{i=0}^{n} i^3 = \Theta(n^4)$

(f) $n^{2^n} + 6 \times 2^n = \Theta(n^{2^n})$

(g) $n^3 + 10^6 n^2 = \Theta(n^3)$

(h) $6n^3/(\log n + 1) = O(n^3)$

(i) $n^{1.001} + n\log n = \Theta(n^{1.001})$

(j) $n^k + \varepsilon + n^k \log n = \Theta(n^k + \varepsilon)$ for all $k$ and $\varepsilon$, $k \geq 0$,and $\varepsilon > 0$

(k) $10n^3 + 15n^4 + 100n^2 2^n = O(100n^2 2^n)$

(l) $33n^3 + 4n^2 = \Omega(n^2)$

(m) $33n^3 + 4n^2 = \Omega(n^3)$

13. 證明下列等式是錯誤的：

(a) $10n^2 + 9 = O(n)$

(b) $n^2 \log n = \Theta(n^2)$

(c) $n^2/\log n = \Theta(n^2)$

(d) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$