04 CHAPTER 演算法導論與基礎資料結構

一般來說，程式製作的過程為程式設計師會先根據使用者的需求來設計演算法 (Algorithm)，然後再挑選一個適合的程式語言，根據程式語言的語法及設計好的演算法來撰寫 (Coding) 程式。由此可知程式設計與演算法之間的關係十分密切。本章將介紹演算法的相關知識與基礎資料結構，包含的主題有演算法基本觀念、演算法分析、常用程式設計方法、發展程式的方法與基礎資料結構 (包含陣列、鏈結串列、堆疊、佇列及樹狀結構)。

4.1 演算法基本觀念

演算法是指是有限數目指令的集合，利用這群指令撰寫的程式可以完成某個特定的工作。演算法有五項基本的條件，必須五項條件都滿足才符合演算法的要求，五項條件分述如下：

1. **輸入 (Input) 條件：**
   0 個或 0 個以上的輸入，也就是說演算法允許沒有輸入。

2. **輸出 (Output) 條件：**
   演算法要求至少應有 1 個輸出。

3. **明確性 (Definiteness) 條件：**
   演算法要求定義必須明確不可模擬兩可 (Ambiguous)。

4. **有限性 (Finiteness) 條件：**
   演算法要求不可存在無窮迴路 (Infinite Loop)，也就是說必須在有限步驟內完作。

5. **有效性 (Effectiveness) 條件：**
   演算法執行的結果應是正確的。

任何一種計算方法必須滿足以上五項條件才符合演算法的要求。如果某一種計算方法執行的結果是錯誤的，就算該計算方法滿足輸入、輸出、明確性及有限性四項條件，而且或許執行的速度也很快，但是一個錯誤的計算方法就不能算是演算法。

「程式中不應有無窮迴路」是普遍被認同的一個觀念，但是事實上「無窮迴路」對於某些程式卻是必須的，比如作業系統本身便是一個反覆執行「無窮迴路」的程式 (她始終在等候使用者提出要求服務的請求)。程式設計師撰寫程式時經常會利用流程圖做為設計程式的輔助工作，流程圖與程式是相關連的。所以說流程圖與程式都允許「無窮迴路」，因此並非所有的流程圖與程式都能滿足演算法的要求。

實例：一個演算法的例子

```
排序副程式 SORT(D, N)
  依序處理 D 中的第 1 筆至第 N 筆資料
    {
    由 D[i] 到 D[N] 中找出最小的值 D[j]
    交換 (D[i],D[j])
    }
```

上面所敘述的演算法若轉換成程式段則相對應的程式段將如下所示：

```
SORT(D, n)
{
  for (i=1; i≤n; i++)
   { j=i;
     for (k=j+1; k≤n; k++)
        if (D[k]<D[j]) j=k;
     T=D[i];
     D[i]=D[j];
     D[j]=T; }
}
```

4.2 演算法分析

由於程式是根據演算法撰寫而成，因此程式與其對應的演算法關係必定非常密切；若演算法設計得宜，則程式的執行效率及記憶體空間的使用情形都會處於較理想的狀況。

分析或評估程式效能的方法可分別就「程式執行所需的時間」(Time) 與「程式執行所需的記憶體空間」(Space) 兩方面來著手。

影響程式執行時間的因素有以下四點：

1. 輸入資料的數量。
2. 所採用的演算法之時間複雜度 (Time Complexity)。
3. 編譯器的優劣。
4. 計算機的執行速度。

通常執行演算法分析時只會考慮輸入資料的數量和演算法之時間或空間複雜度。漸進符號 (Asymptotic Symbols) 常被用來分析演算法的時間或空間複雜度，雖然漸進符號無法精確地表達演算法實際需要的執行時間或記憶體空間，但因可利用簡單的近似值表達演算法所需的時間或空間複雜度，因此被廣泛採用。

最常被使用來表達演算法的漸進符號為「O」(唸作 big-oh)，說明如下：

O：f(n)=O(g(n)) 若且為若存在兩個正整數 N 與 c，使得當 n ≥ N 時，f(n) ≤ c×g(n)。

「O」代表函數的漸進上限，若 $f(n)=O(g(n))$，則 $O(g(n))$ 便代表函數 $f(n)$ 的漸進上限，例如 $f(n)=3n^2+4n+5$，則 $O(n^2)$ 便代表函數 $f(n)$ 的漸進上限。簡單的來說就是，只要 n 的值夠大，用 n^2 的值就可來做為整個多項式 $3n^2+4n+5$ 的近似值。

假設 n 表示要處理的資料量則常見的時間複雜度有以下八種：

編號	符號	意義
1	$O(1)$	常數時間 (Constant time)
2	$O(\log_2 n)$	次線性時間 (Sub-linear Time)，一般以 O(log n) 表示
3	$O(n)$	線性時間 (Linear Time)
4	$O(n\log_2 n)$	線性乘次線性時間 (Sub-linear Time)，一般以 O(n log n) 表示
5	$O(n^2)$	平方時間 (Quadratic Time)
6	$O(n^3)$	立方時間 (Cubic Time)
7	$O(2^n)$	指數時間 (Exponential Time)
8	O(n!)	階層時間

請注意：

$$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

範例 1

寫出下列 f(n) 的時間複雜度：

(1) $5\log n+100$

(2) $8n^3-15$

(3) $20n^2+400$

(4) $20n^3+30n^2+40n\log n+50n$

解

(1) $5\log n+100=O(\log n)$

(2) $8n^3-15=O(n^3)$

(3) $20n^2+400=O(n^2)$

(4) $20n^3+30n^2+40n\log n+50n=O(n^3)$

範例 2

利用以下程式段來說明 $f(n)=2\times n^2+3\times n+500$ 的邏輯意義，在程式段中使用到的 P、Q 及 R 滿足 $P+Q+R=500$，詳細程式段如下：

```
敘述 1;
敘述 2;
…
敘述 P
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        { 敘述 1;
          敘述 2; }
敘述 1;
敘述 2;
…
敘述 Q
for (i=1; i<=n; i++)
    {   敘述 1;
        敘述 2;
        敘述 3; }
敘述 1;
敘述 2;
…
敘述 R
```

解

在此程式內的雙層迴圈之迴圈敘述有兩條，共執行 $2\times n^2$ 次，單層迴圈之迴圈敘述有三條，共執行 $3\times n$ 次，其他一般敘述共有 $P+Q+R=500$ 條，因此執行的總敘述數為 $2\times n^2+3\times n+500$，利用下表做一簡單分析：

n 值	10	100	1000	10000
$(2\times n^2+3\times n+500)$ 值	730	20800	2003500	100030500

由上表知，當 n 值愈大 $2\times n^2$ 的值佔 $(2\times n^2+3\times n+500)$ 值之比重就愈大，也就是說，只要 n 夠大，n^2 這一項便足以用來做為 $(2\times n^2+3\times n+500)$ 之近似值，而且 n 值愈大精確度便愈高。

4.3 常用程式設計方法

運算思維 (Computational Thinking) 是指利用電腦解決問題的思維來解決問題的能力。簡單來說，運算思維就是「設計程式的方法」。

程式產生的過程一般可分為需求 (Requirement)、設計 (Design)、分析 (Analysis)、再修飾與編碼 (Refinement and Coding) 及驗證 (Verification) 五個階段。「需求階段」是根據程式的要求定義出所有可能的輸入及輸出狀況。「設計階段」是根據需求設計出相對應的演算法。「分析階段」是嘗試設計出兩種以上不同的演算法，再由不同的演算法中決定何者最佳。「再修飾與編碼階段」是選擇適當的程式語言開發工具對最佳演算法編碼並撰寫出對應的程式。「驗證階段」是對撰寫出的程式執行證明 (Proving)、測試 (Testing) 及除錯 (Debugging) 三項工作。

常用的程式設計方法有遞迴法 (Recursive Method)、貪婪法 (Greedy Method)、個別擊破法 (Divide and Conquer) 及動態程式法 (Dynamic Programming) 四種，以下僅先對遞迴法做詳細介紹，另外三種程式設計方法則僅因屬於較進階內容，此處僅做概略說明。

1. 遞迴法

允許副程式直接或間接呼叫本身便稱之為遞迴法。目前常用的程式語言開發工具均提供遞迴功能，如 C、C++、Java 及 Visual Basic 等。

範例 1

利用遞迴法設計一個演算法來計算 s=1+2+…+n 的值，其中 n>1 且 n 為整數。

解

```
int f(int n)
{
if (n ==1) return (1)      ;
else return (f(n-1)+n);
}
```

範例 2

利用遞迴法法設計一個演算法來計算 n!=1×2×…×n 的值，其中 n>1 且 n 為整數。

解

```
int f(int n)
{
if (n ==1) return (1)      ;
else return (f(n-1)*n);
}
```

2. 貪婪法

貪婪法的原則是求出現階段的最佳選擇。將求解的過程細分成一系列的子步驟，每個子步驟所做的選擇都是目前最佳的，而且每個子步驟的選擇在往後皆不得被變更。例如 Dijkstra 的單一起點最短路徑演算法，Prime 的最小生成樹演算法和無失真編碼的霍夫曼演算法，都是採用貪婪法。

3. 個別擊破法

將一個問題分解成數個較小的問題，若某些小問題可繼續再被細分，便再將小問題往下再細分成更小的問題，依此類推直到問題獲得解答為止。最後將分解後問題的解答合併成分解前問題的解答，依此類推直到獲得原始問題的解答為止。常見的個別擊破法演算法的範例有快速排序法、二元樹追蹤、圖形的深度優先搜尋及廣度優先搜尋等方法。

4. 動態程式法

動態程式法除了考慮目前的狀況外，還必須考量其他階段的情況後才能做出最佳選擇。例如 0/1 背包問題 (0/1 Knapsack) 便可利用動態程式法來解決。

4.4 發展程式的方法

當程式設計師具備演算法的知識後便可開始設計及發展程式。發展程式常見的方法有「由上而下設計法」(Top Down Design) 及「由下而上設計法」(Bottom Up Design) 兩種作法，分述如下：

1. 由上而下設計法

由上而下的設計法的主要精神為：先確定最高層的功能，然後依序產生各個較低階層的模組與元件。由上而下的設計法是指從事程式設計的過程中，依照程式的邏輯特性將程式細分成幾個較小的問題，再將這些較小的問題同樣依照程式的邏輯特性再往下細分成更小的問題，依此類推直到很容易撰寫程式的單元時為止。

由上而下的設計法的優點為程式可分工由多人共同撰寫、因程式已切割為小單元因此較容易除錯及容易維護；缺點則是程式段較長且執行時間較久。

以「學生資料處理系統」為例，說明由上而下的設計法的處理過程如下：依照學生資料處理系統的邏輯特性將問題切割成處理「基本資料」及「成績資料」兩個較小的問題，接下來「成績資料」同樣依照邏輯特性再往下細分成處理「操行成績」及「學業成績」兩種。程式設計師便可依此設計原則所得之結果來設計程式。

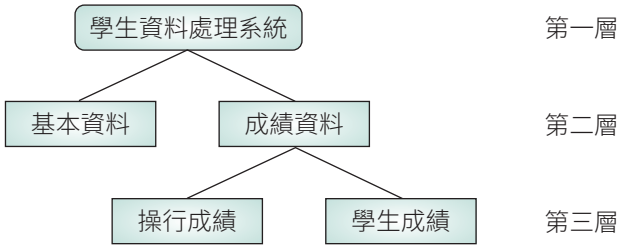


圖 4-1 「學生資料處理系統」架構

2. 由下而上的設計法

由下而上的設計法是指由問題中最容易編寫程式的單元開始設計程式，然後逐級往上層組合成較完整的程式。同樣以上圖「學生資料處理系統」為例，先設計完成「操行成績」及「學業成績」兩個最下層 (第三層) 的程式後，將此兩個程式合併成「成績資料」程式。然後再設計完成第二層「基本資料」程式，最後合併第二層的「基本資料」及「成績資料」兩個程式成為最上層的「學生資料處理系統」。

4.5 基礎資料結構

資料結構所包含的主題有兩大類，一類是表示資料的基本工具，另一類則是常用的演算法。資料結構、演算法及程式設計這三個主題彼此間具有密不可分的關係。在設計程式的過程中經常被用來表示資料的基本工具，例如陣列、鏈結串列、堆疊、佇列及樹狀結構等將在本節中介紹。另外，在撰寫程式的過程中所經常使用的演算法例如搜尋法、排序法及圖形等內容則在第五章中再做介紹。

4.5.1 陣列

一、基本觀念

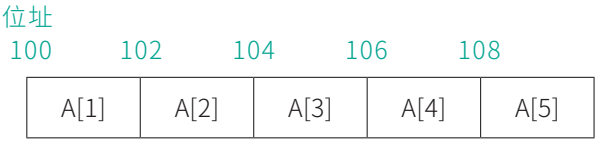
陣列主要是由陣列的名稱，維度 (Dimension)，元素型態以及陣列註標 (Index) 組成。

陣列有兩項特別的限制：

1. 必須配置連續的記憶體空間給陣列元素使用。
2. 陣列中所有元素的型態必須完全相同，也就是說每個元素所佔用的記憶體空間必須是相同的。

範例

若 A 是一個包含 5 個元素的陣列，元素的型態為整數。若整數型態資料佔用記憶體的空間為 2 個位元組，假設配置給陣列 A 的記憶體位址由 100 開始，利用 A[1]、A[2]、A[3]、A[4] 及 A[5] 來表示陣列的 5 個元素，則陣列 A 的所有元素之記憶體空間使用情形將如下圖所示：



由上圖知 A[1]、A[2]、A[3]、A[4] 及 A[5] 佔用了連續的記憶體空間且因為元素的型態相同，因此使用的記憶體空間的大小也相同。

二、陣列儲存方法

陣列在記憶體中的儲存方式可分為「以列為優先」(Row Major Ordering) 及「以行為優先」(Column Major Ordering) 兩種。其中「以列為優先」是在編排陣列中元素之順序時，由最右邊的註標值開始進位，而「以行為優先」則是在編排陣列中元素之順序時，由最左邊的註標值開始進位。一般來說，由於以列為優先法較符合人類的習慣，因此除了 FORTRAN 採用「以行為優先」外，其他的程式語言多是採用「以列為優先」。基於絕大部分的程式語言採用「以列為優先」法來儲存陣列元素，因此本書將只介紹「以列為優先」法。

範例 1

請針對二維陣列 X[1:3, 1:5]，說明在「以列為優先」法中，陣列元素在記憶體中排列的順序。

解

X 為一個二維陣列，第一個維度有三種可能值 (1、2 及 3)，第二個維度有五種可能值 (1、2、3、4 及 5)；因此共有 3×5=15 個元素，此 15 個元素若採用列優先方式儲存，其註標值與相對應之位置如下之說明：

第 1 個元素 [1,1]	第 2 個元素 [1,2]	第 3 個元素 [1,3]	第 4 個元素 [1,4]	第 5 個元素 [1,5]
第 6 個元素 [2,1]	第 7 個元素 [2,2]	第 8 個元素 [2,3]	第 9 個元素 [2,4]	第 10 個元素 [2,5]
第 11 個元素 [3,1]	第 12 個元素 [3,2]	第 13 個元素 [3,3]	第 14 個元素 [3,4]	第 15 個元素 [3,5]

說明：元素 X[1,1] 為第一個元素、X[1,2] 為第二個元素、…，而 X[3,5] 則為第十五個元素。

陣列求址公式

對於陣列中特定的元素若要知道儲存該元素的記憶體空間之位址，必須利用陣列求址公式來計算。陣列求址公式分為一維、二維及多維陣列三種，分別介紹如下：

1. **一維陣列**：陣列為 X(l:u)，其中 l 為陣列之註標下限，u 為陣列之註標上限，w 為每個元素所使用的記憶體空間大小。

   a. 陣列元素個數：$(u-l+1)$。

   b. 陣列佔用記憶體空間量：$(u-l+1)\times w$。

   c. 陣列 X 的位址函數 L：$L(X[i])=a+w\times(i-l)$

   其中 a 為陣列 X 在記憶體中之起始位址。

範例 2

假設一陣列規格如下：int X [1:100]；陣列起始位址為 20，一個整數佔用兩個記憶體空間，則 X 這個陣列佔用的記憶體空間的計算方法如下：

int X [1:100] 代表陣列 X 之註標值介於 1 至 100 之間，因此共有 100-1+1=100 個元素；且每個元素佔用兩個記憶體空間，因此陣列 A 佔 100×2=200 個記憶體空間。根據一維陣列求址公式：

$$L(X[i])=a+w\times(i-l)，$$

將 $a=20$，$l=1$，$w=2$ 代入一維陣列求址公式可得以下之結果：

$$L(X[i])=20+2\times(i-1)$$

若欲求 X[38] 元素之位址，將 i=38 代入上式中可得 X[38] 之位址如下：

$$L(X[38])=20+2\times(38-1)=94$$

因此，X[38] 之位址為 94。

2. **二維陣列**：陣列為 $X(l_1:u_1, l_2:u_2)$，其中 l_1 為陣列左方維度之註標下限，u_1 為陣列左方維度之註標上限，l_2 為陣列右方維度之註標下限，u_2 為陣列右方維度之註標上限，w 為每個元素所使用的記憶體空間大小。

   a. 陣列元素個數：$(u_1-l_1+1)\times(u_2-l_2+1)$。

   b. 陣列佔用記憶體空間量：$(u_1-l_1+1)\times(u_2-l_2+1)\times w$。

   c. 陣列 X 的位址函數，假設 α 為陣列 X 在記憶體中之起始位址：

   「以列為優先」位址函數：

$$L_{row}(X[i,j])=\alpha+w\times[(i-l_1)\times(u_2-l_2+1)+(j-l_2)]$$

範例 3

假設一陣列規格如下：int X [1:100, 5:90]；陣列起始位址為 20，一個整數佔用兩個記憶體空間，則 X 陣列佔用的記憶體空間的計算方法如下：

int X [1:100, 5:90] 代表陣列 X 左方維度之註標值介於 1 至 100 之間，因此共有 100－1+1=100 個元素；右方維度之註標值介於 5 至 90 之間，因此共有 90－5+1=86 個元素且每個元素佔用兩個記憶體空間，因此陣列 A 共有 100×86=8600 個元素，共佔用 8600×2=17200 個記憶體空間。根據二維陣列求址公式：

「以列為優先」位址函數：

$$L_{row}(X[i,j])=\alpha+w\times[(i-l_1)\times(u_2-l_2+1)+(j-l_2)]$$

將 α=20，l_1=1，u_2=90，l_2=5，w=2 代入求址公式可得以下之結果：

$$L_{row}(X[i,j])=20+2\times[(i-1)\times(90-5+1)+(j-5)]$$

若欲求 X[38, 39] 元素之位址，將 i=38，j=39 代入上式中可得 X[38, 39] 之位址如下：

$$L_{row}(X[38,39])=20+2\times[(38-1)\times(90-5+1)+(39-5)]=6452$$

因此，X[38,39] 之位址為 6452。

4.5.2 鏈結串列

鏈結串列 (Linked List) 是寫作程式時，常用的一種資料結構。在鏈結串列中是利用指標來表示元素之下一個元素的位址。常用的格式如下：

上圖中「資料欄位」用來存放資料內容，若資料項目有多個則可以有多個「資料欄位」，而「鏈結欄位」則是用來存放下一筆資料的位址。實際範例如下：

上圖中最後一筆資料的「鏈結欄位」內容為「nil」代表該筆資料的後方已無資料。

鏈結串列這項資料結構被提出的理由如下表：

理由	說明
記憶體空間的管理較有彈性	因為陣列元素存放在記憶體中，需佔用連續的記憶體空間，而鏈結串列的元素不需佔用連續的記憶體位置來存放，只需透過指標值即可得下一個元素的位置，因此鏈結串列對於記憶體空間的管理比較有彈性。
資料的插入或刪除較容易	若要對陣列中的元素執行插入 (Insert) 或刪除 (Delete) 的動作，由於可能會牽涉大量資料的搬移，所以需要耗費較多的時間，難度較高；若是對鏈結串列中的元素執行插入或刪除的動作，由於不會涉及資料搬移的動作，只需變動指標的指向即可，所以比較節省時間，難度較低。

鏈結串列與陣列的比較表如下：

種類 / 比較項目	陣列	鏈結串列
佔用記憶體	較少	較多
記憶體使用率	較低	較高
需要額外的鏈結欄位	否	是
插入資料	較慢	較快
刪除資料	較慢	較快

種類 / 比較項目	陣列	鏈結串列
合併串列	較困難	較容易
分離串列	較困難	較容易
彈性	較差	較佳
循序處理	較快	較慢
隨機處理	較快	較慢
存取方式	直接存取或循序存取皆可	較適合循序存取

4.5.3 堆疊

堆疊 (Stack) 是指一有序串列 (Ordered List)，僅能由一端做加入 (Add) 與刪除 (Deletion) 的動作，此端稱作開口端 (或頂端)，而另一端則稱為封閉端 (或底端)，所以堆疊具有先進後出 (First In Last Out，FILO) 或後進先出 (Last In First Out，LIFO) 的特性。本節將介紹堆疊相關的知識與用途。堆疊的圖形及操作示意圖請參考下圖。

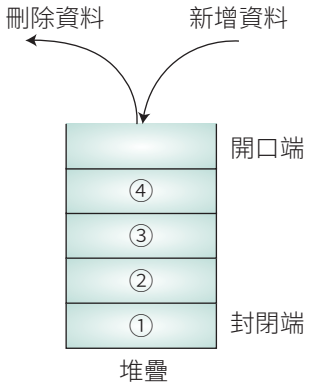


圖 4-2 堆疊操作說明

堆疊的操作

堆疊有下列五項主要操作，分別是 create (新建一個堆疊)、push (加入一新資料項進入堆疊)、pop (由堆疊中刪除一個資料項)、top (讀取出堆疊中最頂端的資料項，但堆疊內容未被破壞) 及 isempty (檢查是否為空堆疊)。相關的程式碼介紹如下：