2

陣　列

陣列是一種非常重要、運用相當廣泛的資料結構。

陣列的抽象定義是一組具有相同資料型態 (data type) 的元素，所組成的有序集合 (ordered set)。在實體電腦的配置中，陣列通常儲存在一塊連續的記憶體上。在程式語言中，我們可以用一個陣列名稱和一個註標 (index) 來唯一表示陣列中的任何一個元素；其中這個陣列名稱，即對應至此陣列在所配置記憶體中的起始位置，而註標值則指出：該陣列元素是自此起始位置數起的第幾個位置，兩相搭配，可以快速地定出任一陣列元素在記憶體中的位址[1]。於是相同性質的資料，可以利用陣列結構儲存，配合迴圈變數做為註標，即可有效率地逐一處理陣列中的各個元素。

本章將介紹陣列的應用技巧，和其在實體記憶空間中配置的關係。前者為各位開出善用陣列資料結構之門，後者引領各位瞭解陣列資料結構如何存放在電腦中，以及程式語言、編譯器如何運作，方便陣列得以有效率地為吾人所用。

2.1 陣列的宣告與使用

任何資料結構的引用，皆需透過程式語言先行宣告，陣列的宣告必須包含三項要素：

(1) 陣列的名稱；

(2) 陣列的大小；

(3) 陣列中元素的資料型態。

我們舉例說明陣列的宣告與使用：

1 因此不論元素存放在陣列中的任何一個位置，它都能在相同的時間內完成儲存或取出的位置運算。

範例 2-1

程式 2-1 陣列內元素值的指定(assignment)

```
1  main()
2  {   int list[5];        //list 為一整數陣列，共有 5 個元素
3      int i;
4      for (i=0; i<5; i++) list[i]=i+1;
5  }
```

此 C 程式中第 1 行的敘述（int list[5];）即宣告了一個陣列—名稱爲 list，大小爲 5，而且陣列的每一元素皆是整數。程式執行後陣列 list 中的 5 個整數元素分別爲迴圈變數 i（i=0,1,2,3,4）加 1，亦即 1,2,3,4,5；如圖 2-1 所示（此圖雖非陣列 list 在記憶體中的實際配置，但已足夠具象地描繪陣列之抽象概念，我們稱之爲陣列 list 的邏輯圖示 (logical representation)）。

list[0]	1
list[1]	2
list[2]	3
list[3]	4
list[4]	5

圖 2-1　陣列 list 的邏輯圖示

各位可以發現陣列名稱（如上例的 list）配合註標，可以方便地指明陣列中的任一個元素；利用迴圈控制變數(loop control variable，如上例的變數 i）做爲會變動的陣列註標，即可透過迴圈的執行，逐一對陣列中的任何元素做適當的運算（見上例第 3 和 4 行）[2]。

2　請注意：在 C 語言中，陣列註標的編號從 0 開始（範例 2.1 第 4 行中迴圈控制變數 i 從 0 開始，到 4 結束）。不同的程式語言有不同的陣列語法。

範例 2-2

程式 2-2 陣列元素的指定與陣列對應元素相加

```
1   #define size 5
2   main()
3   {   int A[size], B[size], C[size];
4       int i;
5       for (i=0; i<size; i++)
6       {   A[i] = i+1;
7           B[size-i-1] = A[i];
8       }
9       for (i=0; i<size; i++)
10          C[i] = A[i]*B[i];
11  }
```

此程式執行完畢後陣列 A、B、C 的內容值將如圖 2-2 (a)、(b)及(c)所示。

A[0]	1	B[0]	5	C[0]	5
A[1]	2	B[1]	4	C[1]	8
A[2]	3	B[2]	3	C[2]	9
A[3]	4	B[3]	2	C[3]	8
A[4]	5	B[4]	1	C[4]	5
(a)		(b)		(c)	

圖 2-2 陣列 A、B 和 C 的邏輯圖示

這個範例提示您：註標是可以先行計算的，如上例第 7 行的

```
    B[size-i-1] = A[i];
```

而且利用常數 (constant) 定義—即上例第 1 行的

```
    #define size 5
```

可增加程式的可讀性，亦可減少程式修改、除錯時可能的疏忽[3]。

範例 2-3

若系上同學的姓名以座號爲註標，存放在陣列 `name` 中：1 號存在 `name[1]`、2 號存在 `name[2]`、…依此類推，`rank[i]`存放著考試第 *i* 高分同學的座號，最高分的同學座號爲 `rank[1]`，次高分同學座號爲 `rank[2]`，則 `name[rank[i]]`可取得此第 *i* 高分同學的姓名。圖 2-3 描繪了 `i=3`，即考試第 3 高分同學的姓名：

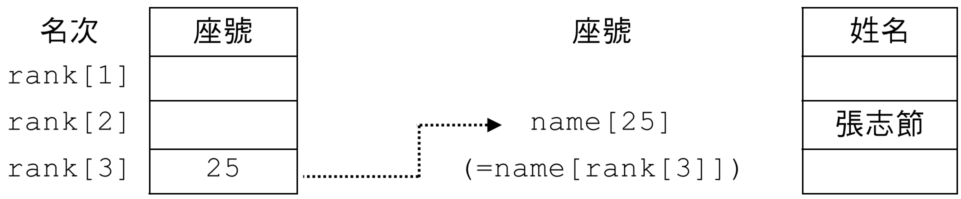


圖 2-3　使用間接參照取得第 3 高分同學的姓名

範例 2-3 中利用了「間接參照」(indirect reference) 的技巧，即陣列 `name` 需要的註標，是以 `i`「直接參照」(direct reference) 陣列 `rank` 後的結果。這也是陣列使用中常用的技巧；這與範例 2-2 中提到「註標可以先行計算」的用法類似。此時陣列 `rank` 所存的是陣列 `name` 的註標，可稱前者爲後者的「索引」(index)—取其可用前者快速地找到後者的含意。目前陣列 `rank` 存放陣列 `name` 的註標，倘若 `rank` 存放的是 `name` 的位址—即指標—會在第 4 章討論。不論是利用註標或指標，間接參照或建立索引皆是重要的概念。

下面我們深入探討挑選排序法（節 2.2.1）並介紹二元搜尋法（節 2.2.2）、矩陣的運算（節 2.2.3）和魔術方塊（節 2.2.4），加深各位對陣列運算的概念和技巧。

3　試想：您的程式有八千行，而有八十個迴圈引用了陣列 *A*，但因需求增加，欲將陣列大小、迴圈次數由 1000 改為 1500；但是您的程式沒有用常數定義或變數，只有八十多個「1000」散佈在八千行的各個角落，會很難維護修改。

2.2 陣列的運算

2.2.1 挑選排序法

將資料由小到大或由大到小的次序排列即為「排序」。排序的各種演算法、複雜度分析等課題將在第 7 章做深入討論。在 1.4.1 節範例 1-2 中，我們曾討論過挑選排序法，在此將焦點放在陣列的使用上，再深入討論這個演算法，並介紹分析演算法的技巧。

我們將程式 1-1 改寫成程式 2-3：

程式 2-3 挑選排序法

```
1  void swap(int *x, int *y)
2  {   int temp;
3      temp = *x;
4      *x = *y;
5      *y = temp;
6  }
7  void selectionSort(int data[], int n)
8  {   int i, j;
9      int min;
10     for (i=0; i<n-1; i++)
11     {   min = i;
12         for (j=i+1; j<n; j++)
13             if (data[j]<data[min]) min = j;
14         swap(&data[i], &data[min]);
15     }
16 }
```

其中第 14 行的資料對調已改寫成程序呼叫（程式 1-1 是採用巨集）：

```
swap(&data[i], &data[min]);
```

參數&data[i]（&data[min]）意指傳入的是變數 data[i]（data[min]）的住址；而程序 swap 的宣告則在第 1~6 行。欲執行挑選排序，可呼叫

```
selectionSort(data, n);
```

其中 data 應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。第 10 行的迴圈變數 i 改成自 0 到 n-2(請想一下原因)。在此我們用圖 2-4 提醒各位程序中參數 (parameter) 的傳遞：

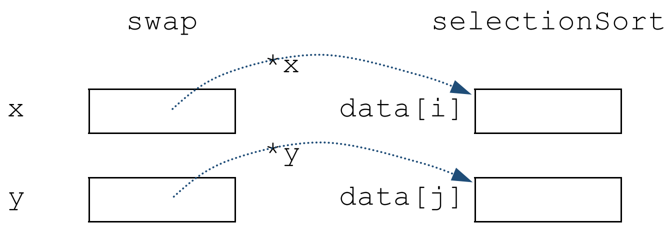
(以指標變數 int *x, *y 記載傳入參數 data[i], data[min]的位址)

宣告　　void swap(int *x, int *y)

呼叫　　swap(&data[i], &data[min]);

(將 data[i], data[min]的位址傳入程序 swap 中)

(a) 傳位法



(b) 位址與間接參照

(以 data 傳入參數陣列 data 的位址、n 記載傳入參數 n 值)

宣告　　void SelectionSort(int data[], int n)

呼叫　　SelectionSort(data, n);

(將陣列 data 位址，n 的值傳入程序 SelectionSort 中)

(c) 傳位與傳值法

圖 2-4　參數傳遞的說明

圖 2-4 (a) 採用了「傳位法」 (passed by address)—所傳的內容爲位址—將 `data[i]`和 `data[min]`的位址(`&data[i]`和`&data[min]`)傳入程序 `swap` 中，而在程序 `swap` 內，用指標變數(`int *`)`x` 和 `y` 分別記住其位址，在第 `3~5` 行以`*x` 和`*y`「間接參照」地執行資料的對調，如圖 2-4 (b)，以確保兩者位址內的資料會被對調(對調後的結果，會反應在呼叫程序 `selectionSort` 中)。圖 2-4 (c) 中陣列 `data` 是以傳位法傳入 `selectionSort` 中—陣列變數 `data` 本身即爲陣列的起始位址[4]，因爲排序的結果必須反映在陣列 `data` 內，傳入位址可方便取得各陣列的值；而 `n` 是以「傳値法」(passed by value) 傳入 `selectionSort` 中，在此 `n` 的值不致被修改[5]。

在節 1.5.1 的範例 1-7 中，我們用程式 1-6、1-7 和 1-8 來求得程式的執行步驟，在節 1.5.2 也說明了我們比較在乎程式在輸入資料增大時，程式執行時間增加的趨勢。所以在分析演算法或程式的時間複雜度時，只須擇定程式中對時間最具決定性的指令(簡稱之爲關鍵指令)，觀察其執行時間的大 O 表示即可。通常這個「對時間最具決定性的指令」，是被最多層迴圈包圍的指令[6]。以程式 2-3 而言，第 `13` 行的比較大小和 `if` 判斷，即爲關鍵指令，它同時也被兩層迴圈包圍；其執行總計次數爲：

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}O(1)$$

$$= \quad O(1)+O(1)+\ \cdots\ +O(1) \qquad (n\text{–}1\text{ 個，}i=0)$$

4 在 C 中，`data` 若宣告為陣列，則變數 `data` 本身即為陣列的起始位址。

5 在這個例子中，`n` 用傳位法傳遞、甚至 `data` 和 `n` 都用全域變數 (global variable)，也不致出錯；但 `n` 用傳值法可增加程式的可讀性，對程序的可再使用性 (reusability) 也較明確。

6 在 1.5.2 節中曾提到，若以 n 為輸入的時間函數 f 中 n 的最高乘冪為 k，則其複雜度為 $O(n^k)$。各位可深思「被最多層迴圈包圍的指令」與其之關係。

$$+\,O(1) + \cdots + O(1) \qquad (n\text{–}2\ 個，i = 1)$$

$$\cdots$$

$$+\,O(1) \qquad (1\ 個，i = n\text{–}2)$$

$$= \frac{n \times (n-1)}{2}$$

$$= O(n^2)$$

範例 2-4

欲對下列 6 筆資料：9、8、4、7、2、3，進行排序，先行存入陣列 data 中：

data[0]	9
data[1]	8
data[2]	4
data[3]	7
data[4]	2
data[5]	3

挑選排序程式中的第 10 行會被執行 5 次 (每次挑出排序資料中最小者)，每次的執行結果如圖 2-5 示：

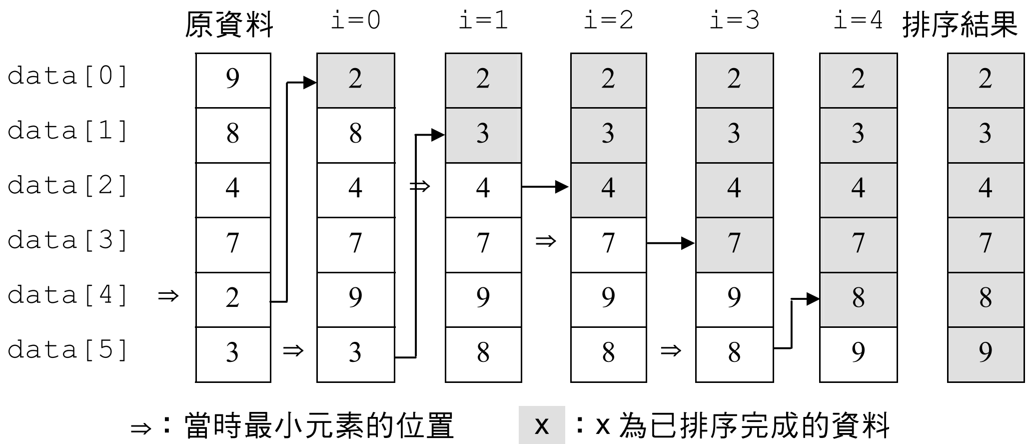


圖 2-5　挑選排序法的執行過程

請注意：雖有 6 筆資料，但第 10 行只需執行 5 次（前第 1, 2, …, 5 小的元素都出現了，剩下的一個即爲最大者，或說是第 6 小者）。

程式 2-3 的挑選排序，只藉著迴圈和陣列，適當更動迴圈控制變數做爲陣列的註標，即可透過比較資料大小、調換資料所在位置，來完成排序的目的。

2.2.2 二元搜尋法

簡單地說，「搜尋」(searching) 是在一群資料找出所需要的。我們可以利用陣列存放這群資料，再配合適合的演算法來搜出所要的資料。假設資料已存放在陣列 data 中，共計 n 筆，而需要找出資料 x 在 data 中的位置。我們先看看直覺的「線性搜尋法」 (linear search) —這個方法是從第一筆資料開始，依序一一比對，直到找到所要的資料、或找不到爲止。下面爲其演算法：

演算法 2-1 線性搜尋法：

輸入： x, data[0], data[1],..., data[n-1]

輸出： 資料 x 所在的位置(註標)或-1(若 x 不在陣列 data 中)

```
1  for (i=0; i<n; i++)
2      if (x == data[i]) return i;
3  return -1;
```

這個演算法自 i=0 起，核對 data[i]是否與 x 相等，若相等，則演算法傳出位置 i 後停止；若不相等，則換下一筆，繼續比對；若陣列所有資料皆無 x，則傳回-1，知會呼叫程序：x 不在陣列 data 中。這個演算法十分簡單，在最差情況時—即 x 不在陣列 data 中—的時間複雜度爲 $O(n)$，因爲此時所有的元素得逐一比對（第 1~2 行共執行 n 次）。當資料未經過排序時，以線性搜尋法來尋找資料，倒也是個可行的方法；但若資料已經過排序（由小到大），則可以用「二元搜尋法」(binary search)，來提高搜尋的速率，其基本想法如下：

➡ 檢查數列中間之資料是否為所求？若是即找到了，否則再看：

- 若中間資料大於 x，對前半部數列進行二元搜尋
- 若中間資料小於 x，對後半部數列進行二元搜尋

對應的演算法可整理如下：

演算法 2-2 二元搜尋法：

輸入：已由小至大排序的陣列 data 及欲搜尋的資料 x

輸出：若 x 在 data 陣列中，則輸出 x 所在位置之註標，否則輸出-1

```
1   lower=0; upper=n-1;
2   while (lower <= upper)
3   {  mid = (lower+upper)/2;
4      if (data[mid] == x)
5          return(mid);
6      else
7          if(data[mid] < x)
8              lower = mid+1;     //  data[mid] < x
9          else
10             upper = mid-1;     //  data[mid] > x
11  }
12  return -1;
```

第 4 行檢查 data[mid]與 x 是否相等，若二者相等，則表示在位置 mid 中已找到 x 了，遂由第 5 行傳回 mid；第 7 行檢查：若 x 介於 data[mid+1]~data[upper]之間，遂把 lower 調至 mid+1（第 8 行）；否則第 9~10 行執行時，表示 data[mid]大於或等於 x，即 x 只可能在 data[lower]~data[mid-1]之間，於是把 upper 調成 mid-1。當 lower > upper 且仍未找到 x 時，表示 x 並不在陣列 data 中，應傳回-1（第 12 行）。

要把這個演算法改寫成完整的 C 程式應十分容易；倒是二元搜尋的概念，

用遞迴的思維來考量也清楚易懂，我們將二元搜尋法用遞迴方式改寫成下面的程序：

程式 2-4 二元搜尋法—遞迴呼叫

```
int BinarySearch(int data[], int x, int left, int right)
{   int mid;
    if (left <= right)
    {   mid = (left+right)/2;
        if (data[mid] == x)  return(mid);
        if (data[mid] < x)
            return BinarySearch(data[], x, mid+1, right);
        else
            return BinarySearch(data[], x, left, mid-1);
    }
    return -1;
}
```

範例 2-5

在已經過排序的陣列 data 中，利用二元搜尋法檢查 x=20 是否在其中。請見圖 2-6 執行二元搜尋法的過程。

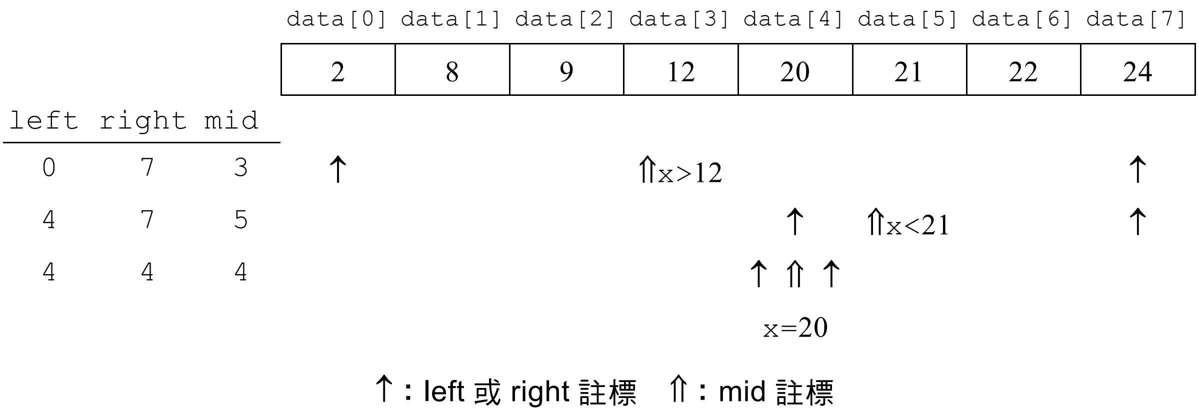


圖 2-6 二元搜尋法的執行過程

請注意：由於陣列 `data` 已排序完成，二元搜尋法的每次比對，皆可保證當時的一半資料不必再予考慮，可以去掉 (prune)；當資料共有 n 筆時，每次比對可去掉一半的資料，$\lceil \log n \rceil$ 次比對後，一定可知所尋是否在其中。即在最差的情況下（`x` 根本不在陣列 `data` 中），也只需 $O(\log n)$ 次的比對，即可完成搜尋。所以二元搜尋法的時間複雜度為 $O(\log n)$，比起線性搜尋法（時間複雜度為 $O(n)$），二元搜尋法的效率比較好。在資料量 (n) 大的時候，其劣優更是可見，但切記：二元搜尋法實施之前，請確定欲搜尋的數列已經過排序。

在演算法中有一類「刪減並搜尋」(prune and search) 策略—每回合可將資料分類，只搜尋有必要的某些類，不必要者可刪減之；二元搜尋法即屬於此策略，因其每回合可保證當時的一半資料可以去掉。

請各位思考一下，排序最好的演算法需 $O(n \log n)$（詳見第 7 章），而線性搜尋法所用 $O(n)$ 的時間，有可能反而划算；例如在週末的電影版報紙廣告中，利用線性搜尋，找出想看電影 M 的播放戲院，應是較有利的方式；先用電影片名的筆劃順序排列，然後用二元搜尋法找出電影 M 的播放戲院，不免事倍功半。但若像電話接線生回答查詢者的電話查詢需求，就非得將資料事先排序，再使用二元搜尋法查詢不可，因為這樣的需求頻率十分高，必須有快速的處理成效。不經排序，直接用循序搜尋法，對多次需求做回應，是非常沒有效率的。若搜尋需求有 k 次，則排序與二元搜尋法的搭配，共需時間 $O(n\log n+k\log n)$，而循序搜尋法需時 $O(kn)$，因此各位可自行判斷二種解決方法的適用時機。

目前我們用到的陣列皆為一維陣列，而二維甚至三維陣列也是常用的資料結構。下一小節我們即有二維陣列的應用。

2.2.3 矩陣的運算

矩陣 (或行列式) 是一種數學上的重要結構，恰可用二維陣列表示之。假設有一 $m \times n$ 的矩陣 A，可用 a_{ij} 表示矩陣 A 中第 i 列、第 j 行的元素，$0 \le i \le m-1$

且 $0 \le j \le n-1$。若矩陣 A 是一個 3×4 的整數矩陣，我們可以 C 語言宣告矩陣 A 如下：

```
int A[3][4];
```

此二維整數陣列 A 共有 3 列 4 行，而 `A[i][j]` 即對應了矩陣 A 之元素 a_{ij}。

二維陣列可以用來解決許多與矩陣運算有關的問題，例如：解聯立方程式、矩陣 FFT (Fast Fourier Transformation) 轉換、圖形 (graph) 中的問題、…等等。在此我們先針對三個簡單且基本的運算進行介紹，加強各位對二維陣列的認識，這三個運算分別為矩陣的轉置 (transpose)、相加與相乘。

2.2.3.1 矩陣的轉置

若 A 為 $m \times n$ 的二維矩陣，a_{ij} 為其第 i 列 j 行元素，$0 \le i \le m-1$ 且 $0 \le j \le n-1$；則 A 的轉置矩陣 B 為 $n \times m$ 的矩陣，b_{ji} 為其第 j 列 i 行元素，且 $b_{ji} = a_{ij}$。若矩陣 B 是 A 的轉置矩陣，可表示為 $B = A^T$。範例 2-6 提供一個簡單的例子。

範例 2-6

$$A = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}_{4\times 3} \qquad B = A^T = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}_{3\times 4}$$

下面是矩陣轉置的演算法：

演算法 2-3 矩陣轉置

輸入：m×n 矩陣 A

輸出：矩陣 B，$B=A^T$

```
將二維矩陣 A 的元素載入二維陣列 A 中：A[i][j] = aij
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        B[j][i]=A[i][j];
return B;
```

各位應可看出這個演算法的時間複雜度為 $O(m \times n)$。

2.2.3.2 矩陣的相加

兩個矩陣 A 及 B 相加的運算，需將各對應元素相加，令矩陣 $C = A+B$，則 $c_{ij} = a_{ij}+b_{ij}$，而且矩陣 A、B 和 C 的大小必須一致。例如下面矩陣 A 及 B 相加得矩陣 C。

範例 2-7

若

$$A = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix}_{2\times3} \quad 且\ B = \begin{bmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \end{bmatrix}_{2\times3}$$

則

$$C \begin{bmatrix} 2+1 & 4+3 & 6+5 \\ 8+7 & 10+9 & 12+11 \end{bmatrix}_{2\times3} = \begin{bmatrix} 3 & 7 & 11 \\ 15 & 19 & 23 \end{bmatrix}_{2\times3}$$

下面是矩陣相加的演算法：

演算法 2-4 矩陣相加

```
輸入：m×n 矩陣 A,B
輸出：矩陣 C，C = A+B
將二維矩陣 A,B 的元素載入二維陣列 A,B 中：A[i][j]=aij；B[i][j]=bij
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        C[i][j] = A[i][j]+B[i][j];
return C;
```

這個演算法的時間複雜度為 $O(m \times n)$。

2.2.3.3 矩陣相乘

兩個矩陣 A 及 B 能夠相乘的前提是 A 的行數必須等於 B 的列數，若 A 的大小為 $m \times n$，B 為 $n \times p$，則 $C = A \times B$ 為 $m \times p$ 的矩陣。其運算表示如下：

$$A = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \cdots & a_{(m-1)(n-1)} \end{bmatrix}, \quad B = \begin{bmatrix} b_{00} & b_{01} & \cdots & b_{0(p-1)} \\ b_{10} & b_{11} & \cdots & b_{1(p-1)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{(n-1)0} & b_{(n-1)1} & \cdots & b_{(n-1)(p-1)} \end{bmatrix},$$

$$C = A \times B = \begin{bmatrix} c_{00} & c_{01} & \cdots & c_{0(p-1)} \\ c_{10} & c_{11} & \cdots & c_{1(p-1)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(m-1)0} & c_{(m-1)1} & \cdots & c_{(m-1)(p-1)} \end{bmatrix};$$

其中

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \times b_{kj} \;\; ; 0 \le i \le m-1,\ 0 \le k \le n\text{-}1,\ 0 \le j \le p-1$$

亦即 c_{ij} 為 A 的第 i 列與 B 的第 j 行對應元素相乘後的和，或者我們可以解讀成 A 第 i 列所成向量，與 B 第 j 行所成向量的內積；下面以實際範例說明。

範例 2-8

若

$$A = \begin{bmatrix} 1 & 3 \\ 5 & 7 \\ 9 & 11 \end{bmatrix}_{3\times 2}, \quad B = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 5 & 4 & 3 & 2 \end{bmatrix}_{2\times 4}$$

則

$$C=\begin{bmatrix} 1\times4+3\times5 & 1\times3+3\times4 & 1\times2+3\times3 & 1\times1+3\times2 \\ 5\times4+7\times5 & 5\times3+7\times4 & 5\times2+7\times3 & 5\times1+7\times2 \\ 9\times4+11\times5 & 9\times3+11\times4 & 9\times2+11\times3 & 9\times1+11\times2 \end{bmatrix}_{3\times4}$$

$$=\begin{bmatrix} 19 & 15 & 11 & 7 \\ 55 & 43 & 31 & 19 \\ 91 & 71 & 51 & 31 \end{bmatrix}_{3\times4}$$

矩陣相乘的演算法如下：

演算法 2-5 矩陣相乘

```
輸入：m×n 矩陣 A，n×p 矩陣 B
輸出：矩陣 C，C = A×B
將二維矩陣 A,B 的元素載入二維陣列 A,B 中：A[i][j] = aij；B[i][j] = bij
for (i=0; i<m; i++)
    for (j=0; j<p; j++)
    {    C[i][j] = 0;
         for (k=0; k<n; k++)
             C[i][j] += A[i][k]*B[k][j];
    }
return C;
```

這個演算法的時間複雜度為 $O(m\times n\times p)$。

2.2.4 魔術方陣

「魔術方陣」(magic square) 指的是一 $n\times n$ 的矩陣，每一方格內可放一個 $1\sim n^2$ 的正整數，使得每一行、列和對角線的整數和皆相等。範例 2-9 提供了兩個魔術方陣。

範例 2-9

8	1	6
3	5	7
4	9	2

(a) 3×3

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

(b) 5×5

圖 2-7 魔術方陣

當 n 是奇數時，H. Coxeter 歸納出了產生魔術方陣的規則：

- 1 填在第 1 列中間；
- 將方陣想成上下、左右、對角相接的圓球，往左上角填入下一個數字；若左上角已遭其它數字佔用，則填在目前方格的下方。

圖 2-7 (b) 的魔術方陣即是如此產生的（而圖 2-7 (a) 則將規則 2 的往左上角移動可改爲往右上角移動）。演算法 2-6 較正式地描述了這組規則。

演算法 2-6 魔術方陣

```
輸入：奇數 n
輸出：n×n 魔術方陣 M
(i, j) = (0, n/2)      // i=0, j=n/2
M[i][j] = data = 1
while (data <= n*n)
{    (k,l)=(i,j)在圓球的下一個左上角位置
     if (M[k][l]已有數字)(i,j)=(i,j)在圓球的正下方位置
     else (i,j)=(k,l)
     M[i][i]=data++
}
return M
```

演算法 2-6 可改寫成程式 2-5 如下：

程式 2-5 魔術方陣

```
#define MaxSize 21
int square[MaxSize][MaxSize];
void MagicSquare(int n)
{   int i, j, k, l, data;
    if ((n>MaxSize) || (n<1))
    { cerr<<"輸入方塊過大或小，不予處理。"<<endl;  return;  }
    else if ((n%2)==0)
    { cerr<<"輸入方塊邊長為偶數，不予處理。"<<endl;  return; }
    for (i=0; i<n, i++)
        for (j=0; j<n, j++)
            square[i][j] = 0;
    i = 0;  j = (n-1)/2;
    square[i][j] = 1;
    data = 2;
    while (data <= n*n)
    {   k = (i-1 < 0) ? n-1 : i-1;
        l = (j-1 < 0) ? n-1 : j-1;
        if (square[k][l] > 0) i = (i+1)%n;
        else { i = k;  j = l; }
        square[i][j] = data++;
    }
    cout<<n<<"x"<<n<<"的魔術方陣"<<endl;
    for (i=0; i<n; i++)
    {   for (j=0; j<n; j++)
            cout<<square[i][j]<<" ";
        cout<<endl;
    }
}
```

程式 2-5 的 16、17 行計算了 k、l 兩個註標值，它們是下一個「左上角」的陣列註標，第 18 行先行檢查該位置是否已填過別的數字？若是，則往正下方填，於是用 `i = (i+1)%n` 確保列的正確註標，而行註標 j 此時是不變的（`[(i+1)%n][j]`位於`[i][j]`的正下方）；若左上角尚未填過數字，則用 `[k][l]` 取代 `[i][j]`；在 20 行處將 data 值填入 `square[i][j]`中。爾後 data 自行加 1，透過 while 迴圈繼續填下一數字。

15~21 行包含了一個 while 迴圈，迴圈控制變數爲 data，data 會從 2 起逐漸加 1 至 n^2；足見其時間複雜度爲 $O(n^2)$。而這個問題須將魔術方陣的 n^2 個方格皆填上數字，至少須 $\Omega(n^2)$ 的時間。由此可知程式 2-5 已是解此問題的最佳演算法，時間複雜度爲 $\Theta(n^2)$。

魔術方陣一旦求得，讀者可自行將之旋轉 90、180、270 度，其結果自然也是魔術方陣。至於如何寫出求得這些魔術方陣的演算法（注意：1 填在第 1 列中間，旋轉 90 (180、270) 度後，它位於最末 1 行（最末 1 列、第 1 行）的中間；移動方向也要隨而更動…），各位不妨嘗試看看。

2.2.5 騎士巡迴

根據西洋棋中騎士的 L 型走法，我們可以走完整個西洋棋盤中 64 (= 8×8) 個位置，一步一格，不致重複；所走過的路徑即稱爲「騎士巡迴」(knight's tour)。這個找出巡迴路徑的問題，在十八世紀初倍受數學家和拼圖迷的注意。這個問題可從棋盤的任何地方開始移動騎士，以 L 型走法，走過棋盤 64 格的每一格，每一格只能走一次，可以在棋盤上標上 1, 2, … , 64，表示騎士所走過的順序。範例 2-10（圖 2-10）有騎士巡迴的例子。

圖 2-8 描繪了西洋棋盤中騎士 K 在 (3, 4) 位置所能走的 8 個 L 型走法(在此以數字 1~8 依順時鐘方向表示）：

	0	1	2	3	4	5	6	7
0								
1				8		1		
2			7				2	
3					K			
4			6				3	
5				5		4		
6								
7								

圖 2-8　西洋棋盤及騎士的 L 型走法

我們先想想若騎士現在位於 (i, j) 上，在一般狀況下，它所能移動的八個方向分別是：$(i–2, j+1)$，$(i–1, j+2)$，$(i+1, j+2)$，$(i+2, j+1)$，$(i+2, j–1)$，$(i+1, j–2)$，$(i-1, j–2)$，$(i–2, j–1)$。請注意，假如 (i, j) 非常靠近棋盤邊緣，則可能有一些方向的移動會讓騎士超出棋盤，這是不允許的。騎士所能走的（最多）8 個方向可以用兩個位移陣列 *move_dx* 和 *move_dy* 來表示，如圖 2-9。那麼在 (i, j) 的騎士下一步可以移動的位置即可用 $(i+move\_dx[k], j+move\_dy[k])$ 來涵蓋，其中 k 是 0 到 7 之間的某個整數值。令 k 為 `for` 迴圈的註標，8 個可能步伐的位置即可在迴圈中容易地決定。

	0	1	2	3	4	5	6	7
move_dx	-2	-1	1	2	2	1	-1	-2
move_dy	1	2	2	1	-1	-2	-2	-1

圖 2-9　騎士步伐與位移矩陣

J.C. Warnsdorff 在 1823 年提出了一個騎士巡迴問題的解法：騎士所要走的下一格，是從下個可行的 L 型位置中挑出可再走下一步的位置數目最少者；亦即：若騎士的下一步有甲、乙、丙三個選擇，而甲、乙、丙再往下各有 3, 2, 4 個選擇，則騎士的下一步會走到乙（其下一步的可行選擇是為最小）。這解法通常可以找到一條路徑，但是並不一定能夠成功！以下用「經驗法則」(heuristic) 來稱呼它。下面是使用 Warnsdoff’s 規則來解決騎士巡迴的經驗法則：

經驗法則 2-7 騎士巡迴問題

輸入：棋盤大小 n、騎士在棋盤 K 上的起點座標 x, y

輸出：n*n 棋盤 K 上騎士巡迴的路徑，在棋格中以 1~64 的編號表示

```
for (each 0≤i≤7, 0≤j≤7) K[i][j]=0 ;  //初始化棋盤
讀進 n, x 和 y; K[x][y]=1;        //讀入起始點
for (2 ≤ step ≤ n*n)
{   cnt = 0; //找出從(x, y)可走下一步的座標（最多 8 個）
    for (0≤k≤7)
    {   (tx, ty)= (x+move_dx[k], y+move_dy[k]);
         if ((0≤tx≤7 && 0≤ty≤7)   && K[tx][ty] == 0))
        {   (next_x[cnt], next_y[cnt]) = (tx, ty);
            cnt++;    //(tx, ty) 在棋盤內且未曾走過，遂記錄之
        }
    }
    if (cnt == 0) {  //回傳 "自(x, y)出發，無騎士巡迴路徑" 訊息 }
    for (0 ≤ t < cnt)  // 針對每個 (next_x[t], next_y[t])
    {   (x, y) = (next_x[t], next_y[t]) //看其下一步有幾個可能
        for (0≤k≤7)
        {   (tx, ty)= (x+move_dx[k], y+move_dy[k]);
            if ((0≤tx≤7 && 0≤ty≤7)&& K[tx][ty] == 0))
                next_move[t]++;
        }
    }
    min = 0;
    for (1 ≤ t < cnt)  // 針對每個 (next_x[t], next_y[t])
    {   if (next_moves[t] < next_moves[min])
            min = t;
    }
    (x, y) = (next_x[min], [next_y[min]]    //移動騎士
    K[x][y] = step;            //記錄騎士的步數
}                 // 自(x, y)繼續嘗試下一步
return K;         // 棋盤上的數字即為自(x, y)出發的騎士巡迴路徑
```

第 5~11 行的 for 迴圈將合法的（由第 7 行檢測）下一步 (tx, ty)一一存入陣列(next_x[cnt], next_y[cnt])中，cnt 也逐次加 1，顯然後者陣列會將所有合法的下一步皆存下來。12 行判斷 cnt 是否為 0，目的在看合法的下一步究竟有沒有，若沒有即列印：該起點無騎士巡迴路徑。第 13~20 行的 for 迴圈則對(next_x[t], next_y[t])再找其可能下一步的個數，記錄在 next_move[t]中。第 21~27 行會找出 next_move[*]（用*表示所有的合法註標）中最小者—令其為 next_moves[min]；而其對應的(next_x[min]], [next_y[min])即為應踏出的下一步（第 27 行）。第 3~28 行的 for 迴圈負責把所有 n×n 個位置走完。

範例 2-10

圖 2-10 列出兩個實作經驗法則 2-7 後所得到騎士巡迴，其一大小為 6×6、另一為 8×8；注意：兩者騎士巡迴的起點皆不同。

25	32	11	2	19	34
10	1	26	33	12	3
31	24	9	18	35	20
8	17	36	27	4	13
23	30	15	6	21	28
16	7	22	29	14	5

(a)

55	14	63	34	1	16	21	36
64	33	56	15	62	35	2	17
13	54	61	58	45	20	37	22
32	57	46	51	42	59	18	3
47	12	53	60	19	44	23	38
28	31	50	43	52	41	4	7
11	48	29	26	9	6	39	24
30	27	10	49	40	25	8	5

(b)

圖 2-10　騎士巡迴的例子

Warnsdorff 經驗法則是個練習活用陣列的好範例，讀者可試著實作此經驗法則，肯定會對陣列的使用更加上手。

2.3 陣列在記憶體中的位置

以抽象的觀點來看，陣列只須是個有序集合—即任一元素有唯一的註標與之對應即可。但一般的程式語言和編譯器 (compiler)，大都將陣列對應到電腦中一塊連續的記憶體區塊，好方便電腦對陣列內元素做存取的動作。因此我們將在本節中討論一維、二維與三維陣列，在實際記憶體中配置的情形。

2.3.1 一維陣列

前面提過陣列的宣告必須包含：(1) 陣列的名稱；(2) 陣列的大小（元素的個數）；和 (3) 元素的型態。以

```
int A[20];
```

爲例，陣列名稱爲 A，大小爲 20，型態爲整數。編譯器看到這樣的宣告，即在記憶體中預留 20 個整數的連續空間，供陣列 *A* 使用；假設這塊連續預留空間之起點爲 $0A00_{16}$（此爲 4 位元組 (byte)，16 進位的記憶體位址表示），若每個整數佔用 4 位元組，則存放陣列 *A* 記憶區塊的邏輯圖示即如圖 2-11。

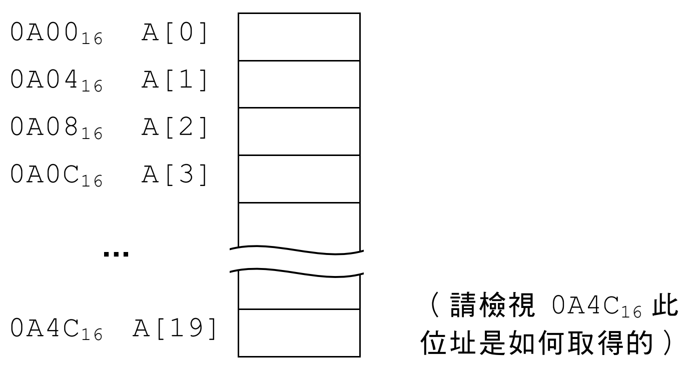


圖 2-11　陣列 A 在記憶體中的 20 個整數連續空間

若一維陣列 A 在記憶體中的起點爲 α ，而且任一元素佔用 l 位元組，若有 n 個元素，則陣列 A 在記憶體中的配置位置，可描繪如圖 2-12。其中 $A[i]$ 在記憶體中的位址爲 $\alpha + i \times l$：

α	$\alpha+l$	$\alpha+2\times l$	⋯		$\alpha+i\times l$	⋯		$\alpha+(n-1)\times l$
A[0]	A[1]	A[2]			A[i]			A[n-1]

圖 2-12　A[i]在記憶體中的位址為 $\alpha+i\times l$

介紹至此，各位可以知道陣列中任一元素，在記憶體中存放的位置，可以利用上述位址公式算出，因而陣列的存取時間與元素存放的位置無關[7]，亦即編譯器可在相同的時間內，決定任何位置陣列元素的位址，其存入（或取出）的時間皆爲一樣。

在 C 語言中利用「位址」亦可對陣列中的元素做存取動作；亦即 `A[i]=0` 與 `*(A+i)=0` 是同義的，變數 `A` 一旦宣告爲陣列，則單獨引用 `A` 變數名即指陣列之起始位置，而 `A+i` 即爲 `A[i]`的位址；在此不必再考量陣列元素的長度，因爲 C 編譯器會在各位宣告陣列 A 的型態時，主動記載其佔用的位元組數，以個人電腦爲例：整數 `int` 佔 4 個位元組，即 `sizeof(int)` 會等於 4，而 `sizeof(char)` 等於 1 、`sizeof(short)` 等於 2、`sizeof(float)` 爲 4、`sizeof(long)` 爲 8、`sizeof(double)` 爲 8、`sizeof(long double)` 爲 16。這些大小也有可能因編譯器或硬體環境不同而有所差異。

2.3.2 二維陣列

二維陣列在邏輯概念上可用二維的表格描繪之，圖 2-13 即爲整數二維陣列 A 的邏輯圖示：

7　這兒有個理想化的假設：硬體存取記憶體中任何位置皆花費一樣的時間。

A	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

圖 2-13 二維陣列 A 的邏輯圖示

然而在實體電腦記憶體的配置中，一般皆以連續的空間存放這些資料，使編譯器在定址（addressing 即找到陣列任一元素的實際位址）時，可以簡單且有效率。假設陣列 *A* 的起始位置在 $\mathtt{A2C4}_{16}$，而且一個整數佔 4 位元組，則圖 2-14 爲圖 2-13 之二維陣列的可能記憶體配置情形；當二維陣列空間轉爲一維記憶體空間時，可採以「列爲優先」(row major) 或採以「行爲優先」(column major) 的配置方式。

A2C4	A[0][0]	1
A2C8	A[0][1]	2
A2CC	A[0][2]	3
A2D0	A[0][3]	4
A2D4	A[1][0]	5
A2D8	A[1][1]	6
A2DC	A[1][2]	7
A2E0	A[1][3]	8
A2E4	A[2][0]	9
A2E8	A[2][1]	10
A2EC	A[2][2]	11
A2F0	A[2][3]	12

(a) 以列為優先

A2C4	A[0][0]	1
A2C8	A[1][0]	5
A2CC	A[2][0]	9
A2D0	A[0][1]	2
A2D4	A[1][1]	6
A2D8	A[2][1]	10
A2DC	A[0][2]	3
A2E0	A[1][2]	7
A2E4	A[2][2]	11
A2E8	A[0][3]	4
A2EC	A[1][3]	8
A2F0	A[2][3]	12

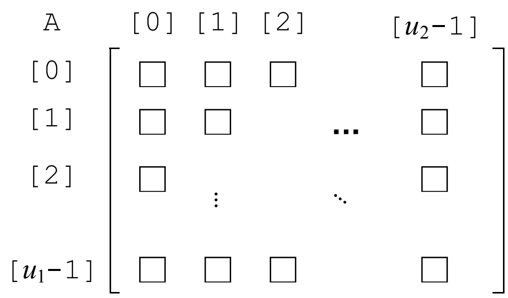
(b) 以行為優先

圖 2-14 二維陣列的記憶體配置

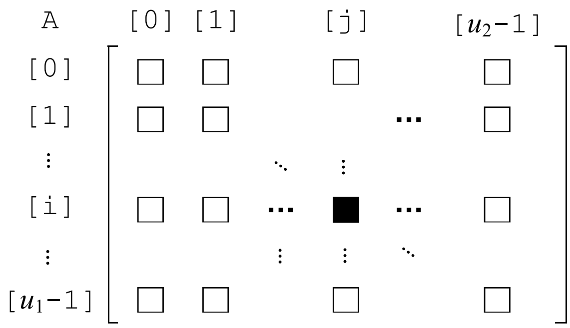
其中以列爲優先的次序，指的是先依列由小到大的順序，再依行由小到大的順序存放資料，即第 0 列所有資料依行序存妥後，再將第 1 列所有資料依行序存妥…，依此類推。

而以行爲優先的次序，則先依行由小到大的順序，再依列由小到大的順序存放陣列的資料。各位可從圖 2-11 中看出兩者之間的差異。本書記憶體位址的探討皆以「列爲優先」爲原則[8]。

我們用圖 2-15 和圖 2-16 來說明如何在二維陣列中，定出任一元素的位址。圖 2-15 (a) 顯示了 $u_1 \times u_2$ 的二維陣列 A；圖 2-15 (b) 顯示了 $A[i][j]$ 在 A 中的對應位置；圖 2-15 (c) 推導出在 $A[i][j]$ 之前的元素個數，亦即在 $A[i][j]$ 之前共計有 $i \times u_2+j$ 個 A 中的元素。

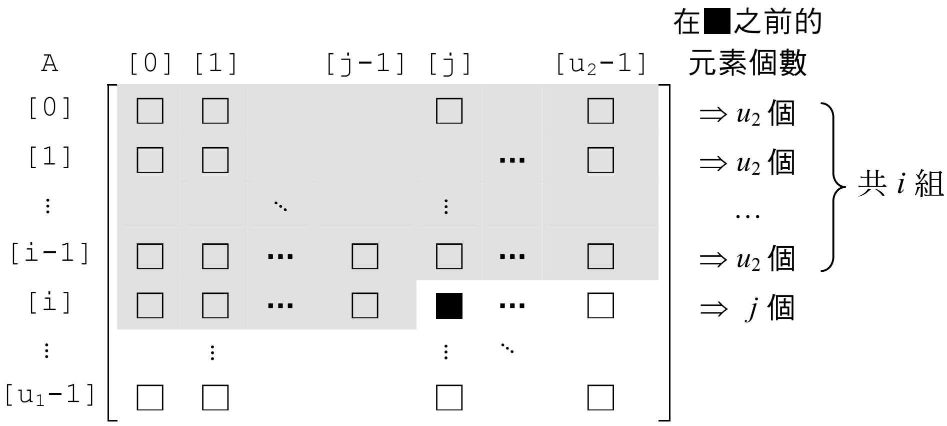


(a) $u_1 \times u_2$ 的二維陣列 A



(b) A[i][j]在 A 中的相對位置

8　現今大多的程式語言皆依以列為優先的原則設計（如 C、C++、Python、Javascript、⋯），以行為優先的有 Fortran。



(c) 在 A[i][j]中■之前的元素個數

圖 2-15　$u_1 \times u_2$ 的二維陣列 A 與 A[i][j]的位置關係

若二維陣列 A 的起始位址爲 α，則 A[i][j] 的位址爲 $\alpha + (i \times u_2 + j) \times l$。請見圖 2-16：

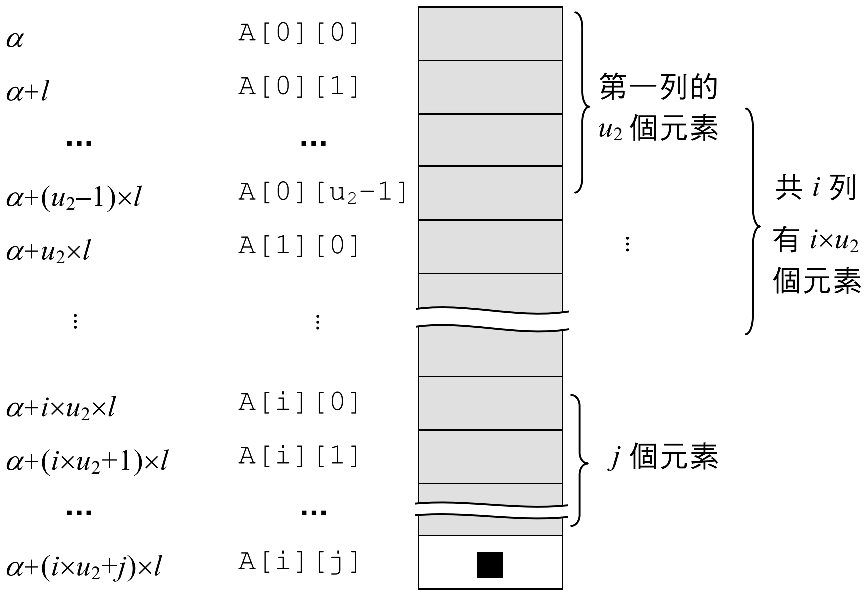


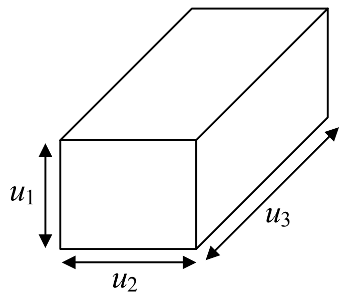
圖 2-16　二維陣列 A 的連續空間配置

（A 起始位址為 α，A[i][j]的位址為 $\alpha + (i \times u_2 + j) \times l$）

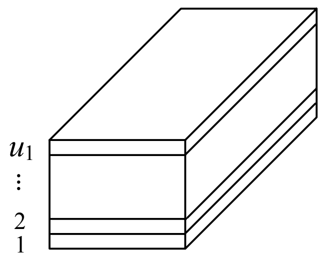
由圖 2-15 (c) 可知：$A[i][j]$ 之前共計有 i 列，而每列有 u_2 個元素，此 i 列共有 $i \times u_2$ 個元素，另外在第 i 列，於第 j 行前有 j 個元素，所以 $A[i][j]$ 之前共有 $i \times u_2+j$ 個元素，逐一自 α 起每 l 個位元組放一個元素，則可得如圖 2-13 的連續空間配置圖。由圖 2-16 可知當 A 陣列的起始位址為 α 時，$A[i][j]$ 的位址為 $\alpha+(i \times u_2+j) \times l$。

2.3.3 三維陣列與高維陣列

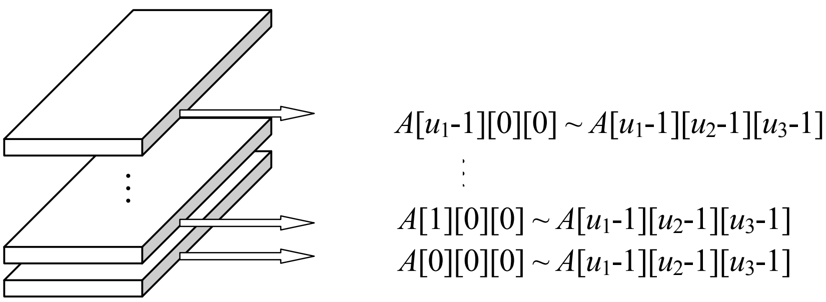
先前介紹二維陣列位址的計算，可以擴展至三維陣列上。圖 2-17 (a) 顯示了一個 $u_1 \times u_2 \times u_3$ 的三維陣列 A；圖 2-17 (b) 和 (c) 則將 A 視為 u_1 個 $u_2 \times u_3$ 二維陣列的組合；圖 2-17 (d) 則將 $A[i][j][k]$ 在 A 中的位置標示出來。



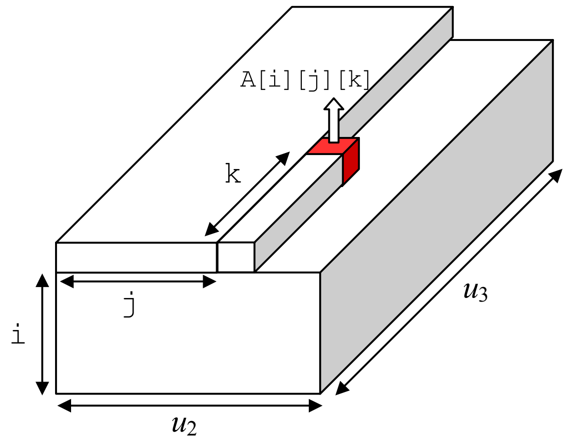
(a) $u_1 \times u_2 \times u_3$ 的三維陣列 A



(b) u_1 個 $u_2 \times u_3$ 二維陣列的組合



(c) 各個 $u_2 \times u_3$ 二維陣列的起迄註標



(d) A[i][j][k]在 A 中的位置

圖 2-17 三維陣列中元素的定址關係

由圖 2-17 (d) 可知，在 $A[i][j][k]$ 之前共有 $i \times u_2 \times u_3 + j \times u_3 + k$ 個 A 中的元素。當陣列 A 的起始位址為 α、每個元素大小為 l 位元組時，$A[i][j][k]$ 的位址為 $\alpha + (i \times u_2 \times u_3 + j \times u_3 + k) \times l$。

綜合以上的討論，我們可以推導出 n 維陣列的位址。首先可知 $A[i_1][i_2]\ldots[i_n]$ 之前會有如下之元素個數：

$$
\begin{aligned}
& i_1 \times u_2 \times u_3 \times \ldots \times u_n \\
+ & i_2 \times u_3 \times u_4 \times \ldots \times u_n \\
+ & i_3 \times u_4 \times u_5 \times \ldots \times u_n \\
& \ldots \\
+ & i_{n-1} \times u_n \\
+ & i_n \\
= & \sum_{j=1}^{n-1} i_j \prod_{k=j+1}^{n} u_k + i_n
\end{aligned}
$$

假設 A 為 n 維陣列，維度為 $u_1 \times u_2 \times \ldots \times u_n$，每個元素大小佔 l 位元組，且

A 的起始位址為 α，則 $A[i_1][i_2]\ldots[i_n]$ 的位址為：

$$\alpha + (\sum_{j=1}^{n-1} i_j \prod_{k=j+1}^{n} u_k + i_n) \times l$$

令

$$w_j = \begin{cases} \prod_{k=j+1}^{n} u_k & \text{if } 1 \le j < n \\ 1 & \text{if } j = n \end{cases}$$

則 $A[i_1][i_2]\ldots[i_n]$ 的位址可整理為：

$$\alpha + l \times \sum_{j=1}^{n} i_j w_j$$

因此在電腦的內部運作中，編譯器可先行找出陣列宣告的上下限值：$u_1, u_2, \ldots, un$，然後利用 $n-2$ 個乘法，求出上式的 $w_1, w_2, \ldots, w_{n-2}$（$w_n = u_n, w_n = 1$，不必求之），先行存下。而計算 $A[i_1][i_2]\ldots[i_n]$ 的位址時，即可利用上式，以 n 個乘法和 n 個加法求得其位址值。圖 2-18 將計算 $A[i_1][i_2]\ldots[i_n]$ 位址時所需要乘法、加法個數的計算標示清楚，方便各位觀察檢驗。

$$\begin{aligned} & i_1 \times u_2 \times u_3 \times \ldots \times u_n \\ + & i_2 \times u_3 \times u_4 \times \ldots \times u_n \\ + & i_3 \times u_4 \times u_5 \times \ldots \times u_n \\ & \ldots \\ + & i_{n-1} \times u_n \\ + & i_n \\ = & \sum_{j=1}^{n-1} i_j \prod_{k=j+1}^{n} u_k + i_n \end{aligned}$$

$\Rightarrow$

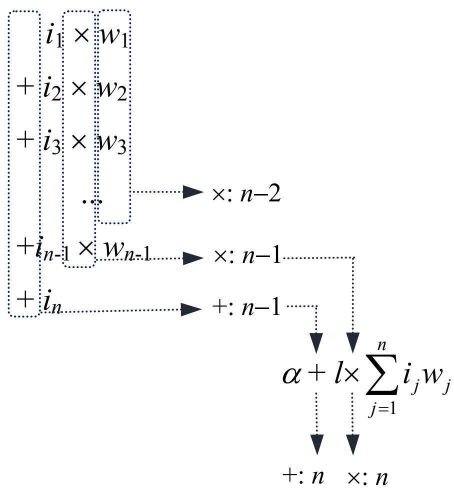


圖 2-18　計算 $A[i_1][i_2]\ldots[i_n]$ 位址所需要用的乘法與加法

本章習題

1. 有多少元素可以存在以下的陣列中（註標範圍 `[x:y]` 中 `x` 表示下界，`y` 表示上界）：

   (a) `A[0:n]`；

   (b) `B[-23:29]`;

   (c) `C[-1:n][1:m]`；

   (d) `D[-n:0][1:3]`；

2. 導出陣列 $A[l_1 \ldots u_1][l_2 \ldots u_2]\ldots[l_n \ldots u_n]$ 的元素 $A[i_1][i_2]\ldots[i_n]$ 的位址公式，$l_j \le i_j \le u_j$，$1 \le j \le n$。假設每個元素佔 l 個字元組，而 α 是 $A[l_1][l_2]\ldots[l_n]$ 的位址，且：

   (a) 陣列 A 是以列爲優先存放；

   (b) 陣列 A 是以行爲優先存放。

3. 寫一個程序來估算每個不同的字元在字串中出現的次數，用適當的資料來測試你的函式。

4. 寫一個程序，輸入爲一個字串 *text* 和一個常數 k，列出在 *text* 中出現次數前 k 高的字元及其出現次數。

5. 寫一個函式，它會接受一個字串 *text* 和兩個整數 *start* 和 *length*，這個函式會從字串 *text* 的第 *start* 個字元起刪除 *length* 個字元，而傳回產生新的字串。

6. 使用最少的記憶空間，寫一個程序：給一個陣列 $A[n]$，請產生 $Z[n]$，使得 $Z[0] = A[n-1], Z[1] = A[n-2], \ldots, Z[n-2] = A[1], Z[n-1] = A[0]$。

7　假設有 n 個串列，$n > 1$，它們用一維陣列 *space*[*m*] 來表示。令 *front*[*i*] 是指到第 *i* 個串列第一個元素的前一個位置，*rear*[*i*] 是指到第 *i* 個串列最後一個元素的位置，假設 $rear[i] \le front[i+1]$，$0 \le i < n$，且 $front[n] = m-1$，這些串列所能做的運算就是插入和刪除。

(a)　找出 *front*[*i*] 和 *rear*[*i*] 的適當的起始和終止條件。

(b)　寫一個程序 `Insert(int i, int j, int item)`，使其能在串列 *i* 的第 (*j*-1) 個元素後面插入 *item*，這個程序在 *space* 已經有了 *m* 個元素的情況下，不允許插入的動作。

8.　承上題，寫一個程序 `int Delete(int i, int j)` 來刪除第 *i* 個串列的第 *j* 個元素，並傳回它的值。

9.　當一個陣列的所有元素都排在其主對角線及其下方或上方，這個矩陣就叫做「三角矩陣」(triangular matrix)。圖 2-19 表示出「下三角」(lower triangular) 和「上三角」(upper triangular) 矩陣，其中非 0 元素標示為 X。

$$
\begin{bmatrix}
X & 0 & 0 & 0 & 0 & & \cdots & & 0 & 0 \\
X & X & 0 & 0 & 0 & & & & 0 & 0 \\
X & X & X & 0 & & & & & 0 & 0 \\
 & & & X & 0 & & & & & \\
\vdots & & & & \ddots & & \ddots & & \vdots & \\
 & & & & & X & 0 & & & \\
X & & & & & & X & 0 & 0 & 0 \\
X & & & & & & & X & 0 & 0 \\
X & & & & & & \cdots & & X & 0 \\
X & X & X & X & X & & & & X & X
\end{bmatrix}
\qquad
\begin{bmatrix}
X & X & X & X & X & & \cdots & & X & X \\
0 & X & & & & & & & & X \\
0 & 0 & X & & & & & & & X \\
 & & 0 & X & & & & & & \\
\vdots & & & \ddots & & \ddots & & & & \vdots \\
 & & & & & X & & & & \\
0 & & & & & 0 & X & & & X \\
0 & & & & & & 0 & X & & X \\
0 & & & & & & & 0 & X & X \\
0 & 0 & 0 & 0 & & \cdots & & 0 & 0 & X
\end{bmatrix}
$$

下三角矩陣　　　　上三角矩陣

圖 2-19　下三角和上三角矩陣

將 $n \times n$ 的下（上）三角矩陣存放在記憶體中，0 的值不存，請為陣列元素 $[i][j]$ 求出其位址公式，$0 \le i, j \le n-1$（每個元素佔 l 個字元組，而 α 為陣列第一個非 0 元素（位於[0][0]）在記憶體中的位址）。

10. 假設 A 和 B 是兩個 $n \times n$ 的下三角矩陣，則它們的元素個數總和為 $n(n+1)$。設計出一種方法把這兩個矩陣同時儲存在一個陣列 $C[n][n+1]$ 裡（提示：把 A 以 C 的下三角方式儲存，B 的轉置矩陣則存在 C 的上三角矩陣）。再設計一個演算法，從陣列 C 中找出 $A[i][j]$ 和 $B[i][j]$ 的值，$0 \le i, j \le n-1$。

11. 一個「三對角線矩陣」(tri-diagonal matrix) 為一個方陣，其中在主對角線及其相鄰的兩個對角線以外的元素均為 0 (如圖 2-20)。令 A 為一 $n \times n$ 的三對角線矩陣，將 A 中三個對角線所形成帶狀區域上的元素，以列為優先，儲存在一個一維陣列 M 中（$A[0][0]$ 存放在 $M[0]$ 中）。設計一個演算法，輸入 $A[i][j]$ 時，可求得 $k-A[i][j]$ 存在 $M[k]$ 中，$0 \le i, j \le n-1$。

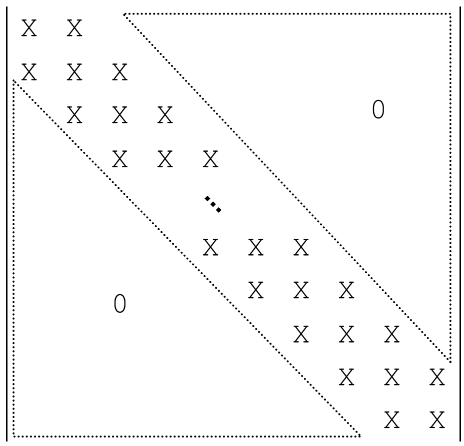


圖 2-20　三對角線矩陣

12. 一個「方形帶狀矩陣」(square band matrix) $D_{n,a}$ 是一個 $n \times n$ 的矩陣，其中所有可能的非零元素，均集中在以主對角線為中心的帶狀區域上（此帶狀區域外的元素皆為 0）；此帶狀區域包括了主對角線與其上和其下各 $a-1$ 條對角線（如圖 2-21 (a) 和 (b)，圖 2-21 (b) 是一 $D_{5,2}$ 的例子）。請回答：

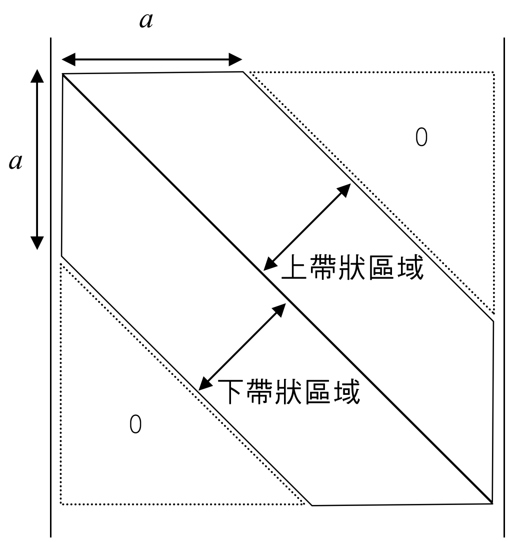
(a) 在此帶狀 $D_{n,a}$ 中有多少個元素？

(b) 對於帶狀 $D_{n,a}$ 中的元素 d_{ij} 而言，i 與 j 之間的關係為何？

(c) 假設帶狀 $D_{n,a}$ 以對角線為主，並從最低的對角線開始，循序儲存在陣列 A 中。例如，圖 2-21 (b) 中的帶狀矩陣 $D_{5,2}$ 的表示法如下：

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$	$A[12]$
10	4	8	7	9	6	0	5	2	11	3	1	12
d_{10}	d_{21}	d_{32}	d_{43}	d_{00}	d_{11}	d_{22}	d_{33}	d_{44}	d_{01}	d_{12}	d_{23}	d_{32}

找出在 $D_{n,a}$ 的下帶狀區域中的任一元素 d_{ij} 的定址公式，即輸入 i 和 j，求出 k，$d_{ij} = A[k]$。



(a) $n \times n$ 方形帶狀矩陣 $D_{n,a}$

$$\begin{bmatrix} 9 & 11 & 0 & 0 & 0 \\ 10 & 6 & 3 & 0 & 0 \\ 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 8 & 5 & 12 \\ 0 & 0 & 0 & 7 & 2 \end{bmatrix}$$

(b) $D_{5,2}$

圖 2-21　方形帶狀矩陣

13. 一個「一般帶狀矩陣」(generalized band matrix) $D_{n,a,b}$ 是一個 $n \times n$ 的矩陣，其中所有可能的非零元素，皆集中在由主對角線以下的 $a-1$ 個對角線，主對角線和主對角線以上的 $b-1$ 個對角線，所形成的帶狀區域上(如圖 2-22)。
    (a) 在此帶狀 $D_{n,a,b}$ 中有多少個元素？
    (b) 對於帶狀 $D_{n,a,b}$ 中的元素 d_{ij} 而言，i 與 j 之間的關係爲何？
    (c) 找出此帶狀 $D_{n,a,b}$ 在一維陣列 B 中的循序表示法。對此表示法，設計一個函數 $address(n, a, b, i, j, B)$，根據傳入的參數 n, a, b, i, j, B，來決定在陣列 B 中，矩陣 $D_{n,a,b}$ 中的 d_{ij} 值。

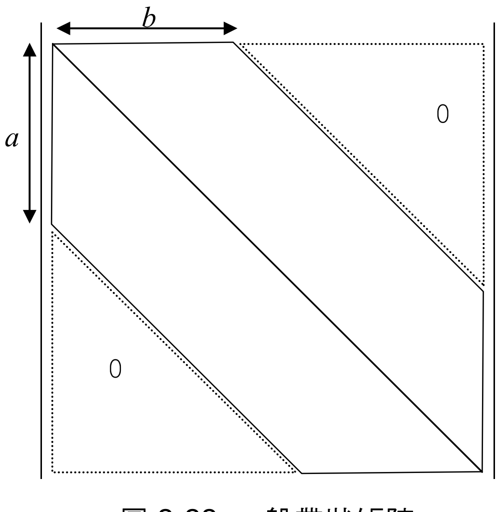


圖 2-22　一般帶狀矩陣

14. 請撰寫程式實作經驗法則 2-7 解決騎士巡迴的問題。