
# Ch09
# 演算法

李官陵 彭勝龍 羅壽之

- 演算法就是解決問題的方法。演算法和程式碼主要的區別在於演算法是給人看的，讓人了解解決這個問題的方法步驟，並且可以被證明它的對錯；而程式碼是給編譯器處理的，它必須符合該程式語言的語法，因此撰寫比較嚴謹。

# 演算法的表示

- 演算法是給人看的，依照人的工作性質，演算法也就有不同的風貌。下面的問題來說明演算法的幾種常見表示法。
- 問題 1：在三個數字中找出最大者

  輸入：三個數字，$a,b,c$

  輸出：最大者的值
- 首先我們來看第一種表示法，稱為「步驟描述法」，解決問題 1 的演算法如圖 9.1：

演算法 1：三選大

輸入：$a$, $b$, $c$ 三個數值

輸出：最大值

方法：

1. 比較 $a$ 和 $b$，設最大值為較大者
2. 比較最大值和 $c$，若 $c$ 較大，則把 $c$ 設為最大值
3. 輸出最大值

圖 9.1　步驟描述法

# 演算法的表示 (續)

- 在這個演算法裡，「三選大」稱為這個演算法的名稱。除了輸入、輸出的描述外，方法說明了這個演算法的步驟，為每一個步驟加上編號是為了方便引用。
- 流程圖是演算法的另一個表示法，通常系統分析師了解各系統模組後，會依照需求設計各模組，其細節處就是演算法。
- 流程圖在系統分析裡是一個很傳統的工具，因此已經有一些標準出來了，例如橢圓形表示開始或結束。長方形表示處理程序，菱形表示決策，而平行四邊形則表示輸入或輸出。

# 演算法的表示 (續)

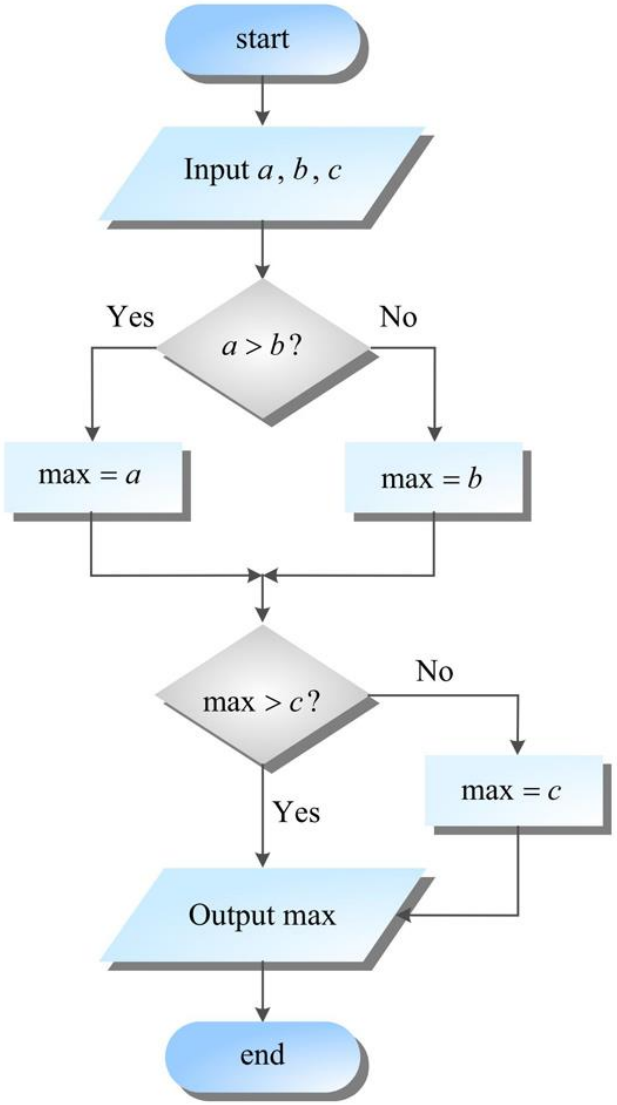


圖 9.2 演算法 1 的流程圖

# 演算法的表示 (續)

- 虛擬碼表示法。

```
Algorithm 1: Find_Max
Input: a, b, c
Output: the maximum
Method:
1. if a > b then
2.     max = a
3. else
4.     max = b
5. endif
6. if c > max then
7.     max = c
8. endif
9. return max
```

圖 9.3　演算法 1 的虛擬碼表示法

- 使用這種方式描述演算法要特別注意縮排結構，像第 2, 4, 7 列的內容，因為是屬於 if...then...else 的結構，我們就可以很清楚知道當條件成立後，演算法會執行什麼內容。

# 演算法的表示 (續)

- 問題 2：在 $n$ 個數字中找出最大者

  輸入：一個含 $n$ 個數字的陣列 $a$ ( 假設陣列是從 1 開始 )

  輸出：最大者的值

- 演算法如圖 9.4。

```
Algorithm 2: Find_Max2
Input: an array a of size n
Output: the maximum
Method:
1. max = a[1]
2. for i = 2 to n
3.     if a[i] > max then
4.         max = a[i]
5.     endif
6. next i
7. return max
```

圖 9.4　用虛擬碼來表示一個演算法

# 演算法的分析

- 同一個問題解決的方法有很多，也就是說有很多演算法可以解決同一個問題，因此就需要一個標準來比較不同的演算法。
- 理論上的分析比較，這就是一般人稱的複雜度分析。因為演算法裡免不了一些條件判斷，真的執行時會依照當時狀況決定執行細目，例如上一節的Algorithm 1 裡的第 7 列會因為第 6 列的判斷成立了才執行，在做分析時，我們採用最糟情況去計算，也就是悲觀的把演算法的每一步都累計來算出整體複雜度。

# 演算法的分析 (續)

- Algorithm 1，因為只有輸入三個數字，整個演算法只做了二個比較 ( 第 1 和第 6 列 )，和最多二個指派指令 ( 分別是第 2 或第 4 列二擇一，另一個是第 7 列 ) 就結束了，它的執行時間和輸入的值也沒有關係，一般稱為常數時間，非常快的意思。

| 演算法步驟 | 執行時間 |
|---|---|
| 1. if $a$ > $b$ then | $c_1$ |
| 2. $max$ = $a$ | $c_2$ |
| 3. else | |
| 4. $max$ = $b$ | $c_2$ |
| 5. endif | |
| 6. if $c$ > $max$ then | $c_1$ |
| 7. $max$ = $c$ | $c_2$ |
| 8. endif | |
| 9. return $max$ | $c_3$ |

圖 9.5 Algorithm 1 的時間估計

# 演算法的分析 (續)

- $c_1$ 表示比較時間，$c_2$ 是指派時間，而 $c_3$ 則是回傳時間。
- Algorithm 1 整個執行時間為 $2 * c_1 + 2 * c_2 + c_3$，算起來稱為常數時間。
- Algorithm 2

| 演算法步驟 | 執行時間 |
|---|---|
| 1. *max* = *a*[1] | $c_2$ |
| 2. for *i* = 2 to *n* | $c_4$ |
| 3. if *a*[*i*] > *max* then | $c_1$ |
| 4. *max* = *a*[*i*] | $c_2$ |
| 5. endif | |
| 6. next *i* | |
| 7. return *max* | $c_3$ |

圖 9.6 Algorithm 2 的時間估計

# 演算法的分析 (續)

- 考慮最糟的情況下，第 3 列的判斷一直都成立，所以整個執行時間為 $c_2 + c_4 * (n-1) + c_1 * (n-1) + c_2 * (n-1) + c_3 = (c_1 + c_2 + c_4) * n + (c_3 - c_4) = c_5 * n + c_6$。在數學上，這個式子稱為線性函數，它跟 $n$ 成正比，當 $n$ 越大，它的值就越大。
- 我們通常把這個執行時間寫成 $f(n) = c_5 * n + c_6$。$n$ 稱為這個演算法的輸入量 (input size)。大部分的演算法執行時間都跟它的輸入量成一個函數關係。

# 演算法的分析 (續)

- 三種符號來表示時間函數。

**定義 1：Big O 表示法**

- 若存在一個常數 c 和一個正整數 $n_0$，使得當 $n \geq n_0$ 時，$f(n) \leq c\, g(n)$，則函數 $f(n)$ 屬於 $O(g(n))$。

- 常見的函數有：

  $O(1)$：常數、$O(n)$：線性、$O(n \log n)$、$O(n^2)$：平方、$O(n^3)$：立方、$O(n^k)$：多項式、$O(2^n)$：指數

# 演算法的分析 (續)

**定義 2：Big Ω 表示法**

- 若存在一個常數 c 和一個正整數 $n_0$，使得當 $n \geq n_0$ 時，$f(n) \geq c\, g(n)$，則函數 $f(n)$ 屬於 $\Omega(g(n))$。

**定義 3：Big Θ 表示法**

- 若存在二個常數 c 和 c′，及一個正整數 $n_0$，使得當 $n \geq n_0$ 時，$c\, g(n) \leq f \leq c'\, g(n)$，則函數 $f(n)$ 屬於 $\Theta(g(n))$。

# 演算法的分析 (續)

- 如果分析使用到 Big Θ，就表示這個分析很緊 (tight)，相當可靠。為什麼說分析得越緊越好呢？因為它同時讓你兼顧了上限與下限，所以必須更精準的分析它。
- 使用 Big *O*、Ω、Θ 表示法就是為了讓分析不要那麼複雜，只要注意大項 ( 就是多項式裡最高次項 ) 就行了。注意大項還有個好處，可以讓我們專注於重要的步驟，例如要分析排序的演算法，我們可以只專注於計算比較的次數，不需要管其他的指令，因而大大的降低了分析的工夫。

# 演算法的分析 (續)

**隨堂練習**

- 請將下面給的函數，用 Big $O$ 表示法表示出來。
- (1) $3n^2 - 5n + 6$　(2) $n + 2n \log n + 100$　(3) $2n^3 + 7n - 9$
- 解答：
- (1) $O(n^2)$　(2) $O(n \log n)$　(3) $O(n^3)$

# 暴力法 (brute force)

- 暴力法又稱窮舉法，顧名思義是找出所有的可能，然後選擇最好的那一個。
- 問題 3： 將 $n$ 個數字由小到大排列出來

  輸入：一個含 $n$ 個數字的陣列 $a[1..n]$

  輸出：一個由小到大排序好的陣列 $a$
- 如果想要把 $n$ 個數字排好，最直接的想法就是首先找出最小那一個，然後把它放在第一個位置；接著在剩餘的 $n-1$ 個值中找出最小者，它當然就是全部的第二小，因此把它放在第二個位置，依此類推。最後剩下那一個一定是最大者，就把它留在最後的位置，而完成了排序的工作。這個演算法每次都在剩餘的值中選擇一個最小者，放在它應該放的位置，所以又叫做「選擇排序法」。

# 暴力法 (brute force) (續)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 1 | 8 | 6 | 2 | 7 |
| **1** | 3 | 4 | 5 | 8 | 6 | 2 | 7 |
| **1** | **2** | 4 | 5 | 8 | 6 | 3 | 7 |
| **1** | **2** | **3** | 5 | 8 | 6 | 4 | 7 |
| **1** | **2** | **3** | **4** | 8 | 6 | 5 | 7 |
| **1** | **2** | **3** | **4** | **5** | 6 | 8 | 7 |
| **1** | **2** | **3** | **4** | **5** | **6** | 8 | 7 |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | 8 |

圖 9.7　選擇排序法

# 暴力法 (brute force) (續)

```
Algorithm 3: Selection_Sort
Input: an array a of size n
Output: the sorted array a
Method:
1. for i = 1 to n-1
2.     min = i
3.     for j = i+1 to n
4.         if a[min] > a[j] then
5.             min = j
6.         endif
7.     next j
8.     temp = a[i]
9.     a[i] = a[min]
10.    a[min] = temp
11. next i
12. return a
```

圖 9.8　選擇排序法的演算法

# 暴力法 (brute force) (續)

- Algorithm 3 的時間複雜度很容易可以從它的二個 for 迴圈來算出，使用 Big $O$ 的表示法，我們可以專注於列 1 到列 7 的運算量，特別是只要計算比較的部分，在最糟糕的情況下，每一回合剩下的元素都要比一下，所以是 $(n-1)+(n-2)+\cdots+2+1=O(n^2)$。

# 貪婪法 (greedy method)

- 顧名思義，因為貪心的心理，每次都挑最好的，而且所挑的都會是解的一部分。換零錢問題裡，假設我們的零錢有 50 圓、10 圓、5 圓和 1 圓等四種硬幣，現在要找 79 元，請問最少要用幾個硬幣？
- 貪婪法的演算法，永遠從目前最好的選擇開始做決定。利用貪婪法解零錢問題，並不一定永遠可以得到最佳解，它要視題目所給的零錢種類而定 ( 在這個例子是最佳解 )。

# 貪婪法 (greedy method) (續)

- 考慮排序問題，跟貪婪法最接近的排序演算法就是「氣泡排序法」，資料位置 $a[1]$ 到 $a[n]$ 像從海底到海面，每次都從 $a[1]$ 開始看二個相鄰的數值，然後把大的換到右邊，因此當我們碰到最大值時，它就會像氣泡一樣一路冒出海面，如此完成第一回合。以此類推，第二回會把第二大值送到倒數第二個位置，最後留下的就是最小值。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 1 | 8 | 6 | 2 | 7 |
| 3 | 4 | 1 | 5 | 6 | 2 | 7 | **8** |
| 3 | 1 | 4 | 5 | 2 | 6 | **7** | **8** |
| 1 | 3 | 4 | 2 | 5 | **6** | **7** | **8** |
| 1 | 3 | 2 | 4 | **5** | **6** | **7** | **8** |
| 1 | 2 | 3 | **4** | **5** | **6** | **7** | **8** |
| 1 | 2 | **3** | **4** | **5** | **6** | **7** | **8** |
| 1 | **2** | **3** | **4** | **5** | **6** | **7** | **8** |

圖 9.9　氣泡排序法

# 貪婪法 (greedy method) (續)

```
Algorithm 4: Bubble_Sort
Input: an array a of size n
Output: the sorted array a
Method:
1. for i = n downto 1
2.     for j = 1 to i
3.         if a[j] > a[j+1] then
4.             temp = a[j]
5.             a[j] = a[j+1]
6.             a[j+1] = temp
7.         endif
8.     next j
9. next i
10. return a
```

圖 9.10　氣泡排序法的演算法

# 貪婪法 (greedy method) (續)

- Algorithm 4 的時間複雜度也可以很容易從它的二個 for 迴圈來算出，在最糟糕的情況下，它和 Algorithm 3 是一樣的，都是 $O(n^2)$。
- 依照貪婪法設計的演算法，在決策上都會有一些特殊的喜好，例如喜歡最大值、偏好最小值、盡可能塞滿等等傾向，於是乎你可能覺得選擇排序法應該是屬於貪婪法吧？為什麼我們把它歸於暴力法，重點就是在做決策的那當下，我們憑藉什麼資訊？在選擇排序法裡，我們每次去挑最小值，都必須看過所有的值，它使用了全域 (Global) 知識；而氣泡排序法在決定換不換時，只跟鄰居比過，它使用了區域 (Local) 知識，所以更有貪婪法的精神。

# 動態規劃 (dynamic programming)

- 它的精神在於將問題依輸入量分成很多小單位的問題。題目一樣，就是輸入量大小的差異，但是藉由小問題的解可以兜出大問題的解，逐次把問題輸入量變大，最後到原問題大小，所得到的解就是原問題的解。這裡只有小問題的解是真算，其他都是查表，就是查小問題的解來兜出另一個大問題的解，因此也有人說動態規劃法就是查表法。

**定義 4：費比納西數字**

- 令 $F_k$ 表示第 $k$ 個費比納西數，$F_0 = 0$，$F_1 = 1$，$F_n = F_{n-1} + F_{n-2}$，$n \geq 2$。

# 動態規劃 (dynamic programming) (續)

```
Algorithm 5: Fibonacci
Input: an integer n
Output: F_n
Method:
1. F[0] = 0
2. F[1] = 1
3. for i = 2 to n
4.     F[i] = F[i-1] + F[i-2]
5. next i
6. return F[n]
```

圖 9.11　動態規劃法求解的演算法

- 因為二個最小的值 $F_0$ 和 $F_1$ 已經得知，從 $F_2$ 的值就可以參考 $F_0$ 和 $F_1$ 而得知。同理 $F_3$ 可以查 $F_1$ 和 $F_2$ 而得知，以此類推最後可以得出 $F_n$。

# 動態規劃 (dynamic programming) (續)

- 具有動態規劃法精神的排序演算法——插入排序法。插入排序法的精神在維持前面為排序好的狀態，然後把下一個插入到前面已排好序列的適當位置上，插入完後排序好的區段又增長了一位，如此依次漸增序列，直到排好為止。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **5** | 3 | 4 | 1 | 8 | 6 | 2 | 7 |
| **3** | **5** | 4 | 1 | 8 | 6 | 2 | 7 |
| **3** | **4** | **5** | 1 | 8 | 6 | 2 | 7 |
| **1** | **3** | **4** | **5** | 8 | 6 | 2 | 7 |
| **1** | **3** | **4** | **5** | **8** | 6 | 2 | 7 |
| **1** | **3** | **4** | **5** | **6** | **8** | 2 | 7 |
| **1** | **2** | **3** | **4** | **5** | **6** | **8** | 7 |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |

圖 9.12　執行插入排序法

# 動態規劃 (dynamic programming) (續)

```
Algorithm 5: Insertion_Sort
Input: an array a of size n
Output: the sorted array a
Method:
1. for i = 2 to n
2.     temp = a[i]
3.     j = i
4.     while a[j-1] > a[j] do
5.             a[j] = a[j-1]
6.             j = j-1
7.     endwhile
8.     a[j] = temp
9. next i
10. return a
```

圖 9.13　插入排序法的演算法

# 動態規劃 (dynamic programming) (續)

- 為什麼插入排序法有動態規劃的精神呢？因為排序 $k$ 個資料的結果是從排序 $k-1$ 個資料的結果來的，因為排序不是找最佳值的問題，所以不是查查表就好，但它確實是從小問題開始，而且第 $k$ 回的答案是從第 $k-1$ 回來產生的，所以我們說它具有動態規劃的精神。

# 分而治之 (divide and conquer)

- 把問題切小，然後各個擊破，最後再把小問題的答案組合成大問題的答案即完成。
- 分而治之和動態規劃的差異在於小問題的使用頻率，通常在動態規劃裡，小問題會被引用二次以上，所以我們需要把它的結果保存起來，等到需要的時候，再去查表，而不用再算一次。在分而治之裡，小問題的解可能只用一次就沒有用了，所以沒有保存的價值。「合併排序法」就是一個有名的分而治之演算法。

# 分而治之 (divide and conquer) (續)

- 當我們把一個序列的資料分成二半，分別去排序後，很重要的一個關鍵就是把二個排序好的小序列合併成一個排序好的大序列，二個序列各設定一個指標指在最小值處，然後比較二個指標所指的值，誰比較小就輸出誰，被輸出較小值的序列，調整指標到下一個位置，繼續做同樣的事，直到某一個序列輸出完畢，再把未輸完的序列通通輸出去，如此完成合併的程序。
- 圖 9.14 是二個 4 個元素的陣列在做合併成序的過程，陰影表示指標，二個陣列內容分別是 {1, 3, 4, 5} 和 {2, 6, 7, 8}。

# 分而治之 (divide and conquer) (續)

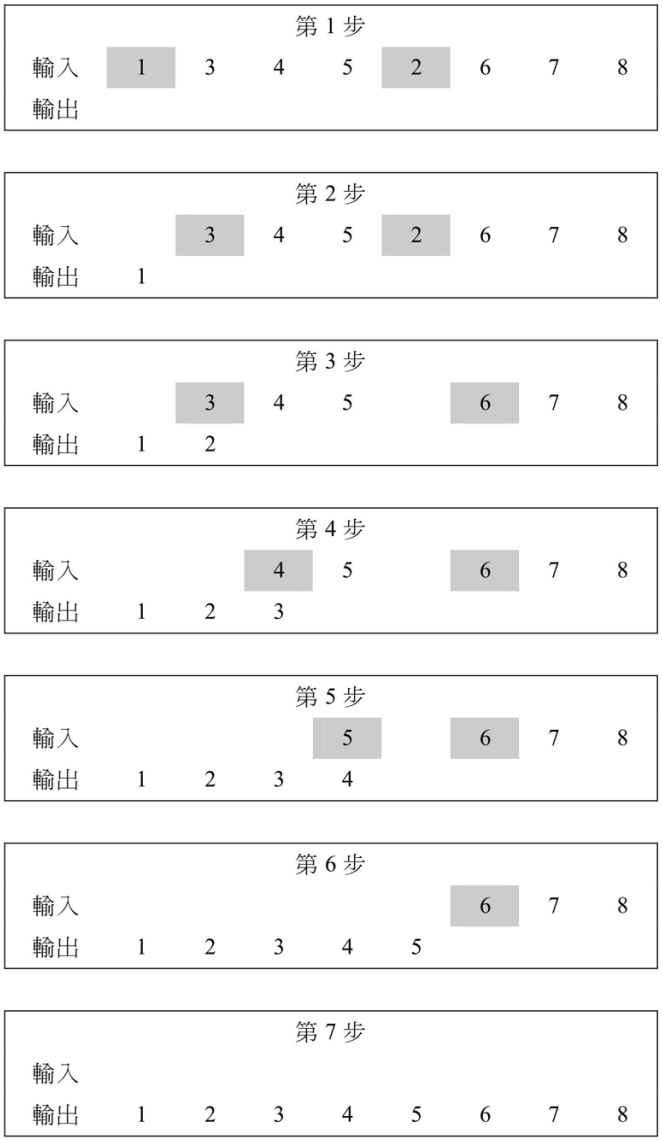


**圖 9.14** 二個 4 個元素的陣列 {1, 3, 4, 5} 和 {2, 6, 7, 8} 在做合併成序的過程，陰影表示指標。

# 分而治之 (divide and conquer) (續)

- 分而治之通常是先分割，然後遞迴解決被分割的小問題，最後再把小問題的解合併成原問題的解。遞迴的過程會一直下去直到問題夠小，可以直接解決為止。在這個排序問題裡，遞迴到最後，排序問題只剩一個資料要排序，顯而易見的，它本身即為答案。因此合併排序法可以省略遞迴的步驟，直接從合併開始，也就是一開始每一個元素代表一個排好的陣列，然後兩兩合併，也就是說從一個元素的陣列合併成二個元素的陣列，然後是四個元素的陣列。

# 分而治之 (divide and conquer) (續)

- 圖 9.15 我們看一個合併排序法的例子。

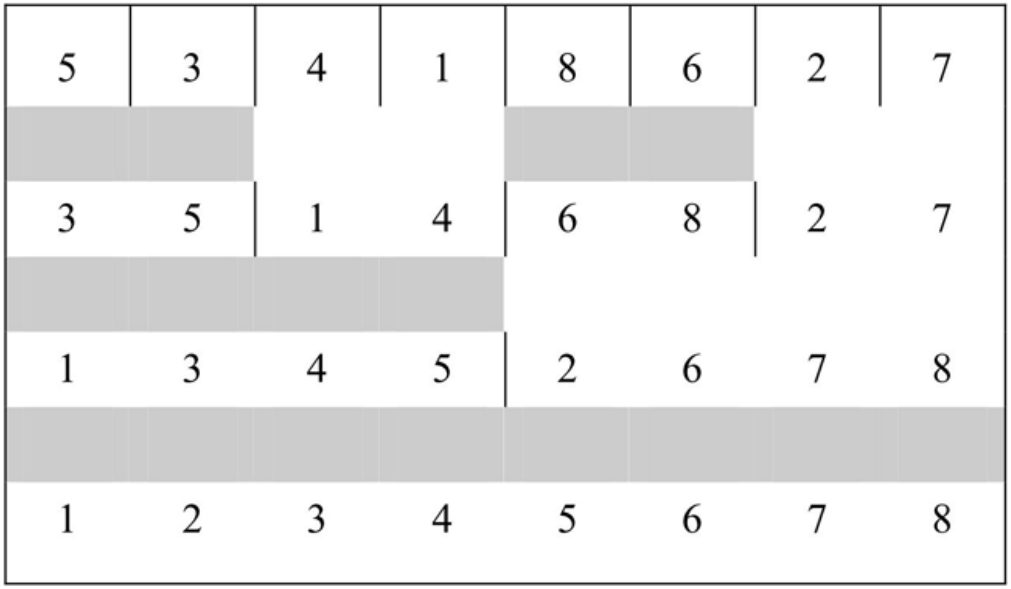


**圖 9.15** 合併排序法

# 分而治之 (divide and conquer) (續)

- 以全部的資料都被處理一次當作一回合來看，不難看出合併排序法總共需要 $\log n$ 回合，因此時間複雜度是 $O(n \log n)$。
- 另一個分而治之的排序演算法稱為「快速排序法」，此演算法稱為快速的原因在於沒有合併的動作，它的精隨在於分割 (partition)。先找一個樞紐值 (pivot)，通常用陣列的第一個值，然後使用二個指標，一個從左邊開始尋找比它還大者，另一個從右邊尋找比它還小者，若這二個指標還未相交，就把這二個值做交換；重覆這個動作直到二個指標相交為止，最後再把樞紐值 ( 在第一個位置 ) 跟右邊指標所指得值 ( 最後一個比樞紐值還小的值 ) 作交換，即完成一個分割的動作。
- 圖9.16 是一個分割程序的例子，其中↑是左指標，而□是右指標。

# 分而治之 (divide and conquer) (續)

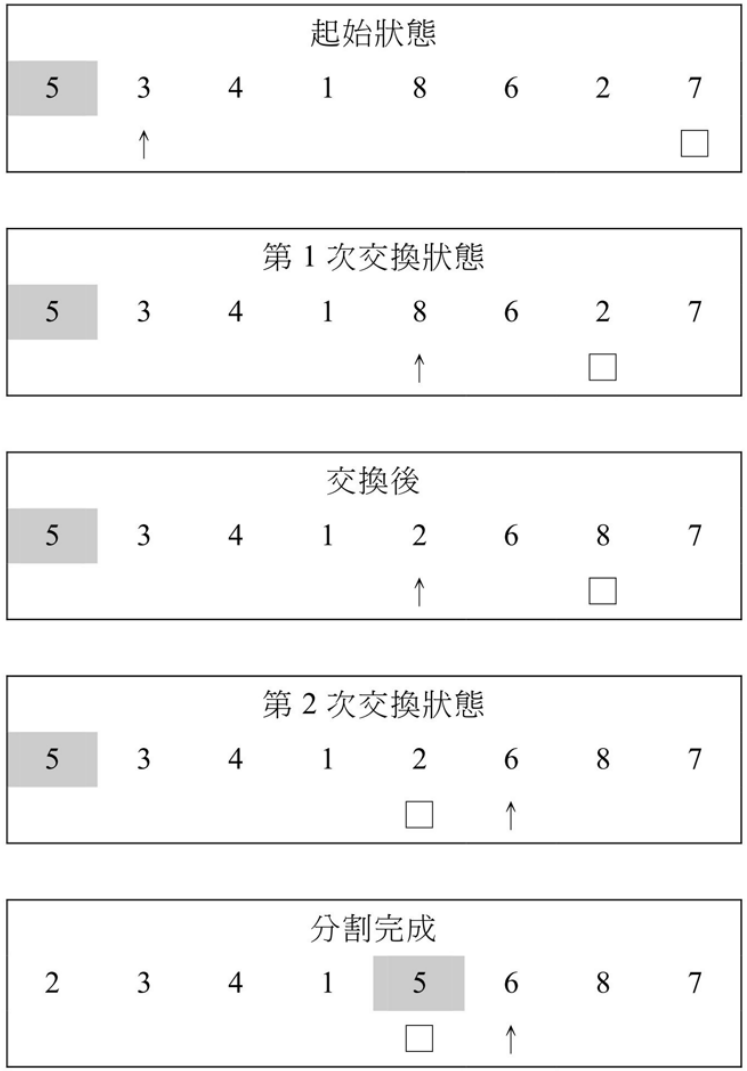


**圖 9.16** 分割程序

# 分而治之 (divide and conquer) (續)

- 當一個分割程序完成後，一個陣列會依照樞紐值分成二個小陣列，左邊陣列的值都比樞紐值小，而右邊陣列的值都比樞紐值大。快速排序法會遞迴的去排序這二個小陣列，直到陣列元素夠少，可以直接排序為止。
- 快速排序法在理論上，它的時間複雜度取決於樞紐值的選取，若每次樞紐值的選取都可以將陣列分成大小相當的二個小陣列，那麼整體時間為 $O(n \log n)$。可是如果樞紐值選得不好 ( 不是最大就是最小 )，整體時間就是 $O(n^2)$。
- 在大部分的應用上，快速排序法幾乎是最快的排序演算法。

# 資料結構與演算法

- 資料結構與演算法是二個彼此相關很深的主題。
- 用堆積排序演算法來說明此關係。
- 使用最大堆積 (Max heap) 來排序。

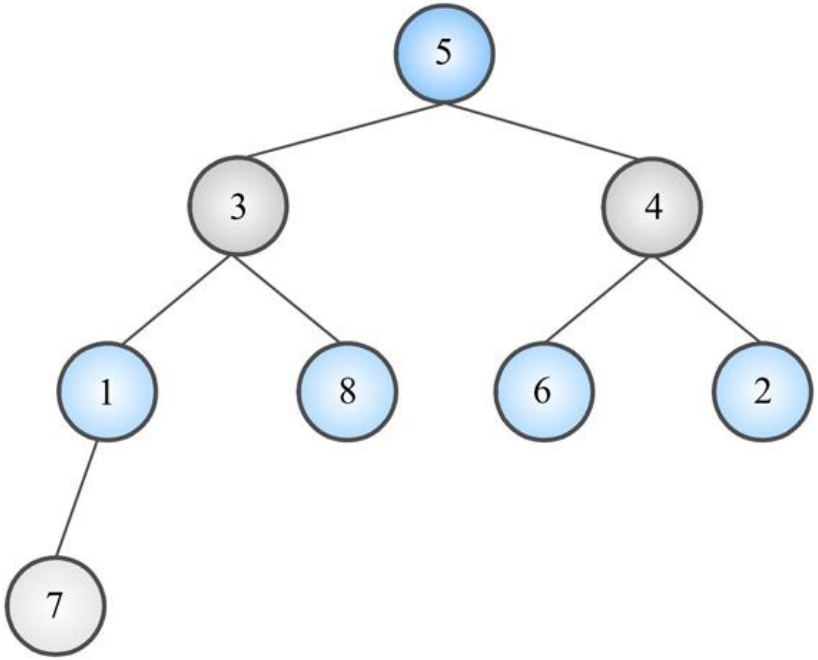


圖 9.17　陣列 (5, 3, 4, 1, 8, 6, 2, 7) 完全二元樹表示法

# 資料結構與演算法 (續)

- 最大堆積這個二元樹裡必須滿足：「任一個內部節點的值都要比它的兒子們的值要大」，因此經過最大堆積的建立之後，上面的完全二元樹就變成圖 9.18 的最大堆積了。

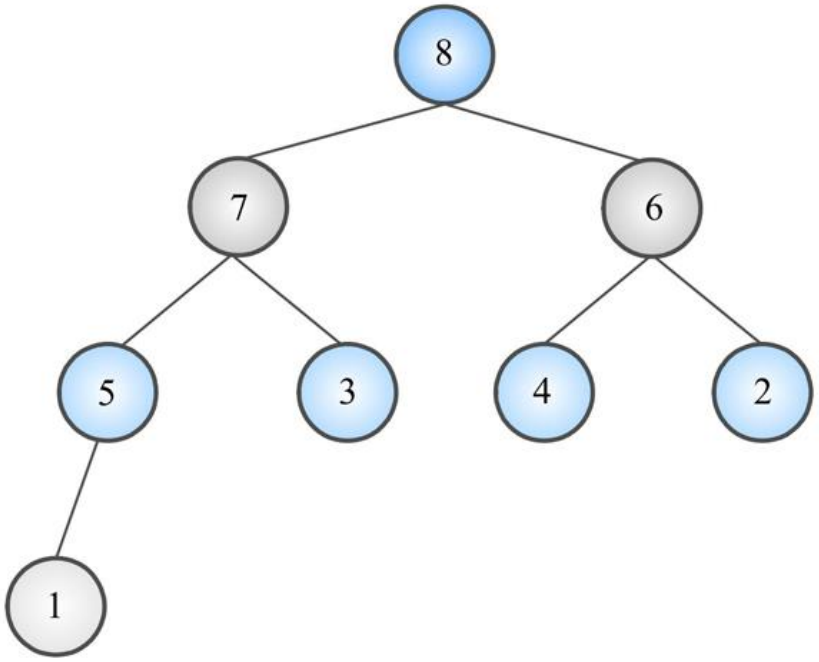


**圖 9.18** 圖 9.17 資料形成的最大堆積

# 資料結構與演算法 (續)

- 使用最大堆積來做排列的好處是，我們可以立即找出這群資料裡的最大值，就是堆積裡的第一個元素。如果我們把 8跟最後一個值 1 作交換，那麼 8 就到它排序後的正確位置了。顯然這個交換會破壞最大堆積的規則，此時就要做堆積的重整。

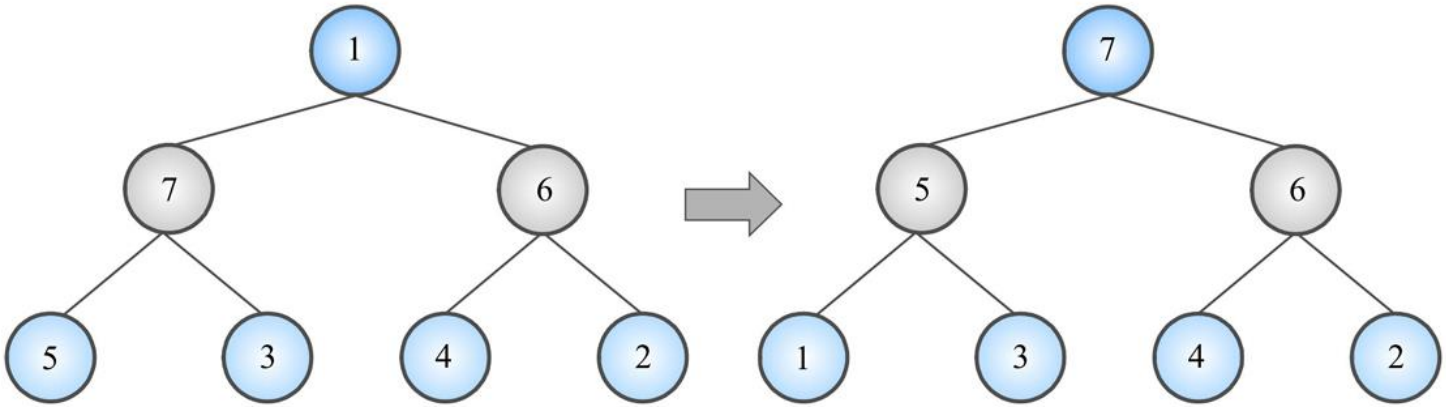


圖 9.19　輸出 8 後之交換與重整結果

# 資料結構與演算法 (續)

- 接著輸出 7 到值為 2 的位置，並且把 2 放到第一個位置。

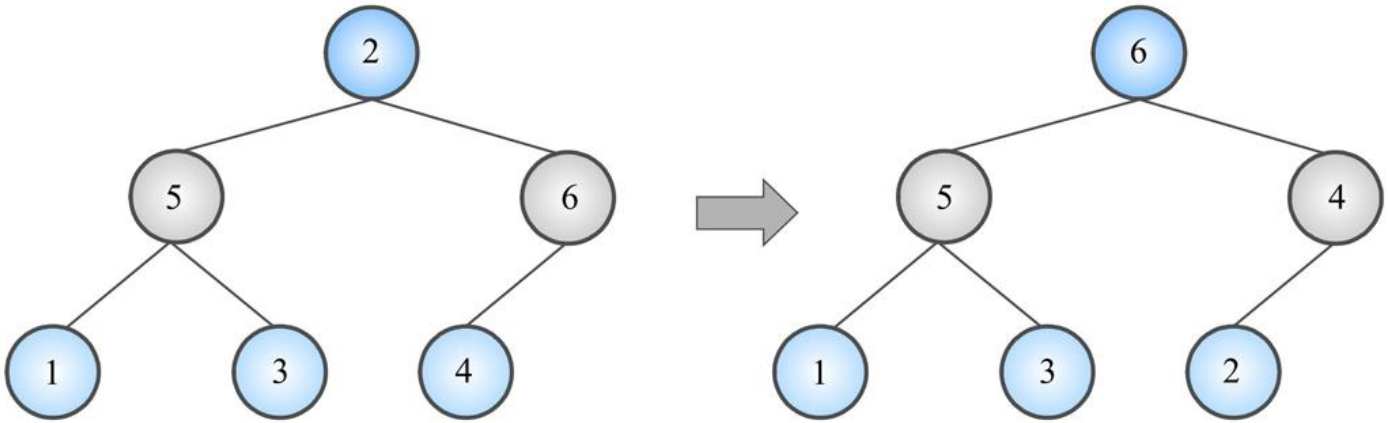


圖 9.20　輸出 7 後之交換與重整結果

# 資料結構與演算法 (續)

- 每次將最大堆積的第一個值和最後一個值交換，即把該最大值送到正確的位置，但每一個交換後，必須做一次重整的動作；此重整動作比較的次數和樹的高度成正比，$n$ 個元素的完全子樹高度是 $\log n$，所以整個演算法花費 $O(n \log n)$ 時間
- 來完成排序的動作，在理論上，堆積排序法和合併排序法都是最佳的排序演算法。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 1 | 8 | 6 | 2 | 7 |
| 8 | 7 | 6 | 5 | 3 | 4 | 2 | 1 |
| 7 | 5 | 6 | 1 | 3 | 4 | 2 | 8 |
| 6 | 5 | 4 | 1 | 3 | 2 | **7** | 8 |
| 5 | 3 | 4 | 1 | 2 | 6 | 7 | 8 |

圖 9.21　原始資料、最大堆積及輸出前三大值後的陣列結果。