7

排　序

「排序」(sorting) 是個既重要又使用頻繁的運算。舉凡資料的整理、搜索，都有排序資料的需求，例如：公司行號年收入、所得稅、學校成績、捐款、運動項目積分、…等日常可見的排名；在電腦中對檔案的名稱、大小、修改日期、…做排序，以利後續的處理；或是在網頁中搜尋關鍵字後呈現網頁的順序…不勝枚舉。對排序過後的資料運用二元搜尋演算法進行搜索，往往是各類搜索問題的核心技術；各種商用的資料庫，也以排序和搜尋的速度相互較量。甚至不少「組合問題」(combinatorial problem) 的解決，都以排序爲其關鍵步驟，例如：在幾何平面的一群點中，找出最近的點對 (closest pair)；在加權圖形中找最小成本延展樹的 Kruskal 演算法的分析中，在最差情況下就得面對排序。在「近似演算法」(approximation algorithms) 求問題近似解的設計中，以排序做爲前置運算 (pre-processing)、或局部搜尋 (local search) 的策略也經常可見。

在本章中我們將重要的排序演算法，及其時間複雜度分析，逐一爲各位介紹。爲簡化演算法的敘述，本章的資料均由存放在一維陣列中的整數做爲排序對象。在諸如資料庫等管理的實際應用上，需要排序的主鍵值 (key) 欄位型態，可能是各種有順序關係的資料型態，例如：整數、實數、字元、日期…等；這將不影響所提演算法的適用性。

7.1 排序的考量

排序的目的是將相同性質的資料，依其順序關係（由小至大或由大至小）排列；隊伍中的身高關係、英文字典中的字詞順序、公司薪資的高低、事件發生距今的遠近…，都是可以排序的資料。在關聯式資料庫 (relational database) 的記錄 (record) 中，每個欄位也都可能是排序的對象；一般而言主欄位 (key field) 大都會加以排序，以方便使用者觀察或取用所要的記錄資料。當資料涉及資料庫時，我們皆以主鍵欄代表整筆紀錄。

依據排序資料存放位置的不同，排序的範疇可分成兩種：

(1) 內部排序法 (internal sorting)：指的是資料全部在主記憶體中。

(2) 外部排序法 (external sorting)：指的是主記憶體中只存放部分資料，而大部分的資料皆在外部記憶體（如：硬碟、磁帶…檔案）中。

兩種排序方式在存取速度方面，有顯著的差異：內部排序法的資料存取皆在主記憶體中進行，而主記憶體屬於隨機存取設備 (random access device)，遂速度較快。而外部排序法的資料大多在硬碟、磁帶等循序存取設備 (sequential access device) 中，所以存取速度會慢上許多，處理資料的量通常也比內部者大上許多。本章的討論以內部排序法爲主。

根據資料在排序前後的相對位置關係，亦有是否具有「穩定性」的考量。例如：

9　12　9*　3　6　8　6*　4　11　7　12*

若排序後的結果爲：

3　4　6　6*　7　8　9　9*　11　12　12*

即原本值相同資料間的先後順序，不受排序影響者，稱該排序演算法具有穩定性。若排序後的結果爲：

3　4　6*　6　7　8　9　9*　11　12　12*

則該排序演算法不具穩定性（6 和 6* 在排序前後的相對位置改變了！）。

至於排序演算法的時間複雜度分析，我們將著眼於排序時所需的「大小比較」(comparison) 或「資料搬動（交換）」(data movement) 的次數計算。下面介紹的排序演算法皆以由小至大排序爲例。

7.2 排序演算法

在下面各小節中，我們會分別介紹：挑選排序法、插入排列法、氣泡排列法、Shell 排列法、合併排列法、快速排列法、基數排列法、堆積排列法等演算法；並分析各個排序演算法的時間複雜度。

7.2.1 挑選排序法

在 2.2.1 節中我們已提過「挑選排序」(selection sort) 的演算法。欲排序的 n 筆資料先行存放在一維陣列中，利用 n 次迴圈完成排序—在第 i 次迴圈時，挑出第 i 小的資料，將之與陣列中第 i 筆資料對調—即可重整陣列使成爲由小至大排序的數列。挑選排序的實作可見程式 2-3，範例 2-4 中則有執行細節的圖例（圖 2-5）。在 2.2.1 節中也提及其時間複雜度爲 $O(n^2)$。

挑選排序演算法除了幾個局部變數 (local variable) 外，並不需要額外的空間—資料的交換都在原存放資料的一維陣列上完成。某個資料在找比它小的資料時，會有資料對調的動作；這個動作可能使得數值相同的資料，不再保持原來的相對先後順序，遂挑選排序法不具穩定性。

7.2.2 插入排序法

「插入排序」(insertion sort) 是將排序的過程視爲：將未排序的資料逐一插入已排序的部分資料中。具體來說，在存放 n 筆資料的一維陣列中，視前 i–1 筆爲已排序資料，插入排序則將第 i 筆資料插入前 i–1 筆中，其中第一筆資料可直接當成已排序，因此 $2 \le i \le n$。

在圖 7-1 中，我們描繪了插入排序法中，第 i 次迴圈做的動作：將第 i 筆未排序資料，找到它在已排序數列（共 i–1 筆）中應排放的適當位置插入之，使其成爲資料個數爲 i 的已排序數列。

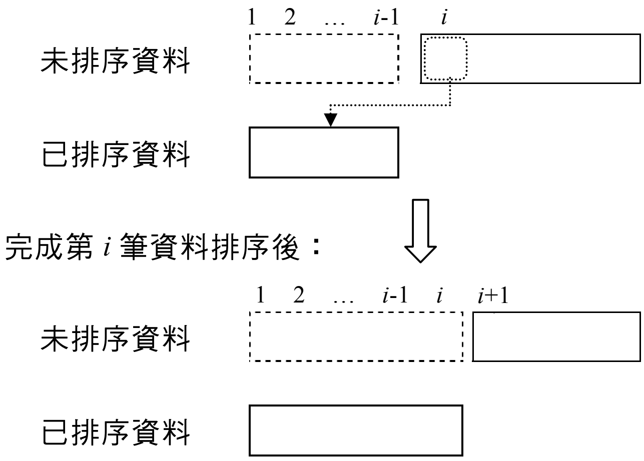


圖 7-1　插入排序法的圖示

茲將其演算法整理如下：

演算法 7-1 插入排序法

輸入：整數陣列 data，長度為 n

輸出：排序陣列 data，若 i<j，則 data[i]≤data[j]，0≤i,j≤n-1

```
for (i=1; i<n; i++)
{  target = data[i];
   j = i;
   while ((j>0) && (data[j-1]>target))
   {   data[j] = data[j-1];
       j--;
   }
   data[j] = target;
}
// for (target=data[i], j=i; (j>0) && (data[j-1]>target); j--)
//     data[j] = data[j-1];
```

我們用圖 7-2 來顯示插入排序法第 i 次迴圈所執行的資料比較與搬動。由於第 1 筆資料位於 data[0]視為已排序，遂從第 2 筆資料 data[1]開始插入排序（第 1 次迴圈處理 data[1]、第 2 次迴圈處理 data[2]、…、第 i 次迴圈處理 data[i]、…），共計 n–1 次。

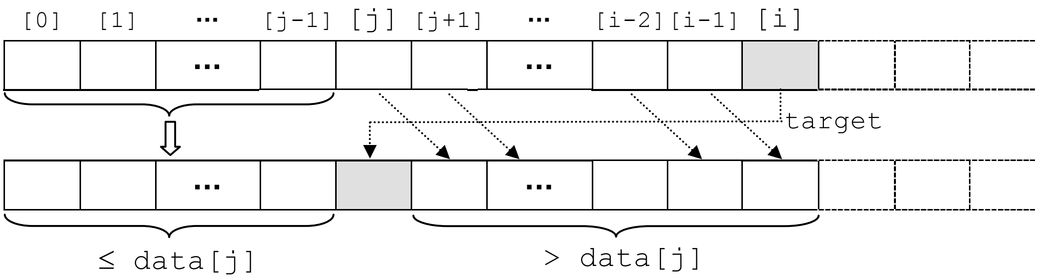


圖 7-2 插入排序第 i 次迴圈的資料比較與搬動

由圖 7-2 可知第 4~8 行的程式碼將 target (target=data[i])插入 data[0]~data[i-1]之間：只要 target 所在位置之前的資料比 target 大（第 4 行 (j>0) && (data[j-1]>target)），就右移一個位置（第 4 行 data[j]=data[j-1]），並往左檢查其他資料（第 6 行）；否則，target 可放置在位置 j（第 8 行），使得 data[0]~data[i]成為排序好的數列。而第 1 行 for 迴圈的 i 由 1 開始到 n-1 結束即可！演算法依序將 data[1]、data[2]、…、data[n-1]，插入其於已排序部份數列中的適當位置，而完成排序動作。請想想：2~7 行是否可用 10~11 行取代？

第 4~7 行的迴圈至多會執行 $O(n)$ 次，而外層(1~9 行)的迴圈會執行 n–1 次，所以插入排序的時間複雜度為 $O(n^2)$；即

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

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

$$=O(n^2)$$

範例 7-1 詳細列出了插入排序法的執行過程。

範例 7-1

圖 7-3 將 8 筆資料經插入排序法完成排序的過程詳細列出：

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	
排序前	8	4	12	6	9	3	7	10	
i=1	4	8							j=0
i=2	4	8	12						j=2
i=3	4	6	8	12					j=1
i=4	4	6	8	9	12				j=3
i=5	3	4	6	8	9	12			j=0
i=6	3	4	6	7	8	9	12		j=3
i=7	3	4	6	7	8	9	10	12	j=6

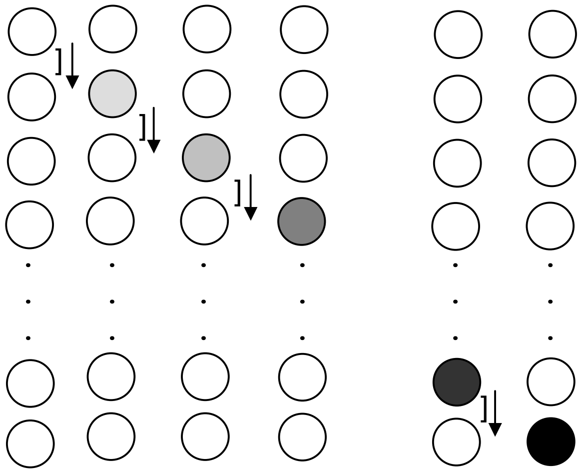
圖 7-3　插入排序的過程

由範例 7-1 可知，插入排序演算法除了 i 和 j 幾個註標變數外，不需要額外的記憶空間。再者任一個未排序的元素，決定它在已由小至大排列數列中的位置時，不會越過小於或等於他自己的資料（第 4 行）；所以值相同的資料在排序時，其相對先後順序依然保存，所以插入排序法具有穩定性。

7.2.3 氣泡排序法

當資料希望以小至大（由上而下）排序時，「氣泡排序」(bubble sort) 要求相鄰的兩元素要維持「上小下大」的順序關係；於是相鄰的兩元素會互相比較大小，將較大的資料往下放。圖 7-4 描繪了這種逐步將大資料往下推去、小資料向上冒出的現象。一旦某資料被推到最下面，過程中他應比其它所有資料都大，他肯定是最大的資料；於是同法找出次大、第三大、…，即可完成排序的動作。氣泡之名實來自其大資料向下沉去、或小資料向上浮出，好似重物投

入水中下沉，而其擠壓之氣泡往上竄出之現象。



表示相鄰兩元素比大小，大者往下放

圖 7-4　氣泡排序法的圖示

氣泡排序的演算法如下：

演算法 7-2 氣泡排序法

輸入:整數陣列 data，共計 n 筆資料

輸出: 排序陣列 data，若 i<j，則 data[i]≤data[j]，0≤i,j≤n-1

```
1   for (i=n-1; i>0; i--)
2       for (j=1; j<=i; j++)
3           if (data[j-1]>data[j])
4               sawp(&data[j-1], &data[j]);
```

第 2~4 行的迴圈將 data[0]~data[i] 中的相鄰資料逐一比較，大者往下移(第 3、4 行)；待此迴圈結束時，data[i]必爲 data[0]~data[i] 中最大者。外層迴圈(第 1~4 行)則令 i 從 n-1 倒數至 0；所以最大、次大、第三大、…元素則可分別往下沉去，所有資料的排序因而完成。

其資料比較的次數可由下式決定：

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

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

$$= O(n^2)$$

所以其時間複雜度爲 $O(n^2)$。

範例 7-2

圖 7-5 將 8 筆資料經氣泡排序法完成排序的過程列出，其中 ↓ 表示相鄰資料比較大小後，將較大資料換到下方、較小資料則換往上：

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

(a) 第一個迴圈：`i=n-1, j=1~n-1`

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

(b) 各迴圈結束之後的結果

圖 7-5 氣泡排序的過程

由圖 7-5 可知：大資料的確往下沉（灰色背景）；而小資料有如氣泡般逐次向上冒出（最小資料 3，最爲顯著）。

氣泡排序法亦不需要額外的儲存空間（除了註標等暫存變數外）。氣泡排序法是穩定的—相同值的資料相鄰時，不會執行對調（第 3 行），遂其相對先後順序不至於受氣泡排序影響。

請各位仔細端詳比較：挑選排序法和氣泡排序法，兩者的相似處在於每次迴圈會決定當時最小或最大的資料。不同處在於挑選排序記的是最小資料的註標，俟確定誰是當時最小後，方執行對調，遂資料交換是每個迴圈一次；而氣泡排序法則不去記誰最大，直接將相鄰兩元素大者往下挪，遂每次迴圈可能有多次的資料交換。再者，挑選排序法不具穩定性，而氣泡排序法則有穩定性。兩者的異同處值得各位觀察比較，應可掌握排序方法在資料交換上著墨的技巧。

7.2.4 Shell 排序法

Shell 排序法是由 Donald L. Shell 於 1959 年所提出的，由上一節氣泡排序法的經驗可知：即令原來的資料尚未完全排序完成，但若有不少資料已位在它

們在排序完成時應處的位置上，則可減少資料搬移的動作。以範例 7-2 來說，在 i=3 之後的運算，已沒有任何資料交換需要執行了。試想：若用氣泡排序法對一由小至大排序完成的資料數列進行排序，則不需要任何資料交換（當然資料比較是少不了的）；但是若對一已由大至小順序排放的數列進行排序，則需要 $O(n^2)$ 次的資料交換。直覺來想：資料的大小順序符合者愈多，可減少資料交換的個數，對所有資料的排序愈有利。Shell 排序法的概念即希望以「分組排序、逐次增加大小順序符合的資料個數」來進行排序：資料先行分成小組，各小組進行排序，由於各組內的資料已依大小順序排放，則整體再併成大組排序時，應可獲得減少資料交換的好處。我們先看範例 7-3 的例子。

範例 7-3

倘若輸入資料共有 12 筆資料：

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

則先將此 12 筆資料分成四組，每隔 4 筆資料者，分在同一組，如圖 7-6 所示（組 1：8、9、1 位於[0]、[4]、[8]）。然後各組內先用插入排序法進行排序—使成「4-組排序」(4-sorted) 的數列（4 組各含 3 筆資料且分別排序的數列）；再用插入排序將之排序成爲 1-組排序 (1-sorted) 數列。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
	8	4	12	6	9	3	7	10	1	5	2	11
組 1 排序	1				8				9			
組 2 排序		3				4				5		
組 3 排序			2				7				12	
組 4 排序				6				10				11
4-組排序數列	1	3	2	6	8	4	7	10	9	5	12	11
1-組排序數列	1	2	3	4	5	6	7	8	9	10	11	12

圖 7-6　Shell 排序的過程

各組以插入排序完成後的數列稱爲「h-組排序數列」(h-sorted sequences)；在範例 7-3 中，h 先定爲 4（間隔 4 筆資料者在同一組），各組分別插入排序形成 4-組排序數列後，再定 h 爲 1（間隔 1 筆資料者在同一組），執行插入排序使成 1-組排序數列，即得排序結果。我們觀察圖 7-6 中的第一筆資料 8，它在 $h = 4$ 小組（組 1 三筆資料）插入排序中，會自[0]移至[4]（或說組 1 中有 1 筆資料比它小—插入排序時會有 1 筆資料移至其前）、爾後 $h = 1$ 的插入排序中，會移至[5]、[6]、[7]（共有 3 筆資料比它小—插入排序時會有 3 筆資料移至其前），總共 4 次搬移；倘若原數列直接採用插入排序，則第一筆資料 8，會自[0]移至[1]、[2]、[3]、…、[6]、[7]（共有 7 筆資料比它小—插入排序時會有 7 筆資料移至其前）共 7 次搬移。各位可察覺 Shell 排序的資料搬移次數相較插入排序爲少。

當資料量頗大時，D. E. Knuth 建議 h-組排序數列中的 h 由小到大可以是 1, 4, 13, 40, 121, …（即 $h_{s+1} = 3h_s+1$，且 $h_1 = 1$，n 爲輸入資料個數），以 $n = 100$ 爲例，可先完成 13-組排序數列、再進行 4-組排序、最後進行 1-組排序。

至於 h-組數列中 h 的相關探討，有數位學者提出不同的看法，在下面演算法之後討論，在此 Shell 排序演算法則視 $h_t, h_{t\text{-}1}, \ldots, h_1$ ($h_1 = 1$) 爲輸入的參數。

演算法 7-3 Shell 排序法

輸入：整數陣列 data，共計 n 筆資料；間隔：$h_t, h_{t-1}, \ldots, h_1$，且 $h_1=1$

輸出：排序陣列 data，若 i<j，則 data[i]≤data[j]，0≤i,j≤n-1

```
for (k=t; k>=1; k--)
{    h = h_k;
     for (i=h; i<n; i++)       //執行 h-組插入排序
     {    target = data[i];
          j = i;
          while ((data[j-h]>target) && (j>0))
          { data[j] = data[j-h];
            j = j-h;
```

```
9             }
10            data[j] = target;
11        }
12  }
```

其中第 3~11 行的迴圈，實爲將資料每間隔 h_k 個分爲同組，各組分別執行插入排序（注意第 6 行的條件：data[j-h]>target 和第 7~8 行中 j 和 j-h 的調整）。外層迴圈則透過 k=t,t-1,...,1（第 1 行）使 $h=h_t$, h_{t-1},...,h_1(=1)（第 2 行），建出 h-組排序數列（第 3~11 行）；最後完成的 1-組排序數列，即爲排序結果。

Shell 排序的時間複雜度與 $h_1, h_2, \ldots, h_t$ 此遞增數列（或 $h_t, h_{t-1}, \ldots, h_1$ 的遞減數列）有關！Knuth 選用的 1, 4, 13 ,40, 121, …要比 1, 2, 4, 8, 16, …好（後者偶數位置的元素不會和奇數位置的元素比較）。當 $t = 2$ 時，選定 $h = 1.72\sqrt[3]{n}$，則 Shell 排序的執行時間爲 $O(n^{5/3})$；這比起單純插入排序法所需的 $O(n^2)$ 要好。這是個非常有效率的結果：多一次前置處理（2-組排序數列先形成），即可使插入排序法（其實就是原始資料直接建出 1-組排序數列）得到改善。若選用 $h_k = 2^k-1$，其中 $1 \le k \le \lfloor \log n \rfloor$，則所使用的比較次數在最差情況只需 $O(n^{3/2})$。至於實際實驗中（n 可能高達 250000），選用 $h_i = (3^i-1)/2$，$1 \le i \le t$，而 t 爲使 $h_{t+2} \ge n$ 的最小 t（即 Knuth 所提的 1, 4, 13, 40, 121,…），可有令人非常滿意的執行速率。學者又提出證明，若整數 h_k 的形式皆爲 2^i3^j 的組合，則比較的次數可在 $O(n\log_2 n)$ 內（不過在實際實驗中，除非 n 非常大，這樣選用 h_k 並得不到滿意的執行速率）。

在此我們並沒有詳述 Shell 排序的時間複雜度分析。然而提醒各位，在實際應用上 Shell 排序是不錯的排序選擇。它不需額外的記憶空間；不過它不具穩定性，理由是分組個別排序時，值相同的資料可能不在同一組，相對先後順序可能因而不再保持。

7.2.5 合併排序法

在前一節中我們已有了「分組各別排序」的觀念；合併排序 (merge sort) 的概念則是：將兩組已各自排序好的數列予以合併，使成為一完整的排列數列。令陣列 A 存放了 m 個已排序的資料，陣列 B 存放了 n 個已排序的資料，合併排序的目的即希望將陣列 A 和 B 各別已排序的資料，合併存放至陣列 C 中，即 C 會共有 $m+n$ 筆已排序的資料。先看一個合併排序的例子：

範例 7-4

A	[0]	[1]	[2]	[3]	[4]	[5]					
	4	8	12	13	15	16					
B	[0]	[1]	[2]	[3]	[4]						
	2	3	5	9	14						
C	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	4	5	8	9	12	13	14	15	16

各位可以很直覺地讓註標由小到大，逐一比對 A、B 中的資料，來得到陣列 C 的結果。我們可以為陣列 A、B 和 C 分別準備註標 i、j 和 k，然後比較 A[i]和 B[j]的大小，凡小者，即置入 C[k]中，而該小者陣列的註標、和 C 陣列的註標 k 都應加 1，再繼續做「比較選小者」，直到陣列 A、B 中其一結束，即可停止比較；而另一陣列的剩餘資料，可直接搬移至陣列 C 的尾端。

演算法 7-4 將上述想法整合陳述。

演算法 7-4 合併兩個已排序的數列

輸入：已排序陣列 A，共 m 筆資料；已排序陣列 B，共 n 筆資料

輸出：排序陣列 C，若 i<j，則 C[i]≤C[j]，0≤i,j≤m+n-1

```
1   void merge(int C[], int k, int A[], int i, int m, int B[],
                  int j, int n)
2   {      while ((i<=m) && (j<=n))
3          {   if (A[i] <= B[j])
4                   C[k++] = A[i++];
5              else
6                   C[k++] = B[j++];
7          }
8          while (i<=m) C[k++] = A[i++];
9          while (j<=n) C[k++] = B[j++];
10  }
```

第 1 行程序 merge 的傳入參數包括：陣列 C、C 的起點註標 k，陣列 A、A 起點註標 i 和 A 的長度 m，陣列 B、B 起點註標 j 和 B 的長度 n。第 2~7 行的迴圈以 (i<=m) && (j<=n)（陣列 A、B 中皆仍有資料）做爲繼續的條件。第 3~6 行做「相關資料比較大小，選出小者放入陣列 C」；第 8 或 9 行直接搬移尚有剩餘資料陣列的所有內容至陣列 C 的尾端（而沒有剩餘資料的陣列會在註標檢查的條件中自然略過）。

一旦瞭解合併的技巧，那麼陣列 A 和陣列 B 的各自排序，亦可靠此合併的技巧來完成—僅僅藉由「合併」完成整個數列的排序！我們分別對陣列 A 和 B 做資料的遞迴分割，直到只剩一筆資料爲止；此時只含一筆資料的數列，自然是已排序好的，把兩個長度爲 1 的排序數列合併，即可呼叫 merge 程序來完成（merge 的傳入參數有長度的考量）；所有長度爲 2 的排序數列分別完成後，再繼續把兩段長度爲 2 的排序數列，合併成長度爲 4 的排序數列；以此類推，即可一路將當初分割的兩個數列合併—所有的運算就是「合併」。

這種設計的概念，實爲演算法中「分割與各自擊破」(divide-and-conquer)的策略。合併排序正是分割與各自擊破策略的經典演算法。

演算法 7-5 合併排序（遞迴）

輸入：data 陣列共計 n 筆資料

輸出：排序陣列 data，若 i<j 則 data[i]≤data[j]，0≤i,j≤n-1

```
1   int  data[n];
2   void merge_sort(int A[], int left, int right)
3   {       int  m;
4           if (left < right)
5           {   m = (left+right)/2;
6               merge_sort(A, left, m);
7               merge_sort(A, m+1, right);
8               merge(A, left, A, left, m, A, m+1, right);
9           }
10  }
11  main()
12  {       read_input_data(data, n);
13          merge_sort(data, 0, n-1);
14  }
```

在主程式第 12 行中，先呼叫 read_input_data 程序，讀入欲排序的陣列 data，共 n 筆資料；第 13 行即呼叫 merge_sort，傳入的參數是陣列 data 和其起點、終點註標。

請注意：演算法 7-5 第 8 行的呼叫

```
merge(A, left, A, left, m, A, m+1, right);
```

傳入的陣列是同一陣列 A 的前半與後半，並存放在同一陣列處。對 C 程式語言而言，傳入程序的陣列名稱即形同傳入其位址，上述對陣列的存取動作會導致資料的錯誤（合併後的資料可能覆蓋尚未處理的資料）！可將演算法 7-4 改寫如下—將傳入的陣列 A、B（實爲同一陣列的前、後半子陣列），在程序內先行複製（至 temp），以複製後的陣列做比較大小的依據，合併後的資料存放回陣列 A 就不致出錯。

演算法 7-4-1 以 C 完成合併已排序的數列

輸入：已排序陣列 A，共 m 筆資料；已排序陣列 B，共 n 筆資料

輸出：排序陣列 C，若 i<j，則 C[i]≤C[j]，0≤i,j≤m+n-1

```
void merge(int C[], int k, int A[], int i, int m, int B[],
           int j, int n)
{   int temp[max_size], p;
    for (p=i; p<=m; p++) temp[p] = A[p];
    for (p=j; p<=n; p++) temp[p] = B[p];
    while ((i<=m) && (j<=n))
    {   if (temp[i]<=temp[j]) C[k++] = temp[i++];
        else C[k++] = temp[j++];
    }
    while (i<=m) C[k++] = temp[i++];
    while (j<=n) C[k++] = temp[j++];
}
```

範例 7-5

圖 7-7 描繪了呼叫演算法 7-5 中合併排序程序 merge_sort(A,0,7)的執行過程（凡加註底線的資料，表示它已排序完成）：

Left	right	m	遞迴呼叫層數	執行行數	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
					24	4	35	1	60	12	52	16
0	7	3	1	1~6	24	4	35	1				
0	3	1	2	1~6	24	4						
0	1	0	3	1~6	24							
0	0	0	4	1~4,10	24							
1	1	1	3	7		4						
1	1	1	4	1~4,10		4						
0	1	-	3	8~10	4	24						
0	3	1	2	7			35	1				
2	3	2	3	1~6			35					
2	2	2	4	1~4,10			35					
3	3	3	3	1~6				1				
3	3	3	4	1~4,10				1				
2	3	-	3	8~10			1	35				
0	3	-	2	8~10	1	4	24	35				
0	7	3	1	7					60	12	52	16
4	7	5	2	1~6					60	12		
4	5	4	3	1~6					60			
4	4	4	4	1~4,10					60			
5	5	5	3	7						12		
5	5	5	4	1~4,10						12		
4	5	-	3	8~10					12	60		
6	7	6	2	7							52	16
6	6	6	3	1~6							52	
6	6	6	4	1~4,10							52	
7	7	7	3	1~6								16
7	7	7	4	1~4,10								16
6	7	-	3	8~10							16	52
4	7	-	2	8~10					12	16	52	60
0	7	-	1		1	4	24	35	12	16	52	60
0	7	-	1	8~10	1	4	12	16	24	35	52	60
			排序完成		1	4	12	16	24	35	52	60

圖 7-7 遞迴合併排序的執行過程

由圖 7-7 可看出：遞迴合併排序是項極為繁瑣的工作，我們可以利用非遞迴的寫作技巧來改善執行的速率。事實上遞迴呼叫的目的僅在分割出夠小的已排序數列（即元素僅有 1 個），一旦夠小的已排序數列出現，之後的運算即是一路合併了。先觀察圖 7-8，它將圖 7-7 的合併過程重新組合：

Left	right	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
		24	4	35	1	60	12	52	16
0	1								
		4	24						
2	3								
				1	35				
0	3								
		1	4	24	35				
4	5								
						12	60		
6	7								
								16	52
4	7								
						12	16	52	60
0	7								
		1	4	12	16	24	35	52	60
排序完成		1	4	12	16	24	35	52	60

圖 7-8　合併排序的合併過程

圖 7-8 提示我們：合併排序其實是靠不斷地合併得到排序的結果。於是我們可將演算法改寫如下（在此我們假設 $n = 2^k$，$k \geq 0$）：

演算法 7-6 合併排序（非遞迴）

輸入：data 陣列共計 n 筆資料

輸出：排序陣列 data，若 i<j，則 data[i]≤data[j]，0≤i,j≤n-1

```
main()
{   int len = 2;
    while (len<=n)
```

```
    {   for (i=0; i<n; i+=len)
            merge(A,i,A,i,i+len/2-1,A,i+len/2,i+len-1);
        len *= 2;
    }
}
```

接下來分析合併排序的時間複雜度。上面的 merge 程序，在合併兩段長度爲 $k/2$ 的排序數列時，得花 $O(k)$ 的時間，使成爲長度爲 k 的排序數列。而合併排序在分割各自擊破時，可視爲將原來長度爲 n 的數列，分割成兩個長度爲 $n/2$ 的子數列，俟這兩個子數列各自排序完成後，再合併這兩個排序子數列，成爲長度是 n 的排序數列。我們可將此關係式寫成：

$$T(n) = 2T(n/2) + O(n)\text{。}$$

解讀成：合併排序長爲 n 的數列所花的時間 $T(n)$，相當於加總分別合併排序 2 個長度爲 $n/2$ 的子數列所需的時間：$2T(n/2)$、以及合併此兩子數列完成排序所要的時間 $O(n)$。而

$$\begin{aligned}
T(n) &= 2T(n/2) + O(n) = 2(2\,T(n/4) + O(n/2)) + O(n) \\
&= 2^2\,T(n/2^2) + O(n) + O(n) \\
&\quad \cdots \\
&= 2^k\,T(n/2^k) + O(n) + O(n) + \ldots + O(n) \\
&= 2^k\,T(n/2^k) + k\,O(n)\text{。}
\end{aligned}$$

當 $n = 2^k$ 時，$k = \log_2 n$；代入上式：

$$\begin{aligned}
T(n) &= 2^{\log_2 n}\,T(1) + \log_2 n\,O(n) \\
&= n\,T(1) + O(n\log_2 n) \\
&= O(n\log n)\text{。}
\end{aligned}$$

因此合併排序的時間複雜度爲 $O(n\log n)$。

倘若 n 未必是 2 的乘方，第 5 行 merge 程序呼叫應將傳入參數有關長度的細節加以修改，演算法 7-6-1 可處理 n 未必是 2 的乘方的合併排序：

演算法 7-6-1 合併排序（非遞迴）—資料大小未必為 2 的乘方

輸入：A 陣列共計 n 筆資料

輸出：排序陣列 A，若 i<j，則 A[i]≤A[j]，0≤i,j≤n-1

```
0  # define MIN(x, y) (x < y ? x : y)
1  main()
2  {   int len = 2;
3      while (len<=n)
4      {   for (i=0; i<n; i+=len)
5              merge(A,i,A,i,i+len/2-1,A,i+len/2,MIN(i+len-1,n-1));
6          len *= 2;
7      }
8      if (len/2<n)
9          merge(A, 0, A, 0, len/2-1, A, len/2, n-1);
10 }
```

第 0 行強調巨集 MIN 得先行定義（其他行號可與演算法 7-6 相互參照比對），方可使用（第 5 行）。因 n 未必是 2 的乘方 (2^k)，第 5 行 merge 程序呼叫時，所傳入的參數將前半子陣列維持是 2 的乘方，而後半子陣列則依其實際長度（自 i+len/2 起，MIN(i+len-1,n-1) 結束）傳入；且應額外考慮第 9~10 行（於 3~7 行的 while 迴圈之外），來執行最後那次合併（其總長度爲 n，而對應註標爲 0~n-1）。

在 merge 程序中，演算法 7-4 用了陣列 B 和 C（呼叫時演算法 7-5 用相同陣列 A 傳入，遂在演算法 7-4-1 仍有陣列 temp）來暫存合併排序執行過程的資料變化，所以合併排序用了 $O(n)$ 的額外記憶體空間。合併排序具有穩定性，因其在合併時不致將相同值的資料對調。

7.2.6 快速排序法

在合併排序中，我們已引用了分割和各自擊破的演算法策略—先分割、各自擊破，爾後再不斷地合併出最後的結果。試想：倘若不必合併是否可以得到節省時間的好處？若分割出的兩個子數列其中某一數列的所有值，完全比另一數列的所有值要小（稱前者爲小子數列、後者爲大子數列），則兩個子數列即可各自排序，不必合併，直接把排序後的大子數列，接在排序後的小子數列之後，即爲整個數列的排序結果！這就是 C. A. R. Hoare 所提出「快速排序」(quicksort) 的概念。其細部設計如下：

首先選取某個元素做爲基準值（通常選第一個元素 `data[0]`），令此基準值爲 `target`，然後將所有比 `target` 小的資料，都放在 `target` 的左邊；而所有不比 `target` 小的資料都放在 `target` 的右邊；如圖 7-9 所示。

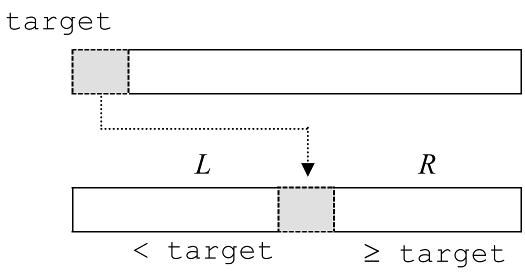


圖 7-9 快速排序法的基本分割步驟

如此一來數列 L（其元素值皆 ＜`target`）和 R（其元素值皆 ≥`target`）則可分別再用此基本分割步驟來進行排序（分割但不須合併）。而此 `target` 一旦移動至數列 L 和 R 之間，即不會再異動；亦即此位置正是 `target` 在排序後所應在的位置。只要數列 L 和 R 各自排序完成，則整個數列即可排序完畢。

至於如何決定 `target` 排序後所應在的位置，使比 `target` 小的資料都在其左邊，而不比 `target` 小的資料都在其右邊，可利用圖 7-10 所描繪的方法來實現。圖 7-10 中的資料存放在陣列 `data[left]~data[right]` 中，`swap(&a,&b)` 是執行交換 `a` 和 `b` 內容的程序。令 `target=data[left]`

且 i 和 j 是兩個陣列註標變數。

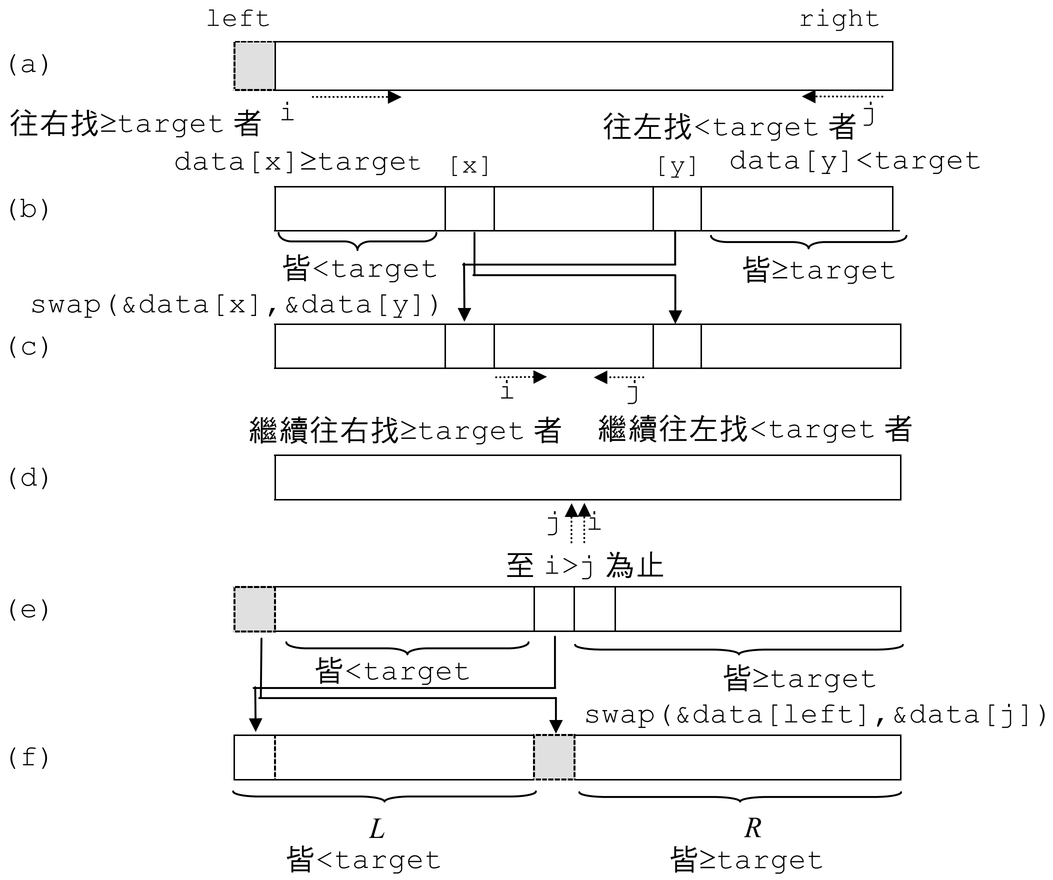


圖 7-10　決定 target 排序後所應在的位置

我們先利用註標 i 自 left 處往右找 ≥target 的資料，找到即暫停；再利用註標 j 自 right 處往左找 ＜target 的資料，找到也暫停；如圖 7-10 (a)。一旦找到這樣的 data[x] 和 data[y]，則交換這兩項資料；使 data[left]~data[x] 的資料皆小於 target，而 data[y]~data[right] 的資料皆 ≥ target；如圖 7-10 (b)。這種尋找、爾後交換的動作，須持續進行，如圖 7-10 (c)，直到 i > j（資料皆已掃描看過）爲止（圖 7-10 (d)）。此時 data[left]~data[j]的資料皆 ＜target，而 data[i]~data[right] 的資料皆 ≥target，如圖 7-10 (e)；於是交換 target（在 data[left]處）

和 data[j]，使成如圖 7-10 (f) 的數列。如此一來，target 在陣列[j]的位置上，在其左的資料皆 < target，即爲上述的數列 L；在其右的資料皆 ≥ target，即爲上述的數列 R。此後 L 和 R 即可各自排序了。L 內的資料再也不必和 R 內者做任何大小比較！而 L 又會分割出 L_L 和 L_R，L_L 內的資料又不必與 L_R (和 R) 內者做大小比較，L_R 內的資料也不必與 L_L (和 R) 內者做比較；R 亦然也。隨著 L 和 R 持續地分割，其長度逐次遞減，考慮第 k 次分割後的左半數列，它不必該次分割的右半數列、也不必與第 k–1、k–2、…、1 次的右半數列比較大小！如此累積的效應，讓快速排序的效能在一般情況下得以傲視群雄！而其最差情況則待後續討論。

綜合上面的討論，快速排序演算法可描述如下。注意：輸入時已將 data[n] 設爲 MaxInt (系統允許的最大整數)，這個設定可使註標 i 的移動，不致在某些狀況中 (例如：往右找不到任何值 ≥ target) 出錯。

演算法 7-7 遞迴快速排序

輸入：整數陣列 data，共 n 筆資料 (data[0]~data[n-1])，data[n]=MaxInt

輸出：排序陣列 data，若 i<j，則 data[i]≤data[j]，0≤i,j≤n-1

```
void QuickSort (int data[], int left, int right)
{   int i, j;
    if (left<right)
    {   i = left+1;
        j = right;
        target = data[left];
        do
        {   while (data[i]<target && i<=j) i++;
            while (data[j]>=target && i<=j) j--;
            if (i<j) swap(&data[i], &data[j]);
        }   while (i<j);
        if (left<j) swap(&data[left], &data[j]);
```

```
13          QuickSort(data, left, j-1);
14          QuickSort(data, j+1, right);
15      }
16  }
```

第 4~12 行的程式碼與圖 7-10 的執行概念是相互對應的！第 13 行和第 14 行是以遞迴的方式呼叫 QuickSort 程序，將數列 *L*（data[left]~data[j-1]）和數列 *R*（data[j+1]~data[right]）以快速排序分別進行排序也。請注意：在第 8 和 9 行中有 i<=j 的條件必須符合，它保證註標所指元素的正確性（即標註範圍必須滿足 i<=j）。

範例 7-6 提供一個 10 筆資料的快速排序過程：

範例 7-6

圖 7-11 演練了 10 個資料執行快速排序的過程。凡加注底線的資料，表示它已安放於排序後所應在的位置上了！

left	right	遞迴呼叫層數	執行行數		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]		
0	9	1	1~7	[	24	4	35	1	60	12	52	16	45	20	]	∞
			8,9				↑i							↑j		
			10				20							35		
			8,9						↑i			↑j				
			10						16			60				
			8~11							↑j	↑i					
			12		12					24						
0	4	2	13	[	12	4	20	1	16	]						
			1~9				↑i	↑j								
			10				1	20								
			8~11				j↑	↑i								

			12	1		12							
0	1	3	13	[1	4	]							
			8~11	j↑ ←→	↑i								
			12	1									
0	-1	4 / 5	13/1~3,16	1									
1	0	4 / 5	14/1~3,16		4								
3	4	3	14			[	20	16	]				
			1~11					↑j	↑i				
			12				16	20					
3	3	4	1~3,16				16						
5	4	4 / 3	1~3,16										
6	9	2	14/1~7						[	52	60	45	35]
			8, 9								↑i		↑j
			10								35		60
			8, 9									↑j	↑i
			12							45		52	
6	7	3	13/1~7						[	45	35	]	
			8, 9								↑j	↑i	
			12							35	45		
6	6	4	1~3,16							35			
8	7	4 / 3	1~3,16										
9	9	3	14/1~7									[	60]
			16										60
		…											
		排序完成		1	4	12	16	20	24	35	45	52	60

圖 7-11 遞迴快速排序的執行過程

注意：我們在上例中的 data[10]處放置了 MaxInt（用 ∞ 表示），其作用可在 left=6、right=9、執行行數爲 8~9 處得知。

這個例子將遞迴快速排序的執行過程，詳細地列出，請各位仔細檢查。請與合併排序法比較：快速排序法沒有合併的必要—各個未經排序的元素，皆會在適當時候擺放到它應在排序後的位置上。然而各位也應發現：一旦找出

target 的位置後，遞迴呼叫的目的實為分割數列，使形成個數較少的 L 和 R 子數列。這個「分割數列」的動作，其實只須簡單地記住「L 和 R 子數列何在」—亦即它們在陣列中的起迄註標，即可用堆疊存放之而取代遞迴呼叫。我們利用堆疊和迴圈控制，改寫演算法 7-7 成為 7-8 如下，期盼改善快速排序的執行效率。

演算法 7-8 堆疊和迴圈實作快速排序

輸入：整數陣列 data，共 n 筆資料(data[0]~data[n-1])，data[n]=MaxInt

輸出：排序陣列 data，若 i<j，則 data[i]≤data[j]，0≤i,j≤n-1

```
left = 0;
right = n-1;
push(left, right);
while (堆疊中仍有元素)
{   (left, right) = pop();
    target = data[left];
    i = left+1;
    j = right;
    do
    {   while (data[i]<target && i<=j) i++;
        while (data[j]>=target && i<=j) j--;
        if(i<j) swap(&data[i], &data[j]);
    }   while (i<j);
    if (left<j) swap(&data[left], &data[j])
    if (j+1<right) push(j+1, right);
    if (left<j-1) push(left, j-1);
}
```

演算法 7-8 中用到的堆疊基本元素為兩個整數的結構變數（即 (left, right) 記住需要排序數列在陣列中的起迄註標），所以 push 和 pop 皆以一

組結構為存取元素。演算法 7-8 與演算法 7-7 中「找出 target 排序後應在的位置」的運算是一樣的，唯堆疊的引用取代了遞迴的呼叫。注意：演算法 7-8 中第 15 和 16 行的 push，與演算法 7-7 的第 13 和 14 行的遞迴呼叫 QuickSort，是相互對應的；但因堆疊是後進先出的機制，所以呼叫兩次 push：先 R 後 L；pop 時會先取得 L 後取得 R，和呼叫兩次 QuickSort：先 L 後 R；所用的參數順序是不一樣的！各位可自行驗證之。也請比較快速排序在遞迴與否時的執行效率。快速排序法的最差情形請看下面範例。

範例 7-7

圖 7-12 描繪了快速排序法遇到的最差情形（凡加注底線的資料，表示它已安放於排序後所應在的位置上了）：

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
60	52	45	35	24	20	16	12	4	1
1	52	45	35	24	20	16	12	4	60
1	52	45	35	24	20	16	12	4	
	4	45	35	24	20	16	12	52	
	4	45	35	24	20	16	12		
		12	35	24	20	16	45		
		12	35	24	20	16			
			16	24	20	35			
			16	24	20				
				20	24				
				20					
1	4	12	16	20	24	35	45	52	60

圖 7-12　快速排序法遇到的最差情形

範例 7-7 並不像範例 7-6 般，可分出約莫等長的 L 和 R 子數列；實際上每次分割只能分割出一段比原長度少 1 的子數列。

接下來我們分析快速排序法的時間複雜度。從範例 7-7 中可知：快速排序演算法在最差情況下（已排序或已反向排序）需執行

$$n + (n-1) + \ldots + 1 = O(n^2)$$

的時間。不過快速排序法的平均時間複雜度爲 $O(n\log n)$（直覺地想：若分割數列皆大約是原數列的一半，則

$$T(n) = 2T(n/2) + O(n) = O(n\log n)$$

（此式在合併排序中分析過）是可近似成立的。此平均時間的分析可在較高階的演算法分析書中找到[1]。而且快速排序法在實地實驗中的確有非常好的表現，所以經常被運用在排序實務上。

快速排序在分割出子數列的資料個數少於 10 個左右時，可直接用挑選或插入排序法排序該子數列（這些排序法在資料個數少時，執行速率尚佳）；如此所需堆疊（遞迴、非遞迴皆然）的空間則可減少。再者：個數少的子數列先行以挑選或插入進行排序，亦可減少堆疊空間。學者對 `target` 的選用也有不同的建議：有人用亂數爲註標取出陣列中對應的數字做爲 `target`、或推薦在 `data[left]`、`data[(left+right)/2]`和 `data[right]`三者之中找出中位數 (median) 做爲 `target`…。這些想法無非設法讓 `target` 可將數列分出長短約莫相同的 L 和 R 子數列，而使快速排序的時間儘可能地接近 $O(n\log n)$。

快速排序法需要額外的堆疊空間[2]，不具有穩定性（因在交換 `data[i]`和

1　若欲排序的數列長度為 k，分割的 L 和 R 數列分別為 s 和 $k-s$，則
$T(k) = T(s) + T(n-s) + O(n)$。

2　請想想：快速排序法需要的堆疊空間至多、至少是多少？若在理想狀態下「分割數列皆大約是原數列的一半」，$O(\log n)$ 的空間大約少不了，有沒有好方法控制在這範圍內？（習題中討論）

data[j]時，可能破壞值相同資料的相對順序關係）。

7.2.7 計數排序法

計數排序法 (counting sort) 利用「計算數字出現的次數」做爲排序的依據。直覺地想：數字 1~10 共出現 30 個，那麼數字 11（如果有的話）在由小到大的排序數列中應排在第 31（或 32，如果有 2 個 11）個位置。若每個數字都把在原始數列中出現的次數統計出來，各個數字在排序數列上的位置即可推算出來，排序也就完成了！有趣的是：不需要比較大小的運算。

先看個簡單的例子：原始數列爲 9, 7, 8, 7, 8, 8 共 6 筆，由於 7 有 2 筆、8 有 3 筆、9 有 1 筆，可推知：7 的排名位置爲 1~2、8 的排位應爲 3~5、而 9 則爲 6，亦即：排序數列爲 7, 7, 8, 8, 8, 9—只需要各數字的出現次數，確實無須比較大小。吾人可用陣列 C 存放各數字的出現次數，以此例而言：

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
C	0	0	0	0	0	0	2	3	1

只須掃描原始數列一回：遇見 9 (7, 8, …) 即在 $C[9]$ ($C[7]$, $C[8]$, …) 上累加 1，就可建立出現次數陣列 C。接下來透過 C—註標由小而大—可知，7 之前沒有任何數字（$C[1]$~$C[6]$皆爲 0），7 有 2 筆遂知其排名位置爲 1~2，8 有 3 筆其排位爲 3~5、9 有 1 筆其排位應爲 6。細心推敲可得：7 的起始排名爲 $C[1]+C[2]+ \ldots +C[6]+1$ （= 1，因爲數字 1~6 皆無，7 爲排序數列中最小者）、8 的起始排名爲 $C[1]+C[2]+ \ldots +C[7]+1$ （= 3，因爲數字 1~7 共有 2 個，8 爲排序數列中第三位）、9 的起始排名爲 $C[1]+C[2]+ \ldots +C[8]+1$ （= 6，因爲數字 1~8 共有 5 個，8 爲排序數列中第六位）；其中 $C[1]+C[2]+ \ldots +C[p-1]$ 的運算稱爲 $C[p]$ 的「前綴和」 (prefix sum)。

值得提醒的是：此例中數列的個數爲 6，但陣列的大小爲 9—實爲數列中最大者也！於是我們得先掃描原始數列（令有 n 筆）一回，決定孰爲數列中最大

者（令爲 q），方能確定陣列 C 所需要的空間—$\max(q, n)$ （或 $O(n+q)$）。各位可以想像一下：若 q 相較於 n 大許多時，時間、空間的需求會如何？

演算法 7-9 嚴謹地將計數排序描述如下。

演算法 7-9 計數排序

輸入：整數陣列 A,共 n 筆資料(A[0]~A[n-1])

輸出：排序陣列 B，若 i<j，則 B[i]≤B[j]，0≤i,j≤n-1

```
1  for (q=0, i=1; i<n; i++)
2          if (data[i]>q) q = data[i];
3  m = MAX(q, n);
4  int * C = int new [m];
5  for (i=0; i<m; i++) C[i] = 0;
6  for (i=0; i<n; i++) C[A[i]] += 1;
7  for (i=1; i<m; i++) C[i] += C[i-1];
8  for (j=1; j<n; j++)
9  {   B[C[A[j]]] = A[j];
10     C[A[j]] += 1;
11 }
12 // 輸出排序結果 B
```

第 1~2 行決定數列中最大者 q，3~4 行決定陣列 C 的大小—實作時用巨集 MAX 挑出 q、n 中較大者）。第 5~6 行以 $O(n)$ 的時間、$O(\max(q, n))$ （或 $O(n+q)$）的空間統計各個數字出現的次數—以自身數字做爲陣列 C 的註標。第 7 行以 $O(n+q)$ 的時間，執行前綴和。第 8~11 行將資料逐一放置在其於排序後數列的定位上—所用時間爲 $O(n)$，而空間爲 $O(n+q)$！總結言之：時間與空間複雜度皆爲 $O(n+q)$，其中 q 爲 n 筆資料中最大者。

在一般的情況，若 $q \le n$ ，計數排序的表現相當好；但若 $q\ (> n)$ 相當大，則空間的需求會相當大[3]。

範例 7-8

下表列出 $n = 11$ 個數字執行演算法 7-9 前 1~7 行後的結果－前綴和後(第 7 行) 陣列 C[i]恰爲數字 i 出現的次數：

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	5	2	2	9	6	3	4	2	3	6	5
C	0	0	3	2	1	2	2	0	0	1	0
+						…					
C	0	0	0	3	5	6	8	10	10	10	11

數字 0,1,2 之前皆沒別的數字 ≤ 它，可自 B[0]起存放

3 之前有 3 個數字 ≤ 3，可自 B[3]起存放

4 之前共有 5 個數字 ≤ 4，可自 B[5]起存放

. . .

下表列出第 1 個數字 5 (A[0]、j=0) 被安放在 B[6] (C[5]爲 6：有 6 個數字在 5 之前 (≤5)，自 B[6]開始存放) ，其變數間對應關係圖示 (B[C[A[0]]]=A[0])，此後 C[5]=6+1=7，表示若再有數字 5，應放到 B[7]。

j=0

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	5	2	2	9	6	3	4	2	3	6	5
C	0	0	0	3	5	6	8	9	9	9	10
B				3		+1	5				

3　倘若做數字 276587539812, 23, 6, 67, …的計數排序，則空間會異常地大！

下表列出第 2 個數字 2（A[1]、j=1）被安置在 B[0]（C[2]爲 0：沒有數字在 2 之前，自 B[0]開始存放），其變數間對應關係圖示（B[C[A[1]]]=A[1]），此後 C[2]=0+1=1，表示若再有數字 2，應放到 B[1]。

j=1

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	5	2	2	9	6	3	4	2	3	6	5
C	0	0	0 +1	3	5	7	8	9	9	9	10
B	2						5				

下表列出第 3 個數字 2（A[2]、j=2）被安放在 B[1]（C[2]爲 1：有 1 個數字在 2 之前 (≤ 2)，自 B[1]開始存放），其變數間對應關係圖示（B[C[A[2]]]=A[2]），此後 C[1]=1+1=2，表示若再有數字 2，應放到 B[2]。注意：數字 2 出現於原數列前者，排序後亦在前；出現於原數列後者，排序後則在後－相同數值的資料其前後關係會維繫住！

j=2

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	5	2	2	9	6	3	4	2	3	6	5
C	0	0	1 +1	3	5	7	8	9	9	9	10
B	2	2					5				

依此邏輯，計數排序即可在不必比較大小，但需要額外空間存放數字出現次數的輔助，搭配前綴和，計算各數字在排序後數列上的位置，而達到排序的目的：

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

計數排序法中額外的空間是必要的，相同數值的資料其前後關係會維繫住，具有穩定性。通常輸出結果會存到另一個陣列—請觀察第 9~10 行陣列 A、B 會交替引用。

7.2.8 基數排序法

基數排序法 (radix sort) 的比較資料大小概念，與之前排序法的概念不太一樣—它的大小比較是與輸入數列的基底位數有關。假設輸入的數字皆為三位數，那麼基數排序法即先對所有資料的百位數進行排序，再對每一組百位數字相同的資料進行十位數字的排序，爾後再針對每一組十位數字相同的資料，進行個位數字的排序。這個順序由高位數開始排至低位數，稱為「高位數優先」(most significant digit first)；事實上先排個位數，再排十位數，爾後百位數；一樣可以得到排序的結果，這個順序則為「低位數優先」(least significant digit first)。我們先看一個範例：

範例 7-9

假設需要排序的資料為：

224　454　321　735　496　524　98　456　339　578　7

我們用低位數優先的基數排序法：先對個位數字排序，再排十位數字、百位數字。各位數字排序的結果則放在編號為 0~9 的 10 個佇列中。「對個位數字排序」只須依原資料順序，把資料加入對應編號的佇列中即可。

rear

	321			524 454 224	735	456 496	7	578 98	339
0	1	2	3	4	5	6	7	8	9

front

接著依佇列編號由小到大的順序，取出所有資料：

321　224　454　524　735　496　456　7　98　578　339

對十位數字排序（依上列資料順序，把資料加入對應編號的佇列中）：

7		524 224 321	339 735		456 454		578		98 496
0	1	2	3	4	5	6	7	8	9

依佇列編號由小到大的順序，取出所有資料：

7　321　224　524　735　339　454　456　578　496　98

再對百位數字排序（依上列資料順序，把資料加入對應編號的佇列中）：

98 7		224	339 321	496 456 454	578 524		735		
0	1	2	3	4	5	6	7	8	9

最後依佇列編號由小到大的順序，取出所有資料，即得排序的結果：

7　98　224　321　339　454　456　496　524　578　735

10 個佇列的使用，在觀念上毫無窒礙，但在實作上有些浪費空間；畢竟欲存的對象僅是輸入的 n 筆資料，我們可用陣列來模擬這 10 個佇列。首先可用一計數陣列 `count[i]`存放上述第 `i` 個佇列內的數字個數（目前排序位數中，數字爲 i 的資料個數；與上節計數排序中的計數陣列同義也）；以本例而言，個位數字排序後的陣列 count 如下：

count	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	0	1	0	0	3	1	2	1	2	1

有了計數陣列 count，即可用陣列 temp 來存放完成目前排序位數排序的結果—我們累加 count 陣列內的值（或說：執行「前綴和」），可取得各數字在 temp 中存放位置的註標。請看下面圖示：

count	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	0	1	0	0	3	1	2	1	2	1
+					...					
index	0	0	1	1	1	4	5	7	8	10

表示個位數字為 0（1）者可自陣列 temp[0]處開始存放

個位數字為 2 者可自陣列 temp[1]處開始存放

...

個位數字為 5 者可自陣列 temp[4]處開始存放

個位數字為 6 者可自陣列 temp[5]處開始存放

因此根據 index 陣列，我們可將原來的資料依 index 內的位置註標存放到 temp 陣列中，（注意每參照一次 index[i]內的位置後，index[i]應自行加 1）。例如第一個數字 224 的個位數字爲 4，count[4]爲 1，遂 224 應放入 temp[1]中，count[4]應自行加 1；第二個數字 454，個位數字爲 4，count[4]已更新爲 2，遂放入 temp[2]中。所有完成個位數字排序後的數字依序放入後，temp 陣列即爲：

temp	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	321	224	454	524	735	496	456	7	98	578	339

與之前用 10 個佇列的結果完全一樣；由於可知用陣列結構來模擬這 10 個佇列是可行的。我們再看對上列 temp 陣列內的資料，進行十位數字依序計數(或執行前綴和），可得：

count	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	1	0	3	2	0	2	0	1	0	2
	+ ↑	↘ ↑	↘ ↑	↘	...			↑	↘ ↑	↘
index	0	1	1	4	6	6	8	8	9	9

那麼對十位數字進行排序的結果，應得新 temp 陣列如下：

temp	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	7	321	224	524	735	339	454	456	578	496	98

再對上列 temp 陣列內的資料，進行十位數字依序計數（記錄百位數各個數字出現的個數、執行前綴和），可得：

count	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	2	0	1	2	3	2	0	1	0	0
	+ ↑	↘ ↑	↘ ↑	↘	...			↑	↘ ↑	↘
index	0	2	2	3	5	8	10	10	11	11

那麼對百位數字進行排序的結果，應得 temp 陣列如下：

temp	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	7	98	224	321	339	454	456	496	524	578	735

這也就是整個數列排序的結果。

根據上面的討論，我們寫出基數排序的演算法：

演算法 7-10　基數排序

輸入：整數陣列 data，共 n 筆資料(data[0]~data[n-1])

輸出：排序陣列 data，若 i<j，則 data[i]≤data[j]，0≤i,j≤n-1

```
1  q = 0;
2  for (i=1; i<n; i++)
3      if (data[i]>q) q = data[i];
4  radix = ⌈log10(q+1)⌉  ;   //radix 是 q 所擁有的位數
5  for (i=1; i<=radix; i++)
6  {   for (j=0; j<10; j++) count[j]=0;
7      for (j=0; j<10; j++)
8      {   digit = data[j]的第 i 位數字;
9          count[digit]++;
10     }
11     index[0] = 0;
12     for (j=1; j<10; j++) index[j] = index[j-1]+count[j-1];
13     for (j=0; j<n; j++)
14     {   digit = data[j]的第 i 位數字;
15         temp[index[digit]++] = data[j];
16     }
17     for (j=0; j<n; j++) data[j] = temp[j];
18 }
```

第 1~4 行決定了數列中最大者 q 的位數爲 radix。6~10 行的迴圈在計算 data 陣列中所有資料的第 i 位數字出現的個數；第 11、12 行在累加 count 陣列至 index 陣列中；第 13~16 行的迴圈則依照第 i 位數字的排列結果，重排 data 陣列的資料至 temp 陣列中（在此也可用上節介紹的計數排序法，來完成各 i 位數字的排序）。第 17 行將陣列 temp 的資料放回 data 陣列中；第 5~18 行的迴圈會執行 radix 次—而每次迴圈包含了數個小迴圈，每個小迴圈都可在 $O(n)$ 的時間內完成。

若輸入的資料最大者為 k 位數，則排序甚至需要 $O(kn)$ 的時間。注意這裡所謂的 k 位數數字其實和 n 是有關的：在電腦中若此 n 個數字分布均勻在正整數 $1\sim n$ 間，最大數即為 n，則 $k = O(\log n)$（或者可解釋成：$\log_2 n$ 個位元，可表示的最大數字為 $2^{\log_2 n} = n$）。所以基數排序的時間複雜度為 $O(n\log n)$。

請注意：在基數排序演算法中，並沒有用到「比較大小」的指令！而且我們用了額外的陣列 `count` 和 `temp`（皆為 $O(n)$），來存放必要的資訊，資料搬移的量頗為可觀。至於相同值的資料，在基數排序中的相對順序不致改變，遂具有穩定性。當資料的位數不大且相近（例如：學號、身份証字號、車牌號碼、發票號碼、…等等），也沒有記憶空間的考量時，以基數排序法進行排序，效率會很好！

請與計數排序法做個比較：計數排序僅需一個回合，但需要 $O(n+q)$ 的空間；基數排序需要 `radix` 個回合，需要 $O(n)$ 的空間；其中 q 是所有資料中最大者，`radix` 為 q 所擁有的位數。兩者皆不須「比較大小」，皆利用空間換取時間。

7.2.9 堆積排序法

在 5.8 節中我們曾介紹過堆積，在最小堆積此資料結構中，父節點的值比其子節點值要小；善加利用這個性質，即可進行排序的運算。

在 5.8.2 節已提及在最大堆積中刪除最大資料後的必要調整動作；在此我們沿用一樣的概念，配合由小到大排序的需求，用最小堆積來執行排序的動作。

堆積排序法 (heap sort) 可分成兩主要步驟討論：

(1) 建立一最小堆積；

(2) 輸出最小元素，更新此最小堆積；

且只須逐次執行步驟 (2) ，即可得到由小到大排序的數列。

資料的儲存亦比照 5.8 節利用陣列 data 儲存之：欲排序的 n 筆資料，放在 data[1]~data[n] 中。注意：最小堆積的邏輯圖示，事實上可對應一完備二元樹（請見 5.8 節）；由 5.8.2 節的經驗，我們以圖 7-13 描繪堆積排序法第 (2) 步驟：傳出最小元素，更新最小堆積的過程：

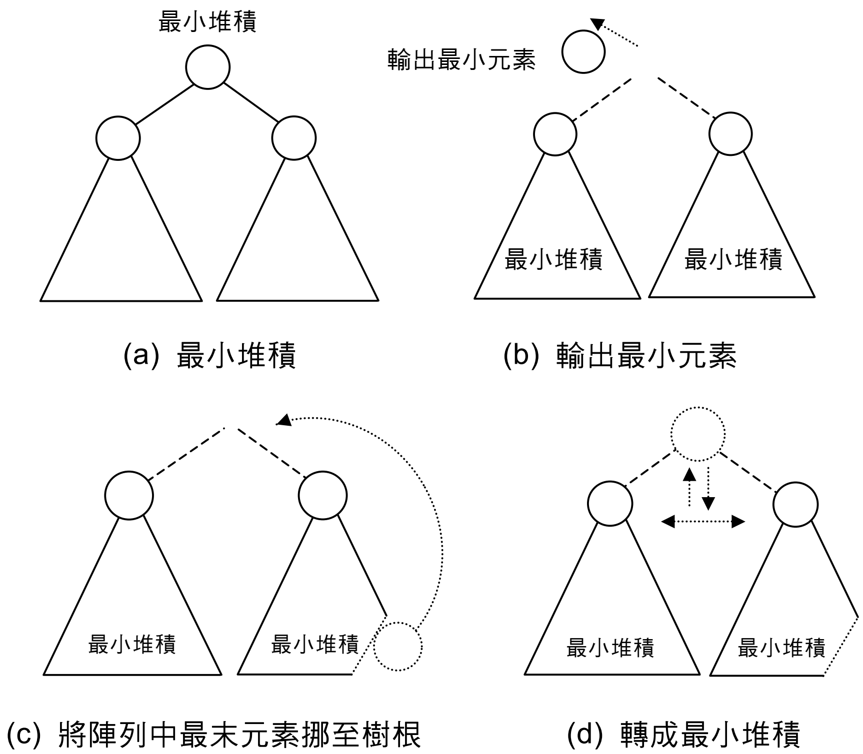


圖 7-13 傳出最小元素，更新最小堆積

其中輸出最小元素亦即樹根（data[1]）的資料後（圖 7-13 (a) 和 (b)），原樹根的兩棵子樹仍為最小堆積；我們可將位於最右下角的樹葉內元素（陣列的最後元素），挪放至樹根處（data[1]），如圖 7-13 (c)。當然這不是個最小堆積；再將此非最小堆積結構，轉換成新的最小堆積，我們把這個轉換的動作（圖 7-13 (d)），寫成下面稱為 restore 的程序。

演算法 7-11　轉換成最小堆積 (restore)

輸入：整數陣列 data[s],data[s+1],...,data[t]，其中以 data[2s]和 data[2s+1]為樹根的兩子樹分別皆為最小堆積

輸出：data[s],data[s+1],...,data[t]形成最小堆積

```
1  void restore (int s, int t)
2  {   int i=s, j;
3      while(i<=t/2)
4      {   if (data[2*i] < data[2*i+1])
5              j = 2*i;
6          else
7              j=2*i+1;
8          if (data[i] < data[j]) break;
9          swap(&data[i], &data[j]);
10         i = j;
11     }
12 }
```

演算法 7-11 的設計頗具彈性：輸入是整數陣列 data[s]~data[t]，它們對應一棵以 data[s]爲樹根的完備二元樹，其中以 data[2s]和 data[2s+1]爲樹根的兩子樹分別爲最小堆積，但以 data[s]爲樹根的樹卻不是！輸出是將 data[s]~data[t]重組成最小堆積。第 2~11 行以父親註標 i (=s)開始，在其不爲樹葉時（i<=t/2）進入迴圈（3~11 行）：挑出左右兒子中較大者（註標爲 j），若 data[i]<data[j]，可離開迴圈（第 8 行；因爲已滿足最小堆積性質）；否則兩者交換（第 9 行），自兒子處繼續 restore（第 10 行）。事實上，這個 restore 程序把兩個堆積—但加上父親卻未必是堆積—的兄弟子樹重建，使以其父親爲樹根的整棵樹就是堆積（其重建後的兩個兄弟子樹當然也是）！它就是建立堆積、進行堆積排序的核心（在演算法 7-12 中介紹）。

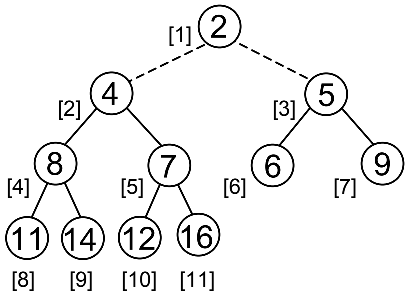
範例 7-10 先舉例說明 restore 的運算過程。

範例 7-10

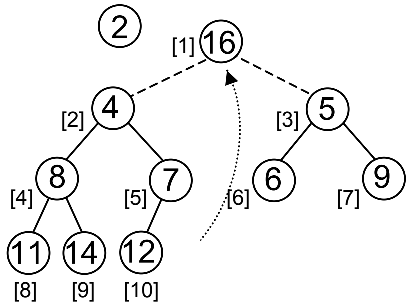
圖 7-14 提供了一個例子，其中共有 10 個數字，已存在陣列 data[1] ~data[11]（如圖 7-14 (a)）中。其對應的最小堆積邏輯圖示（一棵二元完備樹）如圖 7-14 (b) 所示。此時樹根內存放的值乃所有資料最小者，倘若有必要輸出之（例如：排序、選最小邊、選優先順序最高者、…），吾人可用 restore 使剩下的資料仍回復爲最小堆積。將最右樹葉的值 16 移至樹根，如圖 7-14 (c)；這時此二元完備樹已不是最小堆積的結構（我們在圖中以虛線分支代表其上節點非最小堆積，實線分支則依然是最小堆積）。要將之調整成最小堆積，應把兒子中若有比父親小者，取兒子中較小者和父親交換；請看圖 7-14 (d)、(e)、(f) 的描繪。經過調整爲的最小堆積，則如圖 7-14 (g) 所示。

data	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
	2	4	5	7	8	6	9	11	14	12	16

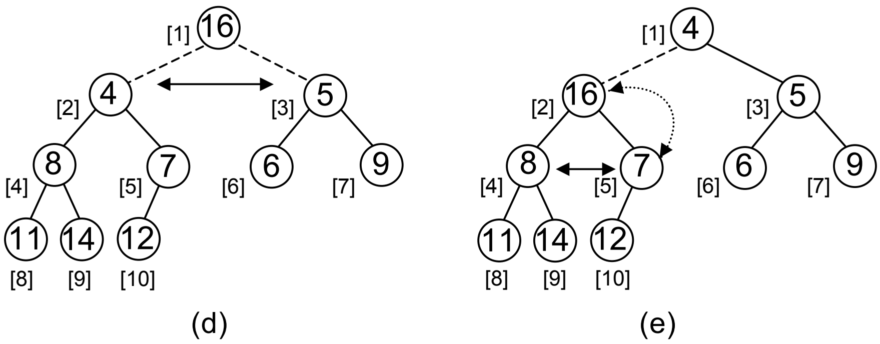
(a) 最小堆積（以陣列存放）



(b) 最小堆積的邏輯圖示



(c) 輸出 2，挪最右樹葉 16 至樹根



（16 的兒子中若有比 16 小者，取兒子中較小者和 16 交換）

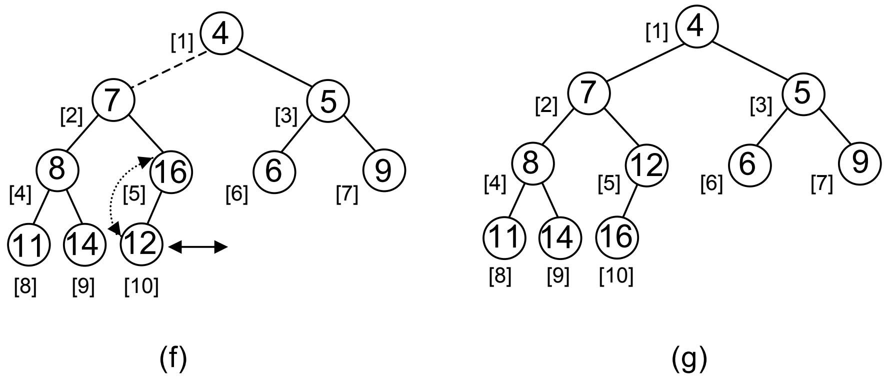
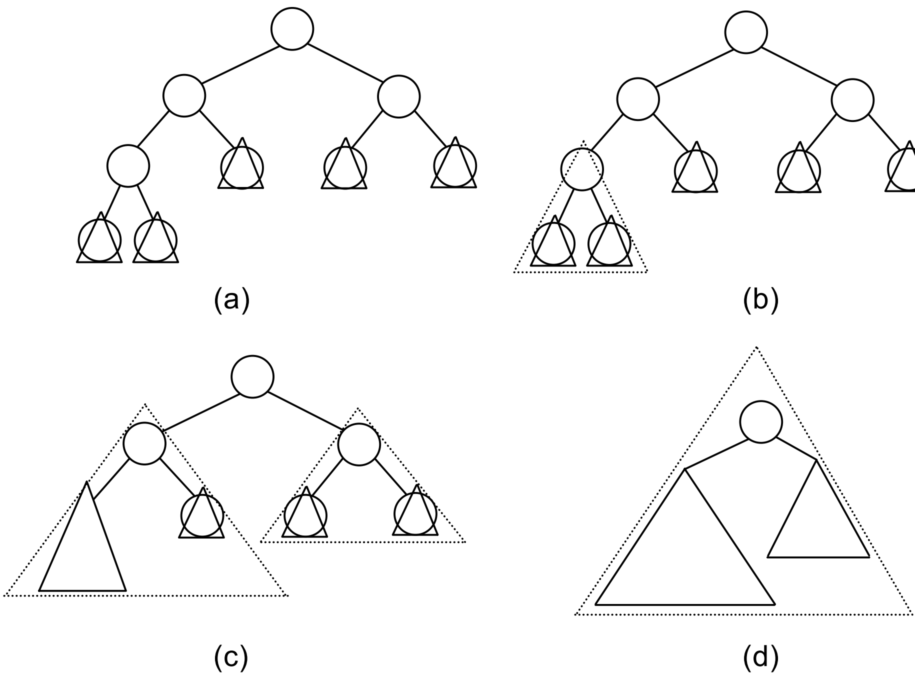


圖 7-14　最小堆積的更新 (`restore` 程序執行的過程)

有了 `restore` 此程序，則堆積排序法中主要步驟 (1)「建立最小堆積」即可引用 `restore` 來完成。基本上 `restore` 程序可將如圖 7-13 中「兩子樹分別是最小堆積，但加上其父親卻未必是最小堆積」這種結構，轉換成最小堆積！因爲樹葉節點本身當然可視爲最小堆積，如圖 7-15 (a) 所示，可利用 `restore` 將「兩兄弟樹葉節點與其父親之未必是最小堆積結構」轉成最小堆積。因此自最後一個中間節點開始，由右往左、逐層往上（樹根處）轉「未必是最小堆積的結構」成爲最小堆積，即可建立所有資料的最小堆積，見圖 7-15 (c)、(d)。圖中實線三角形表示最小堆積；虛線三角形表示非最小堆積。

若最後一片樹葉在 `data[n]` 上，則其父親在 `data[n/2]` 處—此註標恰爲最後一個中間節點之所在（如圖 7-15 (a)、(b)），由它開始自右向左（陣列註

標自 n/2 倒數至 1）逐一執行 restore，即可建立出最小堆積。換句話說：只需對陣列註標 i=n/2,n/2-1,...,1 分別執行 restore(i)，即可將 data[1]~data[n]建出最小堆積。



實線三角形表示最小堆積；虛線三角形表示非最小堆積

圖 7-15 利用 restore 來建立最小堆積

於是整個堆積排序的演算式可撰寫如下：

演算法 7-12 堆積排序

輸入：整數陣列 data，共 n 筆資料(data[1]~data[n])

輸出：排序陣列 data，若 i<j，則 data[i]≤data[j]，1≤i,j≤n

```
for (i=n/2; i>=1; i--)
     restore(i, n);
for (i=n; i>=1; i--)
{    輸出 data[1];
```

```
5         data[1] = data[i];
6         restore(1, i-1);
7     }
```

restore 程序執行的最差情況爲：將在樹根的資料一路往下挪動到樹葉處。這條路徑的長度，恰爲此完備二元樹的高度，即 $O(\log n)$，n 爲樹上的所有節點數（欲排序的資料數目）。

堆積排序法中第 1~2 行旨在建立最小堆積，可在 $O(n)$ 的時間內完成，此結果十分重要，稍後分析之。第 3~7 行逐次輸出當時樹根裏的最小元素，再把當時最右樹葉的元素 (data[i]) 放入樹根，執行 restore(1, i-1)，總計 n 次，需要時間爲 $O(n\log n)$。於是整個堆積排序所需要時間爲 $O(n+n\log n) = O(n\log n)$。堆積排序無須額外的記憶空間，但不具穩定性。

我們解說堆積排序法中第 1~2 行的時間複雜度：令 n 個資料的最小堆積以完備二元樹表示時如圖 7-16，高度爲 d $(=O(\log n))$；呼叫 restore(i, n)（第 2 行）時，i 恰在圖中階層 l 處，則執行 restore 時的最差情況是：註標 i 的資料由階層 l 調整到階層 d 處，時間爲 $c(d-l)$，c 爲一常數(演算法 7-11 中 4~9 行）。

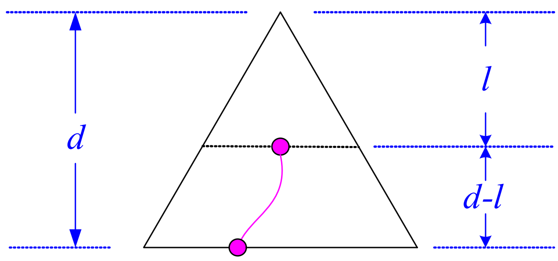


圖 7-16　利用 restore 來建立最小堆積

在階層 l 上的點共爲 2^l 個，爲方便計算，樹根第一層定爲階層 0，於是 $l = 0, 1, 2, \ldots, d-1$。演算法 7-12 中第 1~2 行總共執行步驟爲：

$$\sum_{l=0}^{d-1} c(d-l)2^l = c\sum_{l=0}^{d-1}(d2^l - l2^l) = c(d\sum_{l=0}^{d-1}2^l - \sum_{l=0}^{d-1} l2^l)$$

$$= c(d(2^d-1)-((d-2)2^d+2))$$

$$= c(d2^d-d-(d-2)2^d-2)$$

$$= c(2\times 2^d-d-2))$$

$$\leq c(c_1n-c_2\log n-c_3)$$

$$= O(n)$$

由以上分析得知：建立最小堆積，可在 $O(n)$ 的時間內完成。各位應記得在用 Kruskal 演算法求最小成本延展樹的實驗中，利用最小堆積取得最小權重邊，即比用線性搜尋取最小權重邊、或所有邊排序，再一一取最小邊，都要有效率得多。

7.3 排序究竟可以有多快

在 7.2 節中我們討論了許多排序的演算法，它們的時間複雜度有的是 $O(n^2)$，有的是 $O(n\log n)$；我們自然好奇：可不可能有更快的排序演算法呢？在這一節中，我們要看看排序究竟可以快到什麼地步？可能不斷地快嗎？抑或有某個限制？

大抵而言，我們得靠比較大小和資料交換來完成排序的動作[4]。而比較大小和資料交換的關係，可以用一棵「決策二元樹」來表示之。範例 7-11 的圖 7-17 是一個 $n = 3$ 的大小比較決策二元樹，它恰可描述 3 個資料排序的所有可能。

4　7.2.7 和 7.2.8 節提到的計數和基數排序法沒用到比較大小，它用的是額外的儲存空間和資料搬移。本節的討論即排除這類排序方法。

範例 7-11

令三個數字 A、B、C 要進行排序，則圖 7-17 的決策二元樹，模擬了三個數字的所有大小關係。

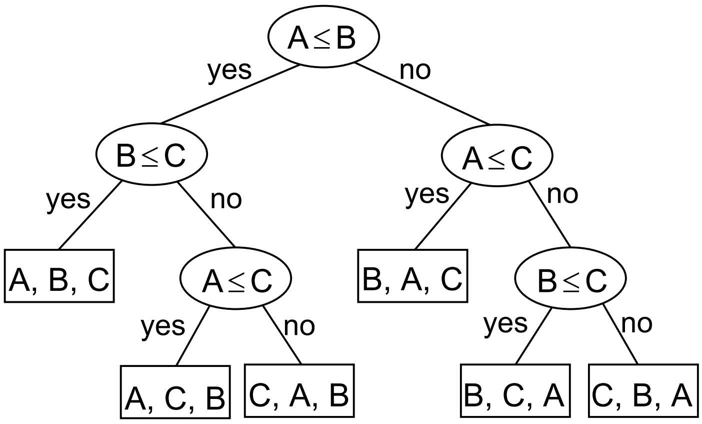


圖 7-17　三個數字排序對應的決策樹

此決策樹中樹葉即為數字的某種大小順序關係，中間節點皆是比較大小的運算。決策樹上自樹根到樹葉的路徑，即為決定該大小順序關係必須花的大小比較運算；而該樹葉節點內的數字順序，即此路徑對應之數字由小至大的排序結果。所有樹葉節點的個數，正是排序資料所有可能的排序組合。

凡是根植於比較大小和資料交換的排序演算法，應可以用一棵決策二元樹，將其演算過程描繪出來；正如圖 7-17 是棵決策二元樹，將其中間節點的運算撰寫成程式，即可推出其對應的排序演算法—實為三個資料的插入排序法也—如演算法 7-13。各位可嘗試設計其它決策樹，而該決策樹可對應出某種排序演算法。

演算法 7-13 三個資料的插入排序

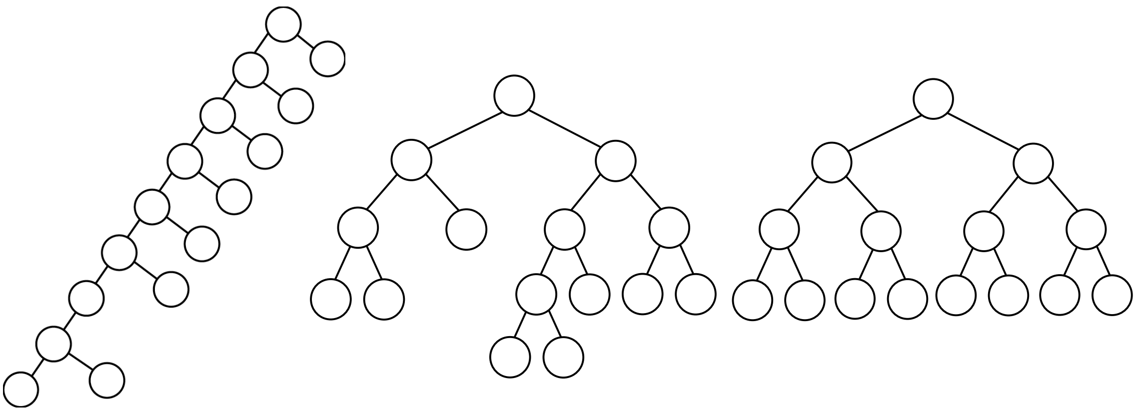
輸入：A[0], A[1], A[2]

輸出：A[0], A[1], A[2]; A[0]≤A[1]≤A[2]

```
a = A[0]; b = A[1]; c = A[2];
if (a<=b)
    if (y<=z) A[0] = a; A[1] = b; A[2] = c;
    else if (x<=z) A[0] = a; A[1] = c; A[2] = b;
          else A[0] = c; A[1] = a; A[2] = b;
else
    if (a<=c) A[0] = b; A[1] = a; A[2] = c;
    else if (b<=c) A[0] = b; A[1] = c; A[2] = a;
          else A[0] = c; A[1] = b; A[2] = a;
//  A[0]≤A[1]≤A[2]
```

決策二元樹上有多條路徑自樹根通往樹葉，每一條路徑會對應一種輸入的排序過程；其中自樹根通往樹葉的最長路徑（也就是此樹的深度），就應是該對應演算法的最差執行時間。此時欲求排序所需的最短時間，相當於找各種決策樹中擁有深度最短者。

圖 7-18 舉出了幾種不同的比較大小決策樹，其樹葉個數皆為 8。而圖 7-18 (a) 的深度為 9、(b) 的深度為 5、(c) 的深度為 4。



(a)　(b)　(c)

圖 7-18　不同的比較大小決策樹

由此可知：當樹葉節點一樣時，決策二元樹愈呈完備，其深度愈小。當樹葉有 x 個時，完備決策樹的深度爲 $\lceil \log_2 x \rceil$。當排序的資料有 n 筆時，其可能的大小順序排列組合爲 n！；這些排列組合的情形，皆可是決策二元樹上的樹葉節點。於是最快的排序方法，應有最短的決策二元樹深度，即 $\lceil \log_2 n! \rceil$。

因爲

$$n! = n\times(n-1)\times(n-2)\times \ldots \times 2\times 1 \geq (n/2)^{n/2}$$

所以

$$\lceil \log_2 n! \rceil \geq \log_2(n/2)^{n/2} = (n/2)\times\log_2(n/2)$$

$$= \Omega(n\log n)。$$

在此我們引用了 1.5.2 節中定義的 Ω 符號來表示下界 (lower bound)，亦即最快的排序方法也要花 $\lceil \log_2 n! \rceil$ 的時間，相當於至少要 $\Omega(n\log n)$ 的時間。

注意我們證明了排序方法至少要 $\Omega(n\log n)$ 的時間，亦即不可能比 $\Omega(n\log n)$ 還快。在此我們沒有提到任何一個演算法，只提及：凡是以比較大小以及資料交換爲基本運算的排序演算法，皆可對應到一棵決策樹；而透過決策樹的性質，決定出任何排序演算法都至少得花 $\Omega(n\log n)$ 的時間–不可能存在決策樹，其最短的高比 $\Omega(n\log n)$ 還低。所以 $\Omega(n\log n)$ 提供了超然於演算法之上的排序時間下界值。

在之前的排序演算法中，堆積排序的時間複雜度就是 $O(n\log n)$，它提供了排序「至多」需要的時間；然而決策樹的性質推導，提供了排序「至少」需要的時間 $\Omega(n\log n)$。我們可以說：堆積排序的 $O(n\log n)$，爲排序找到一個所需時間的上界，而透過決策樹導出的 $\Omega(n\log n)$，爲排序找到一個所需時間的下界。此上、下界恰好相同，於是堆積排序是一種排序的最佳 (optimal) 演

算法。

我們在此對「最佳演算法」給予正式的定義[5]。

定義：一個問題 P 的最佳演算法 A 存在，如果解決 P 的下界等於解決 P 的演算法 A 的上界。

5 建議讀者參閱 1.5.2 節的說明，將演算法上界、問題下界、最佳演算法…等觀念，心領神會，融會貫通。

本章習題

1. 請比較下列排序演算法的異同：

    (a) 挑選排序、氣泡排序和插入排序。

    (b) Shell 排序和合併排序。

2. 請實作本章所提的所有排序演算法。請以亂數產生欲排序的資料，記錄不同資料量 (n) 下，各演算法執行排序所需的時間；做出圖表(n 值為 x 軸，時間為 y 軸)，觀察並討論不同演算法在不同資料量下執行排序的表現。

3. 請說明本章所有排序演算法在輸入數列以下面兩種排序時的表現：

    (a) 排序完成；

    (b) 反向排序完成

4. 請實作不同版本的基數排序法：

    (a) 以 10 佇列存放各個位數排序的結果；

    (b) 用陣列來模擬 (a)。

    並利用題 2 的實驗設計製作圖表，觀察並討論實驗結果。

5. 請實作不同版本的快速排序法：

    (a) 遞迴版；

    (b) 利用堆疊和以迴圈控制版；

(c)　當 `left~right` 間的資料量少於 10 時，直接用挑選或排序；不必再遞迴呼叫（或 push 入堆疊）；

(d)　取一個 0~n−1 的亂數 `i`，令 `target` 爲 `data[i]`；

(e)　令 `target` 爲 `data[0]`、`data[n/2]`和 `data[n-1]`的中位數。

並利用題 2 的實驗設計製作圖表，觀察並討論實驗結果。

6.　請實作不同版本的 Shell 排序法：

(a)　以 1, 4, 13, 40, 121, …爲遞增分組數列；

(b)　以 1, 2, 4, 8, 16, …爲遞增分組數列；

(c)　自行選定遞增分組數列。

7.　請實作不同版本的堆積排序法：

(a)　用陣列實作最小（大）堆積；

(b)　用鍵結串列實作最小（大）堆積。

並利用題 2 的實驗設計製作圖表，觀察並討論實驗結果。

8.　(a)　證明當輸入串列已經排好順序時，快速排序需要 $O(n^2)$ 的時間。

(b)　證明快速排序最差情況時的複雜度爲 $O(n^2)$。

(c)　爲何演算法 7-7 中第 `3` 行的 `left < right` 檢測，在快速排序中是必要的？

9.　證明當較小的子串列先排序時，那麼快速排序中的遞迴，可以用深度爲 $O(1\text{og}n)$ 的堆疊來模擬。

10. 快速排序不是一個穩定的排序法，舉出一個輸入串列的例子說明，其中具有相同鍵值的數個記錄之先後順序不再相同。

11. 請舉例說明基數排序的最差情形。

12. 請說明本章所有排序演算法的最差情形。

13. 設計一個程序，將數列 $(x_1, x_2, \ldots, x_n)$ 向右環形移動 p 個位置，其中 $0 \le p \le n$。此程序應有 $O(n)$ 的時間複雜度，而其空間複雜度應爲 $O(1)$。

14. 如果有 p 部電腦供各位使用，請討論排序在此 p 部電腦中同時執行的可能演算法。