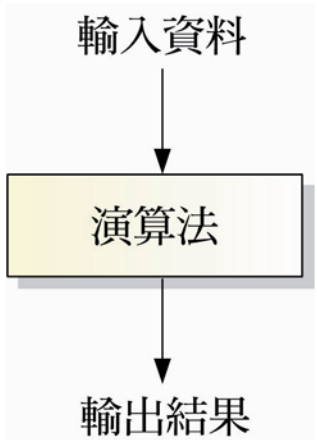
資訊科技概論

第 9 章 演算法

9-1　演算法的概念

- 演算法 (algorithm) 是電腦科學中最重要的基礎觀念
- 演算法的廣義定義：用來解決某個問題或進行某件工作的一連串步驟
  - 食譜：煮出某道餐點的演算法
  - 摺紙
  - 樂譜：是產生音樂旋律的演算法

- 電腦科學方面的演算法：以應用電腦來解決問題的思考方式所想出來的步驟
- 通常是接受一些輸入資料，經過演算法的步驟處理後，產生出預期的輸出資料
- 爲了要在電腦上執行，演算法必須先以某種程式語言 (programming language) 撰寫成程式 (program)，這個過程就是所謂的程式設計 (programming)



電腦演算法的定義

- 電腦演算法都必須符合以下五個條件：
  - 輸入 (Input)
  - 輸出 (Output)
  - 定義必須明確 (Definiteness)
  - 個數有限的步驟 (Finiteness)
  - 有效率 (Effectiveness)
- 電腦演算法的正式定義：

  演算法是一組明確而有次序關係，可以產生結果並會在有限時間內完成的步驟集合

9-2 演算法的表示法

- 演算法的表示法有許多種，並沒有所謂標準的表示法，完全看個人的喜好和思考方式而定。

- 演算法的表示法通常分爲以下幾種：
  - 條列式的步驟
  - 流程圖
  - 虛擬碼
  - 程式碼

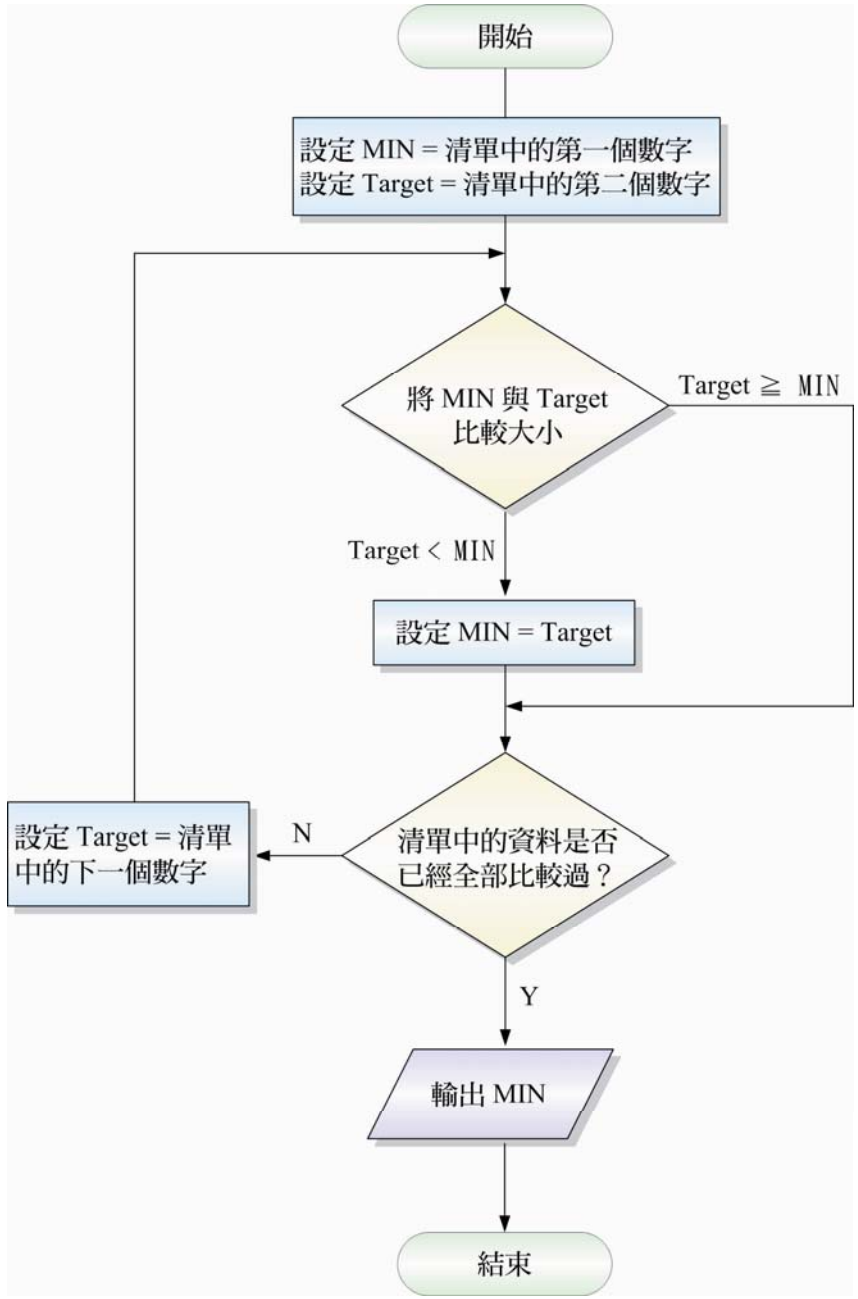
條列式的步驟

- 以一條條敘述的方式來描述問題的解決步驟，通常是以人類的自然語言 (中文、英文等) 所寫成

- 範例：假設要在清單中找出數字最小者，用條列式步驟撰寫的演算法可能是這樣：
  - 第一步：讀入整串數字
  - 第二步：從第一個數字開始，假設最小數MIN爲第一個數字
  - 第三步：將下一個數字與MIN相比較，假如下一個數字比較小，就把MIN改設成它的值。如果相等或較大，則不做任何動作
  - 第四步：繼續讀取下一個數字來與MIN比較，以此類推。直到全部的數字都比較過爲止
  - 第五步：輸出結果就是MIN的值，也是整個數列的最小值

流程圖

- 流程圖 (flowchart) 是演算法的圖形符號表示法，它將演算法中所有的細節隱藏起來，而是以整體宏觀的角度來展現問題的解決流程
- 如圖書館借書流程，銀行的貸款流程



虛擬碼

- 虛擬碼 (pseudo code) 是混合了程式語言與一般口語敘述的表示法
- 虛擬碼沒有固定的語法，通常會使用將來要用來撰寫正式程式碼的程式語言的語法
- 優點是比正式的程式碼容易看懂，而且在表達時思路比較順暢
- 缺點是不夠準確，日後還是須以正規的程式語言改寫

```
MinValue (輸入：一串整數值)
BEGIN
  MIN = 清單中的第一個數字
  Target = 清單中的第二個數字
  LastItem條件爲假
  WHILE (LastItem 條件爲假) DO
  BEGIN
    IF (Target < MIN) THEN
      MIN = Target 的值
    ENDIF
    IF (Target 不是最後一個數字) THEN
      Target = 清單中的下一個數字
    ELSE
      LastItem條件爲真
    ENDIF
  ENDWHILE
  輸出：MIN
END
```

程式碼

- 直接以某種程式語言來撰寫
- 範例：C語言


```
int min_value(int list[], int n)
{
  int  min, i;
  min = list[0];                /* 設定min爲陣列中的第一個元素 */
  for (i = 1; i < n; i++){
    /* 假如陣列中的元素比min的值小，將min設定成這個新的值 */
    if (list[i] < min) min = list[i];
  }
  return(min);      /* 全部檢查完畢，此時的min爲最小值 */
}
```

9-3 演算法的組成元件

演算法最主要的結構：

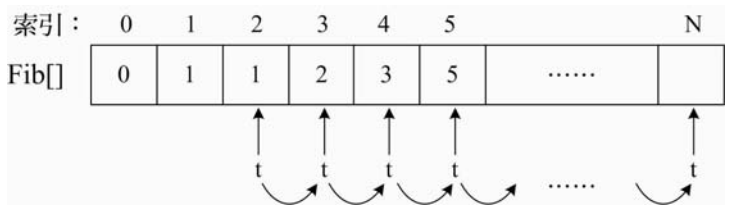
- 條列式 (sequence) 結構：由一連串的指令所組成，這些指令可以是簡單的指派動作，或是另外兩種結構的指令
- 決策式 (decision) 結構：先測試某個條件的值，如果條件符合就進行某組指令，而條件不合時可能會執行另一組指令，通常是使用IF-THEN-ELSE指令來表達
- 重覆式 (repetition) 結構：在符合條件時重複執行某組指令數次，通常是使用WHILE-DO指令來表達

9-4 如何構思演算法

- 開發演算法的過程最具挑戰性的部份在於如何想出解決問題的方法
- 1945年數學家G.Polya歸納出解決數學問題的四個階段，可以套用成爲發展電腦演算法的理論基礎 ：
  - 階段一：深入了解問題
  - 階段二：找出解決問題的方法
  - 階段三：構思演算法的詳細步驟，並轉換爲程式，記得要符合演算法的五大條件
  - 階段四：評估此演算法的結果是否正確，以及這個演算法是否可以應用在其他性質相近的問題上

- 範例：構思出計算費式數列 Fib(N) 值的演算法
- 盡量找出規律性，也就是不管N是多少，都能夠用固定的幾條指令來完成計算

```
int fib_numbers(int Fib[], int N)
{
  int t;
  Fib[0] = 0;
  Fib[1] = 1;
  t = 2;
  /* 重覆迴圈主體一直計算到 t = N 爲止 */
  while (t <= N)
  {
    Fib[t] = Fib[t-1] + Fib[t-2];
    t = t + 1;
  }
  return(Fib[N]);
}
```



9-5 重覆執行的演算法結構

- 電腦擅長處理重複性高的運算或動作
- 大多數的演算法都會牽涉到重複的運算
- 在構思演算法時應盡量把問題歸納出規則，找出可以用重複結構來撰寫的解法
- 重覆執行的演算法結構有兩種：
  - 反覆 (iteration) 結構
  - 遞迴 (recursive) 結構

反覆結構

- 指一群指令以迴圈 (loop) 方式重複的被執行
- 除了指定迴圈本體內應該要執行哪些動作 外，更重要的是要設定迴圈的控制部份
- 迴圈的控制部份包括了三個要素：
  - 設定初值 (initialize)
  - 測試 (test)
  - 修改 (modify)
- 範例：計算N階層的 (N!) 演算法，N 階層公式的定義如下，這種定義是適合以反覆結構的演算法來撰寫：

$$\text{Factorial}(N) = \begin{cases} 1 & \text{若} N = 0 \\ N * (N-1) * (N-2) * \ldots * 2 * 1 & \text{若} N > 0 \end{cases}$$

- 若以反覆結構的方式來撰寫 (如while敘述)，結果如下：

```
int factorial(int n)
{
  int  answer;
  answer = 1;
  if (n == 0)  return (answer);
  /* 控制部份的初值設定是輸入參數 */
  while (n >= 1)                  /* 控制部份的測試動作 */
  {
    answer = answer * n;          /* 反覆結構的定義 */
    n = n - 1;                    /* 控制部份的修改動作 */
  }
  return (answer);
}
```

遞迴結構

- 當問題的特質是切割成多個小範圍的問題個別解決會比解決一個大問題來得容易時，這種解決問題的方式叫做「Divide and Conquer」
- 而對應的演算法撰寫方式就是所謂的遞迴
- 在電腦領域方面適合使用遞迴方式來解決的問題，其條件是分割後的小問題其特性和解法必須與原來的大問題相同才行
- 直接遞迴：函數呼叫它自己本身
- 間接遞迴：函數先呼叫其他函數，而其他函數又回頭來呼叫原始函數

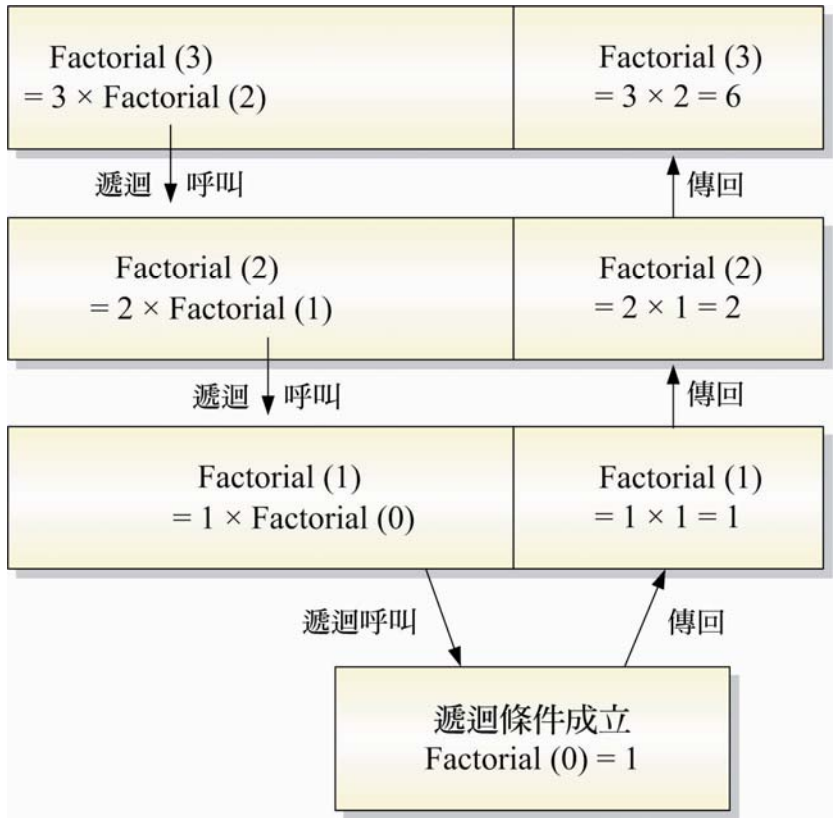
- N階層範例若改用遞迴方式來定義如下：

$$\text{Factorial}(N) = \begin{cases} 1 & \text{若} N = 0 \\ N * \text{Factorial}(N-1) & \text{若} N > 0 \end{cases}$$

- 而改用遞迴結構的方式來撰寫，結果如下：

```
int factorial(int n)
{
  if (n < 1)
    return (1);                    /* 遞迴的結束條件成立 */
  else
    return (n * factorial(n-1))            /* 遞迴定義 */
}
```

- 遞迴程式碼需要有結束條件，若符合程式就會直接計算輸出而不再呼叫它自己本身
- 遞迴結構的呼叫方式是一層層的巢狀結構，等到最內層的遞迴條件成立後，再回到上一層繼續往下執行，以此類推，直到回到最上層爲止
- 範例：計算factorial(3)



9-6 常見的演算法

- 演算法領域裡最常用到的演算法如以下幾類：
  - 找出最大值或最小值
  - 連加
  - 連乘
  - 求平均值
  - 搜尋 (搜尋符合條件的某一筆或數筆資料)
  - 排序 (將一串資料依照規定的順序重新安排位置)

- 當資料量很大的時候，不同的搜尋和排序演算法所花費的時間將大不相同

9-7 搜尋

- 搜尋技術的目的在於可以在多個資料中找出符合特定條件的資料
- 知名的搜尋演算法有：
  - 循序搜尋 (sequential search)
  - 二元搜尋 (binary search)
  - 二元樹搜尋 (binary tree search)
  - AVL樹搜尋 (AVL tree search)
  - 2-3樹搜尋 (2-3 tree search)
  - 紅黑樹搜尋 (red-black tree search)

循序搜尋

- 又稱線性搜尋 (linear search)
- 原理是在陣列內找到值等於目標值target的資料，並傳回其索引，若找不到符合條件的資料，則傳回 -1
- 範例：用C語言撰寫數，參數有三個：
  - 要搜尋的資料 (存放於陣列，整數型態)
  - 資料個數
  - 目標值

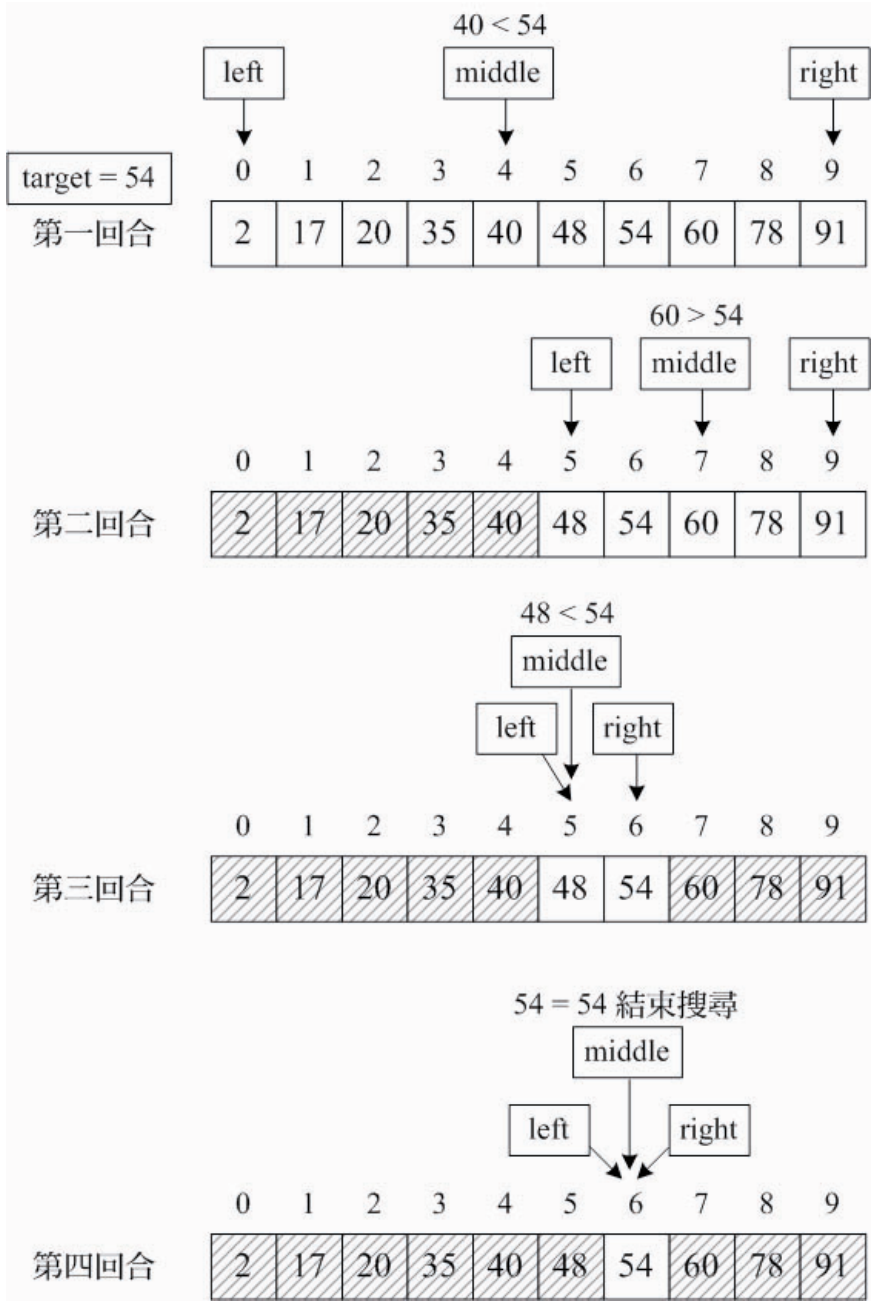
```
int sequential_search(int list[], int n, int target)
{
  int i, index;
  index = -1;   /* index表示符合條件之資料的索引，
                   初始值爲 -1 */
  for (i = 0; i < n; i++){
    if (list[i] == target){          /* 比對陣列內的資料是否等於
    搜尋的目標值 */
      index = i;                              /* 若找到符合條件的資
    料，就將其索引指派給index */
      break;                                  /* 強制離開for迴圈 */
    }
  }
  return(index);                     /* 傳回找到的索引，若爲 -1，
                                     表示找不到 */
}
```

- 範例：list[] = {54, 2, 40, 22, 17, 22, 60, 35}
  - 搜尋的目標爲22
  - 呼叫函數sequential_search(list, 8, 22)
  - 結果傳回值爲3
- 通常用在資料清單並未排序時，而且只適用於小型的資料清單，因爲循序搜尋非常慢

二元搜尋

- 假設要搜尋的資料已經事先排序完畢
- 現在假設是由小到大排序，搜尋時目標值和位於中間的資料做比較：
  - 若目標值比較大，表示可能符合條件的資料是位於陣列的中間到後面，因此搜尋的範圍已經縮小一半。
  - 若目標值比較小，表示可能符合條件的資料是位於陣列的前面到中間，因此搜尋的範圍也是縮小一半。
  - 如果目標值剛好等於中間的資料值，則傳回這個資料值的索引
- 以相同方式在前一回合只剩一半的可能範圍內進行二元搜尋，直到找出等於目標值的資料，然後傳回其索引
- 若找不到等於目標值的資料，則傳回 -1

- 第一回合：範圍是0到9，中間的索引值爲4，因此將list[4] (其值爲40) 與目標值54比較
- 因爲list[4] < 54，因此第二回合的搜尋範圍是索引值5到9的部份
- 第二回合搜尋範圍中間的索引值爲 (5 + 9) / 2 = 7，因此將list[7] (其值爲60) 與目標值54比較
- 因爲list[7] > 54，因此第三回合的搜尋範圍是索引值5到6的部份
- 其餘以此類推



```
/* 此巨集會比較x、y，若x < y，傳回 -1；若x == y，傳回0；
若 x > y，傳回1 */
#define COMPARE(x, y) (((x) < (y)) ? -1: ((x) == (y)) ? 0: 1)

int binary_search(int list[], int left, int right, int target)
{
  int middle;
if (left <= right){
    middle = (left + right) / 2;         /* middle爲陣列中間的索引 */
    switch (COMPARE(list[middle], target)){
      case -1:
        return binary_search(list, middle + 1, right, target);
      case 0:
        return middle;
      case 1:
        return binary_search(list, left, middle - 1, target);
    }
  }
  return -1;
}
```

9-8 排序

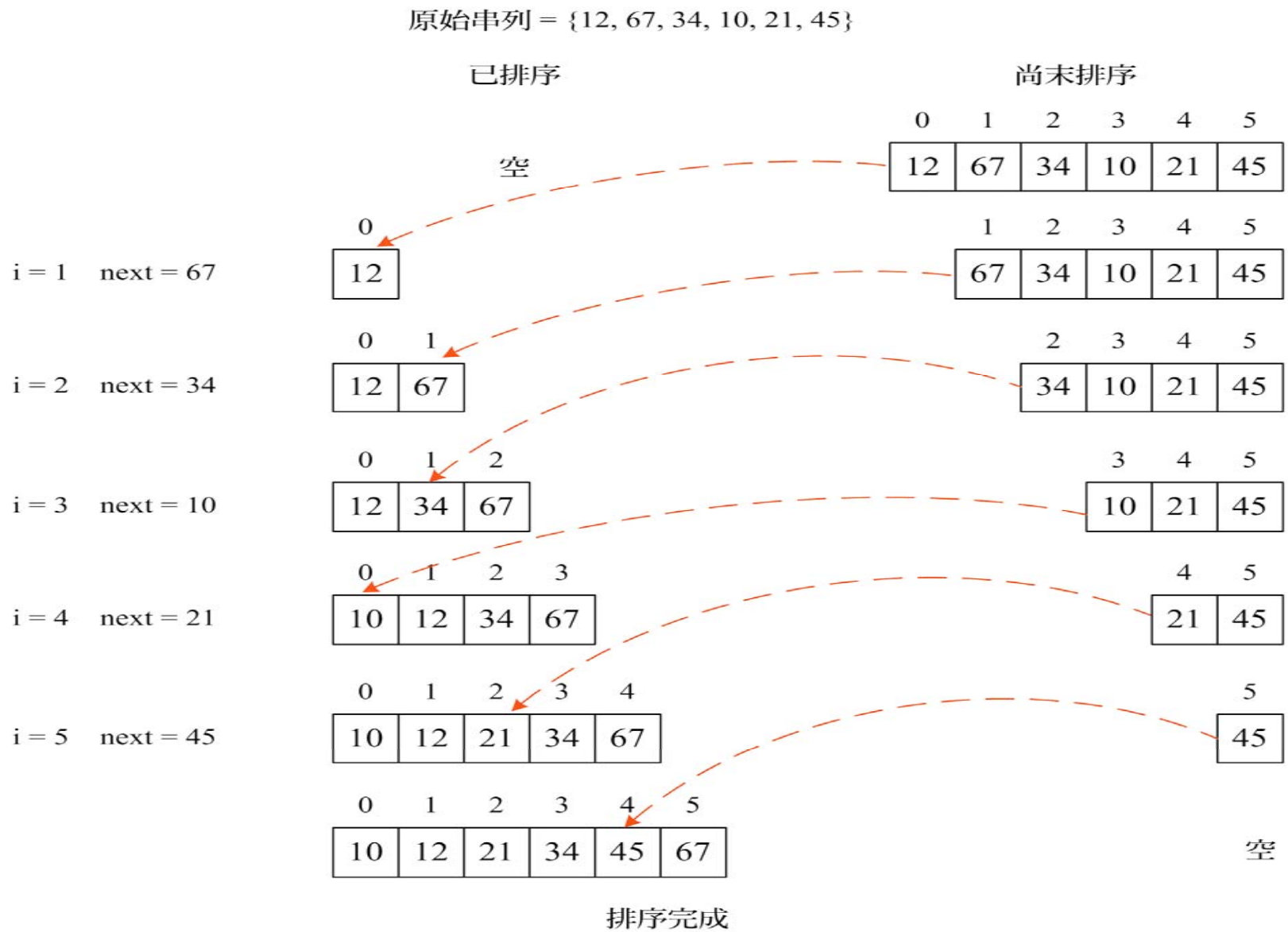
- 排序 (sort) 的目的是將多個資料由大到小或由小到大依序排列
- 知名的排序演算法有：
  - 插入排序 (inserion sort)
  - 氣泡排序 (bubble sort)
  - 快速排序 (quick sort)
  - 合併排序 (merge sort)
  - 謝耳排序 (shell sort)
  - 堆積排序 (heap sort)

插入排序

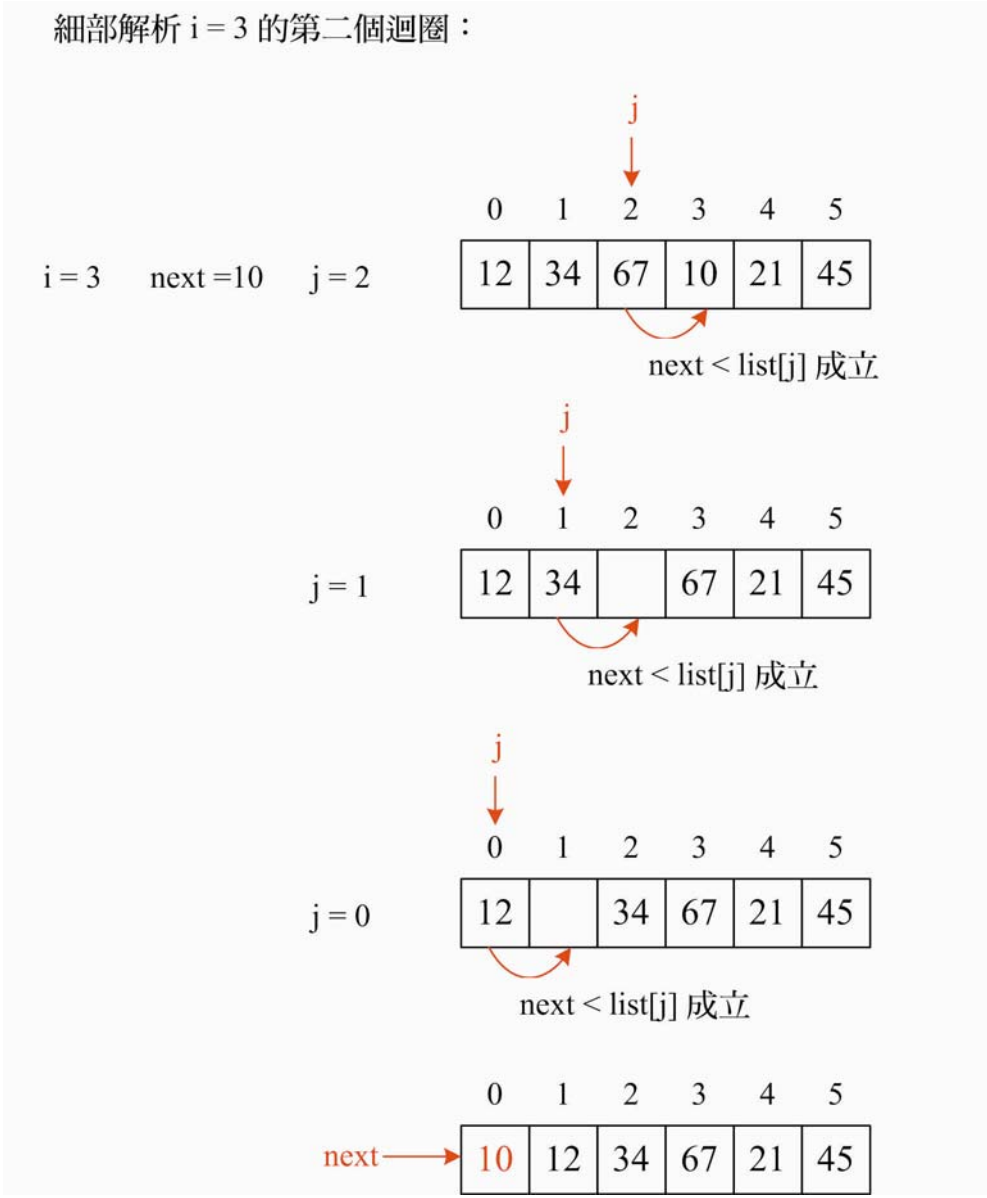
- 取出第一個資料
- 將第二個資料與第一個資料比較，若較大就放在後面，反之若比較小，就將第一個資料往後移插入第二個資料
- 將第三個資料與第二個資料比大小來調整順序，然後再和第一個資料比大小
- 直到所有的資料都插入適當的位置爲止

```
void insertion_sort(int list[], int n)
{
  int i, j, next;
  /* 從第二個撲克牌開始和其前面的撲克牌比大小 */
  for (i = 1; i < n; i++){
    next = list[i];                    /* next就像每次拿到的新撲克牌 */

  /* 第二個迴圈就像之前拿到且已經排序好的撲克牌 */
    for (j = i - 1; j >= 0 && next < list[j]; j--)
      list[j+1] = list[j];         /* 新撲克牌較小，將之前的撲克牌往後移 */

    list[j+1] = next;              /* 最後將空下來的位置讓給新撲克牌 */
  }
}
```

- 範例：list[] = {12, 67, 34, 10, 21, 45}，函數呼叫可以寫成insertion_sort(list, 6)


原始串列 = {12, 67, 34, 10, 21, 45}
已排序
尚未排序
空
i = 1 next = 67
i = 2 next = 34
i = 3 next = 10
i = 4 next = 21
i = 5 next = 45
排序完成

- N個項目的資料清單需要處理N-1回合才能完成

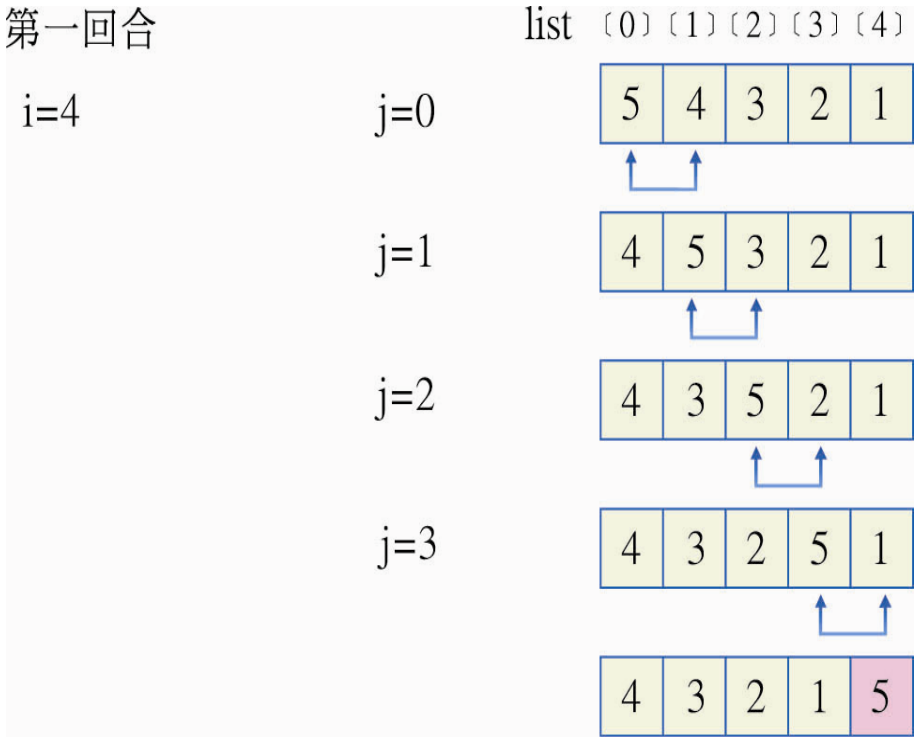


氣泡排序

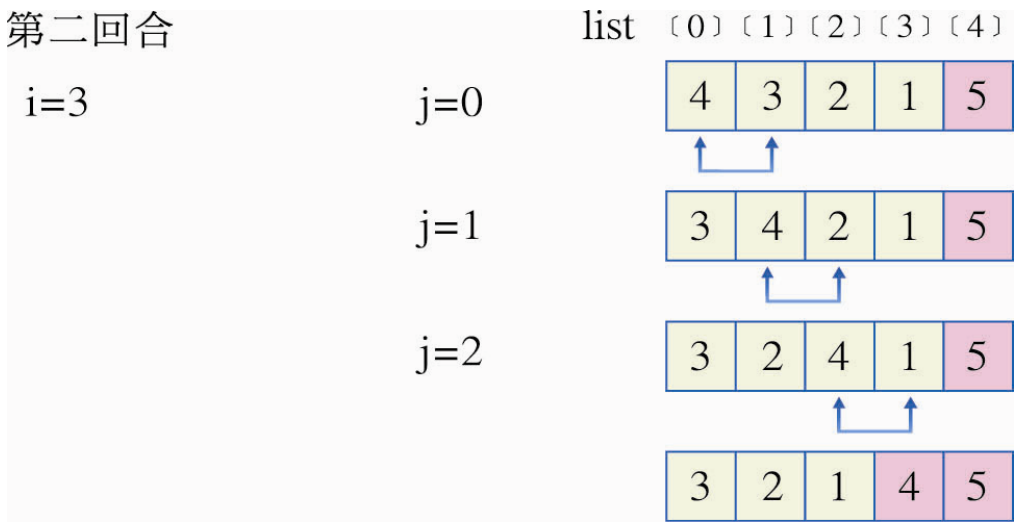
- 原理是將相鄰資料兩兩比較來完成排序
- 若資料的數目爲n，那麼比較的過程分成n - 1個回合，第i個回合會將第i個大的資料像“氣泡”般地浮現在從右邊數回來第i個位置 (由小到大排序)
- 假設要由小到大排序 {3, 5, 9, 4, 7}，第一個回合的任務就是要將第一大的資料 (9) 浮現在從右邊數回來第一個位置
- 要排序的資料陣列爲list[] = {5, 4, 3, 2, 1}，函數呼叫可寫成bubble_sort(list, 5)

```
void bubble_sort(int list[], int n)
{
  int i, j, flag, temp;

  /* 相鄰資料兩兩比較的過程總共有n-1個回合 */
  for (i = n - 1; i >= 1; i--){
    flag = 0;                                  /* flag用來記錄有無發生交換*/

  /* 內部迴圈用來進行每一回合的兩兩比較 */
    for (j = 0; j <= i - 1; j++){
      if (list[j] > list[j + 1]){
        temp = list[j];
        list[j] = list[j + 1];
        list[j + 1] = temp;
        flag = 1;
      }
    }
    if (flag = 0)     break;
  }
}
```

第一回合


list
〔0〕〔1〕〔2〕〔3〕〔4〕
i=4
j=0
5 4 3 2 1
j=1
4 5 3 2 1
j=2
4 3 5 2 1
j=3
4 3 2 5 1
4 3 2 1 5

第二回合


list
〔0〕〔1〕〔2〕〔3〕〔4〕
i=3
j=0
4 3 2 1 5
j=1
3 4 2 1 5
j=2
3 2 4 1 5
3 2 1 4 5

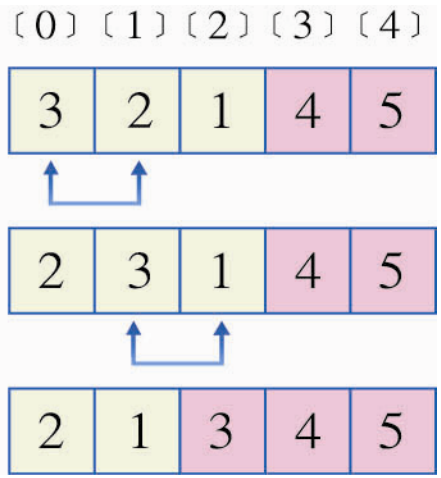
第三回合

i=2

j=0

j=1

list

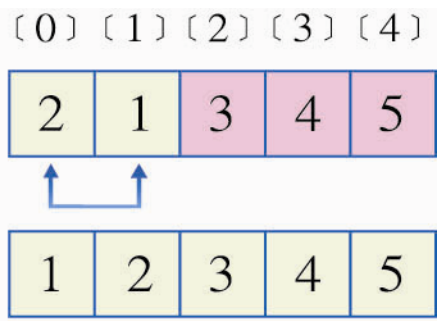


第三回合

i=1

j=0

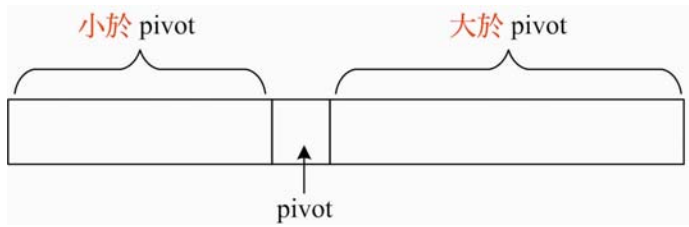
list



排序完畢

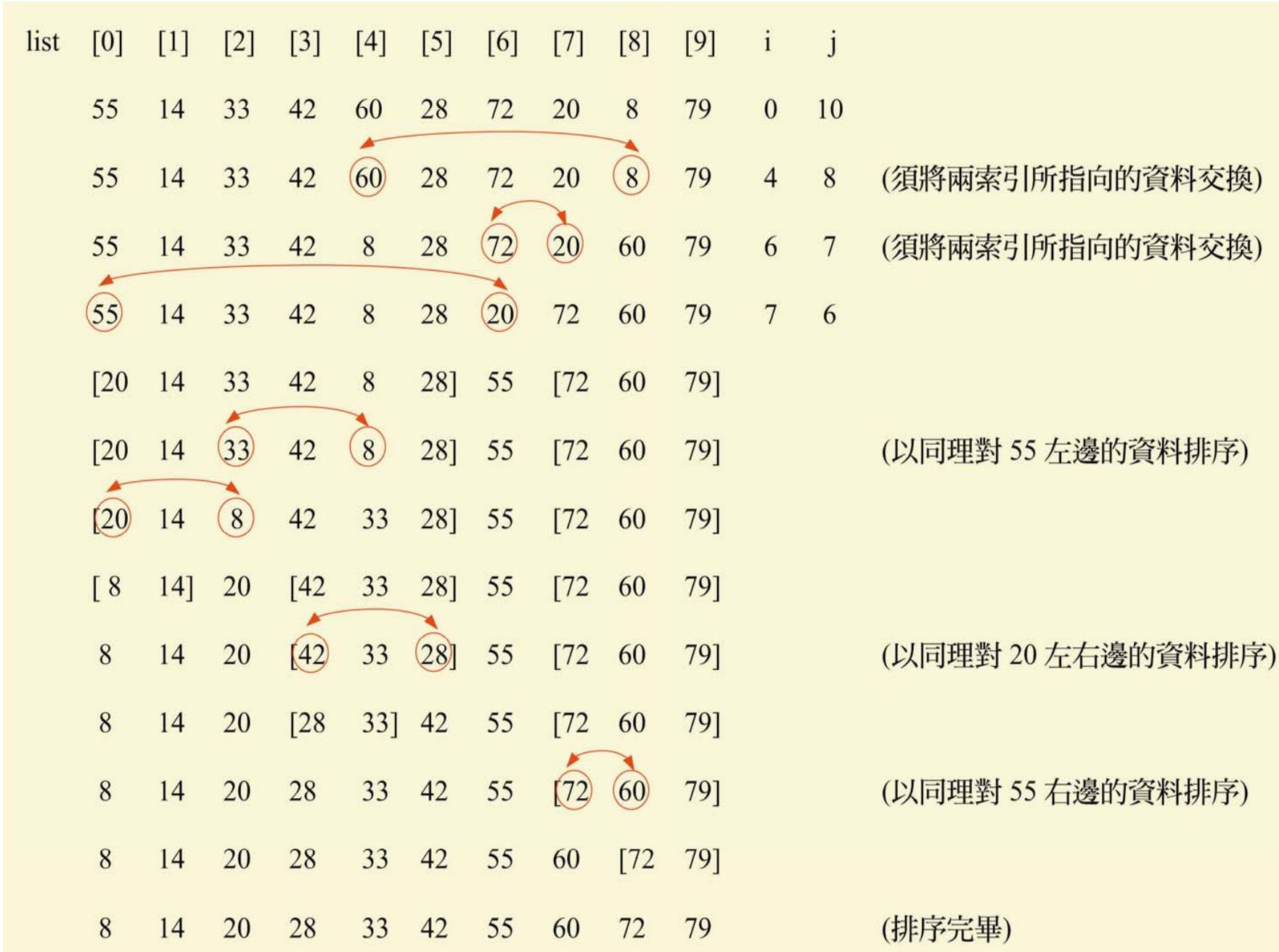
快速排序

- 原理是根據大小找出每一個資料在所有資料中的正確位置。若是由小到大排序，那麼在該位置左邊的資料會比它小，而右邊的資料則會比它大



- 演算法的變數：
  - pivot爲目前要找出正確位置的資料，初始值爲陣列的第一個元素
  - i 是陣列第一個元素的索引
  - j 是陣列最後一個元素的索引加1

- 範例：{55, 14, 33, 42, 60, 28, 72, 20, 8, 79}
- 由陣列的左邊開始尋找比pivot大的資料，找到後令索引 i 指向該資料
- 接著由陣列的右邊開始尋找比pivot小的資料，找到後令索引 j 指向該資料
- 此時若 i 小於 j，就將兩個索引所指向的資料交換，然後重覆前述步驟直到 i 不小於 j 爲止
- 等到 i 大於或等於 j 時，就將 pivot 與索引 j 所指向的資料交換，此時就表示找到了pivot的正確位置
- 接下來再以相同方式分別針對pivot左右兩邊的資料進行快速排序即完成


list [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] i j
55 14 33 42 60 28 72 20 8 79 0 10
55 14 33 42 60 28 72 20 8 79 4 8 (須將兩索引所指向的資料交換)
55 14 33 42 8 28 72 20 60 79 6 7 (須將兩索引所指向的資料交換)
55 14 33 42 8 28 20 72 60 79 7 6
[20 14 33 42 8 28] 55 [72 60 79]
[20 14 33 42 8 28] 55 [72 60 79] (以同理對 55 左邊的資料排序)
[20 14 8 42 33 28] 55 [72 60 79]
[8 14] 20 [42 33 28] 55 [72 60 79]
8 14 20 [42 33 28] 55 [72 60 79] (以同理對 20 左右邊的資料排序)
8 14 20 [28 33] 42 55 [72 60 79]
8 14 20 28 33 42 55 [72 60 79] (以同理對 55 右邊的資料排序)
8 14 20 28 33 42 55 60 [72 79]
8 14 20 28 33 42 55 60 72 79 (排序完畢)

```
void quick_sort(int list[], int left, int right)
{
  int i, j, pivot, temp;

  if (left < right){
    i = left;
    j = right + 1;
    pivot = list[left];

    do{

      do
        i++;
      while (list[i] < pivot);
      do
        j--;
      while (list[j] > pivot);
```

```
/* 若i小於j，將索引i與索引j所指向的資料交換 */
      if (i < j){
        temp = list[i];
        list[i] = list[j];
        list[j] = temp;
      }
    } while (i < j)

/* 若j大於i，將list[left] 與索引j所指向的資料交換 */
    temp = list[left];
    list[left] = list[j];
    list[j] = temp;
    quick_sort(list, left, j - 1);          /* 遞迴呼叫排序左邊*/
    quick_sort(list, j + 1, right);         /* 遞迴呼叫排序右邊*/
  }
}
```

9-9 分析演算法的優劣

- 時間複雜度 (time complexity) 所指的是程式要執行完成所需要的電腦時間
- 空間複雜度 (space complexity) 則是指程式要執行完成所需要的記憶體容量
- 測量演算法的時間複雜度：其中一種最常見的方式就是計算有多少條指令會被執行
  - 與電腦的執行速度無關，但與輸入資料的個數有關

- 加總範例程式的執行步驟個數
- s/e (steps/execution)：每條敘述要花多少個執行步驟
- 頻率 (frequency)：每條程式敘述的執行次數

敘述	s/e	頻率	執行步驟小計
int sum(int list[], int n)	0	0	0
{	0	0	0
int total = 0;	1	1	1
int i;	0	0	0
for(i = 0; i < n; i++)	1	n+1	n+1
total = total + list[i];	1	n	n
return total;	1	1	1
}	0	0	0
			2n+3

Big-O表示法

- Big-O表示法 (Big-O notation)，寫成 O()
- 這是一種用來表示時間複雜度與輸入資料個數之間的關係，例如：
  - O(n)：代表這個程式如果有n個輸入資料，就需要執行n個動作
  - O(n2)：代表有n個輸入資料就需要執行n^2個動作
  - 如上例其時間複雜度爲2n+3，通常就會用O(n)來代表此演算法的時間複雜度

時間複雜度的分析

例如分析循序搜尋演算法的時間複雜度是：

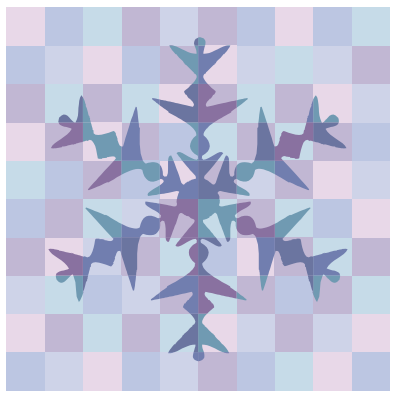
- 最佳情況：第一個就找到目標值，比較動作只有一次，此時的時間複雜度爲1
- 最差情況：找到最後一個還是沒找到，或是在最後一筆才找到，此時已經過n次的比較動作，此時的時間複雜度爲O(n)
- 平均情況：n個元素的陣列，平均花 (n+1)/2次的比較動作，此時的時間複雜度爲O(n)

- 通常比較少分析最佳情況，而是著重在平均情況和最差情況這兩部分
- 二元搜尋演算法的時間複雜度是：
  - 最差情況：至多只需要進行$\log_2 n$次的比對
  - 平均情況：大約是最差情況的一半
  - 因此結果兩者都是$O(\log_2 n)$，亦可直接寫$O(\log n)$
- 插入搜尋演算法的時間複雜度是：
  - 最差情況：$O(n^2)$
  - 最佳情況： $O(n)$
- $\log_2 n < n < n\log_2 n < n^2 < n^3 < 2^n$

各種演算法的O()

- 循序搜尋演算法的時間複雜度爲O(n)
- 二元搜尋演算法的時間複雜度爲O(log n)
- 插入排序演算法的時間複雜度爲O(n2)
- 快速排序演算法的平均情況時間複雜度是$O(n\log_2 n)$，而最差情況是$O(n^2)$
- 氣泡排序演算法的平均情況和最差情況的時間複雜度都是$O(n^2)$
- 在大資料量的情況下，實用的演算法其時間複雜度不能大於$O(n\log_2 n)$
- 實際的執行時間還是得看程式設計的成果而定

第９章　演算法



結束