CHAPTER 11 演算法

演算法

- 演算法就是計算機方法，是設計適合計算機執行的方法，如同神農氏遍嘗百藥的精神一般，計算機科學家針對任何疑難雜症的計算問題，總設法找出最好的解決方法，只是不必以身試毒，而是讓數位計算機代為受罪罷了。
- 在我們的數位世界裡，每一份數位資料的處理，最終都化成某種程度的計算問題，而好的演算法正是數位計算的靈魂。

演算法

- 日益精進的數位處理器，配上精雕細琢的演算法，將是構築未來數位世界很重要的兩把刷子。
- 演算法常需要好的設計與分析，有時也需要腦筋急轉彎，才能找到好解答。

11-1 最大數及最小數找法

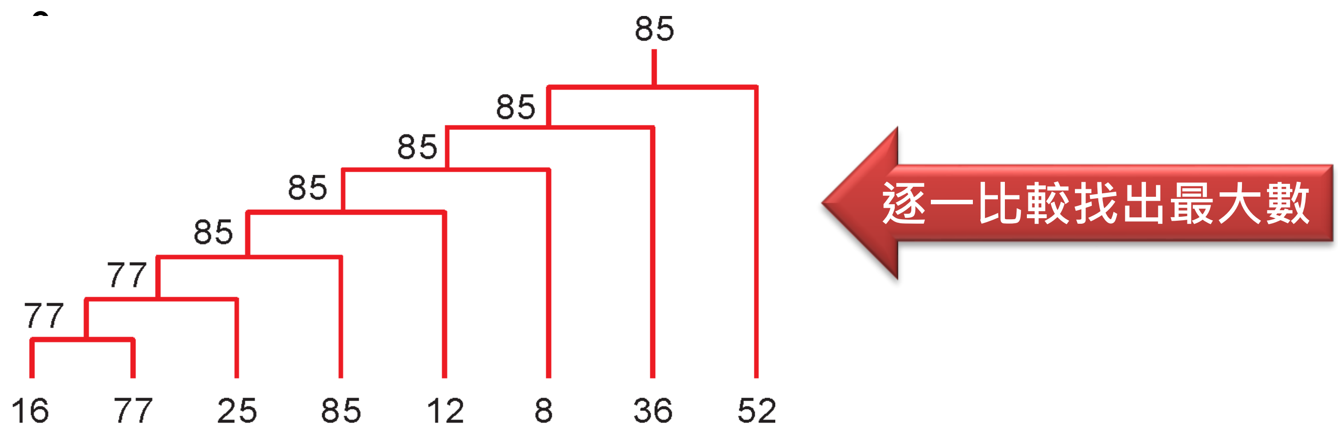
- 作法1─逐一比較法
- 作法2─兩兩比較法

作法1 - 逐一比較法

- 從第一個數看起，記錄到目前為止最大的數，循序往後面的數看去，如果接下來的數比所記錄的最大數更大，則取代之，直到最後一個數，則所記錄的數即為最大數。
- 下圖給了一個八個數的例題「請找出16、77、25、85、12、8、36及52裡的最大數」，一開始16是紀錄上最大的數；等看到77時，77比16大，所以將所記錄的數改成77；接著是25，但25比77小，所以不更改紀錄；接著是85，它比77大，所以最大數改成85，之後85持續為最大數，一直到最後，因此，最大數為85。

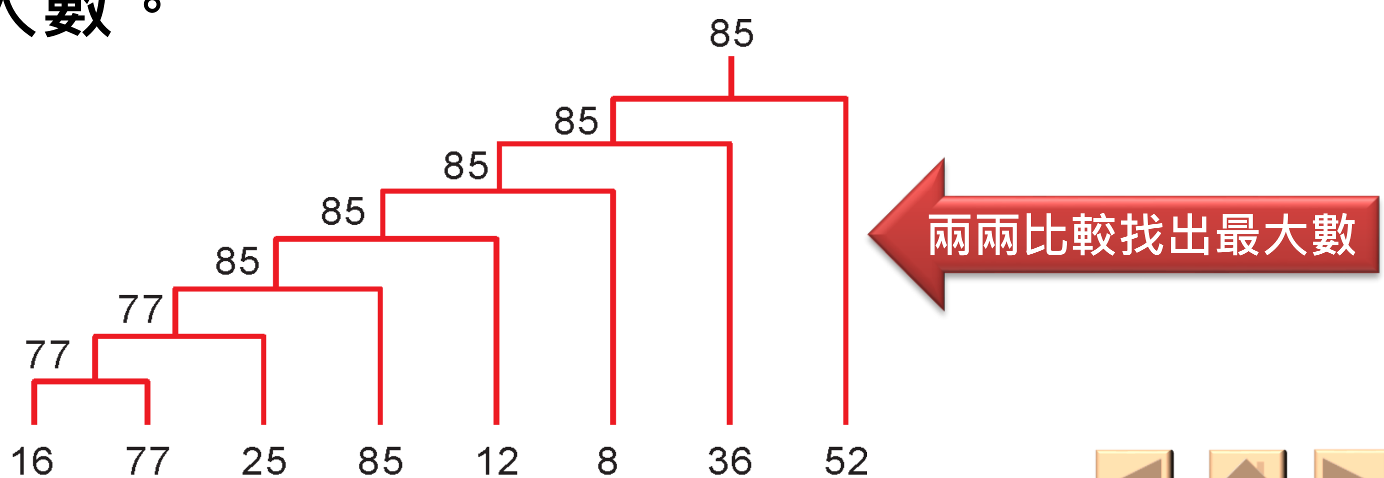
作法1 - 逐一比較法

- 作法1從八個數中找出最大數，共需多少次的比較呢？前面兩個數需要一次，之後每個數都需一次比較，所以共用了7次比較。
- 同理可推，給定n個數，作法1需用n-1次比較找出最大數。



作法2－兩兩比較法

- 兩兩比較，將比較大的數再用同樣作法兩兩比較，直到最後勝出的數即為最大。如下圖所示，第一輪勝出的數為77、85、12及52；第二輪勝出的數為85及52；第三輪勝出的數為85，此數即為最大數。



作法2 - 兩兩比較法

- 作法2從八個數中找出最大數，共需多少次的比較呢？
- 第一輪需要4次比較、第二輪2次、第三輪1次，總共7次，和作法1次數相同。
- 假設n是2的整數次方，給定n個數，第一輪需要n/2、第二輪n/22、第三輪n/23、...，所以共需n/2 + n/22 +n/23 + ... + 1 = n-1，和作法1次數相同！

作法2 - 兩兩比較法

- 考慮下面這個例題：「請找出16、77、25、85、12、8、36及52裡的最大數及最小數」，先看看圖11-1作法1，我們以7次比較找出最大數，再從最大數85外的其他7個數(16、77、25、12、8、36及52)中，以6次比較找出最小數8，這樣共用了7+6=13次比較，是否有更少次數的比較方式呢？

作法2－兩兩比較法

- 現在請回頭再看看剛剛圖11-2的作法2，我們以7次比較找出最大數，要找最小數，只要考慮第一輪輸掉的那些數即可，也就是16、25、8及36這四個數，因此只要再用3次比較即可找出最小數8，這樣共用了7+3=10次比較。

作法2 - 兩兩比較法

- 因此，假設n是2的整數次方，如果我們的問題是從n個數中找出最大數及最小數，要用多少次比較呢？
- 我們可用作法1以n-1次比較找出最大數，再以n-2次比較，從除了最大數之外的n-1個數中，找出最小數，這樣的作法共需(n-1)+(n-2) =2n-3次比較。

作法2 - 兩兩比較法

- 我們也可用作法2以n-1次比較找出最大數，再以n/2-1次比較，從第一輪輸掉的n/2個數中，找出最小數，這樣的作法共需(n-1)+(n/2-1) = 3n/2-2次比較，我們可以證明這是最少的比較次數。

作法2－兩兩比較法

- 考慮下面這個例題：「請找出16、77、25、85、12、8、36及52裡的最大數及第二大數」，在作法1中，我們以7次比較找出最大數，再從最大數85外的其他7個數(16、77、25、12、8、36及52)中，以6次比較找出其中的最大數77(也就是全部的第二大數)，這樣共用了7+6=13次比較，是否有更少次數的比較方式呢？

作法2－兩兩比較法

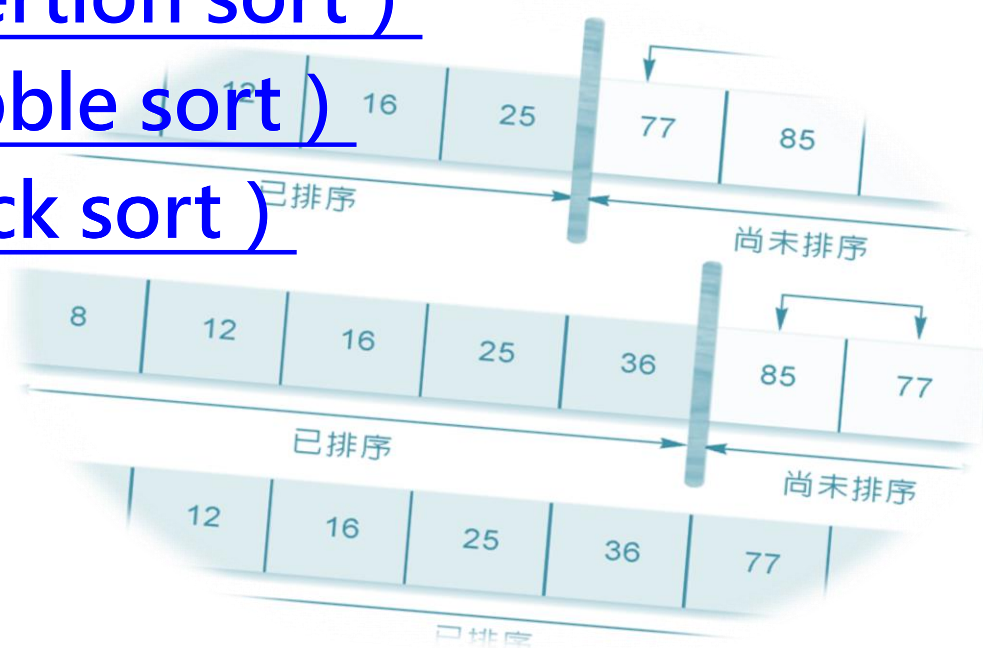
- 在作法2中，我們以7次比較找出最大數，要找第二大數，只要考慮曾輸給最大數的那些數即可，也就是52、77及25這三個數，因此只要再用2次比較即可找出第二大數77，這樣共用了7+2=9次比較。

作法2 - 兩兩比較法

- 因此，假設n是2的整數次方，如果我們的問題是從n個數中找出最大數及第二大數，要用多少次比較呢？以作法1進行，我們需要(n-1)+(n-2) ＝ 2n-3次比較。
- 若以作法2進行，只有$\log_2 n$個數曾輸給最大數，因此，總共需要$(n-1)+(\log_2 n-1) = n+\log_2 n-2$次比較，我們可以證明這是最少的比較次數。

11-2 排序

- 選擇排序法（selection sort）
- 插入排序法（insertion sort）
- 泡沫排序法（bubble sort）
- 快速排序法（quick sort）



11-2 排序

- 排序問題：給定n個數，請將它們由小排到大。排序是電腦經常用到的演算法，資料一旦排序之後，後續尋找便能快速進行。
- 但排序的演算法效率差別很大，當資料量變大時，演算法的好壞將影響執行所需時間甚鉅。

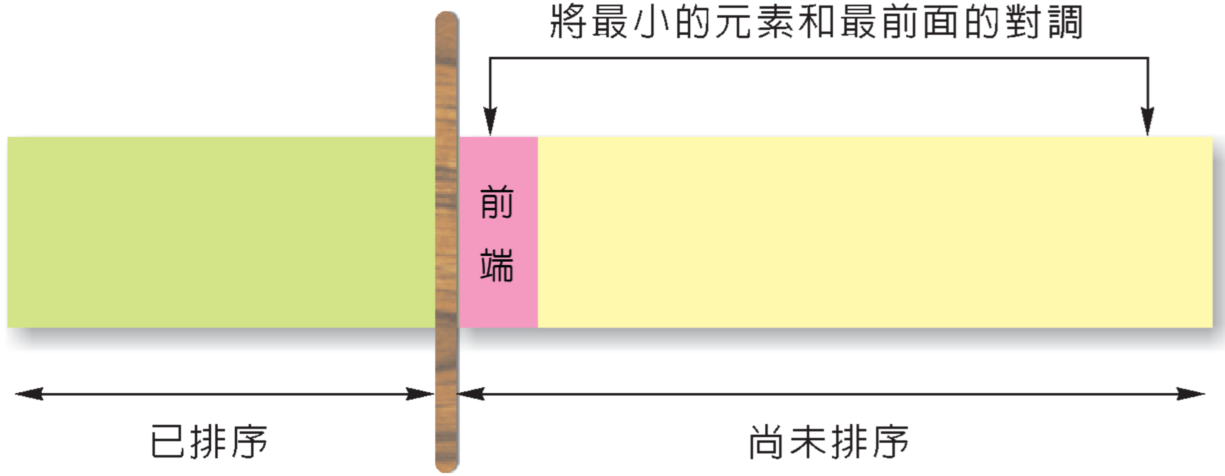
11-2 排序

- 快速排序法
- 先任挑一個資料，將比這資料小的都放在它的前面；比這資料大的放在它後面。
- 然後，針對資料比較小及資料比較大的那兩部分，我們也都使用同樣方法來排序，以此類推。到最後，我們的資料也是由小排到大。
- 這方法平均而言，所做的比較次數會和總筆數乘上總筆數的二基底對數成正比。

選擇排序法(selection sort)

- 選擇排序法將數列切成兩部分：已排序數列及未排序數列，每次從未排序的數列中挑出最小的數，將它移到未排序數列的最前面，這個數不會小於已排序數列的任何數，而且
- 也不會大於未排序數列的任何數，因此它已就定位了，所以可以將它歸入已排序數列，整個關鍵動作如下圖所示。

選擇排序法(selection sort)



選擇排序法將未排序數列的最小數移到序列前端

選擇排序法(selection sort)

- 摘要步驟如下：

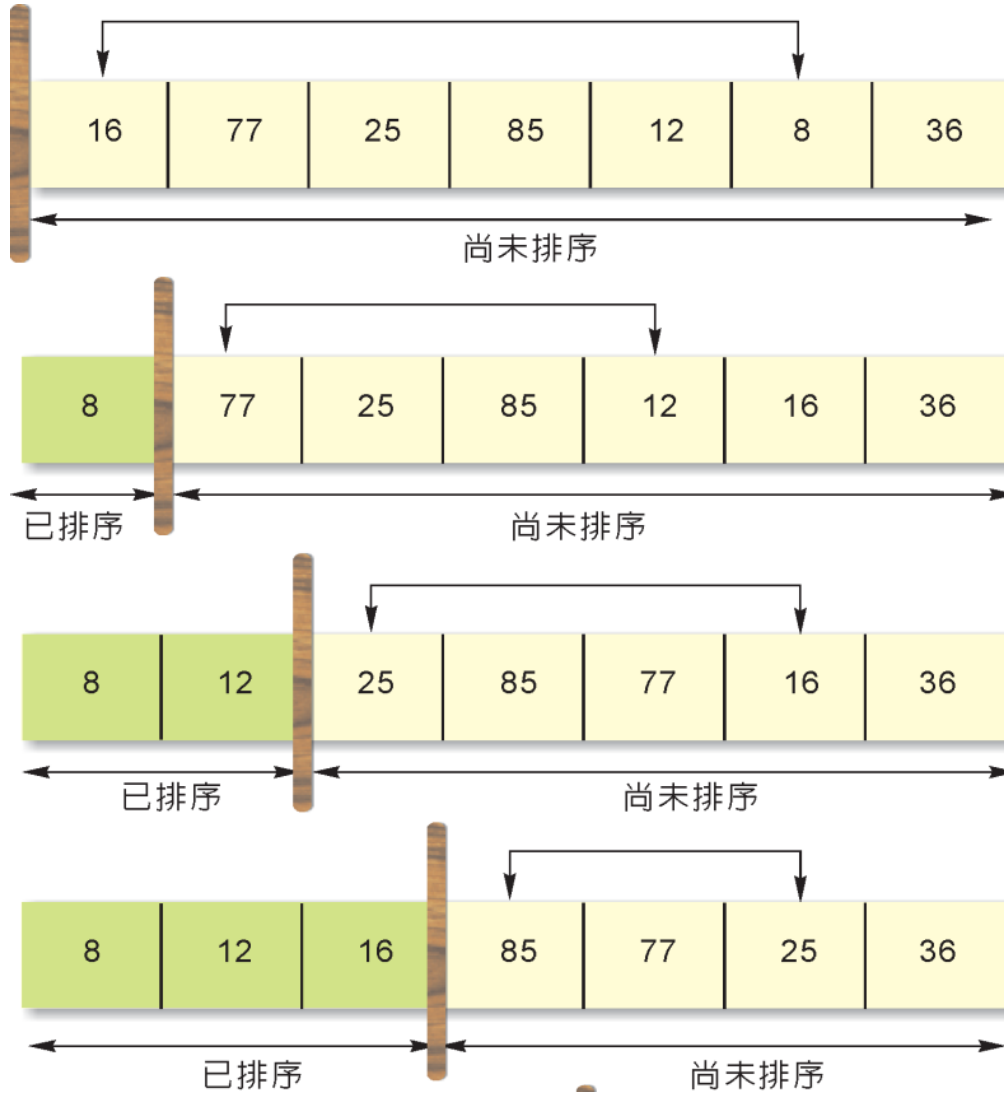
步驟 1	步驟 2	步驟3
一開始整個數列歸類為未排序	從未排序的數中，挑選出最小的數，和未排序數列中的第一個位置元素互調，並將該最小的數歸類到已排序的數列中	重複步驟2，直到所有的數都歸到已排序數列中

一開始，全部數列都算是未排序數列。

其中以8最小，所以它和第一個位置的16互換，造成已排序數列中有8，而未排序數列中有77、25、85、12、16、36。

其中以12最小，所以它和第一個位置的77互換，造成已排序數列中有8、12，而未排序數列中有25、85、77、16、36。

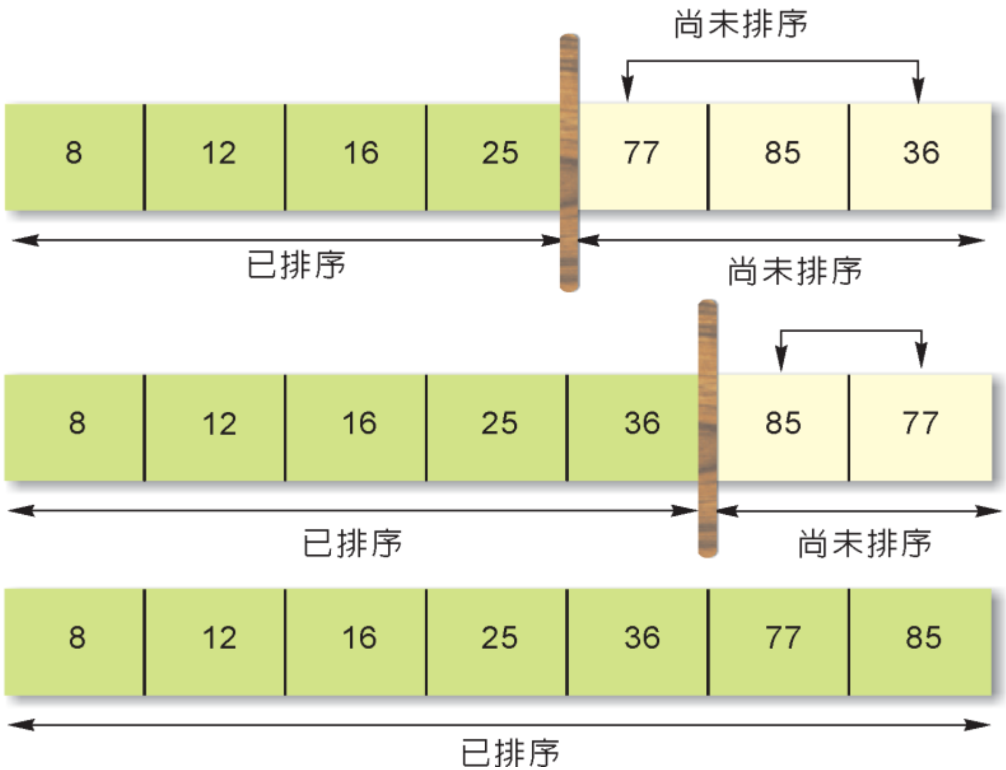
其中以16最小，所以它和第一個位置的25互換，注意在此時16換到未排序數列的最前面，它不比已排序數列的數小，也不比未排序數列的數大，因此它移到了自己的定位，可歸到已排序數列。



其中以36最小，所以它和第一個位置的77互換，造成已排序數列中有8、12、16、25，而未排序數列中有85、77。

其中以77最小，所以它和第一個位置的88互換，造成已排序數列中有8、12、16、25、36、77，而未排序數列中有85。

得到最終的排序結果：8、12、16、25、36、77、85。

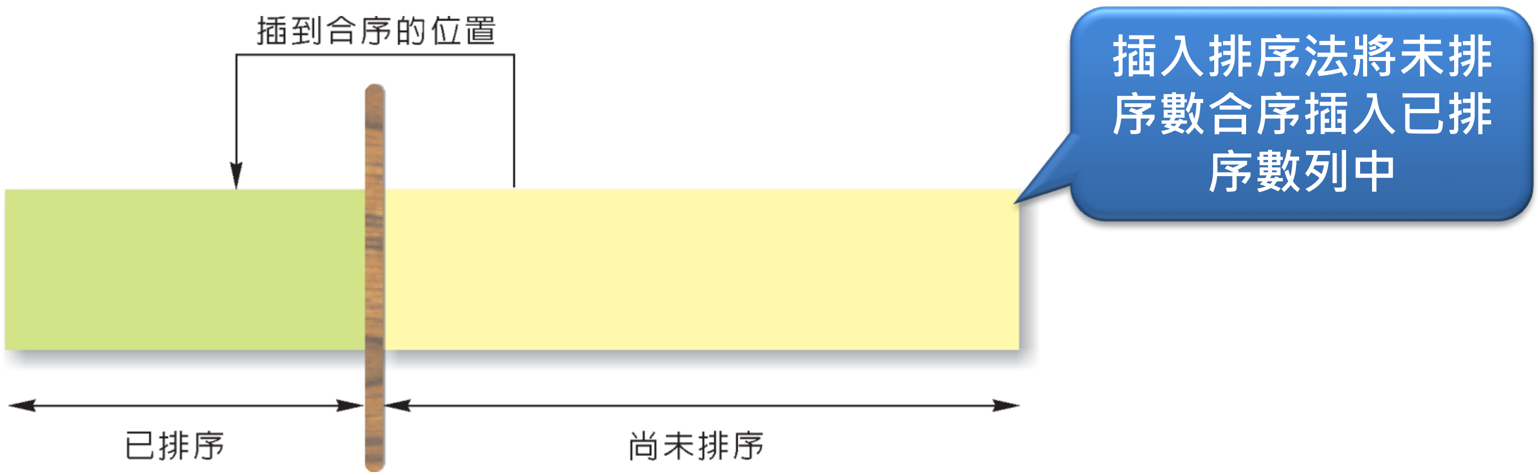


選擇排序法(selection sort)

- 若給定n個數，用選擇排序法大約要做多少次的比較呢？
- 要從n個數中找出最小數，需要n-1次的比較；然後再從剩下的n-1個數中找出最小數，需要n-2次的比較；...，所以總共需要(n-1)+(n-2)+(n-3)+...+1 = (n-1)(n-2)/2次比較，和n2的成長速率成正比。

插入排序法(insertion sort)

- 插入排序法將數列切成兩部分：已排序數列及未排序數列，每次將未排序數列中的第一個數，插入到已排序數列中，使得插入後的已排序數列仍然維持由小排到大的性質。



插入排序法(insertion sort)

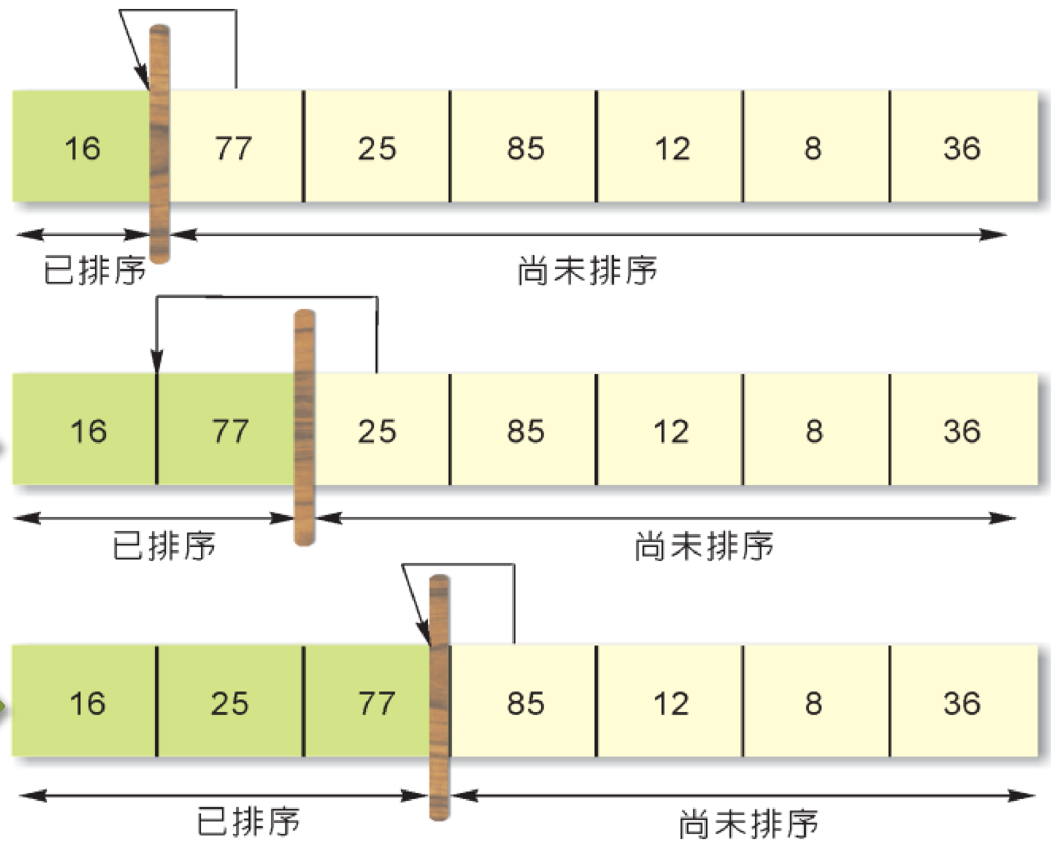
- 摘要步驟如下：

步驟 1	步驟 2	步驟3
一開始只有第一個數在已排序數列裡，其他的數歸類在未排序數列裡	將未排序數列的第一個數，插入到已排序的數列中，使得插入後的已排序數列仍然維持由小排到大的性質	重複步驟2，直到所有的數都歸到已排序數列中

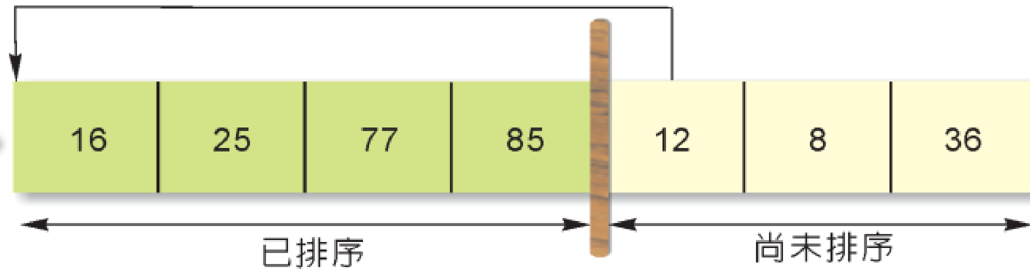
一開始，只有第一個位置的數16在已排序數列裡。

未排序數列的第一個數為77，將它插入已排序數列，因為77最大，所以放在後面，此時已排序數列為16、77。

未排序數列為25、85、12、8、36，再將第一個數25插入到已排序數列裡，得已排序數列為16、25、77。

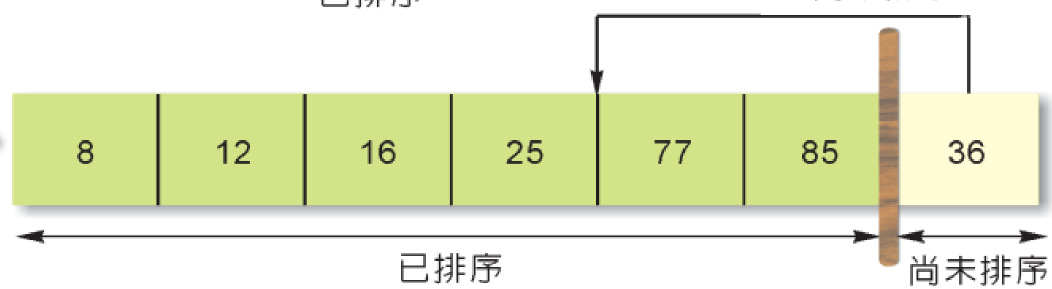


未排序數列為85、12、8、36，再將第一個數85插入到已排序數列裡，此時已排序數列為16、25、77、85。



未排序數列為12、8、36，再將第一個數12插入到已排序數列裡，此時已排序數列為12、16、25、77、85。

未排序數列為8、36，再將第一個數8插入到已排序數列裡，此時已排序數列為8 、 12、16、25、77、85。



得到最終的排序結果：8、12、16、25、36、77、85。

插入排序法(insertion sort)

- 若給定n個數，用插入排序法大約要做多少次的比較呢？
- 一開始，已排序數列的數只有一個，因此我們只要1次比較即可，等到已排序數列的數漸漸多了，我們所需的比較次數逐漸增加，在n個已排序的數中，找尋一個數最合序的位置(也就是在那個位置之前的數都沒有比較大，且之後的數都沒有比較小)，只要$\log_2 n$個比較即可，因此我們最多也不會用超過$n\log_2 n$個比較。

插入排序法(insertion sort)

- 然而這裡的問題是：當你找到一個數合序的位置時，你必須去移動該數插入後，在已排序數列中該位置之後的數都要往後移動一個位置，這樣最慘的情況下，代價可不小。
- 假設給定的數列是由大排到小，則每次都插到已排序數列的最左邊，等於所有已排序數列的數每次都要移動位置，所以總共需1+2+3+…+(n-2)+(n-1)=(n-1)(n-2)/2次移動，和n2的成長速率成正比。

插入排序法(insertion sort)

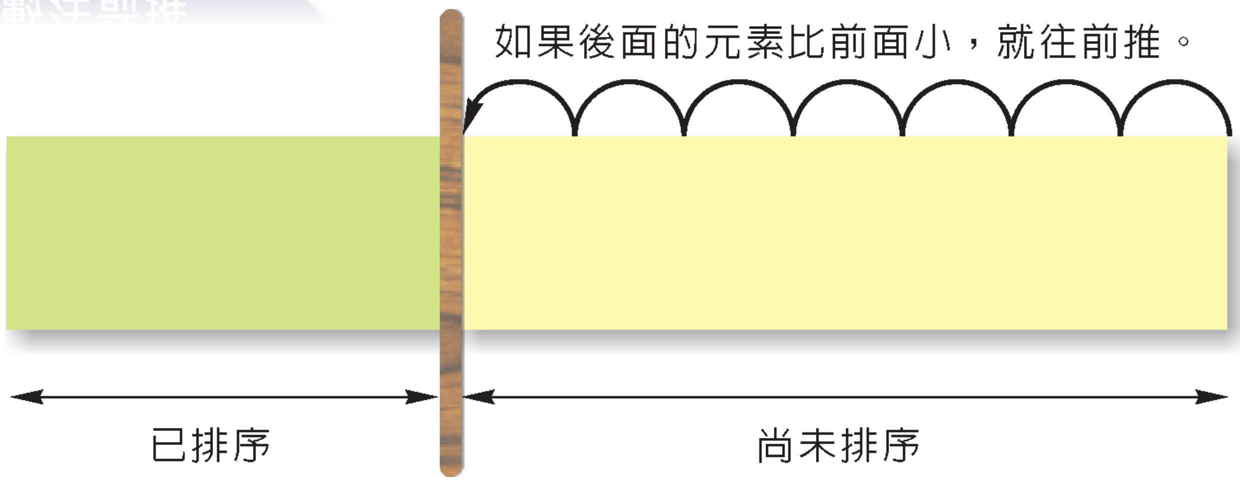
- 細心的讀者可能會想到，排序前半段的時候，已排序數列比較短，所以插入排序法比較有效率，到了後半段，未排序數列比較短，選擇排序法就會比較有效。
- 我們是不是可以綜合這兩種排序法而找到更有效率的方法呢？答案是肯定的，在前半段使用插入排序法，最慘情況共需$1+2+3+...+n/2=(1+n/2)n/4$次移動；在後半段使用選擇排序法，共需$n/2+(n/2-1)+...+1=(1+n/2)n/4$次比較。

泡沫排序法(bubble sort)

- 泡沫排序法將數列切成兩部分：已排序數列及未排序數列。
- 每次從未排序數列中的最後一個數看起，如果它比前面的數小，則往前移，一直看到未排序數列的第一個數為止，在這過程裡，未排序數列最小的數會像泡沫一樣，浮到最前面，這過程和選擇排序法類似。

泡沫排序法(bubble sort)

泡沫排序法將未排序數列數列的小數往前推

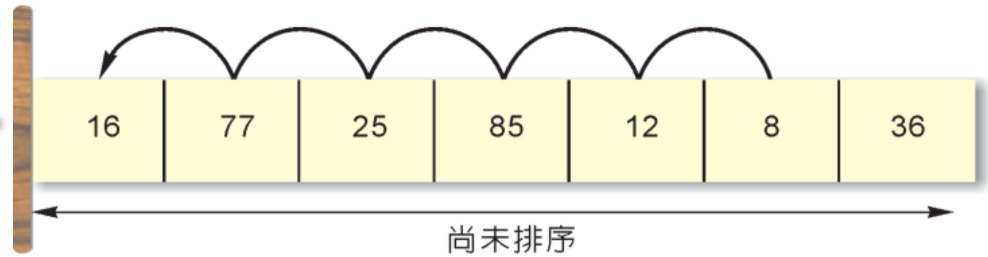


泡沫排序法(bubble sort)

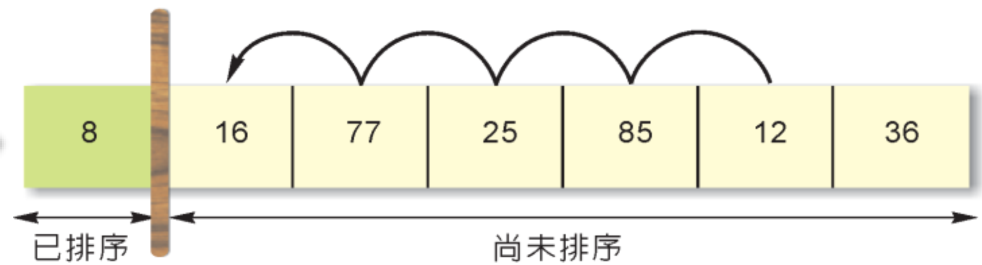
- 摘要步驟如下：

步驟 1	步驟 2	步驟3
一開始整個數列歸類為未排序	從未排序數列的最後一個數開始看起，如果後面的數比前面小，就往前推，在這過程中，最小的數會被推到未排序數列中的第一個位置，將該最小的數歸類到已排序的數列中	重複步驟2，直到沒有往前推的動作為止

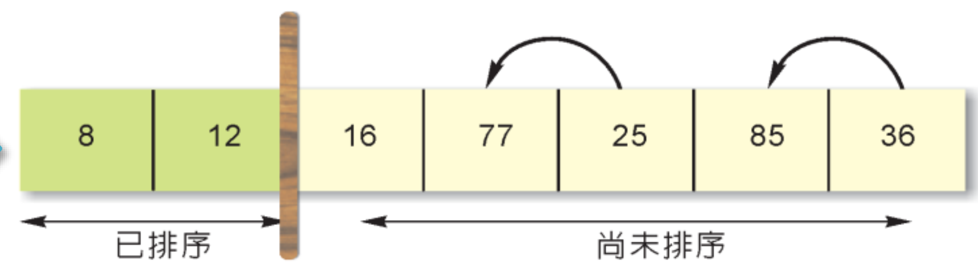
全部數列都是未排序數列，從最後一個數36看起，它沒比8小，所以不做任何動作。



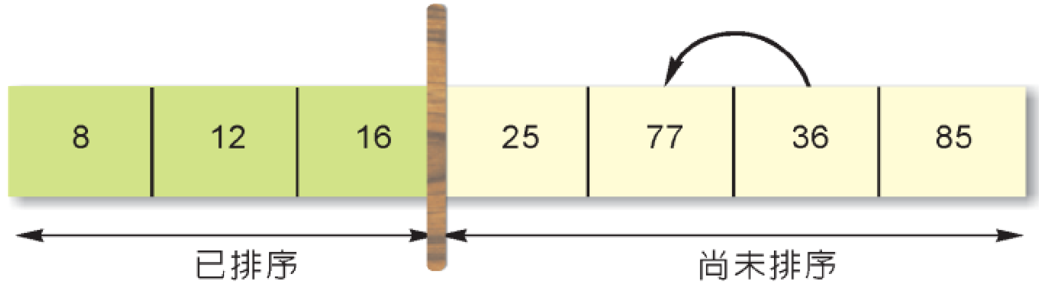
看到8，它比前面的12小，所以往前推，又比85小，再往前推，一直被推到最前面，造成已排序數列中有8。



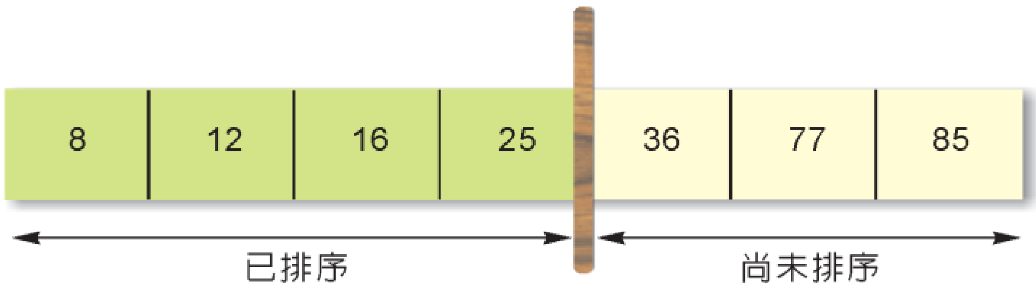
再從最後一個數36看起，它沒比12小，所以不做任何動作，接著看到12，它比前面的85小，所以往前推，又比25小，再往前推，一直被推到最前面。



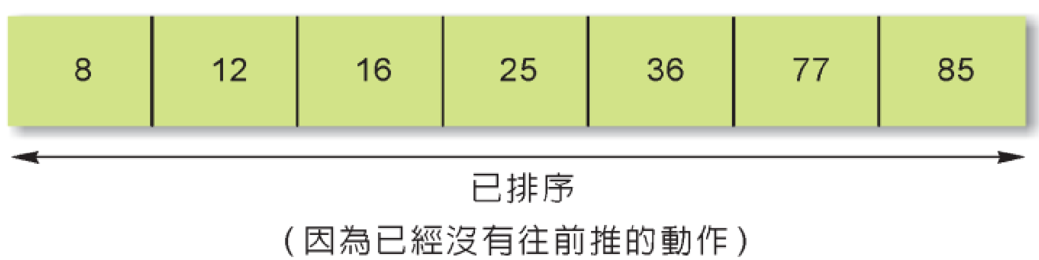
再從最後一個數36看起，它比前面85小，所以往前推，但推完後它比前面25大，所以就停住，再看25，它比前面的77小，所以往前推，但推完後它比前面16大。



再從最後一個數85看起，它沒比36小，所以不做任何動作，接著看到36，它比前面的77小，所以往前推，但推完後它比前面25大，所以就停住。



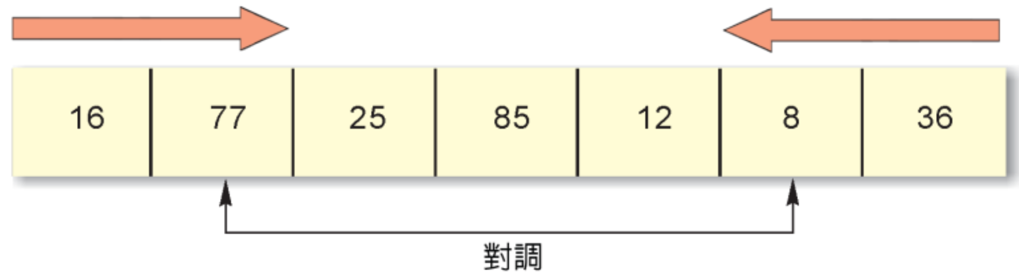
再從最後一個數85看起，發現這一次從85一直看到36，都沒有往前推的動作，表示在未排序數列中，每個數都不比前面的數小，亦即它也已排序，因此得到最終的排序結果：8、12、16、25、36、77、85。



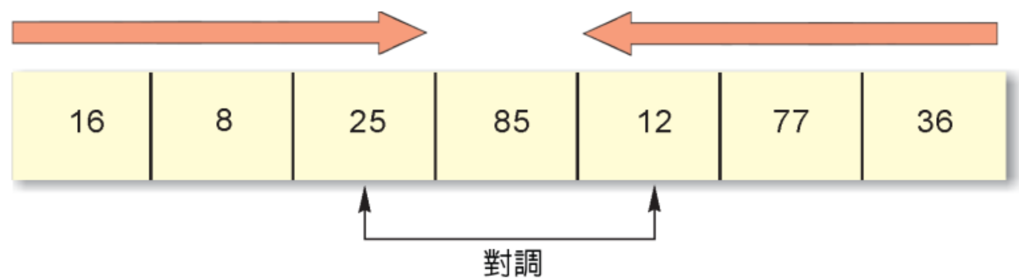
快速排序法(quick sort)

- 快速排序法的概念比較複雜，在此我們並不深入討論細節，僅就整個方向略加敘述，希望讀者能有所體會。
- 它的作法是先取一個數，通常是最前面的那個數，決定這個數該在的位置，等這個數就定位後，比它小的數會在該數的前面，而比它大的數會在該數的後面，再將同樣招數套在前面那一堆以及後面那一堆，等到大家都就定位後，整個數列也已排序了。

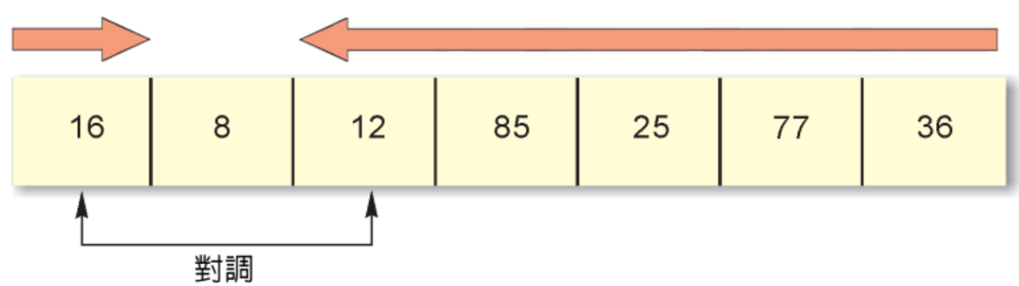
分別從數列的前面和後面往中間看起，從前面過來的會停在比16大的地方，因為我們希望將比16大的數都往後移，因此停在77；從後面過來的，會停在比16小的地方，因為我們希望將比16小的都往前移，因此停在8；此時，將77和8互調。



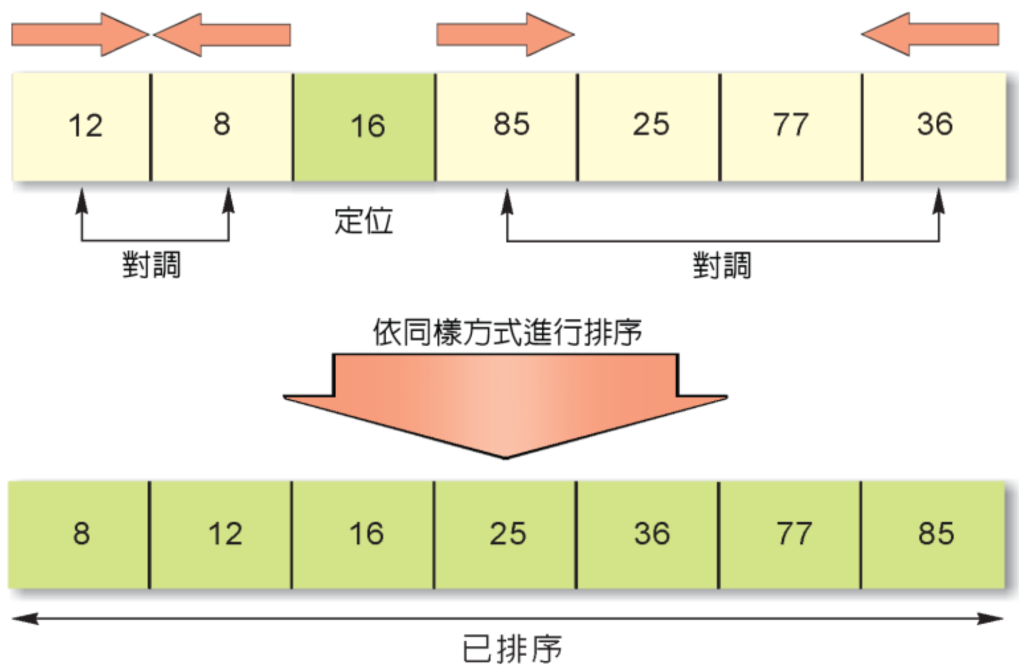
接著再往中間邁進，因此又互調了25和12，最後我們得到16、8、12、85、25、77、36。



再把16和12互調，得到12、8、16、85、25、77、36。



此時16已就定位，剩下的任務就是把12、8及85、25、77、36這兩個子數列排好。



套用同樣的招式將這兩個子數列一一解決。

快速排序法(quick sort)

- 快速排序法通常執行起來，比前面的方法要快許多，若每次就定位的數正好把數列切成兩個大小差不多的子數列，子數列的大小每次縮小一半，等到子數列被切到只剩一個數。
- 假設n是2的整數次方，如果一個數列長度為n，每次若被切為一半，得到所切的次數大約為$\log_2 n$，精細的計算可證明快速排序法所用比較次數的平均和$n\log_2 n$的常數倍成正比。

11-3 二元搜尋法

- 給定一個數列，搜尋問題問的是某個數是否在裡面？例如：給定一個數列16、77、25、85、12、8、36、52，請問5在不在裡面？
- 我們逐一比對後發現5並不在裡面，所以回答5不在這數列裡。
- 請問12在不在裡面呢？我們一一看過去，發現第五個位置有12，因此回答12在這數列裡。

11-3 二元搜尋法

- 這種逐一比較搜尋的方法稱為**循序搜尋法**(sequential search)，在數列很短的時候，還撐得過去，但當數列很長的時候，每次搜尋都要一一比對，則效率就差了。
- **二元搜尋法**（binary search）的運作道理，如果我們先有個排序好的數列，則搜尋起來就有效多了。

11-3 二元搜尋法

- 給定一個排序好的數列，二元搜尋法的步驟如下(實作時須注意儲存數列的陣列是從0的位置算起，或是從1的位置算起。)：

步驟 1
- mid←原排序數列的中間數

步驟 2
- 將所要搜尋的數與mid相比

步驟 3
- 如果搜尋的數與mid相等，則我們已找到，回答該數在數列裡

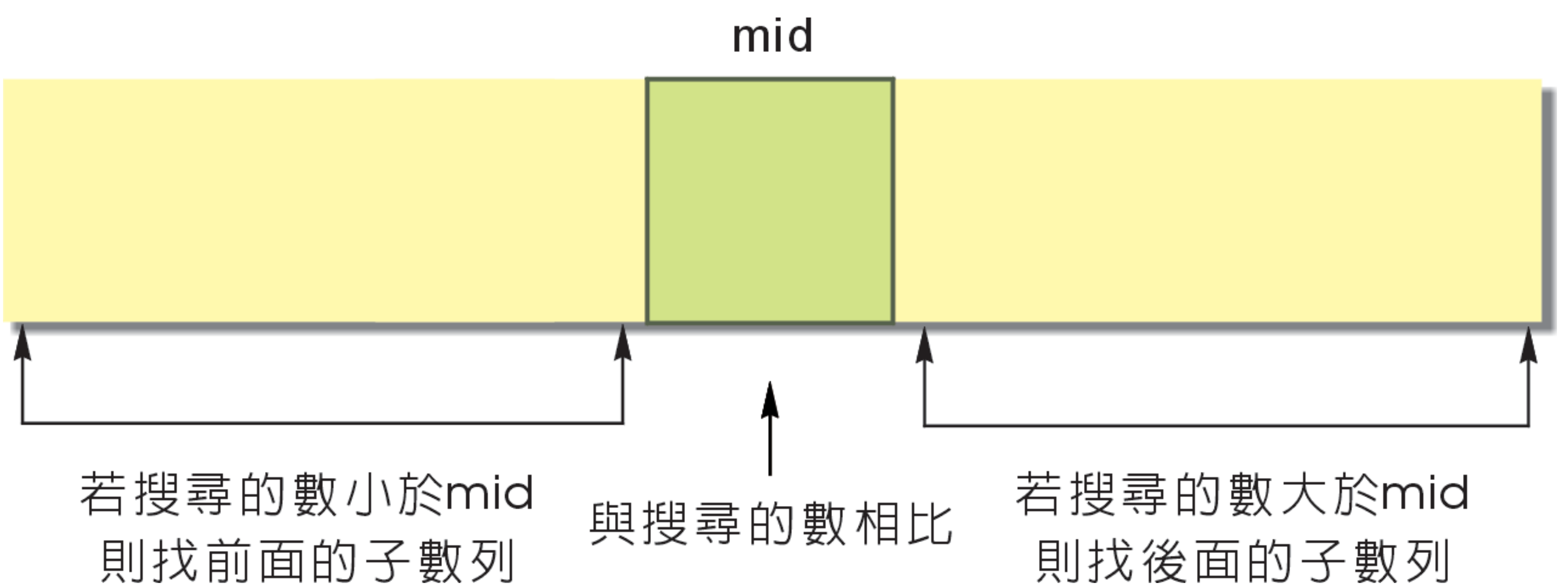
11-3 二元搜尋法

步驟 4
- 如果目前子數列只剩一個數(此時搜尋的數與mid不等)，則回答該數不在數列裡

步驟 5
- 如果搜尋的數小於mid，則只要考慮前半的子數列，mid ←前面子數列的中間數，回到步驟2

步驟 6
- 如果搜尋的數大於mid，則只要考慮後半的子數列，mid ←後面子數列的中間數，回到步驟2

11-3 二元搜尋法



二元搜尋法只要比較一次，問題大小就至少減半

11-3 二元搜尋法

- 假設我們的數列有2^{30}個數，也就是大約十億個數，若該數列已排序，以二元搜尋法找某數是否在裡面，最多需要幾次比較呢？
- 先比一次中間數，所須考慮的子數列個數就從2^{30}減半成為2^{29}，再一次比較，所須考慮的子數列個數最多為2^{28}，…，經過30次比較後，子數列最多只剩2^{0}個，此時再一次比較就可確認某數是否在裡面。

11-3 二元搜尋法

- 因此，二元搜尋法最多只要31次比較，就可判斷某數是否在一個有十億個數的數列裡，很神奇吧！
- 同樣推理，我們也可知道，若已排序的數列長度為n，則二元搜尋法的最多比較次數大約為$\log_2 n$。

11-4 動態規劃技巧

- 動態規劃技巧(dynamic progr amming)的programming在此並不是程式設計的意思，而是代表一種「列表式」的運算。
- 在正式介紹動態規劃技巧之前，我們先從一個簡單的例子來感受列表式的計算為何有時可較有效率地求得我們所要的結果。

11-4 動態規劃技巧

- 費氏數(Fibonacci number)可用下列的遞迴關係(recurrence)來描述：

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_i &= F_{i-1} + F_{i-2} \quad \text{for } i \geq 2
\end{aligned}
$$

- 如果想知道F_{20}的值是多少，有人可能會以程式語言中的遞迴呼叫(recursive call)這麼做：先試著去求得F_{19}，然後再設法求F_{18}，最後再將兩個加起來。

11-4 動態規劃技巧

- 而要如何求得F_{19}呢？這還不簡單嗎？將F_{18}及F_{17}算出來就可以了呀！Wait a minute！F_{18}不是已經算過了嗎？為何現在又要重算了呢？
- 實際上，以遞迴呼叫來處理這樣的問題，重算的次數還真嚇人呢！

11-4 動態規劃技巧

- 如果我們以列表式方法逐一從F_0、F_1、F_2等往F_{20}算去，你會發現在20次運算之內我們就能算出F_{20}的值：

F_0	F_1	F_2	F_3	F_4	F_5	…
0	1	1	2	3	5	…

- 列表式方法最大的作用就是避免重複計算(recomputation)。

11-4 動態規劃技巧

- 基本上，動態規劃技巧有三個主要部分：
  - 遞迴關係(recurrence relation)
  - 列表式運算(tabular computation)
  - 路徑迴溯(traceback)。
- 我們以「最長共同子序列」(Longest Common Subsequence；LCS)問題為例來談談這些特性。

11-4 動態規劃技巧

- 首先先解釋什麼是子序列(subsequence)，所謂子序列就是將一個序列中的一些(可能是零個)字元去掉所得到的序列，例如：pred、sdn、predent等都是president的子序列。
- 給定兩序列，最長共同子序列(LCS)問題是決定一個子序列，使得：
  - 該子序列是這兩序列的子序列；
  - 它的長度是最長的。

11-4 動態規劃技巧

- 當然最長共同子序列不一定是唯一，現在來探討如何找出其中一個最長的子序列，讀者們應能將此方法擴充為找出所有最長共同子序列的方法：

序列一：president
序列二：providence
它的一個LCS為 priden

序列一：algorithm
序列二：alignment
它的一個LCS為 algm

11-4 動態規劃技巧

- 給定兩序列A = $a_1a_2...a_m$及B = $b_1b_2...b_n$，令len(i,j)表示$a_1a_2...a_i$與$b_1b_2...b_j$的LCS之長度，則下列遞迴關係可用來計算len(i, j)：

$$len(i,j)=\begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ len(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } a_i = b_j \\ \max(len(i,j-1),len(i-1,j)) & \text{if } i,j>0 \text{ and } a_i \neq b_j \end{cases}$$

11-4 動態規劃技巧

- 遞迴關係：當某個序列是空序列時，LCS的長度為0；當$a_i=b_j$時，我們將$a_1a_2...a_{i-1}$及$b_1b_2...b_{j-1}$的LCS長度再加上1即可(因為最後的字元相同，可使LCS的長度增加1。)
- 當$a_i \neq b_j$時，這兩個字元不可能配對來貢獻給LCS，所以我們取$a_1a_2...a_i$與$b_1b_2...b_{j-1}$的LCS或$a_1a_2...a_{i-1}$與$b_1b_2...b_j$的LCS等這兩者中較長的一個作為目前LCS的長度。

11-4 動態規劃技巧

- 值得注意的是：len(m,n) 為 $A=a_1a_2...a_m$ 及 $B=b_1b_2...b_n$這兩個序列的LCS之長度。
- 我們可直接用程式語言中的遞迴呼叫(recursive call) 來計算 len(i,j)，但這需要 exponential time(那是很長很長的時間)；而我們若以動態規劃技巧來計算len(i,j)，則在與m×n成常數正比的時間內，我們就能算出len(m,n)。

11-4 動態規劃技巧

- 現在讓我們以**虛擬碼**(pseudo code)寫成的程序LCS-Length來說明如何用**列表式運算**(tabular computation)來算出序列A與序列B的LCS長度，在計算過程中，我們也記錄了最佳長度的貢獻者，以便稍後能藉由**路徑回溯**(traceback)找出LCS。

11-4 動態規劃技巧

procedure *LCS-Length(A, B)*

1. **for** $i \leftarrow 0$ **to** m **do** $len(i,0) = 0$
2. **for** $j \leftarrow 1$ **to** n **do** $len(0, j) = 0$
3. **for** $i \leftarrow 1$ **to** m **do**
4. **for** $j \leftarrow 1$ **to** n **do**
5. **if** $a_i = b_j$ **then** $\begin{bmatrix} len(i,j) = len(i-1, j-1)+1 \\ prev(i,j) = \text{"}\nwarrow\text{"} \end{bmatrix}$
6. **else if** $len(i-1, j) \geq len(i, j-1)$
7. **then** $\begin{bmatrix} len(i,j) = len(i-1, j) \\ prev(i,j) = \text{"}\uparrow\text{"} \end{bmatrix}$
8. **else** $\begin{bmatrix} len(i,j) = len(i, j-1) \\ prev(i,j) = \text{"}\leftarrow\text{"} \end{bmatrix}$
9. **return** *len* and *prev*

LCS-Length計算序列A及序列B的LCS之長度，並記錄最佳值的由來

11-4 動態規劃技巧

- 在LCS-Length中，我們依序由小的i和j算起(這是一種bottom-up的算法)，並以prev陣列來記錄最大值的由來。
- 在談到如何藉由prev陣列做路徑迴溯前，先讓我們用一個例題來進一步說明LCS-Length。
- 假設兩序列為president及providence，下圖描述了LCSLength的運算過程。

11-4 動態規劃技巧

i \ j	0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0	0	0	0	0	0	0	0	0	0	0	0
1 p	0	1 ↖	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←
2 r	0	1 ↑	2 ↖	2 ←	2 ←	2 ←	2 ←	2 ←	2 ←	2 ←	2 ←
3 e	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	3 ↖	3 ←	3 ←	3 ↖
4 s	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	3 ↑	3 ↑	3 ↑	3 ↑
5 i	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↖	3 ←	3 ↑	3 ↑	3 ↑	3 ↑
6 d	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↖	4 ←	4 ←	4 ←	4 ←
7 e	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	5 ↖	5 ←	5 ←	5 ↖
8 n	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	5 ↑	6 ↖	6 ←	6 ←
9 t	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	5 ↑	6 ↑	6 ↑	6 ↑

LCS-Length計算president與providence的LCS

11-4 動態規劃技巧

- 如何藉由路徑回溯(traceback)將最長共同子序列(LCS)建構出來。
- 基本上，從(m,n)沿著prev所記錄的箭頭方向回溯，每當我們碰到斜角箭頭時，表示那個位置$a_i = b_j$，且它也是LCS的一部分，所以我們在往前回溯結束後(我們的回溯過程直到邊界為止)，還得將這個字符印出。

11-4 動態規劃技巧

- 下圖的程序Output-LCS說明了整個回溯過程，起始呼叫為Output-LCS(A, prev, m, n)。

procedure *Output-LCS(A, prev, i, j)*

1 **if** $i = 0$ **or** $j = 0$ **then return**

2 **if** $prev(i, j) =$ "↖" **then** $\begin{bmatrix} Output-LCS(A, prev, i-1, j-1) \\ \text{print } a_i \end{bmatrix}$

3 **else if** $prev(i, j) =$ "↑" **then** *Output-LCS(A, prev, i-1, j)*

4 **else** *Output-LCS(A, prev, i, j-1)*

Output-LCS程序可將整個LCS回溯出來

11-4 動態規劃技巧

j / i	0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0	0	0	0	0	0	0	0	0	0	0	0
1 p	0	1 ↖	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←
2 r	0	1 ↑	2 ↖	2 ←	2 ←	2 ←	2 ←	2 ←	2 ←	2 ←	2 ←
3 e	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	3 ↖	3 ←	3 ←	3 ↖
4 s	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	3 ↑	3 ↑	3 ↑	3 ↑
5 i	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↖	3 ←	3 ↑	3 ↑	3 ↑	3 ↑
6 d	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↖	4 ←	4 ←	4 ←	4 ←
7 e	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	5 ↖	5 ←	5 ←	5 ↖
8 n	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	5 ↑	6 ↖	6 ←	6 ←
9 t	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	5 ↑	6 ↑	6 ↑	6 ↑

Output-LCS的回溯路線序，深色陰影(priden)為LCS所在

11-5 計算難題

- 是不是所有的數位計算問題，我們都能找到有效的解答呢？
- 牛頓曾說：「假如我曾經看得更遠，那是因為站在巨人的肩膀上。」在前輩的耕耘下，有些問題已證明是無解的。
- 例如：判斷程式是否會停的問題 (halting problem)就可證明是無法解答的。

11-5 計算難題

- 而在可解的數位計算問題裡，很多都已依它的計算時間及記憶空間複雜度的難易做歸類了，最有名的歸類要算是NP-Complete問題了。
- 如果您的老闆交代您一個數位計算問題，您苦思多日仍無有效率的解法，您可以試著證明這問題是NP-Complete，然後告訴您的老闆說，即使全世界最厲害的電腦學家，也沒有這個問題的有效解法。

11-5 計算難題

- 是的，所有NP-Complete問題，目前都沒有有效的精確解法，而且只要有一個找到有效解法，那所有NP-Complete問題都有有效解法了。
- 至今已有數以萬計的問題被證明為NP-Complete，雖然大家幾乎都認為這類型的問題並不存在有效解法，但到現在都沒有人可以證明。

11-5 計算難題

- 「旅行推銷員問題」和「小偷背包問題」，看似簡單，但都已證明是NP-Complete。

有一個推銷員，要到各個城市去推銷產品，他希望能找到一個最短的旅遊途徑，訪問每一個城市，而且每個城市只拜訪一次，然後回到最初出發的城市。如果只有幾個城市要訪問，我們很快就可以找出一個最短的旅遊途徑，但如果有很多很多的城市要訪問時，那就會難倒目前所有的數位計算機了。
這問題的關鍵在於：當我們要拜訪很多城市時，可能的拜訪順序組合是天文數字，而我們至今又沒有好的方法，可以快速決定最短的旅遊途徑。

旅行推銷員問題

11-5 計算難題

有個小偷，光顧一家超級市場，他帶了一個背包來裝所偷的東西，假設他的背包最多只能裝三十公斤，而超市內的每樣東西有它的重量及價值，小偷背包問題是要找出最佳的偷法，使得背包內所裝的贓物總價值最高，且總重量又不超過三十公斤。這樣的一個問題居然也是難題！如果小偷用數位計算機來替他決定最好的偷法，在他得到答案前，可能早就被繩之以法了。

小偷背包問題

11-5 計算難題

- 不僅如此，我們還可證明，小偷背包問題和旅行推銷員問題的精確解法，它們的難度是一樣的，也就是說，只要其中有一個存在有效率的精確解法，另一個也會存在有效率的精確解法。
- 實際上，在數位計算世界裡，已有數以萬計的問題，被證明為和旅行推銷員問題同樣難度，真是不可思議吧！

11-5 計算難題

- 如果您的老闆交代您一個數位計算問題，您苦思多日仍無有效率的解法，您可以試著證明它和旅行推銷員問題同樣難度，然後告訴您的老闆說，即使全世界最厲害的資訊學家，也沒有這個問題的有效解法，這樣就不會被炒魷魚喔！
- 面對這一類型難題，我們是否真的束手無策呢？

11-5 計算難題

- 十幾年前，如果證明某一個問題是屬於這一類型的問題，那可算是一篇精采的博士論文。可是現在如果找到了新的難題，那還不夠，還必須提出好的近似解法(在有效率的時間內，找到和最佳解答差不多的近似答案) 。
- 很有意思的是，雖然我們說小偷背包問題和旅行推銷員問題的精確解法一樣難，但它們的近似解法卻南轅北轍。

11-5 計算難題

- 我們可以證明，旅行推銷員問題不太可能存在好的近似解法；而小偷背包問題卻已有很好的近似解法，這也難怪很少有小偷在超市當場被抓包囉。
- 高難度的數位計算問題，是演算法專家最重要的食糧。各式各樣的難題，雖然令人費盡心思，但它的滋味就如同山珍海味。如果沒有問題傷腦筋，那才真是傷腦筋的問題呢！

IT 魔 法 小 百 科

微軟總裁比爾蓋茲在哈佛大學休學前，曾發表過一篇頗具深度的科學論文。趙老勉勵某位大三學生：「微軟總裁比爾蓋茲像你這年紀時，就已寫了一篇好論文！」

該學生投桃報李：「微軟總裁比爾蓋茲像您這年紀，已是世界上最有錢的人了！！」