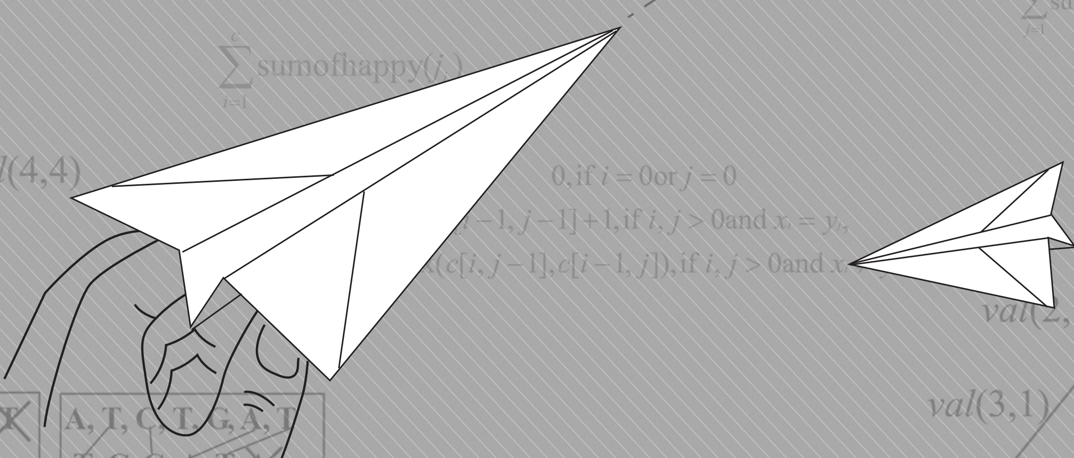
3 動態規劃

章節大綱

「岱宗如何」是泰山派劍法的絕藝，要旨不在右手劍招，而在左手的算數。左手不住屈指計算，算的是敵人所處方位、武功門派、身形長短、兵刃大小，以及日光所照高低等等，計算極為繁複，一經算準，挺劍擊出，無不中的。

金庸　笑傲江湖



3.1 何謂動態規劃？

「什麼是動態規劃 (dynamic programming)？」

簡言之：「計算並儲存小問題的解，並將這些解組合成大問題的解」

當大問題的解，可利用小問題的解組合計算得知時，若將可能用到的小問題之解，先算出並儲存起來，將縮短計算大問題之解的時間。

乍看之來，動態規劃還蠻暴力的。使用動態規劃常經過繁瑣計算後，最後一舉解決問題，有一點點像金庸小說笑傲江湖中的絕招「岱宗如何」。但是需注意的是，小問題之解一旦被計算出並儲存後，就不會再被重複計算。當動態規劃運用得宜時，可有效率地解決問題。

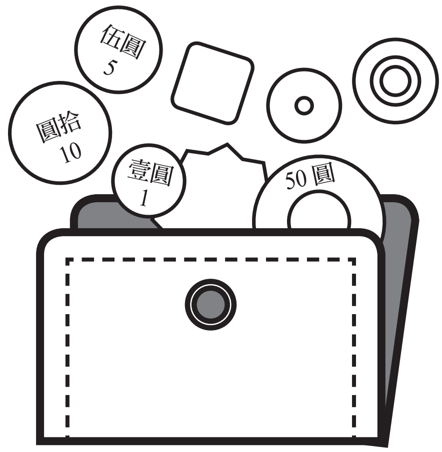
3.2 換零錢

第一個例子是換零錢問題，問題描述如下表。

表 3.1 換零錢問題

問題	老王是一位富翁擁有無限個硬幣，但硬幣種類有限。老王想帶一定金額的零錢 n 去旅行，但為了方便攜帶，希望所帶硬幣的個數愈少愈好。請寫個程式幫他計算，最少需帶多少枚硬幣?
輸入	硬幣面額 $c_1c_2\cdots c_k$ 及零錢的金額(正整數) n 硬幣種類{33, 24, 12, 5, 1}, 零錢的金額 n=36
輸出	所需攜帶最少的硬幣數目 2

任意給一個金額 n，有時不一定可用有限種類的零錢湊齊。例如硬幣種類為{33, 24, 12, 5}時，就湊不出 14 元。但若是當硬幣種類為{33, 24, 12, 5, 1}時，就可以湊出 14 元，甚至所有的正整數(因為有無限個一元的關係)。



如果換零錢的策略為：盡量先兌換大金額的零錢，則會發現這個方法(有時)並不能換到最少的硬幣數目。如表 3.1 的例子，就會換成四個硬幣(36=33+1+1+1)而非最佳的兩個(36=24+12)。

「怎樣才可以使用最少硬幣，組合出總和為 n 的金額呢？」

「所需最少的硬幣的數目是多少?」

「不知道。」

「假想我們知道最佳(少)的硬幣組合，猜想其中的任一個硬幣的面額會是多少?」

「應該是所有硬幣種類其中之一。」

「扣掉此硬幣，剩下的金額為多少?」

「剩下的金額應該是 n 扣掉此硬幣的面額。」

「扣掉此硬幣，剩下所需的硬幣數為多少?」

「不知道。好像是同一個問題但金額小一點。若是知道如何用最少硬幣數組合出此金額就好了。」

最後一句話暗示：「若是知道較小問題的最佳解，將有助於解決大問題」。依表 3.1 中換零錢問題的範例，有以下五種可能：

(1) 36 元是一個 33 元硬幣 和 3 元 所組成的，或是

(2) 36 元是一個 24 元硬幣 和 12 元 所組成的，或是

(3) 36 元是一個 12 元硬幣 和 24 元 所組成的，或是

(4) 36 元是一個 5 元硬幣 和 31 元 所組成的，或是

(5) 36 元是一個 1 元硬幣 和 35 元 所組成的。

「為什麼是五種?」

「因為共有五種硬幣。」

「在五種中，那一種需要的硬幣數最少?」

「不知道。因為不清楚 3, 12, 24, 31, 35 這些金額需要的最少硬幣數。」

「可以事先計算出 3, 12, 24, 31, 35 分別需要多少硬幣?」

從以上五種可能的兌換中，挑選所需硬幣數最少的一種，即是最佳換零錢方式。但是，需要事先計算出這五種金額的最少幣數。

根據上述討論，若讓函數 $f(n)$ 代表組合金額為 n 所需要最少的硬幣數，則下列等式是正確的。

```
f(36)=min{1+f(3), 1+f(12), 1+f(24), 1+f(31), 1+f(35))
```

此處 *min* 是自一集合中選取最小值 (minimum value) 之意。

動態規劃的技巧就是，為了計算 $f(36)$，需事先利用類似的方式計算 $f(3)$, $f(12)$, $f(24)$, $f(31)$, 及 $f(35)$。以下以 $f(31)$ 及 $f(35)$為例說明。因為 31=24+7=12+19=5+26=1+30，所以 $f(31)=min\{1+f(7),\ 1+f(19),\ 1+f(26),\ 1+f(30)\}$。為了計算$f(31)$，需利用類似的式子，事先計算出$f(7)$, $f(19)$, $f(26)$, 及$f(30)$。同樣地，因為 35=33+2=24+11=12+23=5+30=1+34，所以 $f(35)=min\{1+f(2),\ 1+f(11),\ 1+f(23),\ 1+f(30),\ 1+f(34)\}$。為了計算$f(35)$，需利用類似的式子，事先計算出$f(2)$, $f(11)$, $f(23)$, $f(30)$ 及 $f(34)$。

總而言之，下列的兩個式子顯然是正確的。

(1) $f(0)=0$;

(2) $f(n)=1+min\{f(n-c_1),\ f(n-c_2),\ \cdots,\ f(n-c_k)\}$。

當 $n>c_i(1<i<k)$且硬幣面額為 c_1 或 $c_2\cdots$或 c_k。

動態規劃的技巧會將 $f(n-c_1)$, $f(n-c_2)$, $\cdots$, $f(n-c_k)$ 先行計算後儲存起來，再依上述的式子計算出 $f(n)$ 的值 (即找出 $f(n-c_1)$, $f(n-c_2)$, $\cdots$, $f(n-c_k)$ 其中的最小值後，再加上 1)。表 3.2 列出換零錢問題的演算法。

表 3.2 換零錢問題的演算法

輸入	硬幣面額種類 $c_1c_2\cdots c_k$ 及欲攜帶零錢的金額 n (正整數)
輸出	組合金額總和為 n 之最少硬幣數目

步驟	Algorithm coin_changing
	{
	Step 1: 當 z 為 0 或負數時，令 $f(z)=0$。
	Step 2: for k=1 to n
	{
	計算 $f(k)=1+min\{f(k-c_1),\ f(k-c_2),\ \cdots,\ f(k-c_k)\}$。
	}
	}

換零錢問題的演算法，在於利用一個迴圈及前頁中間的式子(2)，先計算比 n 小的所有金額的解(即該金額所需的最少硬幣數)。最後利用這些解，再計算出金額 n 的解。

3.3 數字金字塔

第二個例子是數字金字塔(表 3.3)。

表 3.3 數字金字塔

問題	老王到埃及看金字塔，想從金字塔底向上走到頂。可是每一條路所花的時間都不一樣長。請幫他找到登上金字塔最快的路(所費時間為路徑上數字的總和)。注意每一個向上的路徑只有左上和右上的兩條路
輸入	疊成金字塔的正整數 (注意金字塔的高度不固定) 45 20 33 34 18 30 14 45 09 11
輸出	從任一處底部走到金字塔最頂端最快的路徑 92(=45+20+18+9)，路徑如下圖 45 20 33 34 18 30 14 45 09 11

「這一題可用動態規劃來解嗎?」

「怎樣判斷一個問題可不可以用動態規劃來解?」

「為何換零錢問題可以被成功地解決?」

「利用較小問題的最佳解，來組成大問題的最佳解。」

「這題如果要使用同樣的技巧，下一步要作什麼?」

「找出小問題的最佳解和大問題的最佳解之間的關係。」

3.3.1 找出大問題和小問題之間的關係

弄清楚大問題和小問題之間的關係，顯然是解決本題的重要關鍵。如圖 3-1，抵達金字塔頂點(45)的路只有兩種可能：(A)經過 20 或(B)經過 33 之路徑。

一旦由底部走到 20 和由底部走到 33 之最快路徑都事先被找到，只要從中選一條最快的路，直接登上頂端(45)就是最佳答案了。如果事先計算得知，經過 20 之最快路徑(9→18→20)所費時間為 47，而也知道經過 33 之最快路徑(9→18→33)所費時間為 60，顯然經過 20 的路徑是兩條中較小者。也就是，抵達金字塔頂點(45)最快的路是一條先經過 20 的路徑。

因此，抵達大金字塔頂點 45 最快的路，是從左邊小金字塔頂點 20 最快的路和右邊小金字塔頂點 33 最快的路中，選最短的。此乃大問題之最佳解和小問題之最佳解之間的關係。

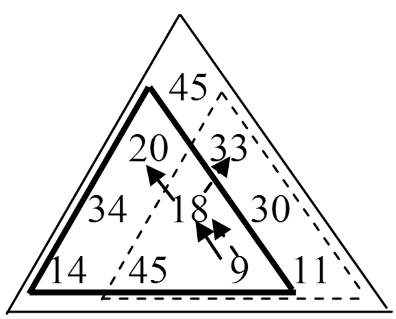


圖 3.1 左右數字金字塔的解有助於找到
大數字金字塔(以 45 為頂)的解

「如何知道小一些的數字金字塔的最佳解？」

「只要知道更小一些的數字金字塔的最佳解，將有助於解決這個問題。」

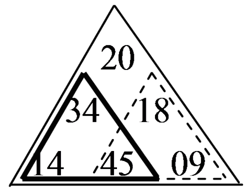


圖 3.2 左右更小數字金字塔的解有助於找到數字金字塔(以 20 為頂的)的解

如圖 3.2 所示，左右更小數字金字塔的解，有助於找到數字金字塔(以 20 為頂的)的解。顯然，左邊小數字金字塔的解是(14→34)這條路(總和 48)，而右邊小數字金字塔的解是(9→18)這條路(總和 27)。比較大小之後，抵達金字塔頂點 20 最快的路是(9→18→20)這條路。

因此，解決數字金字塔的方法，可有系統地計算並儲存所有小問題的解，並將這些解組合成大問題的解。

3.3.2 數字金字塔問題的資料結構

以下討論所需要的兩個資料結構。

一、矩陣變數 $val(x, y)$ 儲存數字金字塔第 x 層第 y 個元素的值(value)。

例如，如圖 3.3 $val(3, 2)$儲存數字 18。

$$
\begin{array}{cccc}
 & val(1,1) & & \\
 & val(2,1) & val(2,2) & \\
val(3,1) & val(3,2) & val(3,3) & \\
val(4,1) & val(4,2) & val(4,3) & val(4,4)
\end{array}
$$

圖 3.3 矩陣 $val(x, y)$儲存數字金字塔第 x 層第 y 個元素的值

二、矩陣變數 $sum(x, y)$儲存從底層走到以第 x 層第 y 個元素為金字塔頂端的最快路徑值。

例如，在圖 3.4 中， $sum(3, 2)$儲存以 $18=val(3, 2)$，$45=val(4, 2)$及 $9=val(4, 3)$三數形成的金字塔(即圖 3.2 中的右三角形)的最快路徑的值(即 $sum(3, 2)=18+9=27$)。換句話說，$sum(x, y)$儲存所有大小金字塔問題的最佳解。

$$
\begin{array}{c}
sum(1,1) \\
sum(2,1) \quad sum(2,2) \\
sum(3,1) \quad sum(3,2) \quad sum(3,3) \\
sum(4,1) \quad sum(4,2) \quad sum(4,3) \quad sum(4,4)
\end{array}
$$

圖 3.4 矩陣 $sum(x, y)$代表從底層走到以第 x 層第 y 個元素為金字塔頂端的最快路徑值

根據以上的討論，下列的式子顯然是正確的。這裡，副程式 $\min(a, b)$傳回 a, b 中較小值。

```
sum(x, y)=val(x, y)+min(sum(x+1, y), sum(x+1, y+1)) if x 不是底層。
sum(x, y)=val(x, y) if x 是底層。
```

Tip 注意上述這個式子準確地描述大問題和小問題之間的關係。

最後，注意在計算 $sum(x, y)$ 值時，可以從底層開始向上計算，即採取由下而上(bottom-up)的方式。理由很簡單，因為上一層的值 $sum(x, y)$需要利用下一層的值 $sum(x+1, y)$及 $sum(x+1, y+1)$來計算得出，所以下一層的值需要先被計算並儲存。如此，利用動態規劃來解數字金字塔的演算法，就可被設計出來(表 3.4)。

表 3.4 數字金字塔的演算法

輸入	疊成金字塔的正整數儲存於 *val*(*x*, *y*)
輸出	從任一個底部走到金字塔最頂端的最快的路徑(所費時間為路徑上數字的總和)
步驟	見下方程式

```
/* max 代表金字塔的層數*/
Algorithm pyramid
{
  for x ← max to 1 step -1             /*由下層開始依次計算*/
  for y ← 1 to max step +1             /*同一層由左向右計算*/
  {
    if x=max 則{sum(x, y)=val(x, y);}   /*最底層無需計算*/
    else {
           sum(x, y)=val(x, y)+min(sum(x+1, y), sum(x+1, y+1));
         }
  }
  輸出 sum(1, 1);
}
```

3.4 最長相同子字串

利用動態規劃的技巧，我們解決了前兩個問題。但是，這個技巧的主要精神到底是什麼呢？以下這句話，至為關鍵。

『利用小問題的最佳解，來組成大問題的最佳解。』

動態規劃得以解決問題的三個步驟，應該是：

一、此問題的 **大問題的最佳解** 可以利用 **小問題的最佳解** 求之。

二、接著，可以利用一個數學式子，將 **大問題的最佳解** 和 **小問題的最佳解** 之間的關係，清楚地表達。

三、最後，系統地，先將 **小問題的最佳解** 先計算出後儲存，再利用它們算出 **大問題的最佳解**。

接下來介紹一個可應用於生命科學上的有趣問題：***最長相同子字串***(the longest common subsequence)。並展現如何遵循這三個步驟，就可以設計出一個動態規劃的演算法。

表 3.5 最長相同子字串

問題	老王找到失散多年的兄弟。為確定兩人的血緣關係，他決定將兩人的基因作比對，請寫一個程式比較兩組基因，並找到此兩組基因(字串)共同擁有的基因組(最長相同子字串)。注意存在此兩組基因的子字串，其先後順序須一致
輸入	兩組字串 $X=<x_1, x_2, \cdots, x_m>$ $Y=<y_1, y_2, \cdots, y_n>$ X= <A, T, C, T, G, A, T> Y= <T, G, C, A, T, A>
輸出	最長相同子字串 $Z=<z_1, z_2, \cdots, z_k>$及其長度。此處所有 $z_i(1 \le i \le k)$需同時出現於 X 及 Y 中，且前後出現的順序不變 Z= <T, C, T, A>，4

以下小節將遵循上述的三個步驟，來設計演算法。

3.4.1 步驟(一)：檢查是否大問題的最佳解，可以利用小問題的最佳解求之。

若用表 3.5 中的範例說明第一步驟，即是問：

> **「X= <A, T, C, T, G, A, T> 及 Y= <T, G, C, A, T, A>的最長相同子字串，可否透過小一些的最長相同子字串得之？」**

答案可以從下面兩個稍小問題中的解，擇其一得之。

(1) X^*=<A, T, C, T, G, A> 及 Y = <T, G, C, A, T, A> 的最長相同子字串(即<T, C, T, A>)，或

(2) X=<A, T, C, T, G, A, T> 及 Y^*= <T, G, C, A, T> 的最長相同子字串(即<T, C, A, T>)。

這是因為 X= <A, T, C, T, G, A, T> 及 Y= <T, G, C, A, T, A>的最長相同子字串，只有兩種可能：(1)沒有 X 最後的字母 T 或 (2)沒有 Y 最後的字母 A(圖 3.5)。理由是：最長相同子字串必須同時出現在兩個字串中。而此範例中的 X 及 Y 尾部的字母不同(即 "T" $\neq$ "A")，故 X 及 Y 的最長相同子字串，不可能同時擁有 X 及 Y 的尾部。

若事先計算出上述兩個(即(1)及(2))最長相同子字串的答案，則其中較長的相同子字串即為 X 及 Y 的最長相同子字串。此大問題的最佳解包含小問題的最佳解之性質，稱為***最佳子結構***(optimal substructure)。

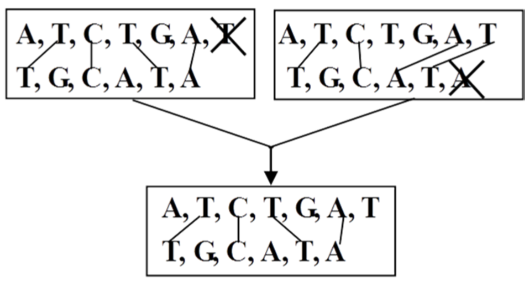


圖 3.5 最長相同子字串問題的最佳子結構性質(當字尾不同時)

另一方面，當兩字串的字尾相同時，也可透過小一些的最長相同子字串得之。例如，當 X=<A, T, C, T, G, A>及 Y =<T, G, C, A, T, A>時，因為 X 及 Y 尾部的字母相同，故可先找出 X^*=<A, T, C, T, G >及Y^*= <T, G, C, A, T>的最長相同子字串<T, C, T>，再於尾部加上共同擁有的 "A"，就可以得到其最長相同子字串為<T, C, T, A>，如圖 3.6 所示。此時大問題的最佳解(即<T, C, T, A>)一樣包含小問題的最佳解(即<T, C, T>)。

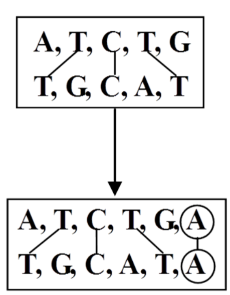


圖 3.6 最長相同子字串問題的最佳子結構性質(當字尾相同時)

Tip 注意此最佳子結構的性質，確認了動態規劃可以解決最長相同子字串問題。

3.4.2 步驟(二)：利用數學式子描述大問題的最佳解和小問題的最佳解之關係

步驟(一)確認了大問題的最佳解包含了小問題的最佳解。如此才有機會，利用小問題的最佳解，來組合拼湊出大問題的最佳解。接下來，步驟(二)便是利用數學式子，將此關係表達清楚。此式子也將在步驟(三)引導著最後解的計算。

當考慮兩個字串 X 及 Y 的最長相同子字串問題時，我們利用 $c[i, j]$ 代表(並儲存)其相關小問題的最佳解的長度。準確地說，$c[i, j]$ 代表字串 X 中前面 i 個字元(即<x_1, x_2, …, x_i>)和字串 Y 中前面 j 個字元(即<y_1, y_2, …, y_j>)所形成之最長相同子字串的長度。

例如，當 X=<A, T, C, T, G, A>及 Y =<T, G, C, A, T, A>時，$c[2, 2]$ 記錄<A, T>和<T, G>的最長相同子字串的長度(即 $c[2, 2]=1$); 而 $c[3, 4]$ 則是記錄<A, T, C>和<T, G, C, A>的最長相同子字串的長度(即 $c[3, 4]=2$)。

當然，很容易地我們可得 $c[i, 0]=c[0, j]=0$ 因為其中的一個字串是空的。當 X 及 Y 的字串字尾相同時，我們可得 $c[i, j]=c[i-1, j-1]+1$(圖 3.6 就是一個例子)。相對地，當 X 及 Y 的字串字尾不相同時，$c[i, j]=\max(c[i-1, j], c[i, j-1])$(如圖 3.5 所示)。以上關係可以整理如下。

$$c[i,j]=\begin{cases} 0, \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1, \text{if } i,j>0 \text{ and } x_i=y_j, \\ \max(c[i,j-1],c[i-1,j]), \text{if } i,j>0 \text{ and } x_i\neq y_j. \end{cases}$$

這個數學式子，再次指出 $c[i, j]$ 的值是可以利用三個小問題的值 $c[i-1, j-1]$, $c[i, j-1]$ 及 $c[i-1, j]$ 所計算出的。這個關係可協助最後一個步驟的設計。

3.4.3 步驟(三)：設計演算法，使得在計算出大問題的最佳解之前，其所需要的所有小問題的最佳解，皆已被事先計算並儲存

最後，在設計一個動態規劃演算法時，需小心安排計算最佳解的順序；即當計算 $c[i, j]$時，其可能需要的三個值 $c[i\text{-}1, j\text{-}1]$, $c[i, j\text{-}1]$及 $c[i\text{-}1, j]$已被事先計算出來並儲存。如圖 3.7 所示，當填每個矩陣中某一格時，須將其上方、左方及左上方的值先填好。

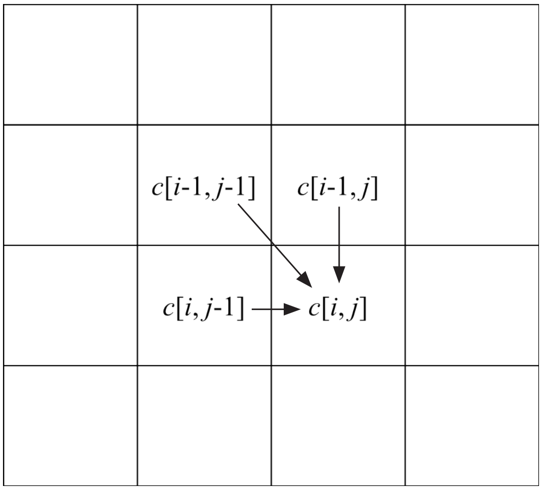


圖 3.7 計算 $c[i, j]$時，其所需的三個值 $c[i\text{-}1, j\text{-}1]$(左上方)，$c[i, j\text{-}1]$(左方)及 $c[i\text{-}1, j]$ (上方)須被事先計算出來。此圖中的箭頭指示計算最佳解的順序

因此，此演算法可以依照由左而右及由上而下的順序，計算出所有的 $c[i, j]$，如圖 3.8 所示。表 3.6 也列出最長相同子字串的演算法。

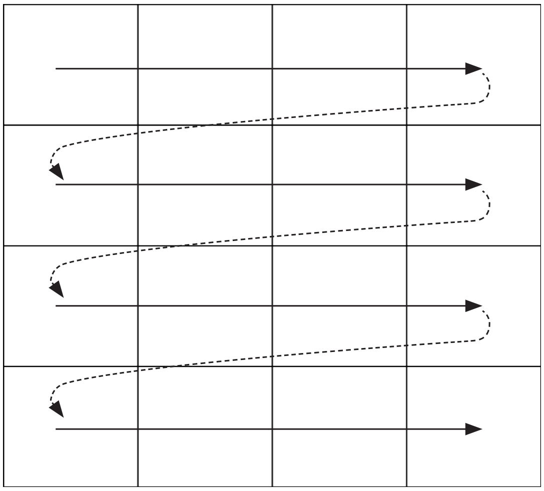


圖 3.8 動態規劃演算法須安排計算最佳值的順序

依照這個方法，當 X=<A, T, C, T, G, A, T>及 Y=<T, G, C, A, T, A>時的所有 $c[i, j]$的值也可計算得出(圖 3.9)。

		T	G	C	A	T	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
T	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
T	0	1	1	2	2	3	3
G	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
T	0	1	2	2	3	4	4

圖 3.9 當 X=<A, T, C, T, G, A, T>及 Y =<T, G, C, A, T, A>時的所有 $c[i, j]$ 的值

表 3.6 最長相同子字串的動態規劃演算法

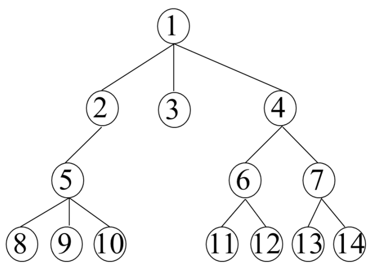
輸入	兩組字串 X=<x_1, x_2, ⋯, x_m>, Y=<y_1, y_2, ⋯, y_n>
輸出	最長相同子字串 Z=<z_1, z_2, ⋯, z_k>的長度
步驟	(見下方程式)

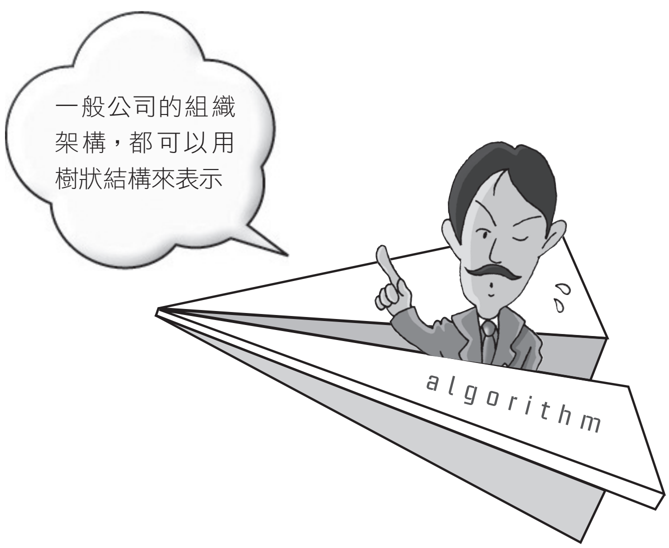
```
Algorithm lcs
{
Step 1: 令 len1，len2 分別為字串 X 及 Y 的長度。
Step 2: 利用兩層迴圈計算出所有的 c[i, j]。
        for(i = 0;i <= len1;i++)             /*由上而下*/
        {
           for(j = 0;j <= len2;j++)          /*由左至右*/
           {
             if(i != 0 && j != 0)
             {
               if(str1[i] == str2[j])       //當字元相等時
                      c[i][j] = c[i-1][j-1] + 1;
               else //當字元不相等時
               {
               if(c[i][j-1] >= c[i-1][j])  //取長度大者
                      c[i][j] = c[i][j-1];
               else
                      c[i][j] = c[i-1][j];
               }
             }
             else
               c[i][j] = 0;            //當首行或首列時設定為 0
           }
        }
Step 3: 將最大子字串的長度 c[len1][len2]輸出;
}
```

3.5 安排公司聚會

一個問題是否可用動態規劃來解，有時不易一眼看出。下一個例子讓我們專注於此判斷。

表 3.7 安排公司聚會問題

<table>
<tr><td>問題</td><td>老王為公司員工安排一個非正式的聚會。公司的組織如同一棵樹，董事長位於樹根。為了使整個聚會成功，人事室利用分數評估每位員工為聚會帶來快樂的程度。另外，為使聚會有愉悅的氣氛，員工和其直屬上司不會同時參加。請設計一個演算法，幫老王安排聚會名單，使得此次聚會的快樂分數其加總最大</td></tr>
<tr><td>輸入</td><td>矩陣 happy[n]代表 n 個員工的產生快樂的分數及矩陣 children[n, n]儲存公司的樹狀組織架構， children[i, j]=1 代表 i 是 j 的直屬上司。n=14

</td></tr>
<tr><td>輸出</td><td>參與聚會的最佳名單(即{1, 2, …, n}的子集合)，使得整體快樂分數加總最大</td></tr>
</table>



「安排公司聚會問題，可用動態規劃來解嗎?」

「應該如何判斷？」

「應該先判斷此問題是否有最佳子結構性質。」

「什麼是最佳子結構性質？」

「檢查是否大問題的最佳解，可以利用小問題的最佳解組合得到。」

「若 n 位員工的公司是大問題，則小問題應是什麼?」「怎樣的小問題的最佳解，對解大問題有幫助？」

「若已知 n-1 位員工的小問題的最佳解，好像對找到 n 位員工大問題的解幫助不大?」

「為什麼？」

「因為公司的組織架構未被考慮；即員工和其直屬上司，不會同時參加聚會的特性。」

「公司的組織架構為何？」

「長得像一棵樹(tree)。」

「一棵樹的小問題可能是甚麼？」

「應該是小樹或子樹(subtree)吧！」

「哪些子樹的解對找到大樹的最佳解有幫助？」

「嗯…」

「大樹和子樹的關係是什麼？」「樹的定義是什麼？」

「一個樹根(root)和其子女(children)為樹根的子樹所組合而成。」

3.5.1　安排公司聚會問題的最佳子結構判斷

以表 3.7 的公司組織為例，1 號員工(即董事長)有兩種可能，即「參加」和「不參加」聚會。

倘若 1 號員工不參加，則以 2 號員工為首的部門(即以 2 為樹根的小樹)的最佳解(即 2 號員工的部門中最大快樂的聚會名單)，同時配合以 3 號員工為首的部門(即以 3 為樹根的小樹)的最佳解，再配合 4 號員工所帶頭的部門 (即以 4 為樹根的小樹)的最佳解，即為整個公司的最佳解(最佳聚會名單)，如圖 3.10 中三個虛線方塊所示。

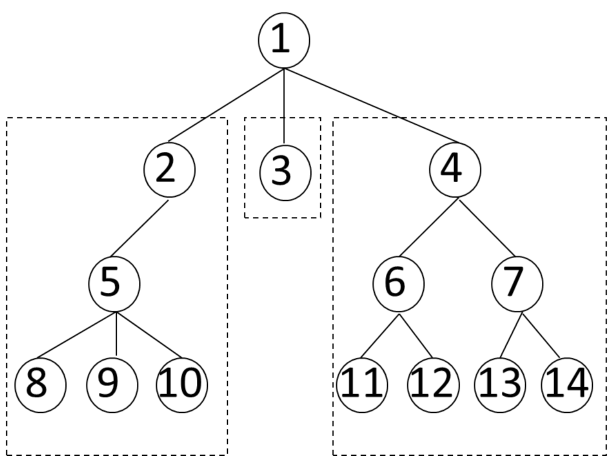


圖 3.10　當 1 號員工不參加聚會時的公司聚會問題

若 1 號員工參加，則 2, 3, 4 號員工不可參加。但是，此時 1 號就可配合 5 號(和 6, 7 號)員工所帶頭的下屬(即以 5, 6, 7 為樹根的小樹)的最佳解，即可得最佳聚會名單，如圖 3.11 所示。

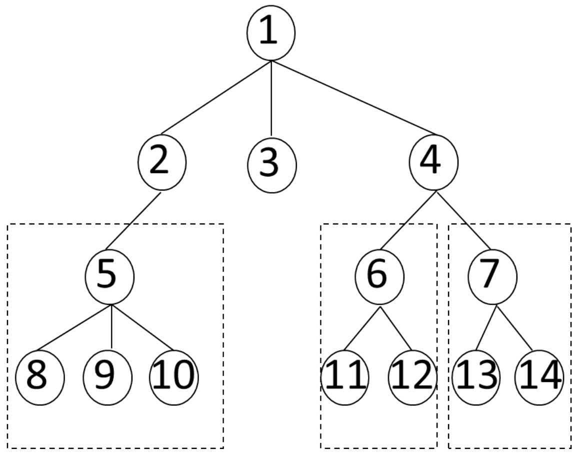


圖 3.11 當 1 號員工參加聚會時的公司聚會問題

從以上這兩種(即 1 號員工參加或不參加)可能聚會名單的快樂分數中，取其大者，即可決定最佳的聚會名單。

如此，我們用小問題(以 2, 3, 4, 5, 6, 7 為樹根的小樹)的最佳聚會名單，找到大問題(以 1 為樹根的大樹)的最佳聚會名單。因此，安排公司聚會問題俱備了最佳子結構性質，適合動態規劃法解之。

3.5.2 安排公司聚會問題的遞迴關係

接下來，進行下一個步驟：利用一個數學式子，來描述大問題的最佳解和小問題的最佳解的關係。

倘若以 v 為根的部門的最大快樂分數，儲存於 sumofhappy(v)中。當 v 是樹葉(leaf)時，顯然 sumofhappy(v)=*happy*(v)。

當 v 不是樹葉時，則必有一個以上的兒子。令其所有兒子為$\{j_1, j_2, \cdots, j_c\}$並令 v 的所有孫子(即其所有兒子的兒子)為 $\{g_1, g_2, \cdots, g_d\}$。根據之前的討論，則下列的式子，可以描述大問題的最佳解和小問題的最佳解的關係。

$$\text{sumofhappy}(v)=\mathit{Max}\{\text{sumofhappy}(j_1)+\text{sumofhappy}(j_2)+\cdots+\text{sumofhappy}(j_c),\ \mathit{happy}(v)+\text{sumofhappy}(g_1)+\text{sumofhappy}(g_2)+\cdots+\text{sumofhappy}(g_d)\}。$$

上列式子也可改寫成：

$$\text{sumofhappy}(v)=\mathit{Max}\{\sum_{i=1}^{c}\text{sumofhappy}(j_i),\ \mathit{happy}(v)+\sum_{j=1}^{d}\text{sumofhappy}(g_j)\}。$$

3.5.3 安排公司聚會問題的計算順序

進行最後的步驟：設計演算法，使得在計算出大問題的最佳解之前，其所需要的所有小問題的最佳解，皆已被事先計算出。

首先，因為安排公司聚會問題，考慮樹狀組織架構，需要一些變數來儲存樹(組織)的資料。令 v 是此公司的樹狀組織的內部節點(internal node)，T_v 則代表以 v 為樹根的樹。以圖 3.10 為例，節點 1 是整棵樹的根(root)，T_1 代表整個公司的大樹。節點 2, 3, 4 是節點 1 的三個兒子，而 T_2 T_3 T_4 分別代表以 2, 3, 4 為樹根的三棵小樹。

用來描述此演算法的若干變數，說明如下：

- *happy*[v]：每一個節點 v 將其快樂指數存於 *happy*[v]中。
- *sumofhappy*[v]：以 v 為根的部門(小樹)的最大快樂指數。
- *children*[v, j]：若 *children*[v, j]回傳 true 則代表節點 v 為節點 j 的父節點。

表 3.8 中的演算法可以計算出最佳聚會清單的快樂指數總和，並存入 *sumofhappy*[1] 中。只要稍加修改，亦可找出最佳聚會清單。

表 3.8 安排公司聚會的動態規劃演算法

<table>
<tr><td>輸入</td><td>矩陣 happy[n]代表 n 個員工的產生快樂的分數
公司的樹狀組織架構 children[n, n]</td></tr>
<tr><td>輸出</td><td>參與聚會的最佳名單(即{1, 2, …, n}的子集合)的整體快樂分數 sumofhappy[1]</td></tr>
<tr><td>步驟</td><td><pre>
Algorithm getSum(int node) /*node 的初始值為 1*/
{
for(i = 1;i <= N;i++)   /*檢查是否此點是一個樹葉*/
{
    if(children(node, i))   /*這個節點不是樹葉*/
    {
       getSum(i);   /*將子節點做為父節點處理*/
       isLeaf= 0;   /*因為不是樹葉，所以標成 NULL*/
    }
}

    if(isLeaf)   /*此點是樹葉時*/
    {
    sumofhappy[node] = happy[node];   /*記錄樹葉的快樂分數*/
    }
    else   /*當此點不是樹葉時*/
    {
       sum2=happy[node];
       for(i = 1;i <= N;i++)
       {
          if(children(node, i))
          /* 若此點是某一點的父節點，則要將其值加入 sum1 */
          {
              sum1 += sumofhappy[i]; /*將子節點的分數加總*/
              for(j = 1;j <= N;j++)
              if(children(i, j)) sum2 += sumofhappy[j];
          }
       }
    if(sum1 >= sum2)   /*取得最高的快樂分數*/
          sumofhappy[node] = sum1;
    else
          sumofhappy[node] = sum2;
    }
}
</pre></td></tr>
</table>

「萬一董事長不可以參加公司聚會，這樣好嗎？」

「怎麼做，董事長一定會參加又不違反公司規定？」

「請人事室調高董事長(帶給員工)的快樂分數就好了。」

「這樣會不會太假了!」

「太棒了，就這麼辦。」

3.6 動態規劃的技巧

「如何判斷一個問題，可用動態規劃來解嗎?」

本章的四個範例，指示我們使用下列步驟，來思考並使用動態規劃技巧。

1. 判斷大問題的最佳解，是否可以利用(多個)小問題的最佳解組合並解出。
2. 若可以，嘗試寫出大問題最佳解及小問題最佳解之間的遞迴關係。
3. 最後，根據上述遞迴關係，設計演算法。先計算小問題最佳解，再計算出大問題的最佳解。

「何種問題使用 動態規劃 特別有效率？」

「動態規劃有什麼優點？」

「什麼是動態規劃？」

「計算並儲存小問題的解，並將這些解組合成大問題的解。」

「什麼是動態規劃的特色？尤其是在加速計算上的優點。」

「計算過小問題不會重算第二次。」

「如此有甚麼好處？」

「避免無謂的重複計算，以加快演算法求解時所需的時間呀！」

「怎樣可充份發揮動態規劃的優點？」

「如果計算過小問題，可被很多的大問題利用來求解的話，也許更可以發揮出動態規劃的優點。」

「怎樣可以知道，一個小問題的解可用於組合多少個大問題的解？」

「我好像寫過兩者之間的關係。啊！大問題最佳解及小問題最佳解之間的遞迴關係式子。」

最後，列出一些可以被動態規劃解的問題，以提供您參考。

1. **矩陣鍊相乘**(matrix-chain multiplication)：多於三個矩陣連乘時，不同順序的矩陣相乘所得的結果相同，但需要的乘法運算總次數不同(一般加減法較省時間，故在此不計算)。此問題要找到最少乘法運算的矩陣相乘順序。
2. **最佳多邊形三角切割**(optimal polygon triangulation)：將一個多邊形切成多個三角形(切點須在頂點上且切割線不可相交)，使其所需的切割線段長度之總和為最小。
3. **全配對最短路徑**(all-pairs shortest-paths)：計算出一圖中，任意兩點(vertex)之間的最短路徑。其中，此圖中不含有總和為負值的迴圈(cycle)。
4. **最佳二元搜尋樹**(optimal binary search tree)：將會被尋找的值，置於二元樹(binary tree)的樹葉(leaf)上。假設已經知道每個被查詢的值的機率，此問題是，建一棵二元搜尋樹，使得其所需的平均找尋時間最小。
5. **資源分配問題**(resource allocation problem)：考慮 m 項資源及 n 個計畫，當某項資源被分配到某個計劃時，會有相對應的利益產生。此問題是找到一個資源分配的方式，使得整體的利益最大。

學習評量

1. 最長嚴格遞增子字串：寫一個程式，自一串整數中找出最長的嚴格遞增子字串 (longest strictly increasing subsequence)，即是從原一串整數中，找到一個子集合；按照原來的順序排列時，是下一個比前一個值大，並且此子集合個數會最大(即排出來的子字串最長)。

```
輸入：
-17 11 9 2 3 8 8 10

輸出：
-17 2 3 8 10
```

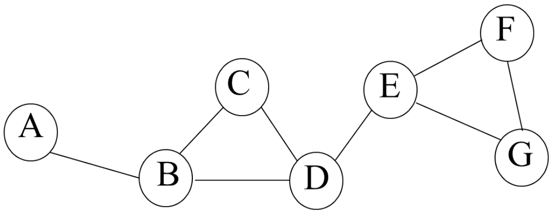
2. 給一個二維的整數矩陣，請找出一個子矩陣 (sub-rectangle) 其矩陣內所有數字的總和為最大。例如，下列矩陣的解是右下方的子矩陣：

```
-1  0   -9
-2  3   14
-2  4   20


3   14
4   20
```

而且其總和為 41。

```
輸入：
3 3                (矩陣的行數及列數)
-1 0 -9            (以下是矩陣的資料)
-2 3 14
-2 4 20

輸出：
41                 (子矩陣最大總和)
```

3. 一台檔案伺服器(file server)希望要被佈置於一個連結(connected)網路的中心， 使得網路上其他設備可以用最少的步數(hop)，得以連接到此伺服器。例如，在下圖中，當此伺服器被置於A，最遠的G最少須花四步連接到A。但是當此伺服器被置於D，最遠的F只需要兩步即可抵達D。點D就是網路的中心點。



輸入：

```
6                    (網路上點的個數，並且以1, 2, 3, …編號)
9                    (網路上線的個數)
1 2                  (以下為每一條網路線的資料)
1 4
2 3
2 4
2 5
3 4
4 5
4 6
5 6
```

輸出：

```
4                    (網路中心點的編號)
```

Memo