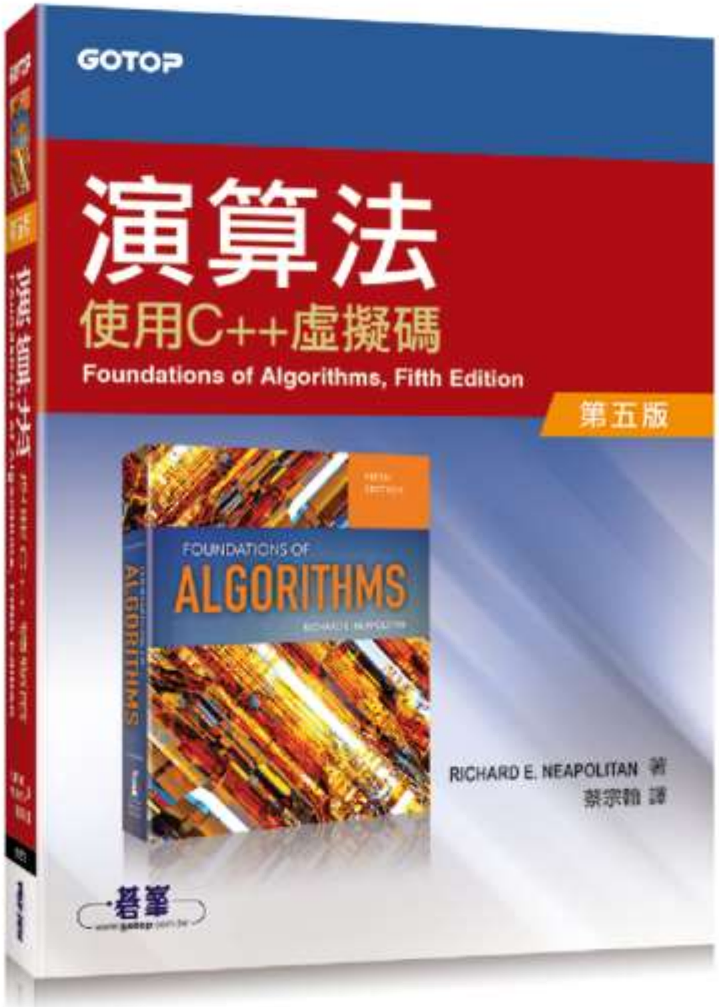


第三章
動態規劃

演算法
使用C++虛擬碼
Foundations of Algorithms, Fifth Edition

第五版

碁峯資訊 版權聲明：本教學投影片僅供教師授課講解使用，投影片內之圖片、文字及其相關內容，未經著作權人許可，不得以任何形式或方法轉載使用。

第三章 動態規劃

各個擊破 vs. 動態規劃

- 由於divide-and-conquer演算法會重複計算某些相同的結果好幾次，造成它極度缺乏效率
- dynamic programming (動態規劃) 與divide-and-conquer
  - 相似處在於，它們會先將一個問題切成數個較小且性質相同的問題
  - dynamic programming會先去計算較小的問題，並且儲存計算的結果。稍後，若有需要先前已算過的部分，就不需重新計算，而可以直接從先前儲存的結果中取得

建構dynamic programming演算法的步驟

- 建立一個遞迴(recursive)的機制，用它來求取一個問題經過切割後，所產生較小但性質相同問題的解。
- 用bottom-up的方式解題，首先由最小的問題開始，逐步向上求取最後整個問題的解。

3.1 二項式係數

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad for \;\; 0 \le k \le n$$

為了避免直接去計算數值龐大的$n!$和$k!$，我們利用上述的式子配合遞迴的特性，逐步地計算出最後的結果。以下是將這個計算式以divide-and-conquer演算法來計算

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \;\; or \;\; k = n \end{cases} \quad (3.1)$$

演算法 3.1 二項式係數 (divide-and-conquer版)

問題：計算二項式係數。

輸入：非負值的整數 n 及 k，其中 $k \le n$。

輸出：*bin* 二項式係數之值 $\binom{n}{k}$。

```
int bin (int n, int k)
{
  if (k == 0 || n == k)
      return 1;
  else
      return bin (n-1, k - 1)+ bin (n - 1, k);
}
```

演算法3.1的問題

- 為了求得$\binom{n}{k}$之值，我們在求解的過程中，計算的次數高達$2\binom{n}{k} - 1$

- 有些需要計算的部份在遞迴的過程中一再被重複地計算。
  - 舉例來說，在計算$bin(n-1, k-1)$和$bin(n-$

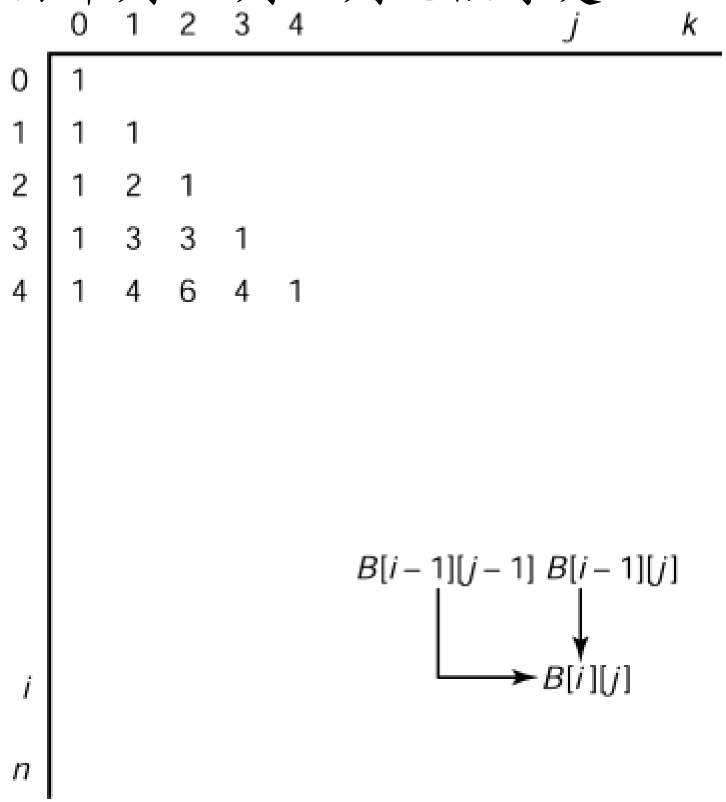
目錄

將演算法3.1改為動態規劃

1. 建立一個遞迴呼叫。我們將等式3.1改寫成使用陣列B的方式

$$B[i][j]=\begin{cases} B[i-1][j-1]+B[i-1][j] & 0<j<i \\ 1 & j=0 \quad or \quad j=i \end{cases}$$

2. 用bottom-up的方式，由陣列一列一列地依序處理，首先從第一列開始。



範例 3.1

求 B[4][2] = $\binom{4}{2}$

計算第 0 列： { 這個步驟是為了完整地模仿演算法的定義。}
{ 雖然 $B[0][0]$ 這個值在稍後的計算中不會使用到。}

$$B[0][0] = 1$$

計算第 1 列：

$$B[1][0] = 1$$
$$B[1][1] = 1$$

計算第 2 列：

$$B[2][0] = 1$$
$$B[2][1] = B[1][0] + B[1][1] = 1 + 1 = 2$$
$$B[2][2] = 1$$

計算第 3 列：

$$B[3][0] = 1$$
$$B[3][1] = B[2][0] + B[2][1] = 1 + 2 = 3$$
$$B[3][2] = B[2][1] + B[2][2] = 2 + 1 = 3$$

計算第 4 列：

$$B[4][0] = 1$$
$$B[4][1] = B[3][0] + B[3][1] = 1 + 3 = 4$$
$$B[4][2] = B[3][1] + B[3][2] = 3 + 3 = 6$$

演算法 3.2 二項式係數 (dynamic programming版)

問題：計算二項式係數。

輸入：非負值的整數 n 及 k，其中 $k \le n$。

輸出：$bin2$，二項式係數之值 $\binom{n}{k}$。

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0.. n] [0.. k];

  for (i = 0; i <= n; i++)
    for (j = 0; j <= minimum(i, k); j++)
      if (j == 0 || j == i)
        B[i] [j] = 1;
      else
        B[i] [j] = B[i -1][j -1] + B[i -1][j];
  return B[n][k];
}
```

- 下表所展示的是在執行過程中，不同的 值，會導致for-j迴圈被執行的次數。

i	0	1	2	3	…	k	$k+1$	…	n
迴圈執行次數	1	2	3	4	…	$k+1$	$k+1$	…	$k+1$

- 因此，for-j迴圈總共被執行的次數可以計算成

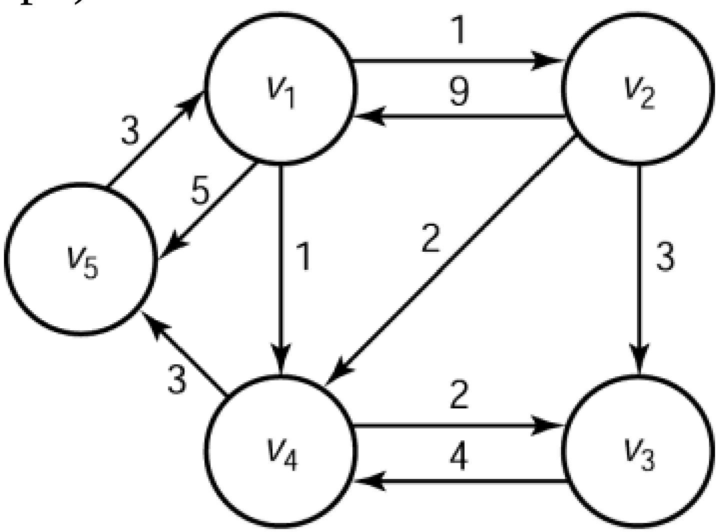
$$1+2+3+4+\cdots+k+\underbrace{(k+1)+(k+1)\cdots+(k+1)}_{n-k+1\text{個}}$$

根據範例A.1，上式等於

$$\frac{k(k+1)}{2}+(n-k+1)(k+1)=\frac{(2n-k+2)(k+1)}{2}\in\Theta(nk)$$

3.2 佛洛伊德最短路徑演算法

- 圖3.2是一個有向權重圖(weighted, directed graph)，其中圓形表示頂點(vertices)，線段表示邊(edge)
- 如果圖形中的邊是具有方向性的，則我們會稱這個圖形為有向圖(directed graph)。
- 在有向圖中，兩頂點間允許兩條方向不同的邊存在。例如在圖3.2中，有一條邊是由頂點v_1 到v_2，此外還同時存在另一條邊是由頂點v_2到v_1。
- 如果圖形中的邊有關聯值，我們就稱此值為權重(weight)，這種圖則稱為加權圖(weighted graph)



- 路徑(path)指的是一系列的頂點編號。也就是說從某一頂點到另一頂點間所會經過的
  - $[v_1, v_4, v_3]$
- 若路徑的起終點相同，我們把這樣的路徑稱為環(cycle)
  - $[v_1, v_4, v_5, v_1]$
  - 如果一個圖形包含著環狀的路徑，我們稱這樣的圖形為環狀圖(cyclic)，反之則稱為非環狀圖(acyclic)
- 如果某路徑為簡單(simple)路徑，則表示在此路徑中，同一頂點不會出現兩次
  - $[v_1, v_2, v_3]$就是簡單路徑$[v_1, v_4, v_5, v_1, v_2]$就不是簡單路徑
- 在加權圖中，某條路徑上所有邊的權重總和稱為長度(length)；而在非權重圖中，路徑長度指的是路徑上有幾條邊

最短路徑

- 一種最佳化問題
  - 候選解就是從某頂點到另個頂點的路徑
  - 值就是該條路徑的長度
  - 最佳值就是這些長度中最小的
- 最顯而易見的演算法
  - 假設有一圖形包含n個頂點，其任一頂點皆有邊直接與其他頂點相連
  - 第一個起點只有一種選擇，頂點A本身；
  - 第二個點時就變成了有n-2個選擇性
  - 第三個點時則有$n-3$個選擇性，依此類推，所有可能的路徑數目為：

$$(n-2)(n-3)\cdots 1=(n-2)!$$

  - 時間複雜度將超越指數時間

dynamic programming的方法

- 首先，我們先建立一個陣列 來表示一個權重圖，其原則如下：

$$W[i][j] = \begin{cases} \text{邊的權重} & \text{如果}v_i\text{到}v_j\text{之間存在著一條邊} \\ \infty & \text{如果}v_i\text{到}v_j\text{之間不存在任何的邊} \\ 0 & \text{如果}i = j \end{cases}$$

- 陣列W稱為相鄰矩陣(adjacency matrix)

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

dynamic programming的方法

- 從陣列W開始計算，逐步得到陣列D的結果
- 在計算的過程中，我們會需要建立一系列n+1個陣列D，我們將用$D^{(k)}$來表示這些陣列
- 其中，並且$D^{(k)}[i][j]$等於v_i與v_j間僅僅使用頂點$\{v_1, v_2, \ldots, v_k\}$作為中間點的最短路徑長度。

- $D^{(n)}[i][j]$所代表的是v_i到v_j間最短路徑長度，在這條路徑中，可經過頂點v_1到v_n中的任一個頂點
- $D^{(0)}[i][j]$代表的是頂點 v_i到v_j 間不能經過其他任何一個頂點所產生最短路徑的長度，換言之，其實就是指頂點 v_i到v_j 這個邊的權重。因此，我們可以建立以下的式子：

$$D^{(0)} = W \text{ 並且 } D^{(n)} = D$$

如何從陣列W求得陣列D

- Step1：建立一個遞迴的程序，使其可以由$D^{(k-1)}$計算出$D^{(k)}$。
- Step2：用bottom-up的方式，重複地執行Step1，也就是由k=1，一直執行到 k=n，逐步地求出$D^{(n)}$。以下是計算過程中產生的變化

$$\begin{array}{ccccc} D^0, & D^1, & D^2, & \ldots, & D^n \\ \uparrow & & & & \uparrow \\ W & & & & D \end{array}$$

- 在執行步驟1時，要考慮兩種可能的狀況：

狀況一

- 至少有一條最短路徑存在於v_i到v_j間，在路徑中允許經過頂點$\{v_1, v_2, \cdots, v_k\}$作為中間點，但並沒有使用到頂點$v_k$

$$D^{(k)}[i][j] = D^{(k-1)}[i][j]$$

- 舉例說明，在圖3.2中

$$D^{(5)}[1][3] = D^{(4)}[1][3] = 3$$

狀況二

- 所有v_1到v_j的最短路徑，路徑中允許使用頂點$\{v_1, v_2, \cdots, v_k\}$做為中間點，並且必需使用到頂點$v_k$
- 圖3.4中，在v_i到v_k的子路徑中，因為不能使用v_k做為中間點(v_k是此子路徑的終點)，所以僅能使用頂點$\{v_1, v_2, \cdots, v_{k-1}\}$。這意味著$v_i$到$v_k$子路徑的最短路徑長度等於$D^{(k-1)}[i][k]$。同理，$v_k$到$v_j$子路徑的最短路徑長度等於$D^{(k-1)}[k][j]$。總結以上的論述，狀況2可以得出以下的結論

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \qquad (3.4)$$

舉例說明，以圖3.2為例

$$D^{(2)}[5][3] = 7 = 4 + 3 = D^{(1)}[5][2] + D^{(1)}[2][3]$$

- 在狀況一與狀況二中找出兩者的最小值來做為 $D^{(k)}[i][j]$的值
- 定義$D^{(k)}$ 與$D^{(k-1)}$ 的關係如下
- $D^{(k)}[i][j] = minimum(\underbrace{D^{(k-1)}[i][j]}_{\text{狀況一}}, \underbrace{D^{k-1}[i][k] + D^{k-1}[k][j]}_{\text{狀況二}})$

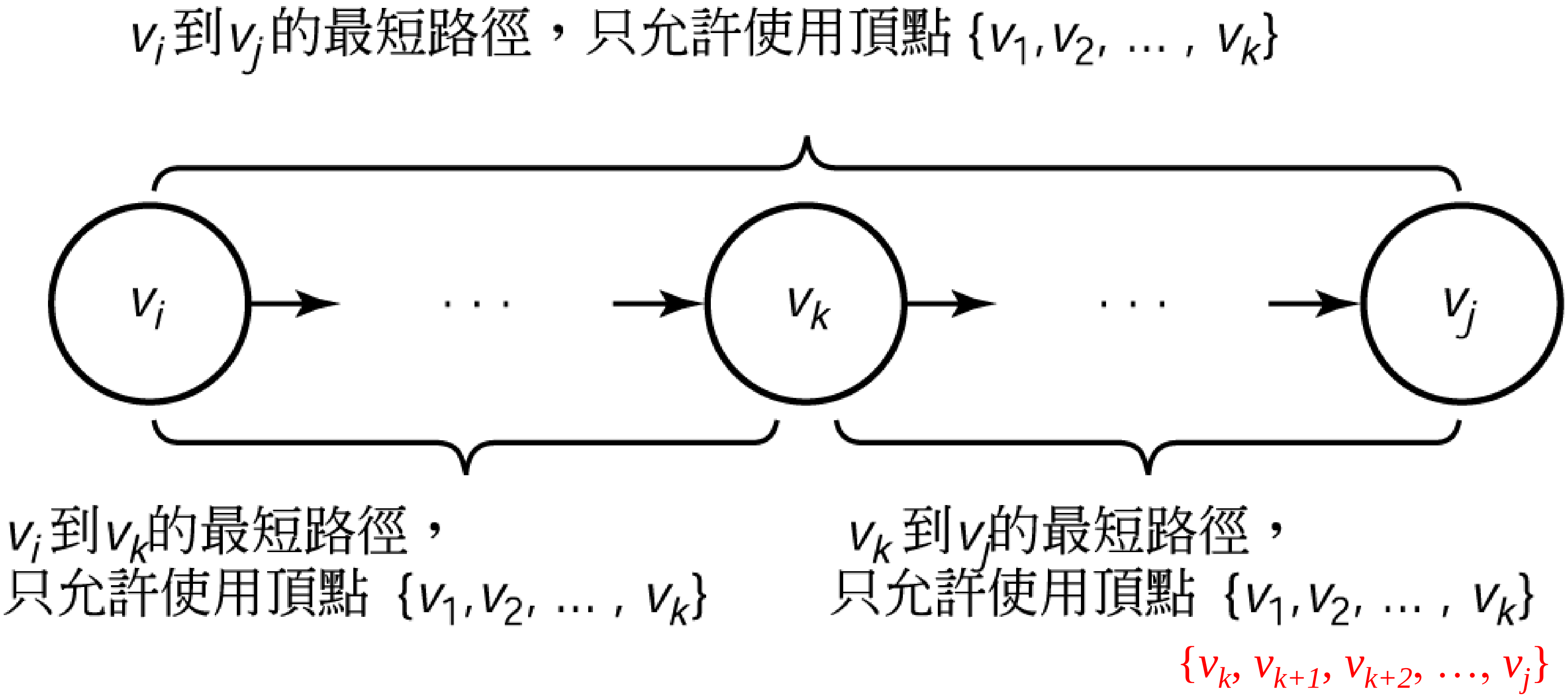


圖3.4 使用頂點v_k的最短路徑

目錄

範例 3.3

- 圖3.2圖形的相鄰矩陣W如圖3.3所示
- 以下我們列舉其中幾個值的計算方式

$$D^{(1)}[2][4] = minimum(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4])$$
$$= minimum(2, 9 + 1) = 2$$
$$D^{(1)}[5][2] = minimum(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2])$$
$$= minimum(\infty, 3 + 1) = 4$$
$$D^{(1)}[5][4] = minimum(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4])$$
$$= minimum(\infty, 3 + 1) = 4$$

當整個陣列$D^{(1)}$被計算出來，緊接著才開始計算陣列$D^{(2)}$，其中$D^{(2)}[5][4]$的計算方式如下

$$D^{(2)}[5][4] = minimum(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4])$$
$$= minimum(4, 4 + 2) = 4$$

於是整個陣列$D^{(2)}$被計算出來。計算程序直到陣列$D^{(5)}$被計算完成，這個結果即是各個頂點之間最短路徑的長度

演算法3.3
佛洛伊德最短路徑演算法

- 問題：在權重圖中，計算各頂點間的最短路徑(其權重皆為非負值)。
- 輸入：一有向權重圖，其中共有n個頂點，此圖形以相鄰矩陣W來表示。
- 輸出：二維陣列D，其列與行的索引值均由1到n，其中$D[i][j]$即表示第i個頂點到第j個頂點間最短路徑的長度。

```
void floyd (int n
            const number W[][],
            number D[][])
{
  index i, j, k;
  D = W;
  for (k = 1; k <= n; k++)
     for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
           D[i] [j] = minimum(D[i][j], D[i][k] + D[k][j]);
```

分析演算法3.3

所有情況的時間複雜度(佛洛伊德最短路徑演算法)

- 基本運算：發生在 for-j 迴圈中的指令。
- 輸入大小：n，圖形中頂點的數目。
- for-j 迴圈被包含在迴圈 for-i 之中，然後整個又被包在 for-k 迴圈中，這三個迴圈分別都執行 n 次，所以

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

演算法3.4
佛洛伊德最短路徑演算法2

- 問題：同演算法3.3，除此之外也要找出最短路徑。
- 額外的輸出：陣列 P ，其列與行的索引值皆為1到 n 。

$$P[i][j] = \begin{cases} v_i \text{ 到 } v_j \text{ 的最短路徑上，索引值最大的頂點編號} \\ \text{（若最短路徑上至少有一頂點存在）} \\ 0 \text{（如果 } v_i \text{ 到 } v_j \text{ 的最短路徑上沒有任何頂點存在）} \end{cases}$$

```
void floyd2 (int n,
             const number W[][],
                   number D[][],
                   index  P[][])
{
  index, i, j, k;

  for (i = 1; i <= n; i++)
     for (j = 1; j <= n; j++)
        P[i] [j] = 0;
  D = W;
  for (k = 1; k <= n; k++)
     for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
           if (D[i][k] + D[k][j] < D[i][j]) {
              P[i][j] = k;
              D[i][j] = D[i][k] + D[k][j];
           }
}
```

目錄

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

圖3.5 將演算法3.4應用在圖3.2上，所產生的陣列P

演算法3.5 印出最短路徑

問題： 在一權重圖中，印出某頂點到另一頂點最短路徑上的所有頂點。

輸入： 由演算法 3.4 所產生的陣列 P，另外再加入兩個索引值 q 和 r，分別 表示圖上兩個頂點的編號。

$$P[i][j] = \begin{cases} v_i \text{ 到 } v_j \text{ 的最短路徑上，索引值最大的頂點編號} \\ \text{（若最短路徑上至少有一頂點存在）} \\ 0 \text{（如果 } v_i \text{ 到 } v_j \text{ 的最短路徑上沒有任何頂點存在）} \end{cases}$$

輸出： v_q 到 v_r 間最短路徑上的所有頂點。

```
void path (index q, r)
{
  if (P[q][r] != 0){
     path (q, P[q][r]);
     cout << "v" << P[q][r];
     path (P[q][r], r);
  }
}
```

3.3 動態規劃和最佳化問題

- 建立一個解最佳化問題的演算法步驟如下：
  1. 建立一個遞迴的機制，用它來求取一個問題經過切割後，所產生較小但性質相同問題的解。
  2. 用bottom-up的方式解題，首先由最小的問題開始，逐步向上求取最後整個問題的解。
  3. 用bottom-up的方式建構出最佳解。
- 若最佳化原則 (principle of optimality)要可以應用在某問題上，它必須符合下列原則：當某問題存在最佳解，則表示其所有的子問題也必存在最佳解
  – 如果v_k是 v_i到 v_j間最短路徑上的點，則 v_i到 v_k以及 v_k到 v_j這兩個子路徑也必定是最短路徑

範例 3.4

- 如何在一個圖形中，找出各個頂點之間「最長」簡單路徑。
- 在這裡限定「簡單」路徑的原因，是因為如果存在著環(cycle)的情況，就可以無限制地任意延長路徑的長度。如此，便難以求得本題的答案。
- 在圖3.6中，v_1 到v_4 間的最長路徑是[v_1,v_3,v_2,v_4]，然而其中的一條子路徑 [v_1,v_3]，並非v_1 到v_3間的最長路徑，因為[v_1,v_2,v_3] 才是v_1 到v_3間的最長路徑。

length[v_1,v_3]=1 並且length[v_1,v_2,v_3]=4

- 因此，最佳化問題原則並無法應用在這類型的問題上

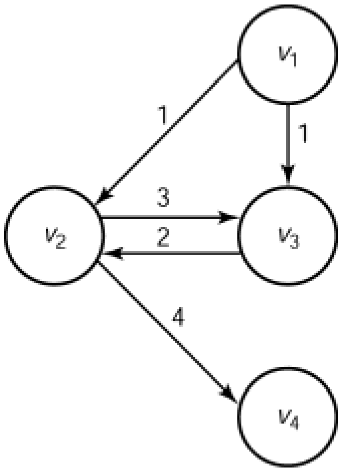


圖3.6 包含cycle的有向權重圖

3.4 連續矩陣相乘

- $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$
- 乘積結果的第一列第一行元素：
  - $1 \times 7 + 2 \times 2 + 3 \times 6 = 29$ →3次乘法
  - 乘積有2×4個元素 →$2 \times 4 \times 3 = 24$次乘法
- 一個$i \times j$的矩陣乘上一個$j \times k$的矩陣，共需
  - $i \times j \times k$個乘法運算

3.4 連續矩陣相乘

- 看看以下這四個矩陣相乘

A	$\times$	B	$\times$	C	$\times$	D
20×2		2×30		30×12		12×8

- 各矩陣下方的數字代表著矩陣的維度。矩陣相乘具有「與相乘順序無關」的特性。舉例說明： 與 所得到的結果是一樣的。
- 以下我們例舉五種不同的相乘順序，不同的順序需要不同的乘法次數。

$A(B(CD))$	$30\times 12\times 8$	+	$2\times 30\times 8$	+	$20\times 2\times 8$	=	3,680
$(AB)(CD)$	$20\times 2\times 30$	+	$30\times 12\times 8$	+	$20\times 30\times 8$	=	8,880
$A((BC)D)$	$2\times 30\times 12$	+	$20\times 12\times 8$	+	$20\times 2\times 8$	=	1,232
$((AB)C)D$	$20\times 2\times 30$	+	$20\times 30\times 12$	+	$20\times 12\times 8$	=	10,320
$(A(BC))D$	$2\times 30\times 12$	+	$20\times 2\times 12$	+	$20\times 12\times 8$	=	3,120

- 六個矩陣相乘的最佳順序可以分解成以下的其中一種型式
  1. $A_1(A_2A_3A_4A_5A_6)$
  2. $(A_1A_2)(A_3A_4A_5A_6)$
  3. $(A_1A_2A_3)(A_4A_5A_6)$
  4. $(A_1A_2A_3A_4)(A_5A_6)$
  5. $(A_1A_2A_3A_4A_5)A_6$
- 第k個分解型式所需的乘法總數，為前後兩部份(一為 $A_1A_2 \cdots A_k$，另一為 $A_{k+1} \cdots A_6$)各自所需乘法數目的最小值相加，再加上相乘這前後兩部份矩陣所需的乘法數目。

$$M[1][k] + M[k+1][6] + d_0d_kd_6$$

- 我們已證明了

$$M[1][6] = \underset{1 \le k \le 5}{minimum}\left(M[1][k] + M[k+1][6] + d_0d_kd_6\right)$$

- 實際上，這個公式並沒有限定第一個與最末個矩陣一定要是A_1與A_6
- 因此，我們可以推廣這個式子以得到下列連乘n個矩陣的遞迴性質。對於$1 \le i \le j \le n$

$$M[i][j] = \underset{i \le k \le j-1}{minimum}\left(M[i][k] + M[k+1][j] + d_{i-1}d_k d_j\right) \quad \text{if } i < j$$
$$M[i][i] = 0 \quad , \text{if } i \ge j \qquad (3.5)$$

- 基於此性質設計的divide-and-conquer演算法是指數時間複雜度的
- 我們使用動態規劃來設計一個效率更高的演算法，用來循序計算M[i][j]的值
  - M[i][j]的計算牽涉到
    - 同一列位於其左及
    - 同一欄位於其下的所有元素
    - 利用此性質，我們可以用以下的方式來計算M中的元素：首先，設定主對角線上的值為0；接著我們從主對角線開始，依序由每一條斜線往上計算(對角線1、對角線2....對角線5)，直到對角線5為止，而它就是最終我們需要的答案—M[1][6]

範例 3.6

- 計算範例3.5中的六個矩陣，下面列出的是dynamic programming演算法執行的步驟，結果詳見圖3.8。
- 計算對角線0：

$$M[i][i] = 0 \text{ for } 1 \leq i \leq 6$$

- 計算對角線1：

$$M[1][2] = \underset{1 \leq k \leq 1}{minimum}\left(M[1][k] + M[k+1][2] + d_0 d_k d_2\right)$$

$$= M[1][1] + M[2][2] + d_0 d_1 d_2$$

$$= 0 + 0 + 5 \times 2 \times 3 = 30$$

- 計算對角線2：

$$M[1][3] = \underset{1 \leq k \leq 2}{minimum}\left(M[1][k] + M[k+1][3] + d_0 d_k d_3\right)$$

$$= minimum(M[1][1] + M[2][3] + d_0 d_1 d_3\,,$$

$$M[1][2] + M[3][3] + d_0 d_2 d_3)$$

$$= minimum(0 + 24 + 5 \times 2 \times 4\,, 30 + 0 + 5 \times 3 \times 4) = 64$$

- 計算對角線3：

$$M[1][4] = \underset{1 \le k \le 3}{minimum}(M[1][k] + M[k+1][4] + d_0 d_k d_4)$$
$$= minimum(M[1][1] + M[2][4] + d_0 d_1 d_4,$$
$$M[1][2] + M[3][4] + d_0 d_2 d_4,$$
$$M[1][3] + M[4][4] + d_0 d_3 d_4)$$
$$= minimum(0 + 72 + 5 \times 2 \times 6,\ 30 + 72 + 5 \times 3 \times 6,$$
$$64 + 0 + 5 \times 4 \times 6) = 132$$

- 計算對角線4：
  - 在對角線4上的$M[1][5]$ 、$M[3][6]$ 的計算方式相似，詳見圖3.8。
- 計算對角線5：
  - 最終，計算對角線5，$M[1][6]$即是矩陣A_1乘到A_6，所需最少的乘法運算次數。

$$M[1][6] = 348$$

演算法 3.6 最少乘法次數

問題： 找出 n 個矩陣相乘所需的最少乘法次數，以及矩陣相乘的順序。

輸入： 表示矩陣的數量 n；索引值由 0 到 n 的整數陣列 d，其中 $d[i-1] \times d[i]$ 表示第 i 個矩陣的維度。

輸出： $minmult$ 表示矩陣相乘所需的最少乘法次數；還有一個二維陣列 P，用來儲存矩陣相乘的最佳順序，它的列索引值由 1 到 $n-1$，而行索引值由 1 到 n。$P[i][j]$ 代表矩陣 i 相乘到矩陣 j 最好的分割方式。

```
int minmult (int n,
             const int d[],
             index P [][])
{
  index i, j, k, diagonal;
  int M[1..n] [1..n];

  for (i = 1; i <= n; i++)
     M[i][i] = 0;
  for (diagonal = 1; diagonal <= n - 1; diagonal++)
      // 位在主對角線上方的第一條斜線

     for (i = 1; i <= n - diagonal; i ++){
        j = i + diagonal;
        M[i][j] =
          minimum (M[i] [k] + M[k + 1][j] + d [i - 1]* d [k] * d [j]);
          i ≤ k ≤ j-1
        P[i][j] = 達到最小乘法次數時的 k 值;
     }
  return M[1][n];
}
```

分析演算法3.6

所有情況的時間複雜度（最少乘法次數）

- 基本運算：我們可以用每個不同k值所執行的指令作為基本運算，其中包含用來檢查是否為最小值的「比較」指令。
- 輸入大小：n，矩陣的數量。
- 在演算法3.6中，我們使用了三層迴圈。因為$j = i + diagonal$。所以k迴圈的執行次數為

$$j - 1 - i + 1 = i + diagonal - 1 - i + 1 = diagonal$$

- 當給定$diagonal$值，for-i迴圈被執行的次數將會是$n - diagonal$。因為$diagonal$的值是由1到$n - 1$，所以基本運算的總執行次數為

$$\sum_{diagonal=1}^{n-1} [(n - diagonal) \times diagonal]$$

- 在習題中，我們將會證明這個式子的值等於

$$\frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

展示如何利用陣列P印出矩陣相乘的最佳順序

- P[2][5] = 4表示矩陣A_2乘到A_5的最佳乘法順序可以分解成

$$(A_2A_3A_4)A_5$$

我們可以藉由拜訪P[1][n]找到第一層的分解方式。由於$n = 6$且$P[1,6] = 1$，因此在最佳相乘順序中，第一層的分解方式為

$$A_1(A_2A_3A_4A_5A_6)$$

接下來，我們要找出矩陣 乘到 的最佳乘法順序，所以必需參考 的值。其值為5，所以可以分解成$(A_2A_3A_4A_5)A_6$

而A_2乘到A_5的分解方式仍待求出。接著我們以相同方式查詢P[2][5] 的值，直到所有的分解方式都已決定，最終可以得到答案

$$A_1((((A_2A_3)A_4)A_5)A_6)$$

演算法 3.7 印出最佳順序

問題：印出 n 個矩陣相乘的最佳順序。

輸入：正整數 n 與由演算法 3.6 所產生的陣列 P，$P[i][j]$ 表示矩陣 i 乘到矩陣 j 的最佳乘法順序中的分割點。

輸出：矩陣相乘的最佳順序。

```
void order (index i, index j)
{
   if (i == j)
      cout << ''A'' << i;
   else {
      k = P[i][j];
      cout << ''('';
      order (i, k);
      order (k + 1, j);
      cout << '')'';
   }
}
```

3.5 最佳二元搜尋樹

- **二元搜尋樹(binary search tree)是一棵由某個有序集合中取出的項(通常稱為key)構成的二元樹。**
  1. 每個節點包含一個key。
  2. 節點 的左子樹中任一節點的key，必需小於等於節點 的key。
  3. 節點 的右子樹中任一節點的key，必需大於或等於節點 的key

演算法 3.8 搜尋二元樹

問題： 在二元搜尋樹中搜尋某個 key（假設此 key 必存在於樹中）。

輸入： *tree*（為一個指標，指向二元搜尋樹）；以及欲搜尋的 key *keyin*。

輸出： 一指標 *p*（指向有包含該 key 的節點）。

```
struct nodetype
{
  keytype key;
  nodetype* left;
  nodetype* right;
};

typedef nodetype* node_pointer ;
```

```
void search (node_pointer tree,
             keytype keyin,
             node_pointer& p)
{
  bool found;

  p = tree;
  found = false;
  while (! found)
    if (p->key == keyin)
        found = true;
    else if (keyin < p-> key);
        p = p-> left;              // 前往左子節點
    else
        p = p-> right;             // 前往右子節點
}
```

- 在*search*() 程序中搜尋一個key所需比較(comparison)指令的數目，稱之為搜尋時間 (search time)
- 我們的目標是使整個二元樹的平均搜尋時間是最小的
- 搜尋一個key的搜尋時間可以寫

$$depth(key) + 1$$

- 假設 $key_1, key_2, key_3, \ldots, key_n$為 n個key依序排列，並且 p_i為key_i 被當作search key的機率。如果 c_i是搜尋key_i 所需比較指令的數目，所以這個樹的平均搜尋時間為

$$\sum_{i=1}^{n} c_i p_i$$

- 我們的目標是將上述的值最小化

較無效率的方法

- 在求取最佳二元搜尋樹時，列出所有可能的二元搜尋樹，然後再找出其中最佳的一個，必須花費至少 n 的指數時間去計算
- 證明

  我們以深度$n-1$的二元搜尋樹為研究對象。在這樣的樹中，除了根節點之外，共有$n-1$層，每一層的節點有兩種選擇，也就是可以做為它的父節點的左子節點或是右子節點。這表示一個二元搜尋樹如果深度為$n-1$的話，則共可以組合出2^{n-1}種不同的型式。

以動態規劃的技巧開發更有效率的演算法

- 假設Key_i到Key_j將被安排到可以最小化$\sum_{m=i}^{j} c_m p_m$的樹
- 其中c_m是在樹中搜尋key 所需比較指令的數目，我們稱這樣的樹是對這些key(Key_i…Key_j)最佳的樹，並將最佳值以$A[i][j]$來表示。由於只需要一個比較指令即可搜尋到Key_i，故$A[i][i] = p_i$。

- 最佳樹的任意子樹對這棵子樹上的眾節點們必須是最佳的。因此，此問題便適用前面所提最佳化原則
- 定義tree 1來表示一個最佳化的樹，其中Key_1 必需是這個樹的根節點；.... tree n 也表示一個最佳化的樹，其中Key_n必需是這個樹的根節點。給定 $1 \le k \le n$，在這裡tree k 的子樹也必需是最佳的，因此在這些子樹中的平均搜尋時間描繪如圖3.13。
- 當$m \neq k$時，就需要一個比較的指令(發生在子樹的根節點)。這個指令使得在tree k中搜尋Key_m的平均時間增加了 $1 \times p_m$。由此可知，我們可以將tree k 的平均搜尋時間整理成以下的式子

$$\underbrace{A[1][k-1]}_{\substack{\text{左子樹的平均}\\\text{搜尋時間}}} + \underbrace{p_1 + \cdots + p_{k-1}}_{\substack{\text{左子樹需要的}\\\text{額外比較時間}}} + \underbrace{p_k}_{\substack{\text{根結點的平均}\\\text{搜尋時間}}} + \underbrace{A[k+1][n]}_{\substack{\text{右子樹的平均}\\\text{搜尋時間}}} + \underbrace{p_{k+1} + \cdots + p_n}_{\substack{\text{右子樹需要的}\\\text{額外比較時間}}}$$

上式等於

$$A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m$$

因為k個樹中必有一個是最佳的，故最佳樹的平均搜尋時間可以寫成以下的式子

$$A[1][n] = \underset{1 \le k \le n}{minimum}(A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} p_m$$

我們可以導出以下的式子

$$A[i][j] = \underset{i \le k \le j}{minimum}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \quad i < j$$

$$A[i][i] = p_i \tag{3.6}$$

$A[i][i-1]$ 及$A[j+1][j]$ 被定義成0

- 利用等式3.6，我們可設計出求出最佳二元搜尋樹的動態規劃演算法。因為計算$A[i][j]$時必須參考第i列位於$A[i][j]$之左以及第j欄位於$A[i][j]$之下的所有項，我們採用一次算一條對角線上的所有值的方式來推進

演算法 3.9 最佳二元搜尋樹

問題： 找出一群 key 的最佳二元搜尋樹，每個 key 都伴隨著一個其做為搜尋鍵的機率。

輸入： n 代表 key 的個數；陣列 p 用來儲存每個 key 被當作搜尋鍵的機率，其中 $p[i]$ 代表 Key_i 的機率。

輸出： 變數 $minavg$，代表最佳二元搜尋樹的平均搜尋時間；二維陣列 R，可以藉由 R 建構出最佳樹，它的列索引值由 1 到 $n+1$，行索引值是由 0 到 n，$R[i][j]$ 記錄某一個 key 的索引值，這個 key 被用來當作 Key_i 到 Key_j 所形成最佳樹的根節點。

目錄

```
void optsearchtree (int n,
                    const float p[],
                    float& minavg,
                    index R[][])
{
  index i, j, k, diagonal;
  float A[1..n + 1][0..n];
  for (i = 1; i < = n; i++){
     A[i][i - 1] = 0;
     A[i][i] = p[i];
     R[i][i] = i;
     R[i][i - 1] = 0;
  }
  A[n + 1][n] = 0;
  R[n + 1][n] = 0;
  for (diagonal = 1; diagonal <= n - 1; diagonal++)
     for (i = 1; i <= n - diagonal; i ++){      //Diagonal-1 就是位在
                                                // 主對角線上方的第一條線
       j = i + diagonal;
       A[i][j] = minimum(A[i][k - 1] + A[k + 1][j]) + Σ_{m=i}^{j} p_m
       R[i][j] = 達到最小乘法次數時的 k 值；
     }
  minavg = A[1][n];
}
```

分析演算法3.9
所有情況的時間複雜度(最佳化二元搜尋數)

基本運算：針對每一個 k 值所需執行的指令數目。這其中包含一個用來測試是否為最小值的比較指令。$\sum_{m=i}^{j} p_m$ 這個值並不需要每次都被重新計算。在習題中，你會發現一個更有效率的計算總和方式。

輸入大小：n，key 的數目。

這個演算法的控制部份幾乎和演算法 3.6 一樣。唯一的不同之處在於，當給定 *diagonal* 和 i 值時，基本運算會被執行 *diagonal*+1 次。這個演算法的分析方式和演算法 3.6 相似。

$$T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

演算法 3.10 建立最佳二元搜尋樹

問題：建立最佳二元搜尋樹。

輸入：n 代表 key 的數目；陣列 *Key* 依序儲存這 n 個 key。陣列 R 是由演算法 3.9 所產生的，其中 $R[i][j]$ 記錄著 Key_i 到 Key_j 所形成最佳樹的根節點之 key 的索引值。

輸出：指標 *tree*，指向包含這 n 個 key 的最佳二元搜尋樹

```
node_pointer tree (index i, j)
{
  index k;
  node_pointer p;

  k = R[i][j];
  if (k == 0)
     return NULL;
  else {
     p = new nodetype;
     p-> key = Key[k];
     p-> left = tree(i, k - 1);
     p-> right = tree(k + 1, j);
     return p;
  }
}
```

目錄

範例3.9

- 假設我們有下列的陣列：

Don	Isabelle	Ralph	Wally
Key[1]	*Key*[2]	*Key*[3]	*Key*[4]

- 並且

$$p_1 = \frac{3}{8} \quad p_2 = \frac{3}{8} \quad p_3 = \frac{3}{8} \quad p_4 = \frac{1}{8}$$

陣列*A*和*R*是由演算法3.9所產生的(如圖3.14)，樹則是由演算法3.10產生的(如圖3.15)，最小平均搜尋時間為 7/4

A

	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

R

	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

圖3.14

目錄

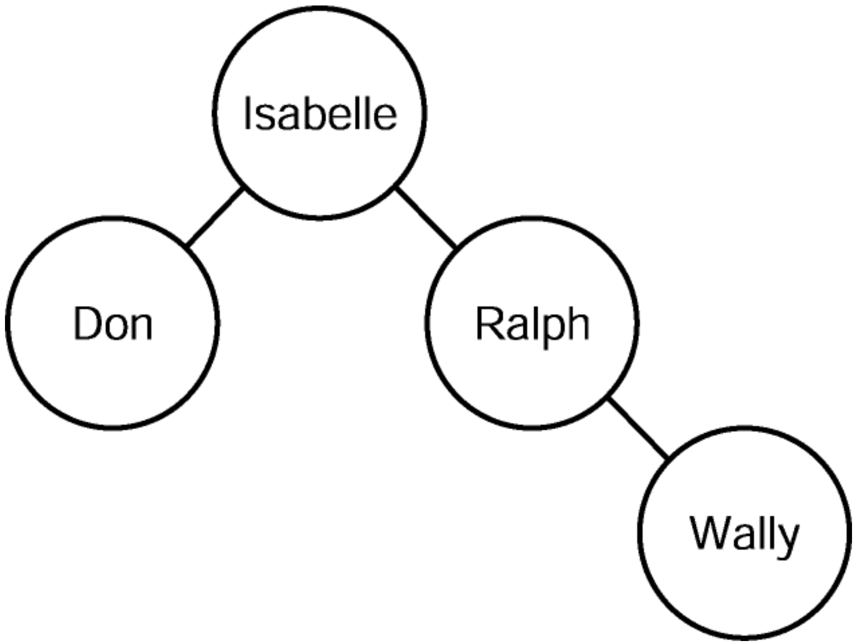


圖3.15
針對範例3.9，應用演算法3.9和3.10所產生的二元樹

3.6 售貨員旅行問題

- 假設有一位售貨員計畫在20個城市間旅行並推銷業務。每個城市間有道路連往其他城市。從售貨員的家鄉城市出發，拜訪了每一個城市一次之後，再回到他的家鄉城市
- 目標
  - 找出最短的路線
- 表示成一個權重圖
  - **頂點**代表城市
  - 在有向圖中，我們稱一個**旅程**(tour)為一條路徑，它是由某一個頂點出發，經過所有頂點一次之後，再回到該頂點。我們也把旅程稱為**漢米爾頓迴路** (Hamiltonian circuit)
  - 最佳旅程(optimal tour)是指該路徑擁有最小長度的特性。售貨員旅行問題的解答，就是要找到一個**最佳旅程**

目錄

$$length[v_1, v_2, v_3, v_4, v_1] = 22$$
$$length[v_1, v_3, v_2, v_4, v_1] = 26$$
$$length[v_1, v_3, v_4, v_2, v_1] = 21$$

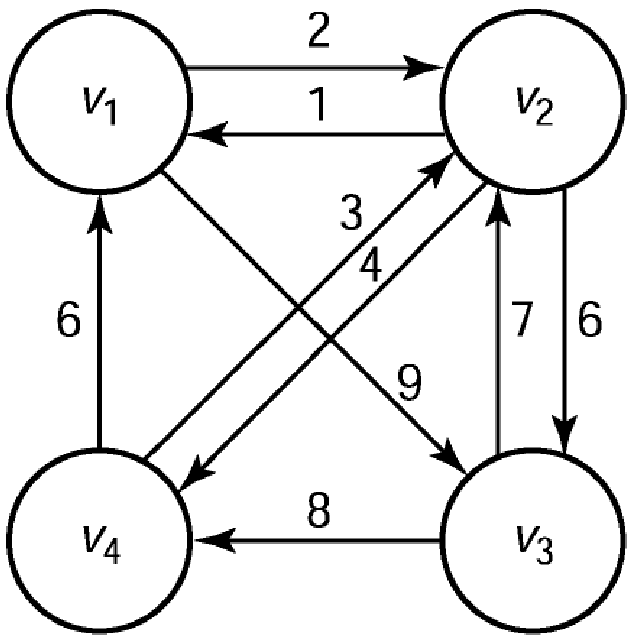


圖3.16 最佳旅程為$[v_1, v_3, v_4, v_2, v_1]$

最簡單的想法：找出所有可能的旅程，然後再找出其中路徑長度最小的

在一般情況下，或許每個頂點都有直接的一條邊通往其他頂點，如果我們要找出所有可能的旅程，第二個點有n-1種可能，第三個點有n-2種可能，...，第n個點只有一種可能。因此，可能的旅程總數為

$$(n-1)(n-2)\cdots 1 = (n-1)!$$

所花費的時間將超過指數時間

設計動態規劃演算法

- 假設在最佳旅程上，頂點v_k是頂點v_1下一個要拜訪的頂點，則旅程上v_k到v_1的這條子路徑也必定是經過所有點至少一次的最短路徑。這種情況正符合最佳化問題原則

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

圖3.17 本圖為圖3.16的相鄰矩陣

V = 所有頂點的集合
A = V的一個子集合
$D[v_i][A]$ = 從v_i出發，經過A中的所有頂點一次，再到達v_1的最短路徑長度

因為$V-\{v_1, v_j\}$包含除了v_1、v_j之外的所有頂點，並且適用最佳化問題原則，因此我們得到

$$\text{最佳}tour\text{ 長度} = \underset{2 \le j \le n}{minimum}(W[1][j] + D[v_j][V-\{v_1,v_j\}])$$

我們將上式進一步地改寫成一般的通式，在這裡$i \neq 1$並且v_i不在A中

$$D[v_i][A] = \underset{j:\, v_j \in A}{minimum}(W[i][j] + D[v_j][A-\{v_j\}]) \quad \text{if } A \neq \emptyset \tag{3.7}$$

$$D[v_i][\emptyset] = W[i][1]$$

範例3.11 找出圖3.17中的最佳旅程

首先，先考慮空集合的情況：

$$D[v_2][\emptyset] = 1$$
$$D[v_3][\emptyset] = \infty$$
$$D[v_4][\emptyset] = 6$$

接下來，考慮所有只包含一個元素的集合：

$$D[v_3][\{v_2\}] = \underset{j:\, v_j \in \{v_2\}}{minimum}(W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$
$$= W[3][2] + D[v_2][\emptyset] = 7 + 1 = 8$$

同樣地，

$$D[v_4][\{v_2\}] = 3 + 1 = 4$$
$$D[v_2][\{v_3\}] = 6 + \infty = \infty$$
$$D[v_4][\{v_3\}] = \infty + \infty = \infty$$
$$D[v_2][\{v_4\}] = 4 + 6 = 10$$
$$D[v_3][\{v_4\}] = 8 + 6 = 14$$

接下來，考慮所有包含兩個元素的集合：

$$D[v_4][\{v_2,v_3\}] = \underset{j:\, v_j\in\{v_2,v_3\}}{minimum}\left(W[4][j] + D[v_j][\{v_2,v_3\} - \{v_j\}]\right)$$
$$= minimum(W[4][2] + D[v_2][\{v_3\}]\,,\, W[4][3] + D[v_3][\{v_2\}])$$
$$= minimum(3+\infty\,,\,\infty+8) = \infty$$

同樣地，

$$D[v_3][\{v_2,\ v_4\}] = minimum(7+10\,,\,8+4) = 12$$
$$D[v_2][\{v_3,\ v_4\}] = minimum(6+14\,,\,4+\infty) = 20$$

最後，計算最佳旅程的長度

$$D[v_1][\{v_2,v_3,v_4\}] = \underset{j:\, v_j\in\{v_2,v_3,v_4\}}{minimum}\left(W[1][j] + D[v_j][\{v_2,v_3,v_4\} - \{v_j\}]\right)$$
$$= minimum(\ W[1][2] + D[v_2][\{v_3,v_4\}]\,,$$
$$W[1][3] + D[v_3][\{v_2,v_4\}]\,,$$
$$W[1][4] + D[v_4][\{v_2,v_3\}])$$
$$= minimum(2+20\,,\,9+12\,,\,\infty+\infty) = 21$$

演算法3. 11

售貨員旅行問題的dynamic programming版演算法

- 問題：在一個有向權重圖中找出最佳旅程(假設所有的權重都是非負值)。
- 輸入：一個有向權重圖，n代表圖中的頂點數目。此圖用一個二維陣列W表示，W的列與行的索引值都是由1到n。$W[i][j]$ 代表從頂點 v_i 到頂點 v_j 的邊之權重。
- 輸出：
  - 變數$minlength$：最佳旅程的長度
  - 二維陣列P，利用陣列P可以建構出最佳旅程。陣列P的列索引值是由1到n，而行索引值是$V-\{v_1\}$的所有子集合。$P[i][A]$儲存的是由頂點v_i出發，通過集合中所有頂點一次，然後到達頂點 的最短路徑上，位在頂點v_i後的第一個頂點。

```
void travel (int n,
             const number W[][],
             index P [][],
             number& minlength)
{
  index i, j, k;
  number D[1..n][subset of V - {v1}];

  for (i = 2; i <= n; i++)
     D[i][Ø] = W[i][1];
  for (k = 1; k <= n - 2; k++)
     for (所有包含 k 個點之 V - {v1} 的子集合 A)
        for (i 使得 i 不等於 1 且 vi 不在 A 中){
           D[i][A] = minimum (W[i][j] + D[j][A - {vj}]);
                     j:vj∈A
           P[i][A] = 造成 minimum 的 j 值;
        }
  D[1][V - {v1}] = minimum (W[1][j] + D[j][V - {v1, vj}]);
                   2 ≤ j ≤ n
  P[1][V - {v1}] = 造成 minimum 的 j 值;
  minlength = D[1][V - {v1}];
}
```

分析演算法3.11

所有情況的時間複雜度
(售貨員旅行問題的dynamic programming版演算法)

- 基本運算：第一個以及最後一個迴圈所花費的時間，跟中間的那個迴圈比較起來其實是無足輕重的，因為中間的迴圈包含了多層次的巢狀迴圈。因此，我們主要考慮的是針對每個v_j，需要執行多少指令來完成計算，包含了一個加法指令。
- 輸入大小：n，代表圖中的頂點數目

每個集合A包含k個頂點，我們需要考慮 個頂點，針對每個頂點，基本運算必需被執行k次。因為包含k個頂點的$V-\{v_1\}$子集合總共有$\binom{n-1}{k}$個，所以基本運算被執行的總次數為

$$T(n)=\sum_{k=1}^{n-2}(n-1-k)k\binom{n-1}{k} \qquad (3.8)$$

不難看出，

$$(n-1-k)\binom{n-1}{k}=(n-1)\binom{n-2}{k}$$

將上式代入等式3.8得到

$$T(n)=(n-1)\sum_{k=1}^{n-2}k\binom{n-2}{k}$$

最後，應用定理3.1，得到

$$T(n)=(n-1)(n-2)2^{n-3}\in\Theta(n^2 2^n)$$

分析演算法3.11

記憶體空間的複雜度

- $M(n)$
  - 陣列$D[v_i][A]$ 及$P[v_i][A]$ 是佔據記憶體空間最主要的部份，所以我們的目標是要確定它們究竟需要多大的記憶空間。
  - 因為$V-\{v_1\}$包含$n-1$個頂點，我們可以應用附錄A中的範例A.10，計算出這些頂點可以組合出2^{n-1}個不同的子集合。
  - 陣列D 和P 的第一個索引值範圍是由1到n，所以

$$M(n) = 2 \times n2^{n-1} = n2^n \in \Theta(n2^n)$$

3.7 序列對齊

- **染色體**
  - 是一條長條線狀，由**去氧核醣核酸(DNA)**所構成的巨型分子
  - 生物表現遺傳特性的載具
- DNA由互補的雙股構成，每股由一條核苷酸序列組成。
- **核苷酸**包含戊糖（脫氧核糖），磷酸基團，和嘌呤或嘧啶鹼基。
- **嘌呤**，腺嘌呤（A）和鳥嘌呤（G），在結構上是相似的。**嘧啶**，胞嘧啶（C）和胸腺嘧啶（T）結構也同樣類似
- 雙股利用核苷酸對的氫鍵結合。腺嘌呤總是和胸腺嘧啶配對，而鳥嘌呤總是與胞嘧啶配對。每對被稱為一個**典型鹼基對（bp）**，而A、G、C和T被稱為**基**

- 圖3.18描繪了DNA片段。實際上，雙股以扭曲的方式圍繞彼此形成一個右手的雙螺旋結構。然而，我們的目的只需要認定它們是字串，如圖所示。染色體被認為僅是一個長的DNA分子
- **DNA序列**是DNA的一段，一個**site**是該序列中是每個鹼基對的位置。DNA序列可以接受**替代**突變**(substitution)**、**插入**突變**(insertion)**、以及**刪除**突變**(deletion)**

圖 3.18 DNA的某一區段

- 考慮在某一特定群體中的每個個體(物種)同樣的DNA序列。每一代中，在產生下一代時，序列中的每個site的每個配子都有機會發生突變。
- 一個可能的結果是：整個群體中，某個基的某個給定site被另一個基所取代。
- 另一種可能的結果是，最後發生了物種分化，整個物種分化成兩個不同的物種。在這種情況下，最終在某個物種中所發生的取代將與在另個物種發生的取代迥異。
- 這意味著，由此兩個物種之個體取出的序列，可能差異很大。我們稱該序列已分道揚鑣。而兩個相對應的序列被稱為**同源序列**

比較來自不同物種個體的同源序列

- 首先必須將序列對齊。這是因為已經分化了，故其中一方或雙方的序列可能發生插入或刪除突變。例如，將人類與貓頭鷹猴的胰島素基因中第一個intron對齊，會得到一個長196個核苷酸的序列，其中163個site並沒有出現gap

範例 3.13

- 範例 3.13 假設我們有下列的同源基因

A A C A G T T A C C

T A A G G T C A.

它們有很多種可能的對齊方式。下列是其中兩種：

```
– A A C A G T T A C C
T A A – G G T – – C A
```

Insertion gap

Unmatching

```
A A C A G T T A C C
T A – A G G T – C A
```

哪一種對齊方式較好？

- 雙方都有5對匹配鹼基對。上面的對齊方式有兩個不匹配的鹼基對，但付出了插入四個gap的代價。另一方面，下面的對齊方式有三個不匹配的鹼基對，但只付出插入兩個gap的代價
  - 例如，假設gap懲罰1點，而不匹配罰3點。我們把一種對齊方式的總懲罰點數稱為該種對齊方式的成本。給定上述的懲罰方式，範例3.13上面的對齊方式的成本為10，而下面的成本為11。所以，上面的對齊方式較好
  - 另一方面，若gap懲罰2點，而不匹配罰1點，不難看出，下面的對齊方式因成本較低而相對較佳

- 假設如下：
  - 不匹配的罰點為1
  - gap的的罰點為2

  首先，我們將兩序列表達為兩陣列：

$$\frac{\text{A}}{x[0]}\ \frac{\text{A}}{x[1]}\ \frac{\text{C}}{x[2]}\ \frac{\text{A}}{x[3]}\ \frac{\text{G}}{x[4]}\ \frac{\text{T}}{x[5]}\ \frac{\text{T}}{x[6]}\ \frac{\text{A}}{x[7]}\ \frac{\text{C}}{x[8]}\ \frac{\text{C}}{x[9]}$$

$$\frac{\text{T}}{y[0]}\ \frac{\text{A}}{y[1]}\ \frac{\text{A}}{y[2]}\ \frac{\text{G}}{y[3]}\ \frac{\text{G}}{y[4]}\ \frac{\text{T}}{y[5]}\ \frac{\text{C}}{y[6]}\ \frac{\text{A}}{y[7]}.$$

  假設opt(i,j)為子序列x[i.. 9]與y[j..7]的最佳對齊成本。則opt(0,0)為x[0.. 9]與y[0..7]的最佳對齊成本，也就是我們想執行的對齊工作

最佳的對齊方式必定從以下的一種方式開始

- x[0]與y[0]對齊。若x[0]=y[0]，則第一個對齊點並沒有任何罰點產生，而若x[0]≠y[0]，則會有一個罰點產生
- x[0]與一個gap對齊，因此在第一個對齊點的罰點為2
- y[0]與一個gap對齊，因此在第一個對齊點的罰點為2

- 假設在x[0..9]和y[0..7]最佳對齊Aopt中，x[0]與y[0]對齊。那麼此種對齊方式將包含x[1..9]與y[1..7]的一種對齊方式B。
- 假設這不是這兩個子序列的最佳對齊方式。則有另一種成本更低的對齊方式C。於是，包含著對齊之x[0]與y[0]的對齊方式C，將造成x[0..9]和y[0..7]會有一個比Aopt更小成本的對齊方式。
- 因此，B必須是最佳的對齊方式。
- 同樣地，如果在x[0 ..9]和y[0 ..7]的最佳對齊方式中，
  - x[0]與gap對齊，則此種對齊方式將包含x[1..9]與y[0..7]的最佳對齊方式。
  - y[0]與gap對齊，則此種對齊方式將包含x[0..9]與y[1..7]的最佳對齊方式。

範例3.14

- 假定下列是x[0..9]和y[0..7]的某種最佳對齊方式：

$$\frac{x[0]}{\mathrm{A}}\quad\frac{x[1]}{\mathrm{A}}\quad\frac{x[2]}{\mathrm{C}}\quad\frac{x[3]}{\mathrm{A}}\quad\frac{x[4]}{\mathrm{G}}\quad\frac{x[5]}{\mathrm{T}}\quad\frac{x[6]}{\mathrm{T}}\quad\frac{x[7]}{\mathrm{A}}\quad\frac{x[8]}{\mathrm{C}}\quad\frac{x[9]}{\mathrm{C}}$$

$$\frac{\mathrm{T}}{y[0]}\quad\frac{\mathrm{A}}{y[1]}\quad -\quad\frac{\mathrm{A}}{y[2]}\quad\frac{\mathrm{G}}{y[3]}\quad\frac{\mathrm{G}}{y[4]}\quad\frac{\mathrm{T}}{y[5]}\quad -\quad\frac{\mathrm{C}}{y[6]}\quad\frac{\mathrm{A}}{y[7]}$$

則下面必定是x[1..9]和y[1..7]最佳對齊方式：

$$\frac{x[1]}{\mathrm{A}}\quad\frac{x[2]}{\mathrm{C}}\quad\frac{x[3]}{\mathrm{A}}\quad\frac{x[4]}{\mathrm{G}}\quad\frac{x[5]}{\mathrm{T}}\quad\frac{x[6]}{\mathrm{T}}\quad\frac{x[7]}{\mathrm{A}}\quad\frac{x[8]}{\mathrm{C}}\quad\frac{x[9]}{\mathrm{C}}$$

$$\frac{\mathrm{A}}{y[1]}\quad -\quad\frac{\mathrm{A}}{y[2]}\quad\frac{\mathrm{G}}{y[3]}\quad\frac{\mathrm{G}}{y[4]}\quad\frac{\mathrm{T}}{y[5]}\quad -\quad\frac{\mathrm{C}}{y[6]}\quad\frac{\mathrm{A}}{y[7]}$$

- 若設定當x[0]=y[0]時，penalty=0；其他情況penalty=1，則可以建立下列的遞迴性質：

$$opt(0, 0) = \min(opt(1, 1) + penalty, opt(1,0) + 2, opt(0,1) + 2)$$

- 雖然我們敘述此遞迴性質時，兩序列的位置是由0開始，但顯而易見地，這性質當我們由任意的位置開始都成立。因此，推廣到一般的情況，

$$opt(i, j) = \min(opt(i + 1, j + 1) + penalty, opt(i + 1, j) + 2, opt(i, j + 1) + 2)$$

- 要完成一個遞迴演算法的發展，我們需要終止條件。設m為x序列的長度，n是y序列的長度。如果我們經過x序列的終點時(i=m)，我們是在y的第j個位置，其中j<n，那麼我們必須插入$n - j$個gap。因此，一個終止條件為：

$$opt(m, j) = 2(n - j).$$

- 同樣地，如果我們經過y序列的終點時(j=n)，我們是在x的第i個位置，其中i<m，那麼我們必須插入$m - i$個gap。因此，另一個終止條件為：

$$opt(i, n) = 2(m - i)$$

演算法3.12
序列對齊問題的各個擊破版演算法

問題：找出兩同源 DNA 序列的最佳對齊方式。

輸入：長度為 m 之 DNA 序列 **x**，長度為 n 之 DNA 序列 **y**。序列以陣列的方式來表達。

輸出：兩序列的最佳對齊方式之成本。

```
void opt ( int i , int j )
{
   if ( i == m)
      opt = 2(n - j ) ;
   else if ( j == n )
      opt = 2(m - i ) ;
   else {
      if ( x [i] == x [j] )
         penalty = 0;
      else
         penalty = 1;
      opt = min( opt ( i + 1, j + 1) + penalty , opt ( i + 1, j) + 2,
            opt ( i , j + 1) + 2);
   }
}
```

該算法的主要問題

- 許多子實例被算了不止一次。例如，在第一層呼叫計算opt(0,0)，需要去計算opt(1,1)、opt(1,0)、以及opt(0,1)等值。第一層遞迴呼叫中要求出opt(1,0)的值，需要去計算opt(2,1)、opt(2,0)、以及opt(1,1)等值。opt(1,1)被獨立算了兩次

改用動態規劃

- 我們建立了一個m+1乘 n+1的陣列，如圖3.19所示

	j	0	1	2	3	4	5	6	7	8
i		T	A	A	G	G	T	C	A	–
0	A									
1	A									
2	C									
3	A									
4	G									
5	T									
6	T									
7	A									
8	C									
9	C									
10	–									

目錄

- 我們在每個序列的結尾加上了一個額外的字符來代表gap。這樣做的目的是讓我們向上的迭代方法有一個出發點。我們會要計算opt(i,j)的值，並將此值存放於此陣列的i,j單元

	j	0	1	2	3	4	5	6	7	8
i		T	A	A	G	G	T	C	A	–
0	A	**7**	8	10	12	13	15	16	18	20
1	A	6	**6**	8	10	11	13	14	16	18
2	C	6	5	**6**	8	9	11	12	14	16
3	A	7	5	**4**	6	7	9	11	12	14
4	G	9	7	5	**4**	5	7	9	10	12
5	T	8	8	6	4	**4**	5	7	8	10
6	T	9	8	7	5	3	**3**	5	6	8
7	A	11	9	7	6	4	2	**3**	4	6
8	C	13	11	9	7	5	3	**1**	3	4
9	C	14	12	10	8	6	4	2	**1**	2
10	–	16	14	12	10	8	6	4	2	**0**

圖3.20求取最佳對齊方式所使用的陣列(已填上所有值)

目錄

如何由此陣列中求得最佳對齊方式

- 首先，我們必須得到造成opt(0,0)的路徑。我們由陣列的左上角開始回溯。
- 我們由可能造成opt(0,0)的三個陣列項中選擇能夠產生正確值的那項。
- 接著，我們在所選的陣列項，重複此步驟。
- 若遇到平手的情況，則任意選擇一個。持續這個步驟直到抵達右下角。
- 得到的路徑所經的陣列項在圖3.20中以粗體顯示。

- 我們演示該路徑中前幾個值是怎麼得到的。
- 首先，將位於第i行和第j列的陣列項以[i][j]表示。接著進行如下：
- 選擇陣列項[0][0]。
- 在路徑中找到第二個陣列項。
- 檢查陣列項[0][1]。因為我們是由此格移到左邊的[0][0]，因此需要插入一個gap，也就是必須在成本中加上2點。因此我們得到

$$opt(0,1) + 2 = 8 + 2 = 10 \neq 7.$$

- 檢查陣列項[1][0]。因為我們是由此格移到上面的[0][0]，因此需要插入一個gap，也就是必須在成本中加上2點。因此我們得到

$$opt(1,0) + 2 = 6 + 2 = 8 \neq 7.$$

- 檢查陣列項[1][1]。因為我們是由此格移到左上的[0][0]，因此成本必須加上penalty的值。由於x[0]=A且y[0]=T，penalty=1，因此我們得到

$$opt(1,1) + 1 = 6 + 1 = 7.$$

- 於是，最佳路徑中的第二個陣列項為[1][1]。

求得對齊方式

- 由陣列右下角開始，我們沿著粗體的路徑。
- 每當做對角線移動到陣列項[i][j]，我們就將第i列的字元放置到x序列，將第j欄的字元放置到y序列。
- 每當往上移動到陣列項[i][j]，我們就將第i列的字元放置到x序列，將gap放置到y序列。
- 每當往左移動到陣列項[i][j]，我們就將第j欄的字元放置到y序列，將gap放置到x序列。
- 若在圖3.20的陣列上操作此程序，我們將求得下列最佳的對齊方式：

A A C A G T T A C C

T A − A G G T − C A