Chap 15. Dynamic Programming 孫光天

Dynamic programming (DP), like the divide-and-conquer, solves problems by combining the solutions to subproblems. (A tabular method). Dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub-subproblems. A dynamic-programming algorithm solves every sub-subproblem just once and then saves its answer in a table which would be used in each iteration. DP is typically applied to optimization problems.

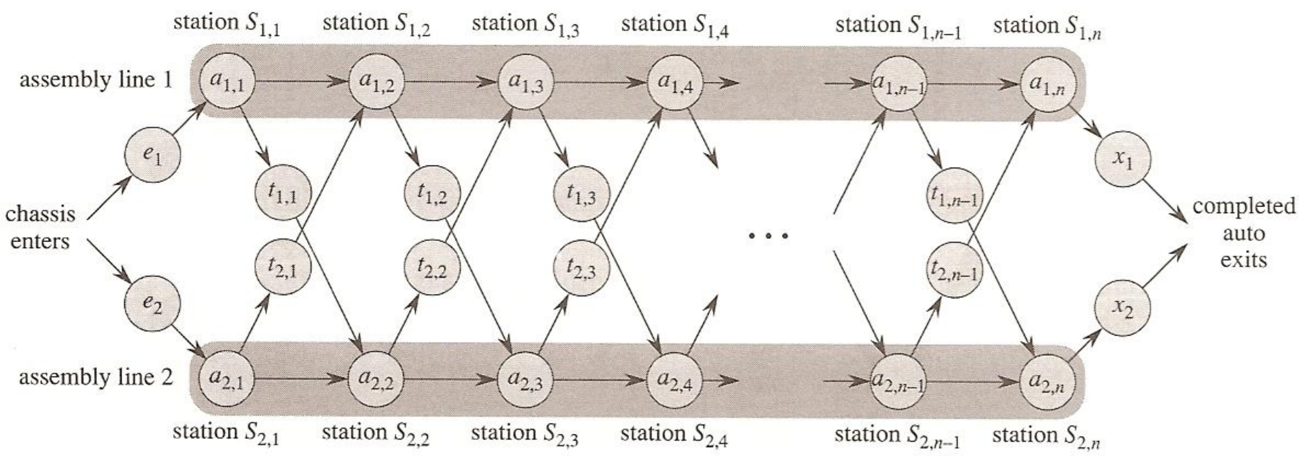
The development of a DP can be broken into a sequence of four steps.

1. **Characterize the structure of an optimization solution.**
   (定義問題的結構(定義變數))
2. **Recursively define the value of an optimal solution.**
   (求解(計算)要用到的所有變數與其遞迴關係)
3. **Compute the value of optimal solution in a bottom-up fashion.**
   (從底部(最基層)開始計算到最終結果)
4. **Construct an optimal solution from computed information.**
   (印出最佳解與如何獲得最佳解之過程)

(必須符合以上這四個步驟(要求))

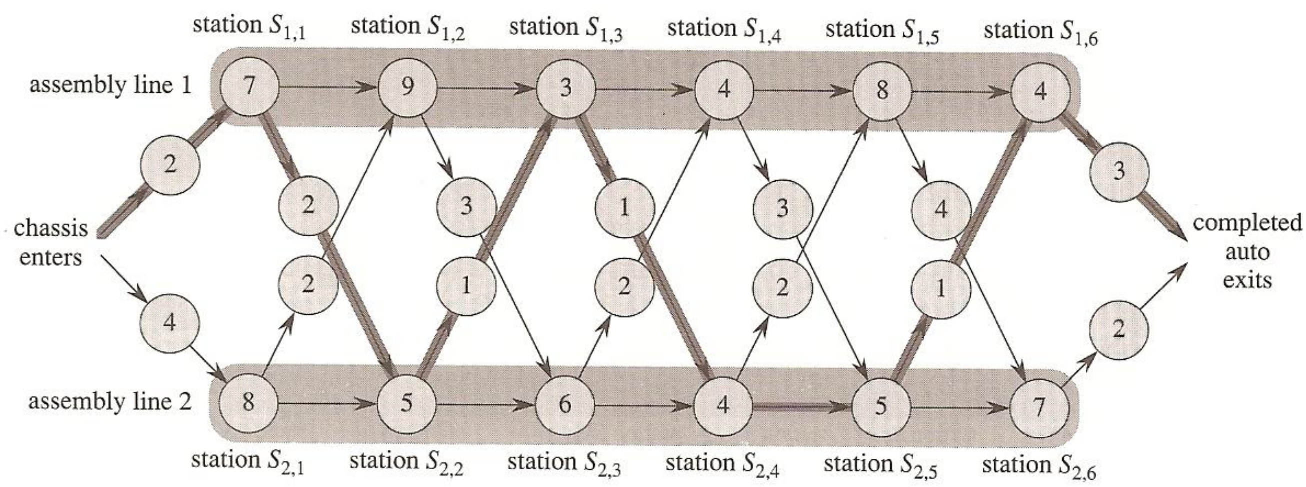
15.1 Assembly-line scheduling (組裝流程，找最短時間)

A factor has two assembly lines, shown Figure 15.1. Each assembly line has n stations, numbered j=1, 2, …, n. We denote the j-th station on line i by $S_{i,j}$. The assembly time required at $S_{i,j}$ by $a_{i,j}$. The time to transfer a chassis (底盤) away from assembly line i after station $S_{i,j}$ is $t_{i,j}$. We want to minimize the total time through the factor (see below Fig. 15.1).



There are 2^n possible ways to choose stations.

For example, (Fig 15.2)



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Apply the DP.

Step 1. The structure of the fastest way through the factory

Characterize the structure of an optimal solution. The fastest way through station $S_{1,j}$ is either

- **The fastest way through station $S_{1,j-1}$ and then directly** through station $S_{1,j}$, or

- The fastest way through station $S_{2,j-1}$ and then transfer from line 2 to line 1 ($t_{2,j-1}$), and then through station $S_{1,j}$.

Step 2. A recursive solution

Let $f_i[j]$ denote the fastest possible time to get a chassis from the starting point through station $S_{i,j}$.

The fastest way through the entire factory, we have

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2).$$

$$\begin{aligned} f_1[1] &= e_1 + a_{1,1} , \\ f_2[1] &= e_2 + a_{2,1} . \end{aligned}$$

$$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$$

for $j = 2, 3, \ldots, n$. Symmetrically, we have

$$f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$$

Combine the above two equations:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 , \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 , \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

Define $l_i[j]$ to be the line number, 1 or 2, which station j-1 is used through station $S_{i,j}$.(到 $S_{i,j}$ 最佳值的前一”站”是從哪一條生產線來)

<see fig. 15.2 (b)>

Step 3. Computing the fastest times

Let $r_i(j)$ be the number of references made to $f_i[j]$ in a recursive algorithm. （使用 recursive 要被呼叫次數）

$$r_1(n) = r_2(n) = 1\ .$$

From the recurrences (15.6) and (15.7), we have

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$$

($r_i(j)$ 將被下一 stage $j+1$ 用到 ，所以，要先計算前面 stage，後面 stage 才有值，所以，要算第 j stage 次數 = 下一 stage j+1 兩個次數相加， 成 指數（2 的次方）成長)；用 DP 如何求解:

(Backward approach, forward implementation (倒回思考,正向執行))

FASTEST-WAY(a, t, e, x, n)

$f_1[1] \leftarrow e_1 + a_{1,1}$
$f_2[1] \leftarrow e_2 + a_{2,1}$
for $j \leftarrow 2$ to n
    do if $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$
        then $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
            $l_1[j] \leftarrow 1$
        else $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
            $l_1[j] \leftarrow 2$
      if $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$
        then $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
            $l_2[j] \leftarrow 2$
        else $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
            $l_2[j] \leftarrow 1$
if $f_1[n] + x_1 \leq f_2[n] + x_2$
    then $f^* = f_1[n] + x_1$
        $l^* = 1$
    else $f^* = f_2[n] + x_2$
        $l^* = 2$

The entire procedure takes Θ(n) time. (由 O(2^n) 降至 O(n))

Step 4: Constructing the fastest way through the factory

Print out the stations used in the fastest way.

PRINT--STATIONS (l, n)

1. $i = l^*$
2. print "line", i, "station", n
3. for $j \leftarrow n$ downto 2
4. do $i \leftarrow l_i[j]$
5. print "line", i, "station", $j-1$

Output:

Line 1, station 6

Line 2, station 5

Line 2, station 4

Line 1, station 3

Line 2, station 2

Line 1, station 1

15.2 Matrix-chain multiplication

$$\begin{bmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,n} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,n} \\ \cdot & & & \\ C_{m,1} & C_{m,2} & \cdots & C_{m,n} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,k} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,k} \\ & & & \\ a_{m,1} & a_{m,2} & \cdots & a_{m,k} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ & & & \\ b_{k,1} & b_{k,2} & \cdots & b_{k,n} \end{bmatrix}$$

We give a sequence of chain (A1, A2, …, An). A product of matrices is fully parenthesized if it is either a single or the product of two fully parenthesized matrix products:

(A1(A2(A3A4))), or (A1((A2A3)A4)), …

A matrix multiplication is (A 每一 row 中元素(column)數目 = B 的 column 中元素(row)數目):

```
MATRIX-MULTIPLY(A, B)
if columns[A] ≠ rows[B]
   then error "incompatible dimensions"
   else for i ← 1 to rows[A]
            do for j ← 1 to columns[B]
                   do C[i, j] ← 0
                      for k ← 1 to columns[A]
                          do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
        return C
```

(所以，乘法運算次數有 *i * j * k* 個)

A chain <A1, A2, A3> of three matrices and the dimensions are:

10 x 100, 100 x 5, and 5 x 50. The multiplications for ((A1A2) A3) is

10 x 100 x 5 = 5000 ; 10 x 5 x 50 = 2500; 5000 + 2500 = 7500

The multiplications for (A1(A2A3)) is

100 x 5 x 50 = 25000; 10 x 100 x 50 = 50000 ; 25000 + 50000 = 75000

The matrix-chain multiplication problem is: given a chain <A1, A2, …, An> of *n* matrices, where for *i* =1, 2, …, *n*, fully parenthesize the product A1, A2, …, An in a way that minimizes the number of scalar multiplications.

Counting the number of parenthesizations

We can split the matrix-chain to subproducts:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases}$$

The solution to the recurrence is $\Omega(2^n)$.

(=
$$C_n = C_0C_{n-1} + C_1C_{n-2} + \cdots + C_{n-2}C_1 + C_{n-1}C_0$$
$$= \sum_{k=0}^{n-1} C_kC_{n-k-1}.$$

$C_0 = 1$ and $C_1 = 1$. Using the generating function (離散 2011 版 Section 7.4 exercise 41 習題解答), then $C_n = C(2n, n) / (n+1)$ (括號位置))

Step 1: The structure of an optimal parenthesization(括號)

Suppose that an optimal parenthesization(括號) of $A_i\ A_{i+1} \ldots A_j$ splits the product between A_k and A_{k+1}. Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems, finding optimal solutions to subproblem instances, and then combineing these optimal subproblem solutions.

Step 2: A recursive solution

We define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. Let m[i, j] be the number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the cost of a cheapest way $A_{1..n}$ is m[1, n].

A dimension of matrix A_i is $P_{i-1} \times P_i$; the matrix product $A_{i..k}\ A_{k+1..j}$ is $P_{i-1} \times P_k \times P_j$. Then,

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \ .$$

一 般 式 :

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \ . \end{cases}$$

Let $s[i,j] = k$ such that $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$.

($s[i,j] = k$: 矩陣乘法由 i 到 j 最佳分割點 k(由第 k matrix 之後切開))

Step 3. Computing the optimal costs

We can use a tabular, bottom up approach, compute the optimal cost. For a matrix Ai has dimension P_{i-1} x P_i. The input sequence <P_0, P_1, …, P_n>, where length[P] = n+1.

We use a table m[1..n, 1..n] for storing m[*i, j*] and a table s[1..n, 1..n] for recording the index *k* achieve optimal cost in computing m[*i, j*].

The procedure could be:

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1                      ⇨ 矩陣串“維度”個數(=矩陣個數+1)
 2  for i ← 1 to n
 3      do m[i, i] ← 0                     ⇨ 邊界條件(boundary condition)
 4  for l ← 2 to n          ▷ l is the chain length.   ⇨ 第一個 loop
 5      do for i ← 1 to n − l + 1          ⇨ 第二個 loop
 6             do j ← i + l − 1
 7                m[i, j] ← ∞
 8                for k ← i to j − 1       ⇨ 第三個 loop
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                          then m[i, j] ← q
12                               s[i, j] ← k
13  return m and s
```

(line 4 is the length of the chain, *l* = 2, it computes m[*i, i+1*]; *l*=3, it computes m[i, i+2]). (由下面(每一 row)往上建立) The running time is O(n^3) (三層 for loop (*l, i, k*)).

< see next page figure 15.3>

Step 4. Constructing an optimal solution

We can trace the optimal solution from the table s[1..n, 1..n]. First, we know the *k* in s[1..n], then partition the chain into s[1, s[1,n]] and s[s[1,n]+1, n]. The procedure is recursive to proceed.

Print-optimal-parens(s, 1, n) print the results:

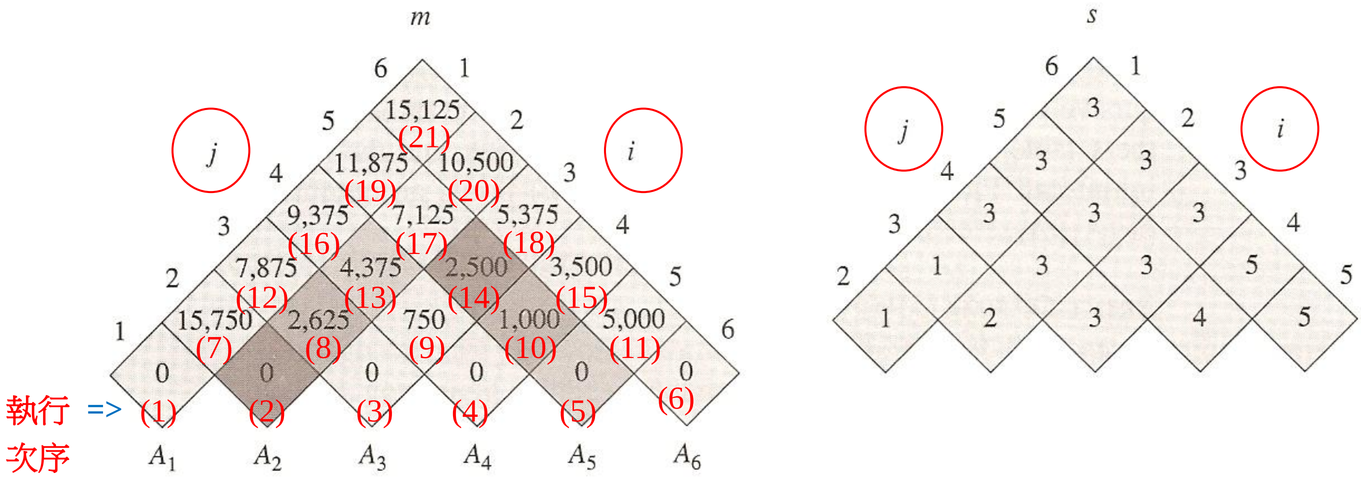


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125\,.$$

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i = j
    then print "A"_i
    else print "("
         PRINT-OPTIMAL-PARENS(s, i, s[i, j])            ⇨ 放入 stack 中
         PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)        ⇨ 放入 stack 中
         print ")"
```

In the example of Figure 15.3, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ the parenthesization $\boxed{((A_1(A_2A_3))((A_4A_5)A_6)).}$

EXERCISE: Coding "Matrix-chain-order(10)" (用 DP 方式寫)

Data: A1(30, 50), A2(50, 20), A3(20, 100), A4(100, 5), A5(5, 40), A6(40, 80), A7(80, 10), A8(10, 50), A9(50, 20), A10(20, 100)

求 Matrix m(*i,j*) & s(*i,j*) & 最佳解

(下週報告: 1. 演算法 與 Source code ; 2. 執行過程(要印出)； 3.結果)

15.3 Elements of DP

We will introduce a variant method, called memorization (備忘錄), for taking advantage of the overlapping-subproblems property.

Optimal substructure

The first step in solving an optimal problem by DP is to characterize the structure of an optimal solution. In DP, we build an optimal solution to the problem from optimal solutions to subproblems.

The common pattern in discovering optimal sub-structure includes:

1. **You show that a solution to the problem consists of a choice. Making this choice leaves one or more subproblems to be solved.**
2. **You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.**
3. **Give this choice, you determine which sub-problems ensure and how to best characterize the resulting space of subproblems.**
4. **You show that the solutions to subproblems used within the**

optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique.

To characterize the space of sub-problems, a good rule of thumb is to try keep the space as simple as possible, and then expand it as necessary. For example, factory schedule problem: we define two subproblem space : $S_{1,j}$ and $S_{2,j}$.

Optimal subproblems are used varied across problem domains in two ways:

1. How many subproblems are used in an optimal solution to the original problem.
2. How many choices we have in determining which subproblem to use in an optimal solution.

Informally, the running time of a dynamic-programming algorithm depends on the product of two factors: the number of subproblems overall and how many choices we look at for each subproblem.

In assembly-line scheduling, we had $\Theta(n)$ subproblems overall, and only two choices to examine for each, yielding a $\Theta(n)$ running time.

For matrix-chain multiplication, there were $\Theta(n^2)$ subproblems ($m(i, j)$, $i, j=1,\ldots,n$), and in each we had at most n-1 choices, giving an $O(n^3)$ running time.

Subtleties(技巧)

One should be careful not to assume that optimal sub-structure applies when it does not. (不能假設不存在的最佳子結構) Consider the following two problems in which we are given a directed graph $G(V, E)$ and vertices $u, v \in V$. 若：

(1) Unweighted shortest path: (沒有權重，即每一路徑權重均相同 =1) an optimal substructure existed.

(2) Unweighted longest simple path: does not exist an optimal sub-structure. (see figure 15.4, for example, if q -> r is the longest path, but q -> r is not the longest path from q to r. The longest path may be q -> s -> t -> r.)

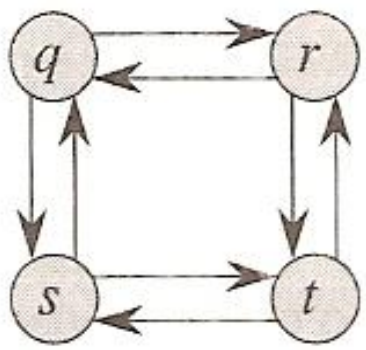
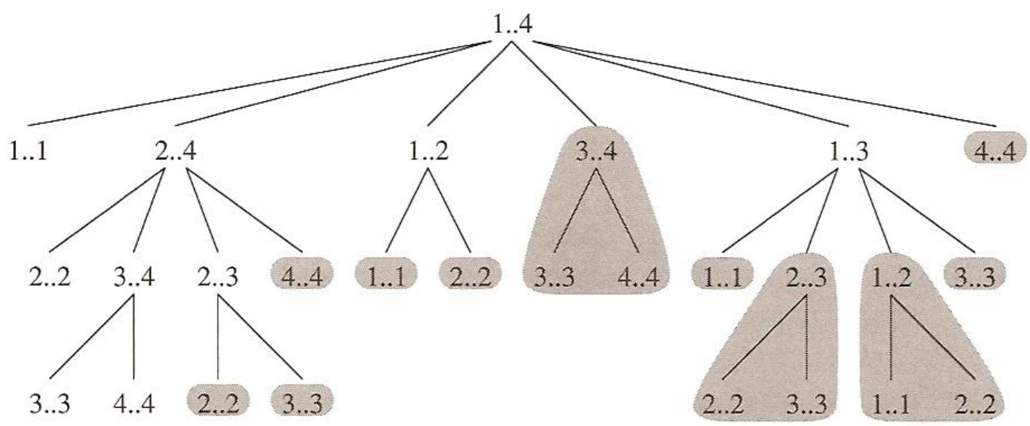


Figure 15.4

Overlapping subproblems

The dynamic programming can be applicable is that the space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over. When a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has *overlapping subproblems*. Figure 15.5 shows that m[2,2], m[3, 3] were recomputed each time.

(下圖 灰色部分均重覆)



The above tree is expanded by the following RECURSIVE-MATRIX-CHAIN(p, i, j) procedure.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i = j
    then return 0
m[i, j] ← ∞
for k ← i to j − 1
      do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
                + p_{i−1} p_k p_j
         if q < m[i, j]
            then m[i, j] ← q
return m[i, j]
```

The recurrence relation is:

$$T(1) \geq 1 ,$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1 .$$

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

Suppose T(n) ≥ 2^n.(Ω下限) By induction, i=1, T(1) ≥ 1= 2^0. For i ≥ 2,

$$\begin{aligned}
& T(n) \geq 2\sum_{i=1}^{n-1} 2^i + n \\
& = 2 \times 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\
& = 2^2 \sum_{i=0}^{n-2} 2^i + n \\
& = 2^2 (2^{n-1} - 2) + n \\
& = 2^{n+1} - 2^3 + n \\
& \geq 2^n
\end{aligned}$$

We use the substitution method is Ω(2^n).

Compare this top-down, recursive algorithm with bottom-up

dynamic programming algorithm. **The latter is more efficient and there are only $\Theta(n^2)$ different subproblems, (total $O(n^3)$) and DP solves each exactly once.**

Reconstructing an optimal solution

Some problems need to reconstruct the optimal solution, for example, the matrix-chain multiplication, we need maintain the table s[*i, j*] saves a significant amount of work (in addition the table m[*i, j*]). By storing in s[*i, j*] the index of the matrix at which we split the product $A_i A_{i+1} \ldots A_j$, we construct each choice in O(1).

Memoization(備忘錄)

(**Memoization** 存放一個特殊作用的結果，以便當下次呼叫，不需執行潛在地昂貴的計算，被貯藏的結果直接被取出(**return**))

There is a variation of DP that often the efficiency of the usual DP approach while maintaining a top-down strategy. This idea is to *memoize* the natural, but inefficient, recursive algorithm. (使用 **DP** 的 **table** 觀念，而非 **recursive algorithm (**較沒效率**))**

A memorized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has not yet to be filled in. When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned. (第一次算出某值後，下次運算時就直接查表，不再重算**)**

The memorized version of RECURSIVE-MATRIX-CHAIN is:

```
MEMOIZED-MATRIX-CHAIN(p)
  n ← length[p] − 1
  for i ← 1 to n
       do for j ← i to n
              do m[i, j] ← ∞
  return LOOKUP-CHAIN(p, 1, n)
```

⇨ 未計算值前 = 無窮大

```
LOOKUP-CHAIN(p, i, j)
  if m[i, j] < ∞
     then return m[i, j]
  if i = j
     then m[i, j] ← 0
     else for k ← i to j − 1
              do q ← LOOKUP-CHAIN(p, i, k)
                        + LOOKUP-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
                 if q < m[i, j]
                    then m[i, j] ← q
  return m[i, j]
```

⇨ 表示已存有資料
直接查表傳回值

MEMOIZED-MATRIX-CHAIN maintains a table m[1..n, 1..n] of computed of m[i, j]. There are $\Theta(n^2)$ calls of first type for table entry m[*i*, *j*], and LOOKUP-CHAIN makes recursive calls, it makes O(n) of them. (Lookup-Chain 雖然為 recursive，但若之前執行過，則不再執行(直接拿記錄，return *m*[*i*, *j*])) The total running time is $O(n^3)$. In general practice, if all subproblems must be solved at least once, a bottom-up DP outperforms (優於) a top-down memorized algorithm by a constant factor. Alternatively, if some subproblems in the subproblem space need not be solved at all, the memorized solution has the advantage of solving only those subproblems that are definitely required.

15.4 Longest common subsequence

A strand of DNA consists of a string of molecules called bases that can be represented as a string over the finite set {A, T, C, G}.(鹼基對 DNA 的成份中有四種含氮鹼基：腺嘌呤 A、胸腺嘧啶 T、鳥嘌呤 G 及胞嘧啶 C。會以 A-T，C-G 方式配對，稱為鹼基對) **For example, the DNA of one organism is S1 = ACCGTCGAGGAACCTTTCG, and another S2 may be S2 = ATTCCGGTCGGGCCTAA. We want to find a longest-common-subsequence. (**共同序列，不須連續。連續 **Chap 32) For example, X={A, B, C, B, D, A, B} and Y={B, D, C, A, B, A}, the longest common subsequence (LCS) is {B, C, B, A}. Then, we can apply the DP to solve the LCS as follows.**

Step 1. Characterizing a longest common subsequence

A brute-force approach is to enumerate all subsequences of X and check each subsequence to see if it is also a subsequence of Y. A subset of indices {1, 2, …, m} of X. There are 2^m subsequences of X. (impractical)

The *i*th *prefix*: X={A, B, C, B, D, A, B}, then X_4 ={A, B, C, B} and X_0 is the empty sequence. Then, the optimal substructure of LCS is:

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

(2 & 3: 我們要求**(X_m, Y_n)**的 **LCS**，要先知道**(X_{m-1}, Y_n)**與**(X_m, Y_{n-1})**的 **LCS)**(將大問題變為較小問題，所以，由最後長度 **m** 或 **n** 開始思考**)**

Step 2. A recursive solution

Theorem 15.1 implies that there are either one or two subproblems to

examine when finding an LCS of X ={x1, x2, …, x_m} and Y ={y1, y2, .., y_n}. If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}. Let us define c[*i, j*] to be the length of an LCS of the sequences X_i and Y_j. The optimal substructure of the LCS problem gives the recursive formula

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

Step 3. Computing the length of an LCS

There are Θ(mn) distinct subproblems, and we can use DP to compute the solutions bottom up. It stores the c[i, j] values in a table c[0..m, 0..n]. It also maintains the table b[1..m, 1..n] to simplify construction of an optimal solution. The procedure returns the b and c tables; c[m, n] contains the length of an LCS of X and Y. (bottom up 建立 c[m, n]，forward implementation)

```
LCS-LENGTH(X, Y)
1   m ← length[X]
2   n ← length[Y]
3   for i ← 1 to m
4        do c[i, 0] ← 0
5   for j ← 0 to n
6        do c[0, j] ← 0
7   for i ← 1 to m                       ⇨ i 為縱座標(row)
8        do for j ← 1 to n               ⇨ 橫向(row)優先求值
9               do if x_i = y_j
10                    then c[i, j] ← c[i − 1, j − 1] + 1
11                         b[i, j] ← "↖"
12                    else if c[i − 1, j] ≥ c[i, j − 1]
13                            then c[i, j] ← c[i − 1, j]
14                                 b[i, j] ← "↑"   ⇨ 值來自上一層
15                            else c[i, j] ← c[i, j − 1]
16                                 b[i, j] ← "←"   ⇨ 值來自前一個
17  return c and b
```

Line 8: 以 **row (j =1 ~ n)** 優先建 **table**。

Lines 10~11: 表示 x_i 與 y_j 有相同 **base (data)**。

Line 12: 表示 X_{i-1} 與 Y_j 比 X_i 與 Y_{j-1} 有更多(或相同)的序列，則

Line 13: c[i,j] = c[i-1, j]

Line 15: 則正好與 **line 12** 相反，X_{i-1} 與 Y_j 比 X_i 與 Y_{j-1} 有較少的序列，所以，**c[i,j] = c[i, j-1]**

The following figure 15.6 shows the tables produced by LCS-LENGTH on the sequences X = {A, B, C, B, D, A, B} and Y = {B, D, C, A, B, A}. The running time of the procedure is O(mn).

/* 若要連續共同片段，**line 12~16** 改為一行 **line 12 : c[i, j] = 0** */

/* 若要與某已知一片段做比對，參考 **chap 32** */

j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	(2) 0	0	0	0	0
1	A	0	(3) ↑ 0	(4) ↑ 0	(5) ↑ 0	↖ (6) 1	(7) ←1	↖ (8) 1
2	B	(1) 0	↖ (9) 1	(10) ←1	(11) ←1	↑ 1	↖ 2	←2
3	C	0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 4. Constructing an LCS

Begin at b[m, n] and trace through the table following the arrows. The initial invocation is PRINT-LCS(b, X, length[X], length[Y]).

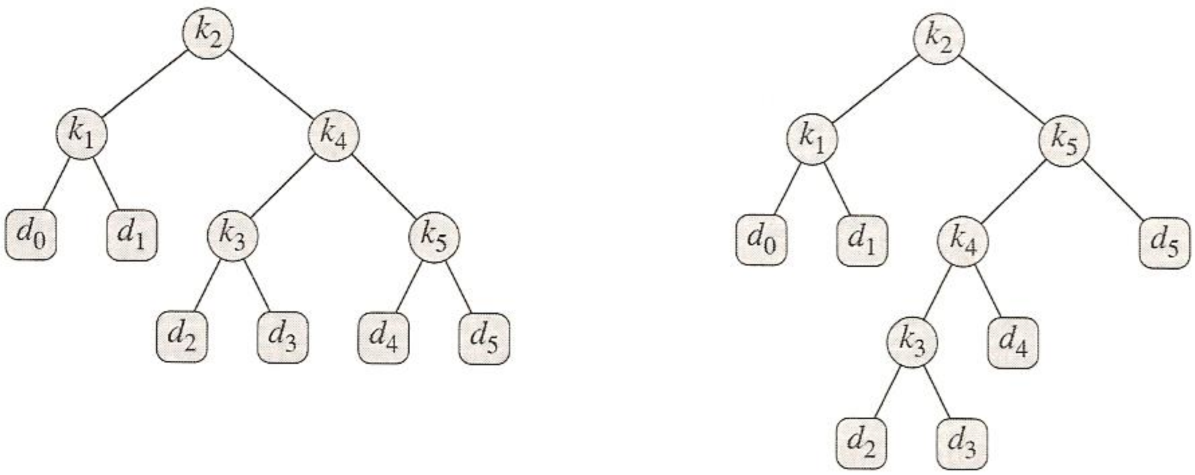
```
PRINT-LCS(b, X, i, j)
if i = 0 or j = 0
    then return
if b[i, j] = "↖"                      ⇨ 兩序列有相同 data
    then PRINT-LCS(b, X, i − 1, j − 1)
        print x_i                     ⇨ ↖表示有相同 data，x_i 印出
elseif b[i, j] = "↑"
    then PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

EXERCISE: X={A, B, C, A, C, B, D, A, B} & Y={B, D, C, A, B, D,A}

Coding program & show the solution & b[m, n] & c[m, n] (上圖)

15.5 Optimal binary search trees

When searching a key in a binary search tree is one plus the depth of the node containing the key.(深度加 1 (因為 root 也比一次) 為搜尋到此 key 的 cost) We want words that occur frequently in the text to be placed nearer the root. It is known as an optimal binary search tree. For each key K_i , we have a probability p_i that a search would be for K_i. A dummy key d_i represents all value between K_i and K_{i+1} with a probability q_i. Figure 15.7 shows the results:



(a) (b)

Figure 15.7 Two binary search trees for a set of $n = 5$ keys with the following probabiliti

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

⇨ key k_i 機率
⇨ dummy key d_i 機率

(a) A binary search tree with expected search cost 2.80. **(b)** A binary search tree with expecte cost 2.75. This tree is optimal.

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 .$$

$$\begin{aligned} \mathrm{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\operatorname{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (\operatorname{depth}_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n} \operatorname{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n} \operatorname{depth}_T(d_i)\cdot q_i \,, \qquad (\end{aligned}$$

For example,

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

We want to construct a binary search tree whose expected search cost is smallest. We call such a tree an *optimal binary search tree*.

Solving it by DP:

Step 1: The structure of an optimal binary search tree.(類似矩陣串)

Consider any subtree, it contains keys K_i, … K_j must also have as its leaves dummy keys d_{i-1}, .., d_j. If an optimal binary search tree T has a subtree T' containing keys K_i, … K_j, then this subtree T' must be optimal. So, how to select the root of subtree become the issue of finding the optimal binary search tree. Giving keys K_i, … K_j, and the root K_r, then the left subtree of the root K_r contains the keys K_i, … K_{r-1} (and dummy keys d_{i-1}, .., d_{r-1}), and the right subtree contains the keys K_{r+1}, .., K_j (and dummy keys d_r, .., d_j). If the root is K_i,

then the left subtree only contains the dummy key d_{i-1}(no actual keys). If the root is K_j, then the right subtree only contains the dummy key d_j.

Step 2: A recursive solution.

Let e[*i, j*] be the expect cost of searching an optimal binary search tree containing the keys K_i, .. K_j. Ultimately(最終), we wish to compute e[1, n]. When $j = i-1$, we just have the dummy key d_{i-1}, the expected search cost is e[i, i-1] = q_{i-1}.(邊界條件) Then the expected search cost of the subtree $T_{i,j}$ is the sum of probabilities as:

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \;. \quad \text{(第 0 層的 cost (機率總和))}$$

所以，$W(i, i-1) = q_{i-1}$. Thus, if K_r is the root of an optimal subtree containing keys K_i, .., K_j, we have: [在 root 的下一層，比較次數加 1]

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

Noting that

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j) \;,$$

Then, e[*i*, *j*] become (左右子樹比 K_r 下一層，機率 *w*(*i*, *j*)要多加一次):

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) \;.$$

Consider the boundary condition $j = i-1$.

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \;, \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j \;. \end{cases}$$

Step 3: Computing the expected search cost of an optimal binary search tree.

We store e[i, j] values in a table e[1.. n+1, 0 ..n].

(e[n+1, n] is used to store the dummy keys d_n , and

e[1, 0] is used to store the dummy keys d_0).

w(i, j) stores the probability in a table w[1.. n+1, 0 ..n].

root(i, j) stores the root for keys i ~j in a table root[1.. n, 1 ..n].

The pseudocode of the optimal binary search tree is:

```
OPTIMAL-BST(p, q, n)
1  for i ← 1 to n + 1
2       do e[i, i − 1] ← q_{i−1}
3          w[i, i − 1] ← q_{i−1}
4  for l ← 1 to n
5       do for i ← 1 to n − l + 1
6               do j ← i + l − 1
7                  e[i, j] ← ∞
8                  w[i, j] ← w[i, j − 1] + p_j + q_j
9                  for r ← i to j
10                      do t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
11                         if t < e[i, j]
12                           then e[i, j] ← t
13                                root[i, j] ← r
14 return e and root
```

Line 8: 求 *W(i, j)*，可由 *W(i, j-1)* + p_j + q_j 累計求來(由下面(每一 row)往上建立：W(1,0), W(2,1),… ,W(n+1,n) & e(1,0), e(2, 1),e(3,2), …, e(n,n-1) [(初始值)為邊界條件= $q_0, q_1, q_2, …, q_n$];

子樹大小由 *l*=1(一個資料節點)開始算，逐漸增加 *l*=2,…, *l*=n;

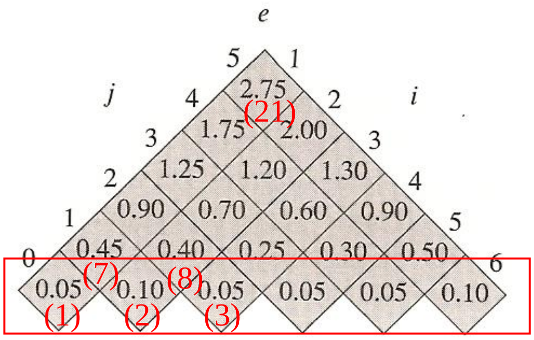
l=1：W(1,1), W(2,2), …; *l*=2: W(1,2), W(2,3), …, W(n-1,n);…;

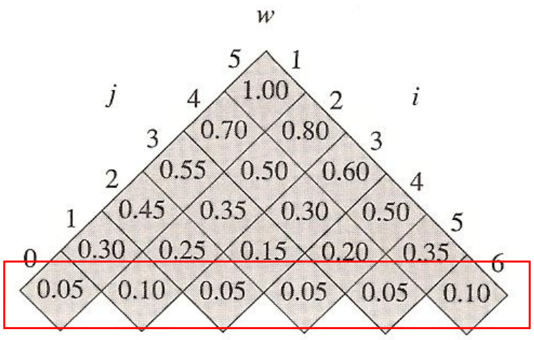
l=n-1: W(1,n-1), W(2,n); *l*=n: W(1,n))。 The index *l* is the width of subtree $T_{i,j}$, (類似前面 矩陣串乘法問題，root *r* 就是切割點位置) and *r* is the root of subtree. For *l* =1, we compute e[i, i] and w[i, i]. 接下來，再求 *l* =2 的所有可能, *l* =3 … The runnme is $O(n^3)$.

For example, e[1, 1] = e[1, 0] +e[2,1]+w[1,1] = 0.05 + 0.1 + 0.3 = 0.45（上圖 15.7(b)(左子圖)為答案）

所以，e[1, 3]就有三種可能，找 min:

e[1, 0] +e[2,3]+w[1,3] ; e[1, 1] +e[3,3]+w[1,3]; e[1, 2] +e[4,3]+w[1,3]



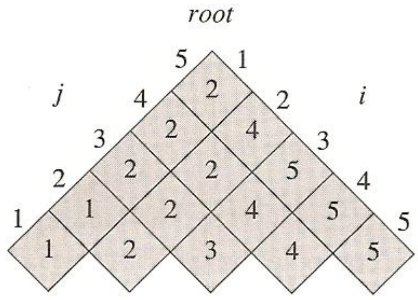


邊界條件

執行次序

同 p.15-9

Fig. 15.3



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80