CH3、Dynamic Programming

動態規劃

考題重點(目錄)

一、Dynamic Programming 的基本概念
二、背包問題
三、LSC 及其應用
四、Matrix-Chain Multiplication
五、其他重要問題

1. Bellman-Ford
2. Floyd-Warshall
3. OBST
4. …

Dynamic Programming 共 6 個例子：

1. 0/1 背包
2. LSC
3. Matrix Chain Multiplication
4. Bellman-Ford
5. Floyd-Warshall
6. OBST

Greedy 共 6 個例子：

1. Fractional 背包
2. Huffman
3. Kruskal’s
4. Prim’s
5. Sollin’s
6. Dijkstra’s

Dynamic Programming **的基本概念**

一、What is Dynamic Programming？

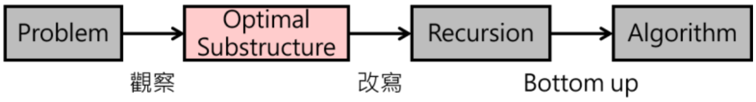
Dynamic Programming 是一種將已計算出的結果，記錄在表格中的技巧，目的是為了避免重複計算相同子問題，以 Bottom-Up 方式進行運算

二、Why Dynamic Programming？

用 Fibonacci Number 的例子來看：求 F5

1. 用 Divide-and-Conquer 求：要展開太多次(14 次)、Overlapping Subproblem，屬於 Top-Down 方式
2. 用 Dynamic Programming：使用表格，重複使用已計算出之結果

三、設計 Dynamic Programming Algorithm 的流程



Optimal Substructure 為"一個問題的最佳解如何由其 Subproblem 的最佳解所構成"

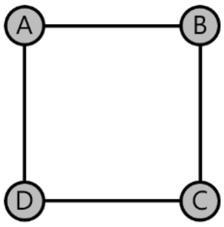
例：Shortest Path Proglem

Optimal Substructure：

A 經 C 到 B 點的最短路行 = A 到 C 之最短路徑 + C 到 B 點之最短路徑

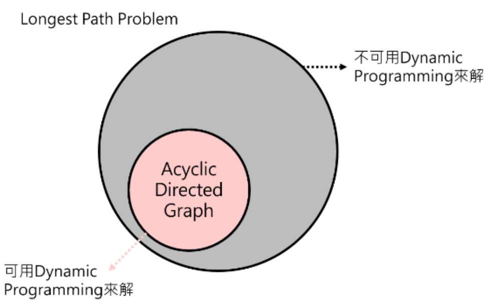
例(98 交大)：Longest Path Problem 沒有 Optimal Subtructure

⇨ 無法用 Dynamic Programming 來解



A 到 D = ABCD，但不等於：A 到 C(ABC) + C 到 D(CBAD)

Note：



要在『各種情況下都可以』用 Dynamic Programming 解，才可以說此問題『可用 Dynamic Programming 解』

例(99 交大 p3-5.1.2)：True/False

1. Dynamic Programming always provides polynomial time algorithms.
2. Huffman coding for compression is a typical Dynamic Programming algorithm.
3. Dynamic programming uses tables to design algorithms.
4. Optimal substructure is an important element of Dynamic Programming algorithm.
5. The single source shortest path problem has the property of optimal substructure.

1. *False：未必，反例為：Subset-Sum Problem 為 NPC，用暴力法：$O(2^n)$，用 Dynamic Programming：$O(n2^{n/2})$。Dynamic Programmin 較有效率，但仍為 Exponential Time*
2. *False：為典型的 Greedy Algorithm*
3. *True*
4. *True*
5. *True(見上例)*

Knapsack Problem

一、Problem Statement

1. Fractional KP
   Input: n 個 item(第 i 個重 w_i，值 v_i)及 W(最大負重)
   Output: 最大 profit
   限制：
   (1) 取物總重 ≤W
   (2) 取物時可只取物品的部分(*)
   有一個 Item 重 3kg，可只取其 2kg
2. 0/1 KP
   同上，但取物時得『全取』

二、Fractional KP

1. 解法：Greedy
   從目前 V_i/w_i 最高物品開始取(重複)，直到物品拿完 or 負重=W
2. 演算法：p3-10
   Time Complexity：$\Theta(n \lg n)$
3. 例：W=5，解 Fractional KP

Item	V_i	w_i
1	10	2
2	6	1
3	12	3

1. *取 Item 2, 1kg，剩下負重 4kg, \$6*
2. *取 Item 1, 2kg，剩下負重 2kg, \$16*
3. *取 Item 3, 2kg，剩下負重 0kg, \$24*

4. 例(98 交大)：p3-66.10
   Maximize $\sum v_i x_i$ *(拿 Item i 得多少\$)*
   Subject to $\sum w_i x_i \le W$ *(拿 Item i 負了多少 kg 的重)*, $0 \le x_i \le 1$ *(取 Itme i 的幾分之幾)*
   最佳解的 x_1 = 1, if $w_i \le W$*(可全拿 item 1)*；
   = W/w_1, if $w_1 > W$*(用盡所有力氣取 item 1)*

三、0/1 KP

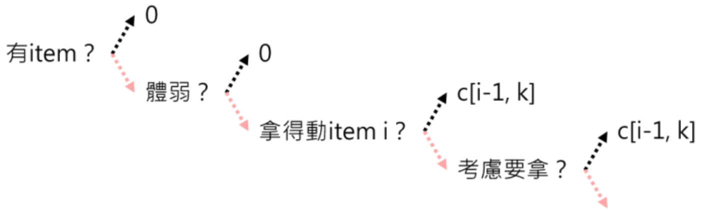
1. 0/1 KP 無法用 Greedy 解
   P3-7.11
2. 用 Dynamic Programming 解
   (前提：Item 的重量是正整數)

[Recursion]

例(98 交大)：

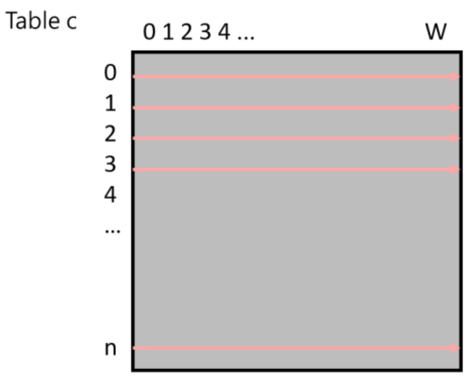
令 c[i, k]為考慮 Item 1~i 且負重 k 下的最大 profit

C[i, k] = 0, if i=0(*無物*) or k=0(*體弱*)
= c[i-1, k], if k<wi(*拿不動第 i 項物品*)
= max(c[i-1, k](*不取 item i*), vi+c[i-1, k-wi](*x 取 item i，拿了 item i，造成可負重下降*)),
if k≥wi(*拿得動 item i*)



[Algorithm]

Bottom-up



程式：

```
for k ← 0 to W                                    //無物時
        c[0, k] ← 0;
for i ← 1 to n                                    //體弱
{
        c[i, 0] ← 0;
        for k ← 1 to W
        {
                if(k<wi)                          //拿不動 item i
                        c[i, k] ← c[i+1, k];
                else                              //拿得動 item i
                        c[i, k] ← max(c[i-1, k], vi+c[i-1, k-wi]);
        }
}
```

Time Complexity：Θ(nW)
Space Complexity：Θ(nW) *(指 Table c 的大小)*

例：W=5

Item	V_i	w_i
1	10	2
2	6	1
3	12	3

解 0/1 KP

	0	*1*	*2*	*3*	*4*	*5*
0	*0*	*0*	*0*	*0*	*0*	*0*
1(10)	*0*	*1.*	*2.*			
2(6)						
3(12)						

1. *$k=1 < w_1=2 \Longrightarrow c[1, 1]=c[0, 1] =0$*
2. *$k=2 \geq w_1=2$ (拿得動), $c[1, 2] = max(c[0, 2], 10+c[0, 0]) = 10$*

以此類推，得到表格如下：

	0	*1*	*2*	*3*	*4*	*5*
0	*0*	*0*	*0*	*0*	*0*	*0*
1(10)	*0*	*0*	*10*	*10*	*10*	*10*
2(6)	*0*	*6*	*10*	*16*	*16*	*16*
3(12)	*0*	*6*	*10*	*3.*	*18*	*4.*

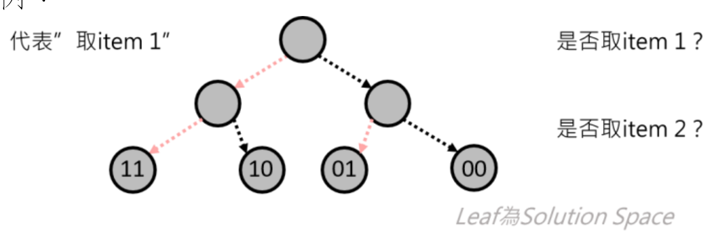
3. *$k=3 \geq w3=3$ (拿得動), $c[3, 3]=max(c[2, 3], 12+c[2, 0]) = 16$*
4. *$k=5 \geq w3=3$ (拿得動), $c[2, 5]=max(c[2, 5], 12+c[2, 2]) = 22$*

最後推出表格如下

	0	*1*	*2*	*3*	*4*	*5*
0	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	*0*	*10*	*10*	*10*	*10*
2	*0*	*6*	*10*	*16*	*16*	*16*
3	*0*	*6*	*10*	*16*	*18*	*22*

Note：
1. *此法僅限於 Item 重為正整數時*
2. *0/1 KP 為 NPC (Θ(nW) 為 pseudo-polynomical)*

3. 用 Branch-and-Bound 解(CH8-1)
   (1) Logically：將求最佳解的過程視為是在 State-Space Tree 中做 Search
   例：



   (2) Pratically：設計一個 Bounding function 去估以目前狀態可到佳解的可能性，每次展開 Bounding function 最大的 Node，且永不展開 Bounding function ≤ 目前最佳解的 Node

(3) 以 0/1 KP 為例：

先依 v_i/w_i 大小將 Item 重新排序

例：W=4

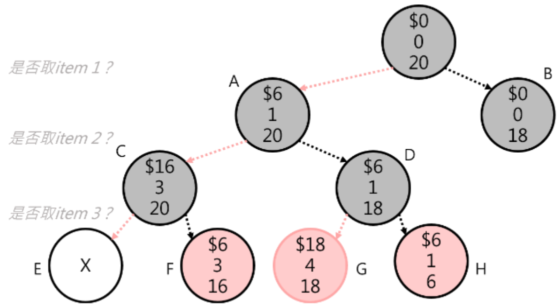
Item	v_i	w_i
1	6	1
2	10	2
3	12	3

設計 Bounding function 如下：

Bf(N) = 目前在 N 可得的$ + 以 Fractional KP 拿剩下 item 可得的$

Note：Bounding function 值為以目前狀態而言，可拿到$的 Upper Bound

每一個 Node 要記錄下的 State：



1.展開 Root

2.展開 A(因為 Bounding function 值最大)

3.展開 C

4. E 為 Infeasible Solution(超重)

5. F 為一解，設 MAX=16

6.展開 D

7. G 和 H 均為一解，設 MAX=18

8.不用展開 B(因為其 Bounding function≤MAX)

⟹ 18(取 Item 1 和 Item 3)

Note：

1. *對於 NPC 的 Problem 來說，可用 Branch-and-Bound 解*
2. *在此架構上，Bounding function 的設計會是影響效能的最大關鍵*
3. *Time Compleity：O(leaf 數)：leaf 數看問題本身*
   ex：組合性的：2^n、排列性的：n!

Longest Common Subsequence

一、Terms

1. Sequence
   Ex：X=<a, b, c, a>
2. Subsequence
   Ex：<a, c>為 X 的 Subsequence*(從 X 中取出，順序不變)*
3. Prefix*(從前方取順序一段)*
   Ex：X_3=<a, b, c>
4. Common Subsequence
   Ex：Y=<a, c, b, c>，則<a, c>為 X 和 Y 的 Common Subsequence
5. Longest Common Subsequence
   Ex：<a, b, c>為 X 和 Y 的 LCS*(LCS 不一定唯一)*

二、Naïve Approach

1. 列舉出 X 所有 Subsequence*($\Theta(2^m)$)*
2. 列舉出 Y 所有 Subsequence*($\Theta(2^n)$)*
3. 找 1、2 相同的 Subseqeunce 中，最長者*(一定要 Exponential Time)*

三、Dynamic Programming

[Recursion]

令 c[i, j]為 LC(X_i, Y_j)的長　　*//X 的前 i 個字、Y 的前 j 個字*

c[i, j] = 0　　, if i=0 or j=0
//有一者為空序列

= c[i-1, j-1]+1　　, if x_i=j×i
//Xi 的最後一字 = Yj 的最後一字

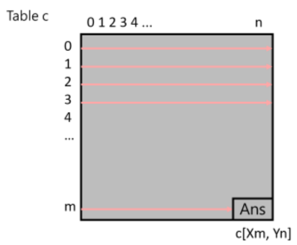
= max(c[i-1, j], c[i, j-1])　　, if $X_i \neq Y_j$
//最後一字不同

例：X=<a, b, c, a>、Y=<a, c, b, a> => LCS(X, Y) = <a, b, a>
長度：c[X_3, Y_3]+1

X=<a, b, c, a>、Y=<a, b, b ,c>
LCS(X_i, Y_{j-1}) = LCS(<a, b, c, a>、<a, b, b>) = <a, b>
LCS(X_{i-1}, Y_j) = LCS(<a, b, c>、<a, b, b, c>) = <a, b, c>
⇨ LCS(X, Y) = <a, b, c>

[Algorithm]：(Bottom-up)

Table c：



程式：

```
for j ← 0 to n                  //i=0，第一行=0
        c[0, j] ← 0;
for i ← 1 to m                  //j=0，第一列=0
        c[i, 0] ← 0;
for i ← 1 to m
        for j ← 1 to n
        {
                if(xi=yj)
                        c[i, j]  ←  c[i-1, j-1]+1;
                else
                        c[i, j]  ←  max(c[i-1, j], c[i, j-1]);
        }
```

Time Complexity：$\Theta(mn)$
Space Complexity：$\Theta(mn)$　*//指 table c 的大小*

Note：口訣：相同斜上再加 1、不同誰大就抄誰(Copy)

<table>
<tr><td></td><td>j-1</td><td>j</td></tr>
<tr><td>i-1</td><td></td><td>|</td></tr>
<tr><td>i</td><td>-</td><td>\</td></tr>
</table>

Ex：X=<a, b, a, c>、Y=<a, b, c, a>，求 LCS(X, Y)

<table>
<tr><td></td><td></td><td></td><td>a</td><td>b</td><td>c</td><td>a</td></tr>
<tr><td></td><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>a</td><td>1</td><td>0</td><td>1\</td><td>1-</td><td>1-</td><td>1\</td></tr>
<tr><td>b</td><td>2</td><td>0</td><td>1|</td><td>2\</td><td>2-</td><td>2-</td></tr>
<tr><td>a</td><td>3</td><td>0</td><td>1\</td><td>2|</td><td>2-</td><td>3\</td></tr>
<tr><td>c</td><td>4</td><td>0</td><td>1|</td><td>2|</td><td>3\</td><td>3-</td></tr>
</table>

LCS(X, Y)的長度=3

LCS 求法：從最後一格出發，按方向前進，遇\則圈字、遇 0 則停 => 圈的字為 LCS

<table>
<tr><td></td><td></td><td></td><td>a</td><td>b</td><td>c</td><td>a</td></tr>
<tr><td></td><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td></td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>a</td><td>1</td><td>0</td><td>1\</td><td>1-</td><td>1-</td><td>1\</td></tr>
<tr><td>b</td><td>2</td><td>0</td><td>1|</td><td>2\</td><td>2-</td><td>2-</td></tr>
<tr><td>a</td><td>3</td><td>0</td><td>1\</td><td>2|</td><td>2-</td><td>3\</td></tr>
<tr><td>c</td><td>4</td><td>0</td><td>1|</td><td>2|</td><td>3\</td><td>3-</td></tr>
</table>

⇨　*LCS=<a, b, c>*

四、LCS 的應用

1. Longest Increasing Subsequence(LIS)

例(99 中央)：input: X=<5, 1, 3, 2, 4>、output: LIS(X)=<1, 2, 4>

Algorith

1. *Y ← sort(X); //由小到大*
2. *Return LCS(X, Y);*

Time Complexity：$\Theta(n^2)$

1. *$\Theta(n \lg n)$*
2. *$\Theta(n^2)$)*

2. Longest Common Substring

例：input: X=<a, b, a, c>、Y=<a, b, c, a>、output:<a, b>

(<a, b, c>不為 Longest Common Substring)

			a	b	c	a
		0	1	2	3	4
	0	0	0	0	0	0
a	1	0	1\	1-	1-	1\
b	2	0	1|	2\	2-	2-
a	3	0	1\	2|	2-	3\
c	4	0	1|	2|	3\	3-

1. *找最長連續斜上(\)*
2. *標上(|)或左(-)者，其值為 0(代表在 X 或 Y 斷掉)*

Note：also see: CH3-5、p3-58.4

Matrix-Chain Multiplication

一、Problem Statement

Input：n 個 matrix 的 size P[0: n](其中 A_i 的 size 為 $P_{i-1} \times P_i$)

Output：算出 $A_1 \times A_2 \times \ldots \times A_n$ 所需最少純量乘法數

例：給定 3 個 Matrix 的 size 如下：

A1: 10 × 100；A2: 100×5；A3: 5 × 50，求算出 A1 × A2 × A3 所需最少純量乘法數？

列出所有乘法順序：

*A：p×q、B：q ×r，則 A*B 需要 p*q*r 個純量乘法*

A1 ×(A2 ×A3)：100 ×5 ×50(括號) + 10×100×50 = 75000

(A1×A2)×A3：10 ×100 ×5(括號) + 10×5×50 = 7500

Note：長 n 的 Matrix-Chain，則有 $C_{n-1} = 1/[(n-1)+1] \times C_{n-1}^{2(n-1)}$種相異的乘法順序

⇨ *列出所有乘法順序要 Exponential Time!*

Note(p3-64.7)：若每個 Matrix 均同 ⟹ 乘法順序不會影響所需乘法

⇨ *欲加快 Matrix-Chain 運算的速度，要使用加快 2 個矩陣相乘的演算法(ex：Strassen's Algorithm)*

二、Dynamic Programming 解法

[Recursion]

令 m[i, j]為算出 $A_i \times \ldots \times A_j$ 所需最少，則：

m[i, j]	= 0,	if i=j	//長為 1 時
	= min(m[i, k] + m[k+1, j] + $P_{i-1} \times P_k \times P_j$),	if i≤k≤j-1	//長≥2 時

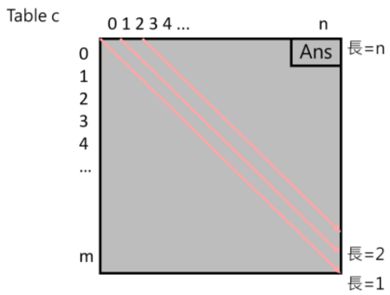
Note：用以下情境來理解：

用 Divide-and-Conquer 求 $A_i \times \ldots \times A_j = [A_i \times \ldots \times A_k] * [A_{k+1} \times \ldots \times A_j]$

遞迴[前] * [後]再合併

[演算法]

Bottom-Up(以『長』做 Bottom-Up)



程式：

```
for i ← 0 to n                                  //長=1
        m[i, j] ← 0;
for l ← 1 to m                                  //matrix chain 長=2~n
{
        for i←1 to n-l+1                        //起點+終點=長 l 的 matrix chain：Ai*...*Aj
        {
                j ← i+l-1;
                m[i, j] ← 無限大;                //衛兵
                for k ← i to j-1                //切法
                {
                        q←m[i, k]+,m[k+1, j]+Pi-1*Pk*Pj;        //此切法的最少乘法數
                        if(q < m[i, j])                         //目前的較好
                        {
                                m[i, j] ← q;                    //記下次數
                                s[i, j] ← k;                    //記下切點
                        }
                }
        }
}
```

Time Complexity：$\Theta(n^3)$

Space Complexity：$\Theta(n^2)$

例：A1：3×3、A2：3×7、A3：7×2、A4：2×9、A5：9×4

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

	2	3	4	5
1	1.			
2				
3				2.
4				

1. $m[1, 2] = min(m[1, k]+m[k+1, 2]+P_0*P_1*P_2) = m[1, 1]+m[2, 2]+P_0*P_1*P_2 = 0+0+3*3*7 = 63$
2. $m[3, 5] = min(m[3, k]+m[k+1, 5]+P_2*P_k*P_5) =(k=3): m[3, 3]+m[4, 5]+P_2*P_3*P_5=0+72+56=128$
   $=(k=4): m[3, 4]+m[5, 5]+P_2*P_4*P_5=126+0+252=378$

[用看的]

3. $m[1, 4]$：求 $A_1*\ldots*A_4$ 的最少乘法數：
   $k=1$：$[A_1][A_2\ A_3\ A_4] = m[1, 1]+m[2, 4]+3*3*9 =$
   $k=2$：$[A_1\ A_2][A_3\ A_4] = m[1, 2]+m[3, 4]+3*7*9 =$
   $k=3$：$[A_1\ A_2\ A_3][A_4] = m[1, 3]+m[4, 4]+3*2*9 = 114$

以此類推，求得表格

	1	2	3	4	5
1	0	63	60	114	156
2		0	42	96	138
3			0	126	128
4				0	72
5					0

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

因此，最少乘法數：156；最佳乘法順序：((A1)×(A2×A3))×(A4×A5)

例 p3-61.6：

15125

Subset Sum Problem(**子集和問題**)

問題：給定一個 Set S={a1, …, an}與一 k 值，問 S 的子集合中，是否有一集合元素和等於 k？

想法：

為 0/1 背包問題之延伸問題，解法上一樣採 Recursive，並且判斷 ak 元素包含、與不包含之情況：若不包含，則找『扣除 ai 後之集合=k』；若包含，則找『扣除 ai 後之集合=k-ai』

程式：

```
bool isSubsetSum(int set[], int n, int sum)
{
    //基本判斷回傳值
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;

    //若集合最後一數大於 Sum，直接扣除後再找(不過此舉有些多餘，但可增加速度)
    if (set[n-1] > sum)
        return isSubsetSum(set, n-1, sum);

    /*檢查以下兩種包含 ai 與不包含 ai 之狀況
        (a) 包含則兩邊都扣除後 Recursvie
        (b) 不包含則只從 Set 中扣除 ai 後繼續 Recursive     */
    return isSubsetSum(set, n-1, sum) || isSubsetSum(set, n-1, sum-set[n-1]);
}

// Demo 用的主程式
int main()
{
   int set[] = {3, 34, 4, 12, 5, 2};
   int sum = 9;
   int n = sizeof(set)/sizeof(set[0]);
   if (isSubsetSum(set, n, sum) == true)
        printf("Found a subset with given sum");
   else
        printf("No subset with given sum");
   return 0;
}
```

討論：

Time Complexity：O(sum×n)　　*//為 NP 問題，且為 psudo polynomial*