$$\begin{array}{r} 393 \\ \times\ 252 \\ \hline \end{array}$$

3×2 出現4次

第一次就將3×2的答案記下來

Algorithms
Chapter 15
Dynamic Programming

動態規劃法 = 填表法 = 一種有技巧的暴力法

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Rod cutting**
- Matrix-chain multiplication
- Elements of dynamic programming 使用動態規劃法的要素
- Longest common subsequence
- Optimal binary search trees

台北到高雄的最短距是350公里

Dynamic Programming$_{1/2}$

路徑(一)：台北→台中→台南→高雄
路徑(二)：台北→花蓮→屏東→高雄

- Not a specific algorithm, but a technique, like divide-and-conquer. 像divide and conquer，非演算法，而是一種解決問題的技巧
- Dynamic programming is applicable when the subproblems are not independent. 適用於子問題重覆出現的時候
- A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table.
  对相同子問題，只解決一次，且將答案放入表格中
- "Programming" in this context refers to a tabular method, not to writing computer code. Programming指的是填表法
- Used for **optimization problems**: 用來解決最佳化問題
  - Find **a** solution with **the** optimal value.
  - Minimization or maximization. 可能是最大化(利益)或最小化(成本)

最佳解可能不止一個，但最佳解的值只有一個

Dynamic Programming$_{2/2}$

- **Four-step method**

1. Characterize the structure of an optimal solution.
   問題的最佳解也包含子問題的最佳解
2. Recursively define the value of an optimal solution.
   用子問題的答案定義最佳解
3. Compute the value of an optimal solution in a bottom-up fashion.
   先算子問題，再算原問題
4. Construct an optimal solution from computed information.
   用子問題的答案產生最佳解

Rod cutting$_{1/2}$

- How to cut steel rods into pieces in order to maximize the revenue you can get ?
  - Each cut is free.
  - Rod lengths are always an integral number of inches.長度為整數
- **The rod-cutting problem problem**
  - **Input:** A length n and table of prices p_i, for $i = 1, 2, \ldots, n$.
  - **Output:** The maximum revenue obtainable for rods whose lengths sum to n.

也可以都不切

- If p_n is large enough, an optimal solution might require no cuts.
- We can cut up a rod of length n in 2^{n-1} different ways.
  - can choose to cut or not cut after each of the first $n - 1$ inches.

n-1個切點:每一個都可以選擇切或不切

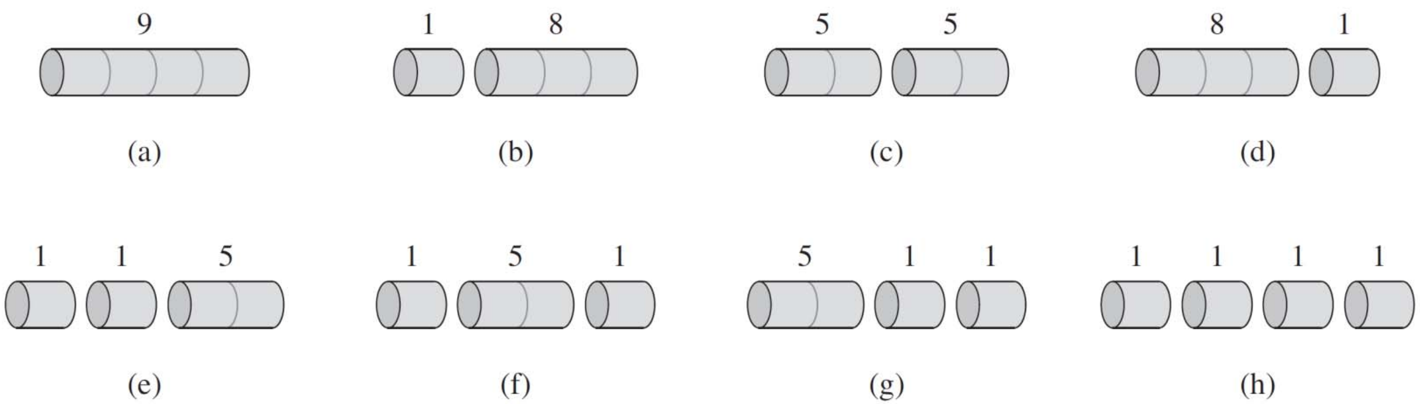
Rod cutting$_{2/2}$

- Consider the case when n = 4.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

  - Here are all 8 ways to cut a rod of length 4.
  - The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

最好的切法



假設台北到高雄最短距離經过台中

台北到高雄最短距 = 台北到台中最短距 + 台中到高雄最短距

Structure of an optimal solution

- Let r_i be the maximum revenue for a rod of length i.

r_i：長度為 i 的最高價錢

- **Step 1:** Characterize the structure of an optimal solution.
  - Suppose a cut is made at distance j inches in an optimal solution of size n. 假設一個最好的切法在長度為 j 的地方切了一刀
  - The optimal revenue $r_n = r_j + r_{n-j}$.
  - An optimal solution to a problem contains within it an optimal solution to subproblems. 問題的最佳解包含子問題最佳解
  - This is **optimal substructure**.

$r_n = r_j + r_{n-j}$

長度為 n 的最高價錢 = 長度為 j 的最高價錢 + 長度為 $n-j$ 的最高價錢

將問題的size變小：用子問題的答案定義最佳解

Recursive solution

- **Step 2:** Recursively define the value of an optimal solution.

最好的切法可能是

- Can determine optimal revenue r_n by taking the maximum of
  - p_n: the price we get by not making a cut, 不用切
  - $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of n − 1 inches, 切在距離 1
  - $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inch and a rod of n − 2 inches,… 切在距離 2
  - $r_{n-1} + r_1$.
- More generally, $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$.

所有可能性中取最大：共有n種

換一種看法

A simpler way to decompose the problem

- Every optimal solution has a leftmost cut. 每個最佳解都有最左的一刀
  - A first piece of length *i* cut off the left-hand end, and a remaining piece of length *n* − *i* on the right. 假設最左的那一刀切在距離 i
  - Need to divide only the remainder, not the first piece.
  - Leaves only one subproblem to solve, rather than two subproblems.
  - $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$.

左邊不用再切，只需切右邊

```
CUT-ROD(p, n)
    if n == 0
        return 0
    q = −∞
    for i = 1 to n
        q = max(q, p[i] + CUT-ROD(p, n−i))
    return q
```

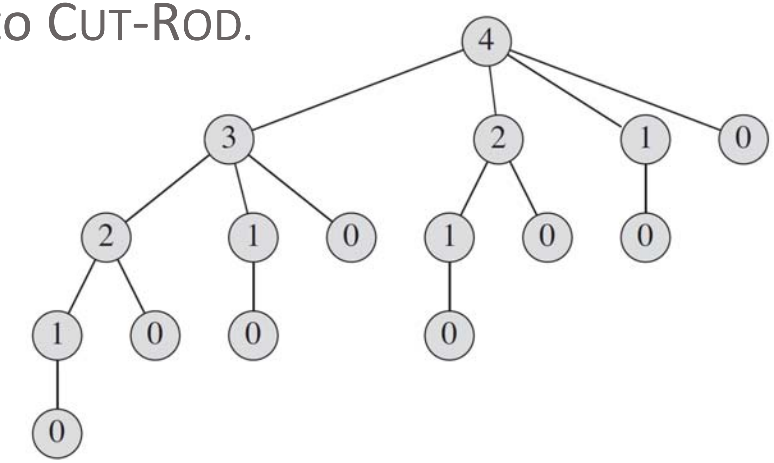
→左邊的長度由1~n

Recursive top-down implementation

由上往下遞迴

Running time of CUT-ROD(p, n)

- For n = 40, the program could take more than an hour.
  n=40時，电脑要跑至少1小時
- Each time you increase n by 1, the program's running time would approximately double. 每增加1，要多2倍
- Why is CUT-ROD so inefficient? 沒效率的原因：重覆算子問題
  - It solves the same subproblems repeatedly.
- Running time: n=4時，右边可能為 3，2，1，0
  - $T(n)$: total number of calls made to CUT-ROD.
  - $T(n) = \begin{cases} 1 & \text{if } n = 0, \\ \underbrace{1}_{\text{自己}} + \underbrace{\sum_{j=0}^{n-1} T(j)}_{\text{右边子問題}} & \text{if } n > 1. \end{cases}$
  - $T(n) = 2^n$. (exercise 15.1-1)



Dynamic programming 使用動態規劃法

- Using dynamic programming for optimal rod cutting
  - Instead of solving the same subproblems repeatedly, arrange to solve each sub-problem just once. 每個子問題只算一次
  - Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem. 遇到相同子問題就查表
  - "Store, don't recompute" ➔ time-memory trade-off. 計算與記憶体空間不可兼得
  - Can turn an exponential-time solution into a polynomial-time solution.
- Two basic approaches: **top-down with memoization**, and **bottom-up method**.

2種方法 { top-down with memoization
bottom-up method

Top-down with memoization

將答案存在表中

- Save the result of each subproblem in an array or hash table.
- The procedure first checks whether it has previously solved this subproblem. 計算前先檢查是否有算过
  - Yes : return the saved value.
  - No : compute the value in the usual manner.

有：回傳答案
沒有：進行計算

- **Memoizing** is remembering what we have computed previously.
  - Storing the solution of length *i* in array entry *r*[*i*].

```
MEMOIZED-CUT-ROD-AUX(p,n, r)
    if r[n] ≥ 0
        return r[n]          有：回傳答案
    if n == 0
        q = 0                沒有：進行計算
    else q = −∞
        for i = 1 to n
            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p,n − i, r))
    r[n] = q
    return q
```

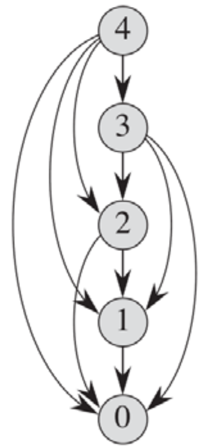
```
MEMOIZED-CUT-ROD(p,n)
    let r[0..n] be a new array
    for i = 0 to n
        r[n] = −∞
    return MEMOIZED-CUT-ROD-AUX(p,n, r)
```

Bottom-up method

- **Step 3:** Compute the value of an optimal solution in a bottom-up fashion. 先算子問題，再算原問題
- The procedure solves subproblems of sizes j = 0, 1,..., n, in that order. 先算 size = 1，再算 size = 2，…
- When solving a subproblem, have already solved the smaller subproblems we need. 先解決子問題：不需遞迴

```
BOTTOM-UP-CUT-ROD(p, n)
    let r[0..n] be a new array
    r[0] = 0
    for j = 1 to n
        q = −∞
        for i = 1 to j
            q = max(q, p[i] + r[j − i])
        r[j] = q
    return r[n]
```

j = size
i = 左邊的長度
j 種可能性



The subproblem graph.

Running time

- Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.

size = 1 ⇒ 1 種可能性 size = 2 ⇒ 2 種可能性

- **Bottom-up**
  - Doubly nested loops 兩層 for loop $= \underbrace{1}_{size=0} + 1 + 2 + 3 + \dots + n = \Theta(n^2)$
  - Number of iterations of inner for loop forms an arithmetic series.

    等差級數
- **Top-down**
  - MEMOIZED-CUT-ROD solves each subproblem just once.
  - It solves subproblems for sizes 0, 1,…,n.
  - To solve a subproblem of size n, the for loop iterates n times.
  - Total number of iterations also forms an arithmetic series.

Memoized: ① 每一個子問題都只有算一次

② Size 為 n 的子問題有 n 種可能性

③ $\underbrace{1}_{size=0} + 1 + 2 + 3 + \dots + n = \Theta(n^2)$

Reconstructing a solution$_{1/2}$

- **Step 4:** Construct an optimal solution from computed information. 用子問題的答案產生最佳解
- Saves the first cut made in an optimal solution for a problem of size *i* in *s*[*i*]. 將size為i的最左那一刀記在S[i]

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
    let r[0..n]  and s[0..n] be new arrays
    r[0] = 0
    for j = 1 to n
        q = −∞
        for i = 1 to j        j種可能性
            if q < p[i] + r[j − i]        如果價錢更高
                q = p[i] + r[j − i]           ①更新價錢
                s[j] = i                      ②記住最左那一刀
        r[j] = q        記住最好價錢
    return r and s
```

Reconstructing a solution$_{2/2}$

- To print out the cuts made in an optimal solution.

```
PRINT-CUT-ROD-SOLUTION (p, n)
    (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
    while n > 0
        print s[n]  →最左那一刀
        n = n − s[n]→右边子問題
```

- The call EXTENDED-BOTTOM-UP-CUT-ROD(p, n) return

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- A call to PRINT-CUT-ROD-SOLUTION(p, 10) would print just 10.
- A call with $n = 7$ would print the cuts 1 and 6.

Outline

- Rod cutting
- **Matrix-chain multiplication**
- Elements of dynamic programming
- Longest common subsequence
- Optimal binary search trees

$[A]_{p\times q} \times [B]_{q\times r} = [C]_{p\times r}$

Matrix-chain multiplication

- When we multiply two matrices A and B, if A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix.
  - The number of scalar multiplications is ***pqr***.

要算出C要算出pqr次，大小為pr

$[A_i]_{p_{i-1} \times p_i}$

- **Matrix-chain multiplication problem**

一連串的矩陣

$$\begin{array}{cc} & p_i \\ p_{i-1} & \boxed{A_i} \end{array}$$

  - **Input:** A chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices.
    (matrix A_i has dimension $p_{i-1} \times p_i$)
  - **Output:** A fully parenthesized product $A_1, A_2, \ldots, A_n$ that minimizes the number of scalar multiplications. 括號⇒乘法順序

將 $A_1 \ldots A_n$ 完全括弧，使得用到的純量乘法次數最少

- For example: The dimensions of the matrices A_1, A_2, and A_3 are 10×100, 100×5, and 5×50, respectively.
  - $((A_1A_2)A_3) = 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500.$
  - $(A_1(A_2A_3)) = 100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000.$

$[A_2, A_3]_{100\times 50}$ $\qquad$ $[A_1]_{10\times 100} \cdot [A_2A_3]_{100\times 50} = [A_1A_2A_3]_{10\times 50}$

Counting the number of parenthesizations

- **Brute-force algorithm:** 暴力法：將所有可能性都試過
  - Checking all possible parenthesizations
- Time: $\Omega(2^n)$. (Exercise 15.2-3)

  $p(k)\ \boxed{A_1 \ldots A_k}\ \boxed{A_{k+1} \ldots A_n}\ p(n-k)$

  $\Rightarrow 1 \le k \le n-1$
  - Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$.
  - A fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts.
  - The split between the two subproducts may occur between the kth and $(k + 1)$st matrices.

    $p(n) = n$ 個矩陣完成括弧的可能數

    (純量乘法可能數)
  - Thus, we have $P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$

    前 k 個的可能數　　後 n-k 個的可能數

Step 1: The structure of an optimal solution

- An optimal solution to an instance contains optimal solutions to subproblem instances. 問題的最佳解包含子問題的最佳解
- For example:
  - If $((A_1A_2)A_3)(A_4(A_5A_6))$ is an optimal solution to $A_1, A_2, \ldots, A_6$.
  - Then, $((A_1A_2)A_3)$ is an optimal solution to A_1, A_2, A_3 and $(A_4(A_5A_6))$ is an optimal solution to A_4, A_5, A_6.

若最佳解為 $((A_1A_2)A_3)(A_4(A_5A_6))$

$\Rightarrow$ $((A_1A_2)A_3)$ 為 $A_1 \sim A_3$ 的最佳解

$(A_4(A_5A_6))$ 為 $A_4 \sim A_6$ 的最佳解

$$p_{i-1}\boxed{A_i}^{\,p_i}\ \boxed{A_{i+1}}^{\,p_{i+1}}\cdots\boxed{A_j}^{\,p_j} = p_{i-1}\boxed{A}^{\,p_j}$$

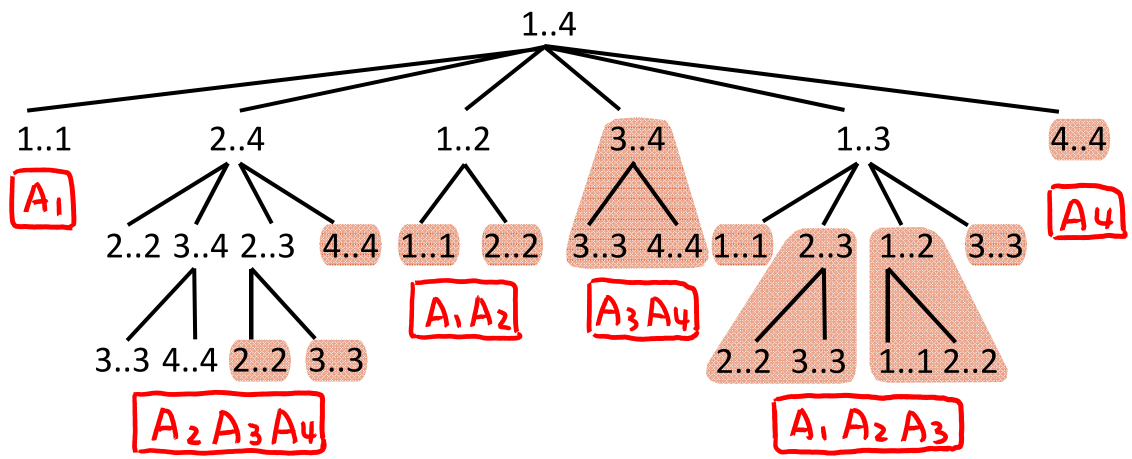
Step 2: A recursive solution

- Define $m[i, j]$ = the minimum number of scalar multiplications needed to compute $A_iA_{i+1}\ldots A_j$. m[i,j]= $A_i \sim A_j$ 最少的純量相乘數

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

左邊最小乘法數+右邊最小乘法數+相乘的乘法數

- The recursion tree for the computation of $m[1,4]$.



*程式的流程是DFS(深度優先): 1..1→2..4→2..2→3..4→3..3→4..4→2..3→2..2→3..3→4..4

Step 3: Computing the optimal costs

- Based on the recursive formula, we could easily write an exponential-time recursive algorithm to compute the compute the minimum cost $m[1, n]$ for multiplying $A_1A_2 \ldots A_n$.

i,j不同 i,j相同

- There are only $\binom{n}{2} + n = \Theta(n^2)$ distinct subproblems, one problem for each choice of i and j satisfying $1 \leq i \leq j \leq n$.

- We can use dynamic programming to compute the solutions bottom up. 使用動態規劃法,由下往上,先算子問題,再算原問題

暴力法花指數時間
但子問題的個數只有 n^2 個

$\boxed{A_1}\boxed{A_2A_3} = 0 + 2625 + 30\times 35\times 5 = 7875,\ k=1$

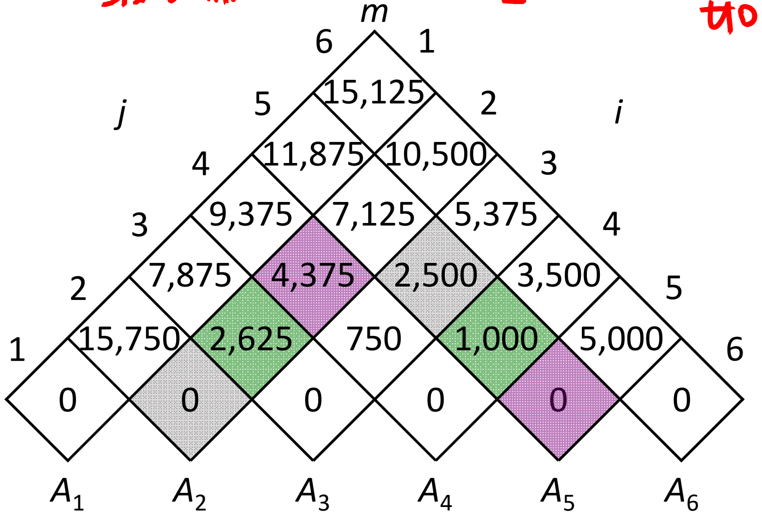
$\boxed{A_1A_2}\boxed{A_3} = 15750 + 0 + 30\times 15\times 5 = 18000,\ k=2$

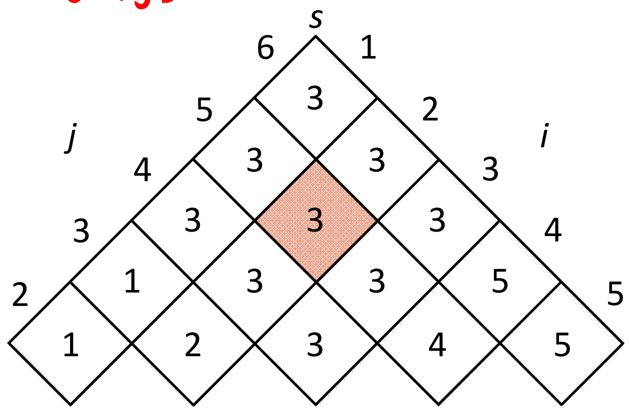
Dependencies between the subproblems

最少需要幾個純量乘法

如何算 m[i,j] 最好的 k (使純量乘法最少)

由下往上填表





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

$P_0 = 30,\ P_1 = 35,\ P_2 = 15,\ P_3 = 5,\ P_4 = 10,\ P_5 = 20,\ P_6 = 25$

- $s[i, j]$: index k achieved the optimal cost in computing $m[i, j]$.

$$m[2,5] = \min\begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35\cdot 15\cdot 20 = 13000, & \boxed{A_2}\ \boxed{A_3A_4A_5} \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35\cdot 5\cdot 20 = 7125, & \boxed{A_2A_3}\ \boxed{A_4A_5} \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35\cdot 10\cdot 20 = 11375. & \boxed{A_2A_3A_4}\ \boxed{A_5} \end{cases}$$

*先算長度=1，再算長度=2 … 再算長度=n

MATRIX-CHAIN-ORDER pseudocode

```
MATRIX-CHAIN-ORDER(p)
    n ← length[p] − 1
    for i ← 1 to n
        m[i, i] ← 0
    for ℓ ← 2 to n      /* ℓ is the chain length*/
        for i ← 1 to n − ℓ + 1
            j ← i + ℓ − 1
            m[i, j] ← ∞
            for k ← i to j − 1
                q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                if q < m[i, j]
                    m[i, j] ← q
                    s[i, j] ← k
    return m and s
```

長度為1

長度為ℓ的起頭和結尾

i：起頭

j：結尾

長度為2~n

算 m[i,j]

k有j-i種

- The loops are nested three deep, and each loop index (ℓ, i, and k) takes on at most $n - 1$ values.
- Time: $O(n^3)$.

表的大小 $O(n^2)$，算每一個要 $O(n)$，
因為每一格有k種且 $k < n$
$\Rightarrow O(n^2) \cdot O(n) = O(n^3)$

用已知資訊建構最佳解

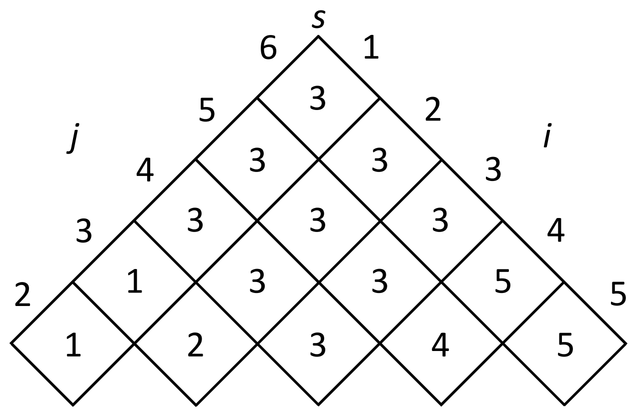
Step 4: Constructing an optimal solution

- Each entry $s[i, j]$ records the value of k such that the optimal parenthesization of $A_iA_{i+1}\cdots A_j$ splits the product between A_k and A_{k+1}.

(左 右)

```
PRINT-OPTIMAL-PARENS(s, i, j)
    if i = j
        print "A_i"
    else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
        PRINT-OPTIMAL-PARENS(s, s[i, j]+1, j)
        print ")"
```

若i=j時,直接印



- The call PRINT-OPTIMAL-PARENS(s, 1, n) prints the parenthesization $((A_1(A_2A_3))\,((A_4A_5)A_6))$.

(左 右)

Outline

- Rod cutting
- Matrix-chain multiplication
- **Elements of dynamic programming**
- Longest common subsequence
- Optimal binary search trees

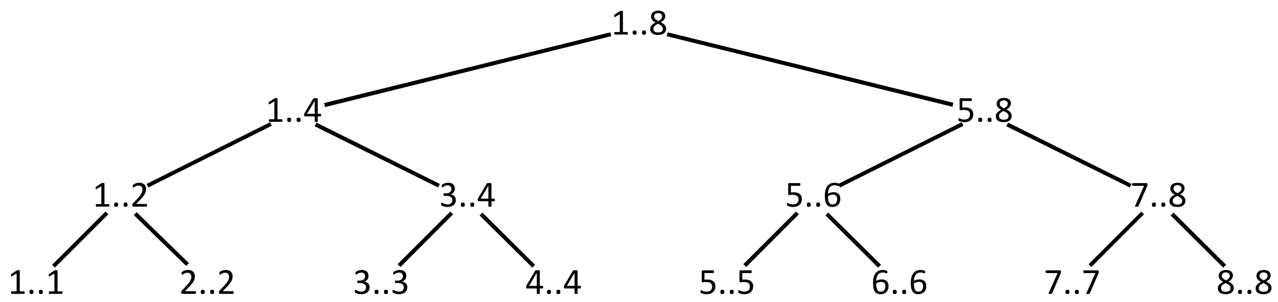
Elements of dynamic programming$_{1/2}$

- **Optimal substructure** 問題的最佳解也包含子問題的最佳解
  - An optimal solution to a problem contains an optimal solution to subproblems.
    - If $((A_1A_2)A_3)(A_4(A_5A_6))$ is an optimal solution to $A_1, A_2, \ldots, A_6$, then $((A_1A_2)A_3)$ is an optimal solution to A_1, A_2, A_3 and $(A_4(A_5A_6))$ is an optimal solution to A_4, A_5, A_6.
- **Overlapping subproblems** 重覆子問題
  - A recursive algorithm revisits the same problem over and over again. 遞迴演算法一再遇到相同問題
  - Typically, the total number of distinct subproblems is a polynomial in the input size. 但子問題的個數只有n的多項式個
  - In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion. matrix-chain 的子問題会重覆

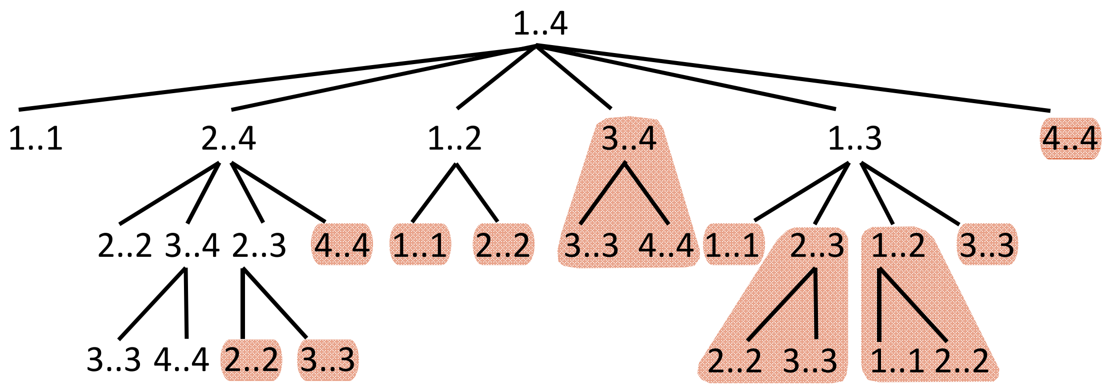
divide-and-conquer 会一直遇到新的子問題

Elements of dynamic programming$_{2/2}$

- Example: merge sort 問題不一樣



- Example: matrix-chain 重覆子問題



Outline

- Rod cutting
- Matrix-chain multiplication
- Elements of dynamic programming
- **Longest common subsequence**
- Optimal binary search trees

Longest-common-subsequence 最長相同子序列

- A **subsequence** is a sequence that can be derived from another sequence by deleting some elements.

子序列：將原本 sequence 的某些元素刪除

- For example:
  - $\langle K, C, B, A \rangle$ is a subsequence of $\langle \boldsymbol{K}, G, \boldsymbol{C}, E, \boldsymbol{B}, B, \boldsymbol{A} \rangle$.
  - $\langle B, C, D, G \rangle$ is a subsequence of $\langle A, C, \boldsymbol{B}, E, G, \boldsymbol{C}, E, \boldsymbol{D}, B, \boldsymbol{G} \rangle$.
- **Longest-common-subsequence problem**
  - **Input:** 2 sequences, $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.
  - **Output:** A maximum-length common subsequence of X and Y.
- For example: $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$.
  - $\langle B, C, A \rangle$ is a common subsequence of both X and Y.
  - $\langle B, C, B, A \rangle$ is an longest common subsequence (**LCS**) of X and Y.

長度為4 最長相同子序列

Step 1: Characterizing an LCS

- **Brute-force algorithm:** 暴力法將所有可能性都看過
  - For every subsequence of X, check whether it is a subsequence of Y.
- Time: $\Theta(n2^m)$.
  - 2^m subsequences of X to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.

x的子序列個數 2^m
檢查是不是y的子序列：$O(n)$
∴ 共花 $\Theta(n \cdot 2^m)$

- Given a sequence $X = \langle x_1, x_2,..., x_m \rangle$, we define the ith **prefix** of X, as $X = \langle x_1, x_2,..., x_i \rangle$.

x的第i個前置

- For example:
  - $X = \langle A, B, C, B, D, A, B \rangle$.
  - $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

Optimal substructure of an LCS

若 $x_m \neq y_n$，z 有可能 $\begin{cases} x_{m-1} \text{ 和 } y_n \text{ 的 LCS} \\ x_m \text{ 和 } y_{n-1} \text{ 的 LCS} \end{cases}$

- **Theorem 15.1** 如果 $x_m = y_n$，可以 x_{m-1} 和 y_{n-1} 的 LCS 來形成 Z_{k-1}

  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}.

- For example:
  - $X = \langle A, B, C, B, D, A, B \rangle$, $Y = \langle B, D, C, A, B \rangle$ and $Z = \langle B, C, A, B \rangle$ is an LCS of X and Y. If $x_7 = y_5$, then $z_4 = x_7 = y_5$ and $Z_3 = \langle B, C, A \rangle$ is an LCS of X_6 and Y_4.
  - $X = \langle A, B, C, B, D, A, D \rangle$, $Y = \langle B, D, C, B, A \rangle$ and $Z = \langle B, C, A \rangle$ is an LCS of X and Y. If $x_7 \neq y_5$, then $z_3 \neq x_7$ implies that $Z_3 = \langle B, C, A \rangle$ is an LCS of X_6 and Y_5.
  - $X = \langle A, B, C, B, D, A, A \rangle$, $Y = \langle B, D, C, A, B \rangle$ and $Z = \langle B, C, A \rangle$ is an LCS of X and Y. If $x_7 \neq y_5$, then $z_3 \neq y_5$ implies that $Z_3 = \langle B, C, A \rangle$ is an LCS of X_7 and Y_4.

在 $x_m \neq y_n$ 的情形下，$z_k \neq x_m$ 與 $z_k \neq y_n$ 至少有一個成立，
⇒ 因為兩個都是 common sequence，所以選較長者

Step 2: A recursive solution

- Define $c[i, j]$ = length of LCS of X_i and Y_j. We want $c[m, n]$.

x_i 和 y_j 的 LCS 長度

由定理而來

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \quad \text{case ①} \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \quad \text{case ② ③} \end{cases}$$

選較長者

Step 3: Computing the length of an LCS

- Based on the recursive formula, we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences.

  如果將遞迴公式直接寫成程式,則程式的時間複雜度將會是指數時間

- There are only $\Theta(mn)$ distinct subproblems.

  c[i,j],其中 $i=1\sim m$, $j=1\sim n$, ∴子問題個數 $\Theta(mn)$ 個

- We can use dynamic programming to compute the solutions bottom up. 使用動態規劃法,由下往上,先算子問題,再算原問題

LCS-LENGTH pseudocode

```
LCS-LENGTH(X, Y)
    m ← length[X]; n ← length[Y]
    for i ← 1 to m
        c[i, 0] ← 0
    for j ← 0 to n
        c[0, j] ← 0
    for i ← 1 to m
        for j ← 1 to n
            if x_i = y_j
                c[i, j] ← c[i – 1, j – 1] + 1
                b[i, j] ← "↖"
            else if c[i - 1, j] ≥ c[i, j - 1]
                c[i, j] ← c[i – 1, j]
                b[i, j] ← "↑"
            else  c[i, j] ← c[i, j – 1]
                b[i, j] ← "←"
    return c and b
```

初始化

先算 $i=1$

$i=2$

⋮

$i=m$

j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←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

x_i 與 y_j 相比

不同：上和左中取大的 ⇒ case ② ③

相同：左上的長度+1 ⇒ case ①

↑：記錄大的是從何處所取

- Time: $O(mn)$.

Step 4: Constructing an LCS

- Whenever we encounter a "↖" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS. 將回溯路徑印出

```
PRINT-LCS(b, X, i, j)
    if i = 0 or j = 0
        return
    if b[i, j] = "↖"
        PRINT-LCS(b, X, i – 1, j – 1) 往左上
        print x_i  直接印
    elseif b[i, j] = "↑"
        PRINT-LCS(b, X, i – 1, j) 往上
    else PRINT-LCS(b, X, i, j – 1) 往左
```

i \ j		0	1	2	3	4	5	6
	y_j		*B*	*D*	*C*	*A*	*B*	*A*
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑0	↑0	↑0	↖1	←1	↖1
2	*B*	0	↖1	←1	←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

- This procedure prints "*BCBA*".

Outline

- Rod cutting
- Matrix-chain multiplication
- Elements of dynamic programming
- Longest common subsequence
- **Optimal binary search trees**

Optimal binary search trees

- **Input:** A sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order.
  A sequence $D = \langle d_0, d_1, \ldots, d_n \rangle$ of $n + 1$ dummy keys.
  - $k_1 < k_2 < \cdots < k_n$.
  - d_0 = all values < k_1. d_n = all values > k_n.
  - d_i = all values between k_i and k_{i+1}.
  - For each key k_i, a probability p_i that a search is for k_i.
  - For each key d_i, a probability q_i that a search is for d_i.
- **Output:** A BST with minimum expected search cost.
  - $$\begin{aligned} \text{E[search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=1}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} q_i + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} \text{depth}_T(d_i) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} \text{depth}_T(d_i) \cdot q_i \end{aligned}$$

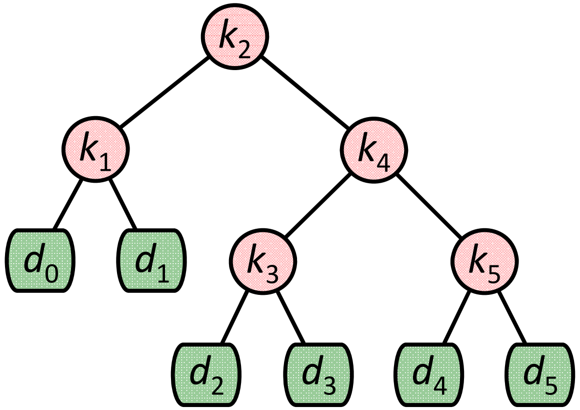
$$\boxed{\sum_{i=1}^{n} p_i + \sum_{i=1}^{n} q_i = 1}$$

$0.15 \times 2 + 0.1 \times 1 + 0.05 \times 3 + 0.10 \times 2 + 0.20 \times 3$
$+ 0.05 \times 3 + 0.10 \times 3 + 0.05 \times 4 + 0.05 \times 4 + 0.05 \times 4 + 0.10 \times 4 = 2.80$

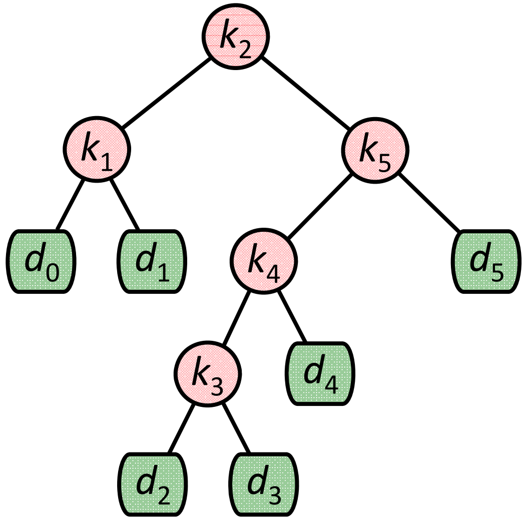
An example

k_i : 成功

d_i : 失敗



Expected search cost 2.80.



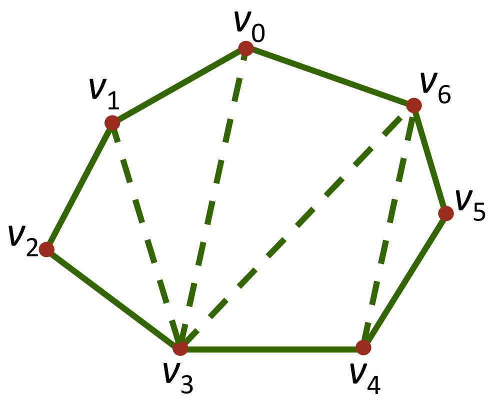
Expected search cost 2.75.
This tree is optimal.

- Two binary search trees for a set of n = 5 keys with the following probabilities:

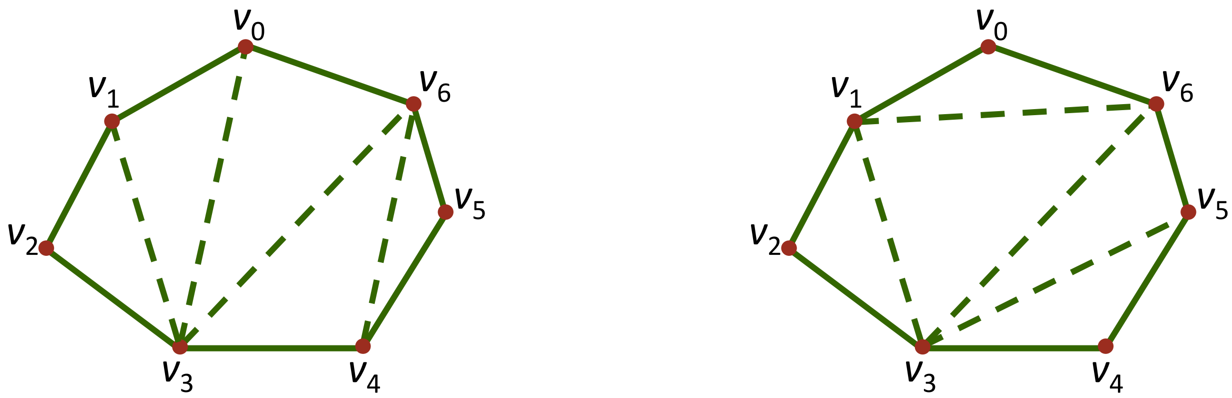
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Optimal polygon triangulation$_{1/2}$ 將多邊形三角化

- If $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ is a **convex polygon**, it has n **sides**, $\overline{v_0 v_1}$, $\overline{v_1 v_2}$, …, $\overline{v_{n-1} v_0}$.
- Given two nonadjacent vertices v_i and v_j, the segment $v_i v_j$ is a **chord** of the polygon.
- A **triangulation** of a polygon is a set T of chords of the polygon that divide the polygon into disjoint triangles.



Optimal polygon triangulation$_{2/2}$



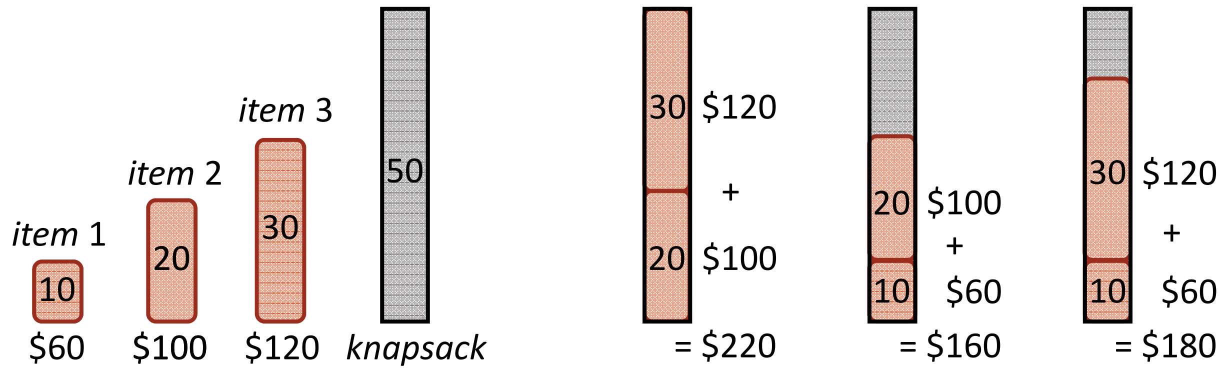
Two ways of triangulating a convex polygon.

- **Optimal polygon triangulation problem**
  - **Input:** A convex polygon $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$.
    - A weighting function w defined on triangles formed by sides and chords of P. 三角化後週的和為最小
  - **Output:** A triangulation that minimizes the sum of the weights of the triangles in the triangulation.

动態規劃法=填表法

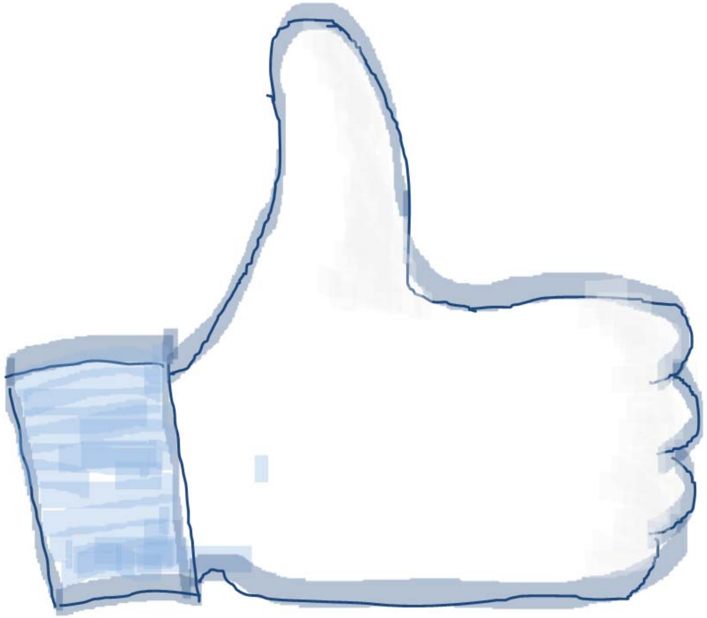
0-1 knapsack problem-- using DP 背包問題

- **Input:** A set $A = \{a_1, a_2, \ldots, a_n\}$ of n items and a knapsack of capacity C. 背包容量
  - Each item a_i is worth v_i dollars and weighs w_i pounds.
    各物品有其重量及價值
- **Output:** A subset of items whose total size is bounded by C and whose profit is maximized. 如何取有最大價值
  - **Each item must either be taken or left behind.**
    每個item只能選擇取或不取
- For example:



講義超精美

我沒蓋你



2014.01.06.

魏琳瑑.


对阿 对阿 对阿 对阿 对阿 对阿 对阿 对阿 对阿 对阿
清池 Good Good 講義
对阿 对阿 对阿
对阿 对阿 对阿
对阿 对阿 对阿
对阿 对阿 对阿
对阿 对阿 对阿
对阿 对阿 对阿
对阿
对阿
精美
对阿 对阿
对阿 对阿
对阿 对阿 对阿
对阿 对阿 对阿
大家要認真看講義喔!
2014. 12. 17