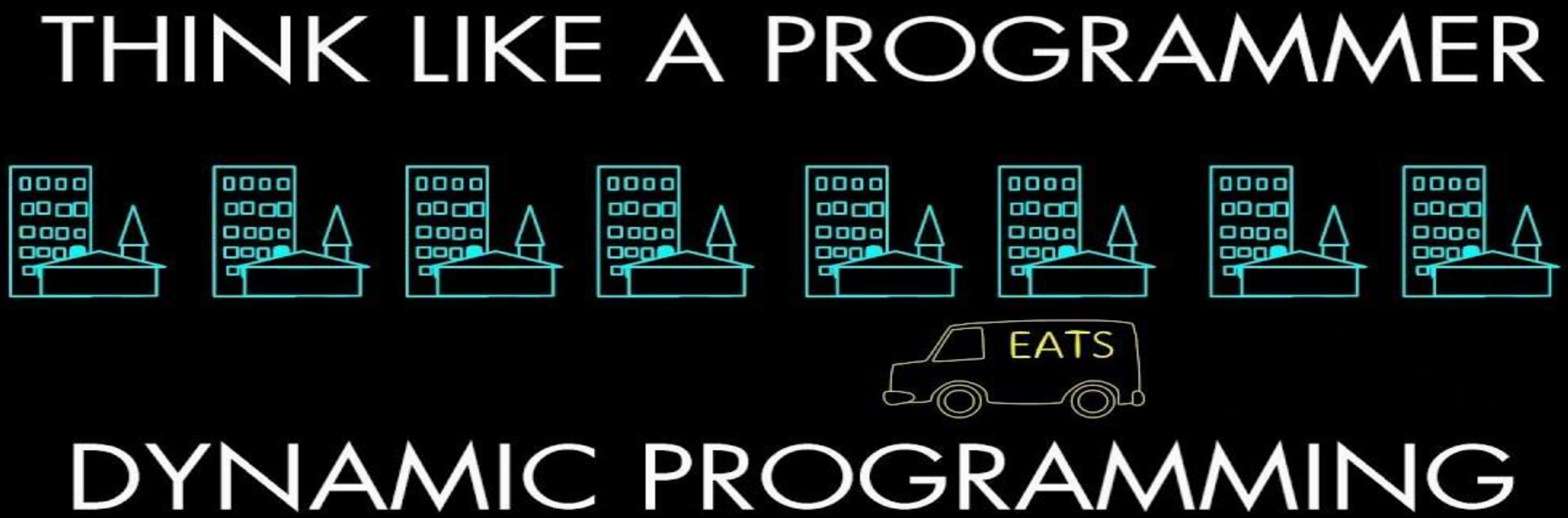


Algorithm Design and Analysis
Dynamic Programming (2)

http://ada.miulab.tw   slido: #ADA2020

國立臺灣大學 National Taiwan University

Yun-Nung (Vivian) Chen

Announcement

- Homework 1 ADA Party
  - Due on 10/29 (Thur) 14:20
- Mini-HW 4 released
  - Due on 10/29 (Thur) 14:20
- Homework 2 released
  - Due on 11/10 (Tue) 14:20 (2 days before midterm)

Outline

- Dynamic Programming
- DP #1: Rod Cutting
- DP #2: Stamp Problem
- DP #3: Sequence Alignment Problem
  - Longest Common Subsequence (LCS) / Edit Distance
  - Viterbi Algorithm
  - Space Efficient Algorithm
- DP #4: Matrix-Chain Multiplication
- DP #5: Weighted Interval Scheduling
- DP #6: Knapsack Problem
  - 0/1 Knapsack
  - Unbounded Knapsack
  - Multidimensional Knapsack
  - Fractional Knapsack

動腦一下 – 囚犯問題

- 有100個死囚，隔天執行死刑，典獄長開恩給他們一個存活的機會。
- 當隔天執行死刑時，每人頭上戴一頂帽子(黑或白)排成一隊伍，在死刑執行前，由隊伍中最後的囚犯開始，每個人可以猜測自己頭上的帽子顏色(只允許說黑或白)，猜對則免除死刑，猜錯則執行死刑。
- 若這些囚犯可以前一天晚上先聚集討論方案，是否有好的方法可以使總共存活的囚犯數量期望值最高？

猜測規則

- 囚犯排成一排，每個人可以看到前面所有人的帽子，但看不到自己及後面囚犯的。
- 由最後一個囚犯開始猜測，依序往前。
- 每個囚犯皆可聽到之前所有囚犯的猜測內容。

Example: 奇數者猜測內容為前面一位的帽子顏色 → 存活期望值為75人

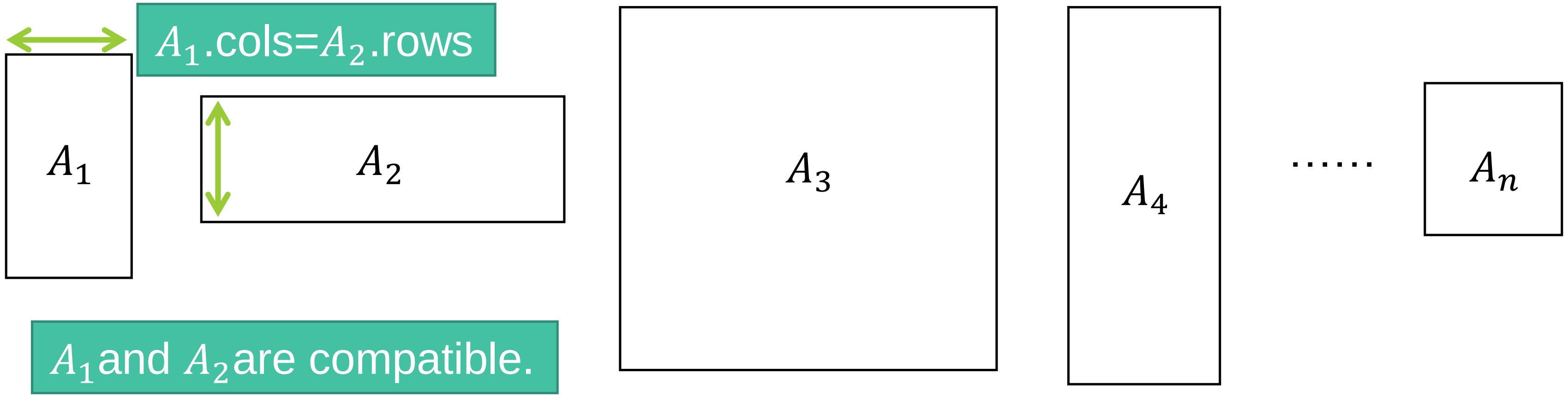
有沒有更多人可以存活的好策略?

DP#4: Matrix-Chain Multiplication

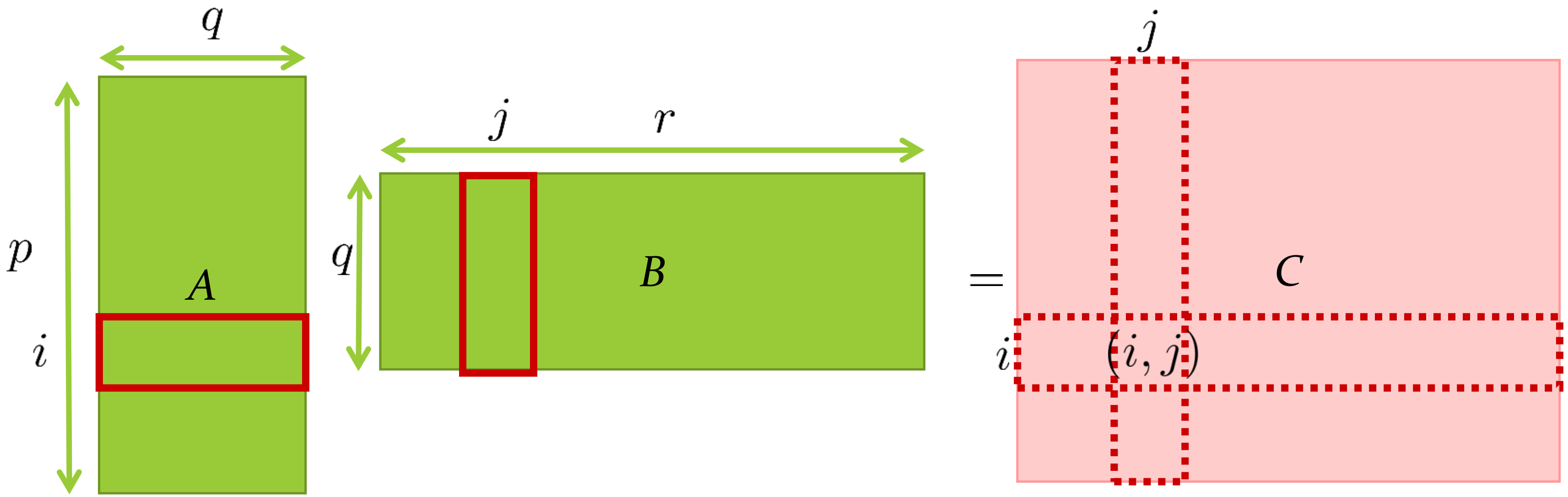
Textbook Chapter 15.2 – Matrix-chain multiplication

Matrix-Chain Multiplication

- Input: a sequence of *n* matrices $\langle A_1, \ldots, A_n \rangle$
- Output: the product of $A_1 A_2 \ldots A_n$



Observation



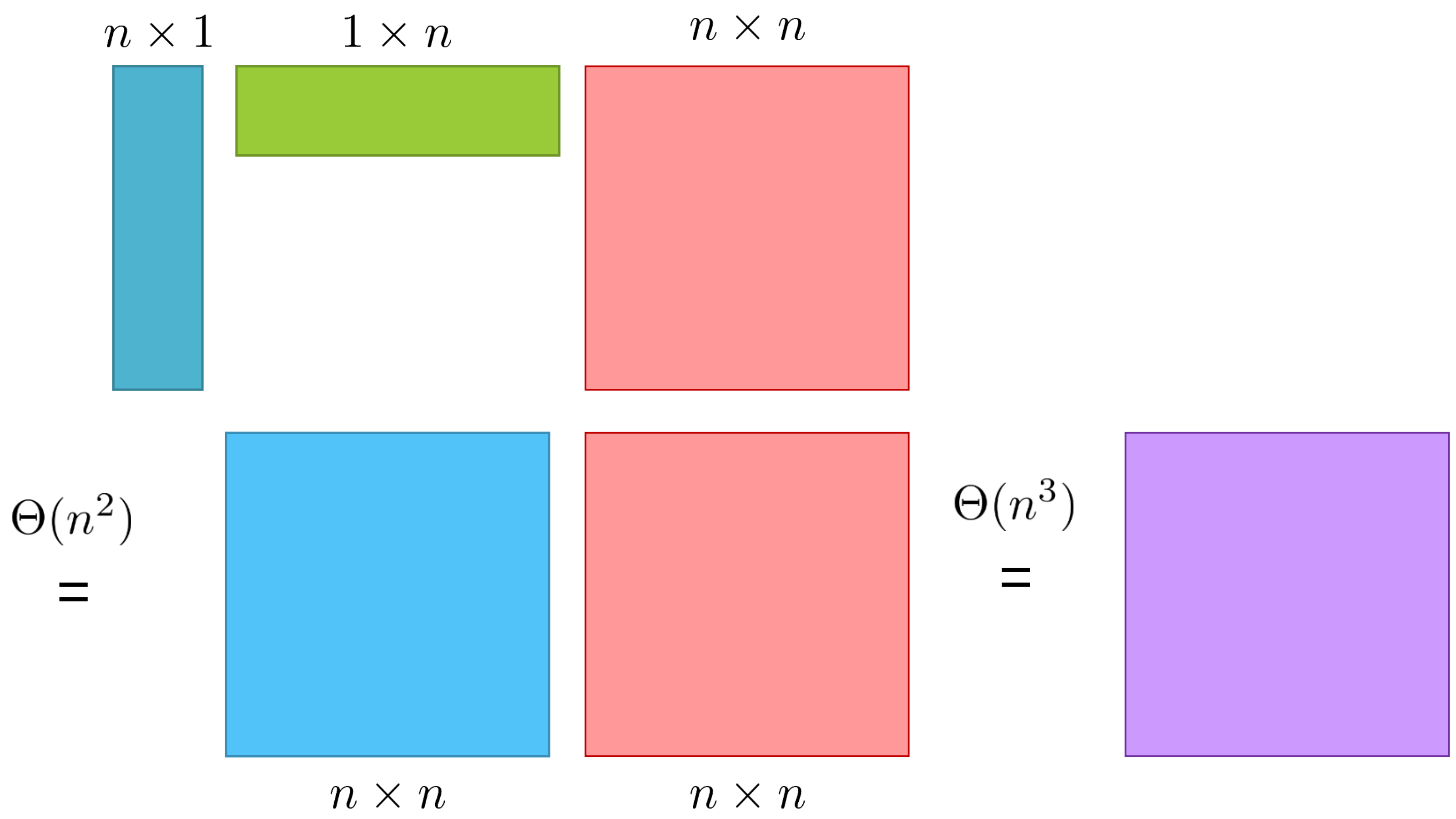
$$C(i,j) = \sum_{k=1}^{n} A(i,q) \cdot B(k,j)$$

- Each entry takes q multiplications
- There are total pr entries

➡ $\Theta(q)\Theta(pr) = \Theta(pqr)$

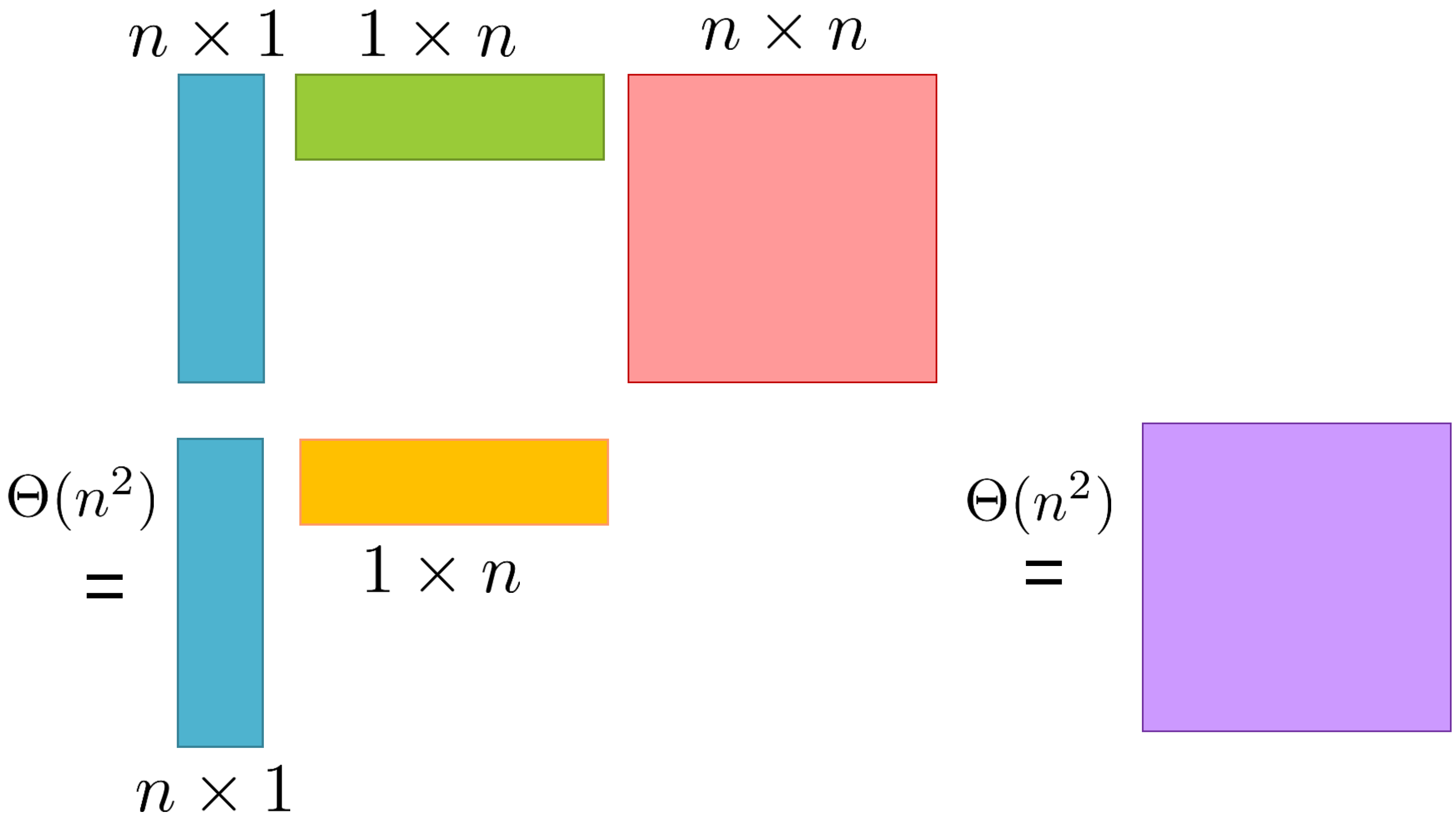
Matrix multiplication is associative: $A(BC) = (AB)C$. The time required by obtaining $A \times B \times C$ could be affected by which two matrices multiply first .

Example



- Overall time is $\Theta(n^2) + \Theta(n^3) = \Theta(n^3)$

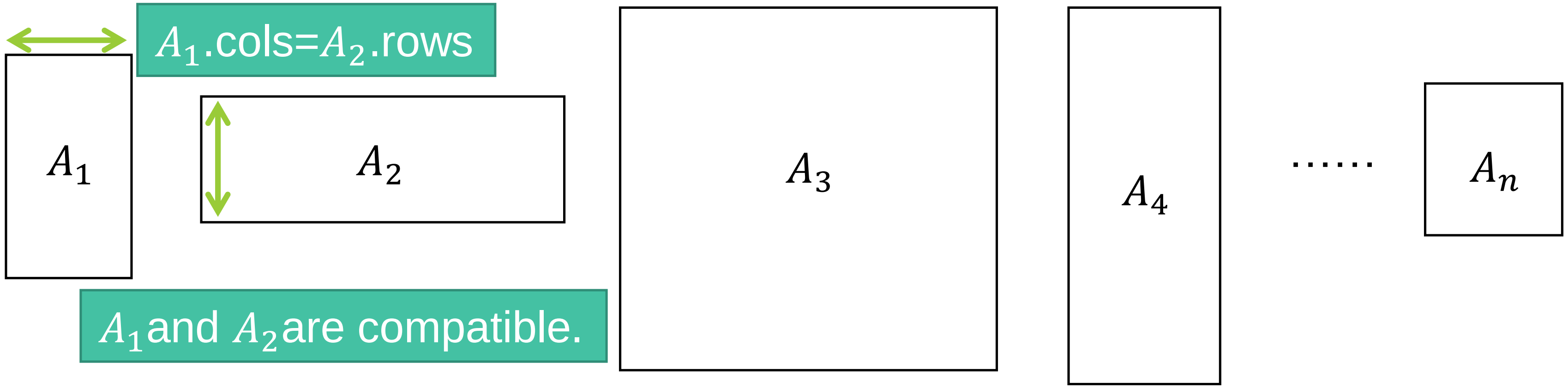
Example



- Overall time is $\Theta(n^2) + \Theta(n^2) = \Theta(n^2)$

Matrix-Chain Multiplication Problem

- Input: a sequence of integers $l_0, l_1, \ldots, l_n$
  - l_{i-1} is the number of rows of matrix A_i
  - l_i is the number of columns of matrix A_i
- Output: an order of performing $n-1$ matrix multiplications in the minimum number of operations to obtain the product of $A_1 A_2 \ldots A_n$



Do not need to compute the result but find the fast way to get the result!
(computing "how to fast compute" takes less time than "computing via a bad way")

Brute-Force Naïve Algorithm

- P_n: how many ways for n matrices to be multiplied

$$P_n = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P_k P_{n-k} & \text{if } n \geq 2 \end{cases}$$

$$(A_1 A_2 \cdots A_k) \qquad (A_{k+1} A_{k+2} \cdots A_n)$$

- The solution of P_n is Catalan numbers, $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$, or is also $\Omega(2^n)$ Exercise 15.2-3

Step 1: Characterize an OPT Solution

Matrix-Chain Multiplication Problem
Input: a sequence of integers $l_0, l_1, \ldots, l_n$ indicating the dimensionality of A_i
Output: an order of matrix multiplications with the minimum number of operations

- Subproblems
  - `M(i, j)`: the min #operations for obtaining the product of $A_i \ldots A_j$
  - Goal: `M(1, n)`
- Optimal substructure: suppose we know the OPT to `M(i, j)`, there are k cases:

$$i \le k < j$$

$A_i A_{i+1} \ldots A_k$	$A_{k+1} A_{k+2} \ldots A_j$

  - Case k: there is a cut right after A_k in OPT

    左右所花的運算量是`M(i, k)`及`M(k+1, j)`的最佳解

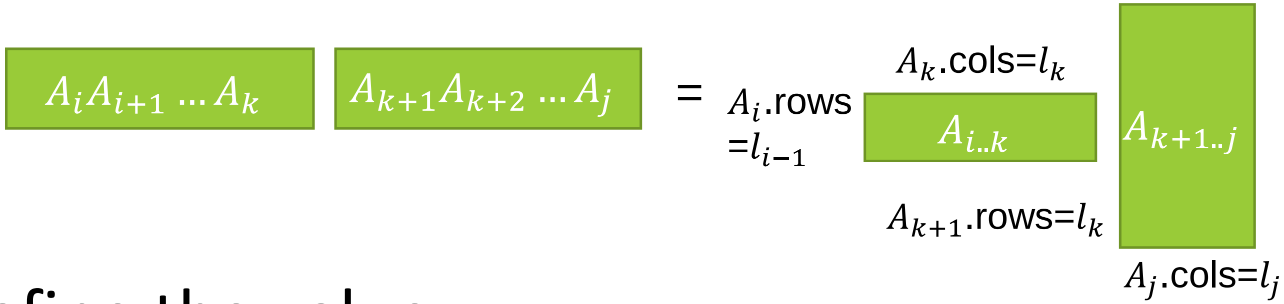
Step 2: Recursively Define the Value of an OPT Solution

Matrix-Chain Multiplication Problem
Input: a sequence of integers $l_0, l_1, \dots, l_n$ indicating the dimensionality of A_i
Output: an order of matrix multiplications with the minimum number of operations

- Suppose we know the optimal solution to `M(i, j)`, there are k cases:
  - Case k: there is a cut right after A_k in OPT
    左右所花的運算量是`M(i, k)`及`M(k+1, j)`的最佳解

$$M_{i,j} = M_{i,k} + M_{k+1,j} + l_{i-1} l_k l_j$$

$$A_{i \dots k} A_{k+1 \dots j}$$



- Recursively define the value

$$M_{i,j} = \begin{cases} 0 & i \geq j \\ \min_{i \leq k < j} \left(M_{i,k} + M_{k+1,j} + l_{i-1} l_k l_j \right) & i < j \end{cases}$$

Step 3: Compute Value of an OPT Solution

Matrix-Chain Multiplication Problem
Input: a sequence of integers $l_0, l_1, \ldots, l_n$ indicating the dimensionality of A_i
Output: an order of matrix multiplications with the minimum number of operations

- Bottom-up method: solve smaller subproblems first

$$M_{i,j} = \begin{cases} 0 & i \geq j \\ \min_{i \leq k < j} \left(M_{i,k} + M_{k+1,j} + l_{i-1} l_k l_j \right) & i < j \end{cases}$$

- How many subproblems to solve
  - #combination of the values i and j s.t. $1 \leq i \leq j \leq n$

$$T(n) = \underbrace{C_2^n}_{i \neq j} + \underbrace{n}_{i = j} = \Theta(n^2)$$

Step 3: Compute Value of an OPT Solution

```
Matrix-Chain(n, l)
  initialize two tables M[1..n][1..n] and B[1..n-1][2..n]
  for i = 1 to n
    M[i][i] = 0 // boundary case
  for p = 2 to n // p is the chain length
    for i = 1 to n - p + 1 // all i, j combinations
      j = i + p - 1
      M[i][j] = ∞
      for k = i to j - 1 // find the best k
        q = M[i][k] + M[k + 1][j] + l[i - 1] * l[k] * l[j]
        if q < M[i][j]
          M[i][j] = q
return M
```

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

Dynamic Programming Illustration

How to decide the order of the matrix multiplication?

j

i

$M_{i,j}$	1	2	3	4	5	6	…	n
1	0							
2		0						
3			0					
4				0				
5					0			
6						0		
⋮							0	
n								0

Step 4: Construct an OPT Solution by Backtracking

```
Matrix-Chain(n, l)
  initialize two tables M[1..n][1..n] and B[1..n-1][2..n]
  for i = 1 to n
    M[i][i] = 0 // boundary case
  for p = 2 to n // p is the chain length
    for i = 1 to n - p + 1 // all i, j combinations
      j = i + p - 1
      M[i][j] = ∞
      for k = i to j - 1 // find the best k
        q = M[i][k] + M[k + 1][j] + l[i - 1] * l[k] * l[j]
        if q < M[i][j]
          M[i][j] = q
          B[i][j] = k // backtracking
  return M and B
```

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

```
Print-Optimal-Parens(B, i, j)
  if i == j
    print A_i
  else
    print "("
    Print-Optimal-Parens(B, i, B[i][j])
    Print-Optimal-Parens(B, B[i][j] + 1, j)
    print ")"
```

$$T(n) = \Theta(n)$$

Exercise

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 x 35	35 x 15	15 x 5	5 x 10	10 x 20	20 x 25

j (columns), i (rows)

$M_{i,j}$	1	2	3	4	5	6
1	0	15,750	7,875	9,375	11,875	15,125
2		0	2,625	4,375	7,125	10,500
3			0	750	2,500	53,75
4				0	1,000	3,500
5					0	5,000
6						0

$B_{i,j}$	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

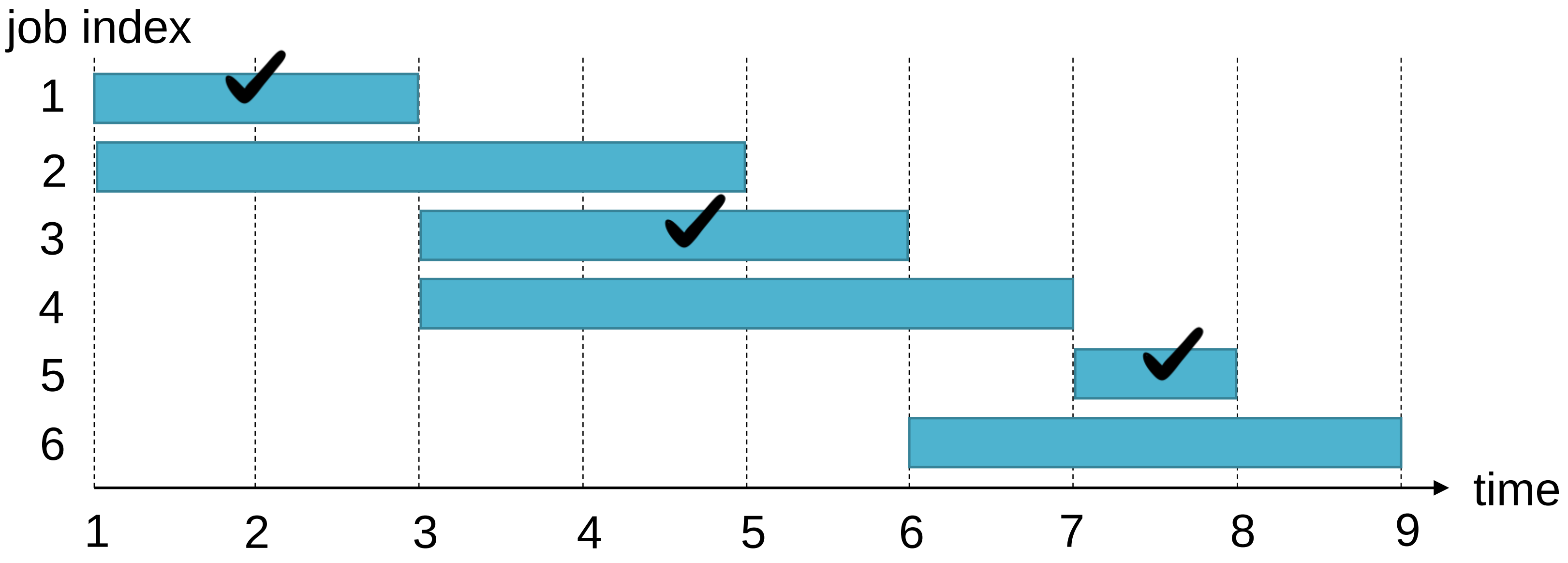
DP#5: Weighted Interval Scheduling

Textbook Exercise 16.2-2

Interval Scheduling

- Input: n job requests with start times s_i, finish times f_i
- Output: the maximum number of compatible jobs
- The interval scheduling problem can be solved using an "**early-finish-time-first**" greedy algorithm in $O(n)$ time

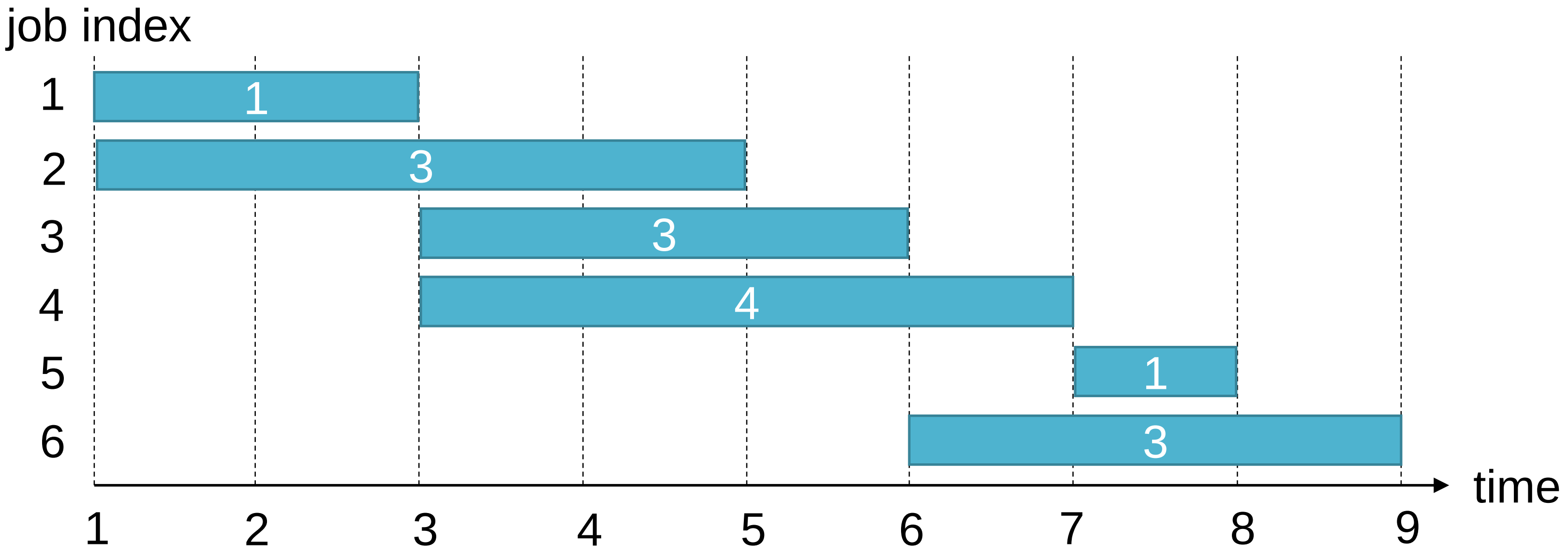
"Greedy Algorithm"
Next topic!



Weighted Interval Scheduling

- Input: n job requests with start times s_i, finish times f_i, and values v_i
- Output: the maximum total value obtainable from compatible jobs

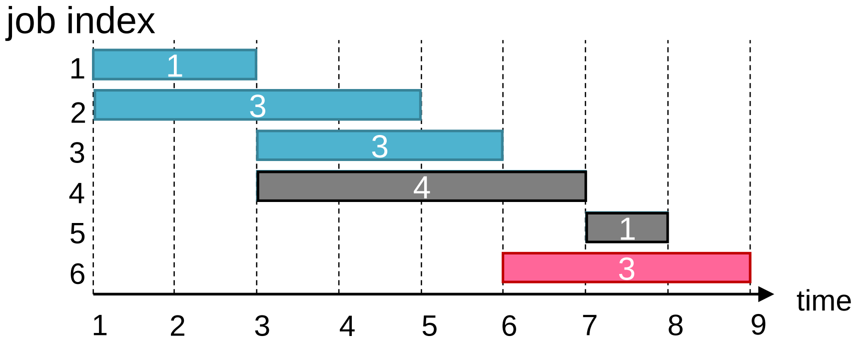
Assume that the requests are sorted in non-decreasing order ($f_i \leq f_j$ when $i < j$)
$p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
e.g. $p(1) = 0$, $p(2) = 0$, $p(3) = 1$, $p(4) = 1$, $p(5) = 4$, $p(6) = 3$



Step 1: Characterize an OPT Solution

Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the maximum total value obtainable from compatible

- Subproblems
  - `WIS(i)`: weighted interval scheduling for the first i jobs
  - Goal: `WIS(n)`
- Optimal substructure: suppose OPT is an optimal solution to `WIS(i)`, there are 2 cases:
  - Case 1: job i in OPT
    - OPT\\$\{i\}$ is an optimal solution of `WIS(p(i))`
  - Case 2: job i not in OPT
    - OPT is an optimal solution of `WIS(i-1)`



Step 2: Recursively Define the Value of an OPT Solution

Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the maximum total value obtainable from compatible

- Optimal substructure: suppose OPT is an optimal solution to `WIS(i)`, there are 2 cases:
  - Case 1: job i in OPT
    - OPT\\$\{i\}$ is an optimal solution of `WIS(p(i))`

$$M_i = v_i + M_{p(i)}$$

  - Case 2: job i not in OPT
    - OPT is an optimal solution of `WIS(i-1)`

$$M_i = M_{i-1}$$

- Recursively define the value

$$M_i = \begin{cases} 0 & \text{if } i = 0 \\ \max(v_i + M_{p(i)}, M_{i-1}) & \text{otherwise} \end{cases}$$

Step 3: Compute Value of an OPT Solution

Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the maximum total value obtainable from compatible

- Bottom-up method: solve smaller subproblems first

$$M_i = \begin{cases} 0 & \text{if } i = 0 \\ \max(v_i + M_{p(i)}, M_{i-1}) & \text{otherwise} \end{cases}$$

i	0	1	2	3	4	5	…	n
M[i]								

```
WIS(n, s, f, v, p)
  M[0] = 0
  for i = 1 to n
    M[i] = max(v[i] + M[p[i]], M[i - 1])
 return M[n]
```

$$T(n) = \Theta(n)$$

Step 4: Construct an OPT Solution by Backtracking

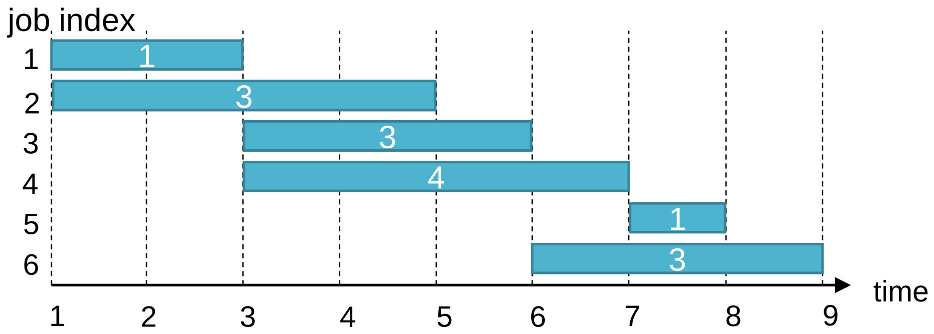
Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the maximum total value obtainable from compatible

- Bottom-up method: solve smaller subproblems first

$$M_i = \begin{cases} 0 & \text{if } i = 0 \\ \max(v_i + M_{p(i)}, M_{i-1}) & \text{otherwise} \end{cases}$$

i	0	1	2	3	4	5	6
M[i]	$\boxed{0}$	$\boxed{1}$	3	$\boxed{4}$	5	6	$\boxed{7}$

$v_1 + M_{p(1)} = 1 + M_0$ $v_3 + M_{p(3)} = 3 + M_1$ $v_6 + M_{p(6)} = 1 + M_3$



Step 4: Construct an OPT Solution by Backtracking

Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the maximum total value obtainable from compatible

```
WIS(n, s, f, v, p)
  M[0] = 0
  for i = 1 to n
    M[i] = max(v[i] + M[p[i]], M[i - 1])
 return M[n]
```

$$T(n) = \Theta(n)$$

```
Find-Solution(M, n)
  if n = 0
    return {}
  if v[n] + M[p[n]] > M[n-1] // case 1
    return {n} ∪ Find-Solution(p[n])
  return Find-Solution(n-1) // case 2
```

$$T(n) = \Theta(n)$$

DP#3: Knapsack (背包問題)

Textbook Exercise 16.2-2

Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the <u>maximum value</u> for the knapsack with capacity of W

- Variants of knapsack problem
  - 0-1 Knapsack Problem: 每項物品只能拿一個
  - Unbounded Knapsack Problem: 每項物品可以拿多個
  - Multidimensional Knapsack Problem: 背包空間有限
  - Multiple-Choice Knapsack Problem: 每一類物品最多拿一個
  - Fractional Knapsack Problem: 物品可以只拿部分

Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the <u>maximum value</u> for the knapsack with capacity of W

- Variants of knapsack problem
  - **0-1 Knapsack Problem: 每項物品只能拿一個**
  - Unbounded Knapsack Problem: 每項物品可以拿多個
  - Multidimensional Knapsack Problem: 背包空間有限
  - Multiple-Choice Knapsack Problem: 每一類物品最多拿一個
  - Fractional Knapsack Problem: 物品可以只拿部分

Step 1: Characterize an OPT Solution

0-1 Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i
Output: the max value within W capacity, where each item is chosen **at most once**

- Subproblems `ZO-KP(i)`  `ZO-KP(i, w)`

  consider the available capacity

  - `ZO-KP(i, w)`: 0-1 knapsack problem within w capacity for the first i items
  - Goal: `ZO-KP(n, W)`
- Optimal substructure: suppose OPT is an optimal solution to `ZO-KP(i, w)`, there are 2 cases:
  - Case 1: item i in OPT
    - OPT$\setminus\{i\}$ is an optimal solution of `ZO-KP(i - 1, w - w_i)`
  - Case 2: item i not in OPT
    - OPT is an optimal solution of `ZO-KP(i - 1, w)`

Step 2: Recursively Define the Value of an OPT Solution

0-1 Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i
Output: the max value within W capacity, where each item is chosen **at most once**

- Optimal substructure: suppose OPT is an optimal solution to `ZO-KP(i, w)`, there are 2 cases:
  - Case 1: item i in OPT
    - OPT\\{i} is an optimal solution of `ZO-KP(i - 1, w - w_i)` $M_{i,w} = v_i + M_{i-1,w-w_i}$
  - Case 2: item i not in OPT
    - OPT is an optimal solution of `ZO-KP(i - 1, w)` $M_{i,w} = M_{i-1,w}$
- Recursively define the value

$$M_{i,w} = \begin{cases} 0 & \text{if } i = 0 \\ M_{i-1,w} & \text{if } w_i > w \\ \max(v_i + M_{i-1,w-w_i}, M_{i-1,w}) & \text{otherwise} \end{cases}$$

Step 3: Compute Value of an OPT Solution

0-1 Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i
Output: the max value within W capacity, where each item is chosen **at most once**

- Bottom-up method: solve smaller subproblems first

$$M_{i,w} = \begin{cases} 0 & \text{if } i = 0 \\ M_{i-1,w} & \text{if } w_i > w \\ \max(v_i + M_{i-1,w-w_i}, M_{i-1,w}) & \text{otherwise} \end{cases}$$

i\w	0	1	2	3	…	w	…	W
0								
1								
2			$M_{i-1,w-w_i}$			$M_{i-1,w}$		
i						$\boxed{M_{i,w}}$		
n								

Step 3: Compute Value of an OPT Solution

0-1 Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i
Output: the max value within W capacity, where each item is chosen **at most once**

- Bottom-up method: solve smaller subproblems first

$$M_{i,w} = \begin{cases} 0 & \text{if } i = 0 \\ M_{i-1,w} & \text{if } w_i > w \\ \max(v_i + M_{i-1,w-w_i}, M_{i-1,w}) & \text{otherwise} \end{cases}$$

i	w_i	v_i
1	1	4
2	2	9
3	4	20

$W = 5$

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	4	4	4	4	4
2	0	4	9	13	13	13
3	0	4	9	13	20	24

Step 3: Compute Value of an OPT Solution

0-1 Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i
Output: the max value within W capacity, where each item is chosen **at most once**

- Bottom-up method: solve smaller subproblems first

$$M_{i,w} = \begin{cases} 0 & \text{if } i = 0 \\ M_{i-1,w} & \text{if } w_i > w \\ \max(v_i + M_{i-1,w-w_i}, M_{i-1,w}) & \text{otherwise} \end{cases}$$

```
ZO-KP(n, v, W)
  for w = 0 to W
    M[0, w] = 0
  for i = 1 to n
    for w = 0 to W
      if(wi > w)
        M[i, w] = M[i-1, w]
      else
        M[i, w] = max(vi + M[i-1, w-wi], M[i-1, w])
  return M[n, W]
```

$$T(n) = \Theta(nW)$$

Step 4: Construct an OPT Solution by Backtracking

```
ZO-KP(n, v, W)
  for w = 0 to W
    M[0, w] = 0
  for i = 1 to n
    for w = 0 to W
      if(wi > w)
        M[i, w] = M[i-1, w]
      else
        M[i, w] = max(vi + M[i-1, w-wi], M[i-1, w])
  return M[n, W]
```

$$T(n) = \Theta(nW)$$

```
Find-Solution(M, n, W)
  S = {}
  w = W
  for i = n to 1
    if M[i, w] > M[i - 1, w] // case 1
      w = w - wi
      S = S U {i}
  return S
```

$$T(n) = \Theta(n)$$

Pseudo-Polynomial Time

- Polynomial: polynomial in the **length of the input** (#bits for the input)
- Pseudo-polynomial: polynomial in the **numeric value**

- The time complexity of 0-1 knapsack problem is $\Theta(nW)$
  - n: number of objects
  - W: knapsack's capacity (non-negative integer)
  - polynomial in the numeric value

  = pseudo-polynomial in input size

  = exponential in the length of the input
- Note: the size of the representation of W is $\log_2 W$

$= 2^m \quad = m$

Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the <u>maximum value</u> for the knapsack with capacity of W

- Variants of knapsack problem
  - 0-1 Knapsack Problem: 每項物品只能拿一個
  - **Unbounded Knapsack Problem: 每項物品可以拿多個**
  - Multidimensional Knapsack Problem: 背包空間有限
  - Multiple-Choice Knapsack Problem: 每一類物品最多拿一個
  - Fractional Knapsack Problem: 物品可以只拿部分

Step 1: Characterize an OPT Solution

Unbounded Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i has **unlimited supplies**
Output: the max value within W capacity

- Subproblems
  - `U-KP(i, w)`: unbounded knapsack problem with w capacity for the first i items
  - Goal: `U-KP(n, W)`

0-1 Knapsack Problem	Unbounded Knapsack Problem
each item can be chosen **at most once**	each item can be chosen **multiple times**
a sequence of binary choices: whether to choose item i	a sequence of i choices: which one (from 1 to i) to choose
Time complexity = $\Theta(nW)$	Time complexity = $\Theta(n^2W)$

Can we do better?

Step 1: Characterize an OPT Solution

Unbounded Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i has **unlimited supplies**
Output: the max value within W capacity

- Subproblems
  - `U-KP(w)`: unbounded knapsack problem with w capacity
  - Goal: `U-KP(W)`
- Optimal substructure: suppose OPT is an optimal solution to `U-KP(w)`, there are n cases:
  - Case 1: item 1 in OPT
    - Removing an item 1 from OPT is an optimal solution of `U-KP(w - w`$_1$`)`
  - Case 2: item 2 in OPT
    - Removing an item 2 from OPT is an optimal solution of `U-KP(w - w`$_2$`)`

  :
  - Case n: item n in OPT
    - Removing an item n from OPT is an optimal solution of `U-KP(w - w`$_n$`)`

Step 2: Recursively Define the Value of an OPT Solution

Unbounded Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i has **unlimited supplies**
Output: the max value within W capacity

- Optimal substructure: suppose OPT is an optimal solution to `U-KP(w)`, there are n cases:
  - Case i: item i in OPT
    - Removing an item i from OPT is an optimal solution of `U-KP(w - w_1)`

$$M_w = v_i + M_{w-w_i}$$

- Recursively define the value

$$M_w = \begin{cases} 0 & \text{if } w = 0 \text{ or } w_i > w \text{ for all } i \\ \max_{1 \le i \le n, \boxed{w_i \le w}} (v_i + M_{w-w_i}) & \text{otherwise} \end{cases}$$

只考慮背包還裝的下的情形

Step 3: Compute Value of an OPT Solution

Unbounded Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i has **unlimited supplies**
Output: the max value within W capacity

- Bottom-up method: solve smaller subproblems first

$$M_w = \begin{cases} 0 & \text{if } w = 0 \text{ or } w_i > w \text{ for all } i \\ \max_{1 \le i \le n, w_i \le w}(v_i + M_{w-w_i}) & \text{otherwise} \end{cases}$$

w	0	1	2	3	4	5	…	W
M[w]								

i	w_i	v_i
1	1	4
2	2	9
3	4	20

$W = 5$

Step 3: Compute Value of an OPT Solution

Unbounded Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i has **unlimited supplies**
Output: the max value within W capacity

- Bottom-up method: solve smaller subproblems first

$$M_w = \begin{cases} 0 & \text{if } w = 0 \text{ or } w_i > w \text{ for all } i \\ \max_{1 \le i \le n, w_i \le w}(v_i + M_{w-w_i}) & \text{otherwise} \end{cases}$$

w	0	1	2	3	4	5
M[w]	0	4	9	13	18	22

$\max(4+0)$

$\max(4+4, 9+0)$

$\max(4+9, 9+4)$

$\max(4+13, 9+9, 17+0)$

$\max(4+18, 9+13, 17+4)$

i	w_i	v_i
1	1	4
2	2	9
3	4	20

$W = 5$

Step 3: Compute Value of an OPT Solution

Unbounded Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i has **unlimited supplies**
Output: the max value within W capacity

- Bottom-up method: solve smaller subproblems first

$$M_w = \begin{cases} 0 & \text{if } w = 0 \text{ or } w_i > w \text{ for all } i \\ \max_{1 \le i \le n, w_i \le w}(v_i + M_{w-w_i}) & \text{otherwise} \end{cases}$$

```
U-KP(v, W)
  for w = 0 to W
    M[w] = 0
  for w = 0 to W
    for i = 1 to n
      if(wi <= w)
        tmp = vi + M[w - wi]
        M[w] = max(M[w], tmp)
  return M[W]
```

$$T(n) = \Theta(nW)$$

Step 4: Construct an OPT Solution by Backtracking

```
U-KP(v, W)
  for w = 0 to W
    M[w] = 0
  for w = 0 to W
    for i = 1 to n
      if(wi <= w)
        tmp = vi + M[w - wi]
        M[w] = max(M[w], tmp)
  return M[W]
```

$$T(n) = \Theta(nW)$$

```
Find-Solution(M, n, W)
  for i = 1 to n
    C[i] = 0 // C[i] = # of item i in solution
  w = W
  for i = i to n
    while w > 0
      if(wi <= w && M[w] == (vi + M[w - wi]))
        w = w - wi
        C[i] += 1
  return C
```

$$T(n) = \Theta(n + W)$$

Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the maximum value for the knapsack with capacity of W

- Variants of knapsack problem
  - 0-1 Knapsack Problem: 每項物品只能拿一個
  - Unbounded Knapsack Problem: 每項物品可以拿多個
  - **Multidimensional Knapsack Problem: 背包空間有限**
  - Multiple-Choice Knapsack Problem: 每一類物品最多拿一個
  - Fractional Knapsack Problem: 物品可以只拿部分

Step 1: Characterize an OPT Solution

Multidimensional Knapsack Problem
Input: n items where i-th item has value v_i, weighs w_i, and size d_i
Output: the max value within W capacity and with **the size of D**, where each item is chosen at most once

- Subproblems
  - `M-KP(i, w, d)`: multidimensional knapsack problem with w capacity and d size for the first i items
  - Goal: `M-KP(n, W, D)`
- Optimal substructure: suppose OPT is an optimal solution to `M-KP(i, w, d)`, there are 2 cases:
  - Case 1: item i in OPT
    - OPT$\setminus\{i\}$ is an optimal solution of `M-KP(i - 1, w - w_i, d - d_i)`
  - Case 2: item i not in OPT
    - OPT is an optimal solution of `M-KP(i - 1, w, d)`

Step 2: Recursively Define the Value of an OPT Solution

Multidimensional Knapsack Problem
Input: n items where i-th item has value v_i, weighs w_i, and size d_i
Output: the max value within W capacity and with **the size of $\boldsymbol{D}$**, where each item is chosen at most once

- Optimal substructure: suppose OPT is an optimal solution to `M-KP(i, w, d)`, there are 2 cases:
  - Case 1: item i in OPT
    - OPT\\$\{i\}$ is an optimal solution of `M-KP(i - 1, w - wi, d - di)`

      $M_{i,w,d} = v_i + M_{i-1,w-w_i,d-d_i}$
  - Case 2: item i not in OPT
    - OPT is an optimal solution of `M-KP(i - 1, w, d)`

      $M_{i,w,d} = M_{i-1,w,d}$
- Recursively define the value

$$M_{i,w,d} = \begin{cases} 0 & \text{if } i = 0 \\ M_{i-1,w,d} & \text{if } w_i > w \text{ or } d_i > d \\ \max(v_i + M_{i-1,w-w_i,d-d_i}, M_{i-1,w,d}) & \text{otherwise} \end{cases}$$

Exercise

Multidimensional Knapsack Problem
Input: n items where i-th item has value v_i, weighs w_i, and size d_i
Output: the max value within W capacity and with **the size of $\boldsymbol{D}$**, where each item is chosen at most once

- Step 3: Compute Value of an OPT Solution
- Step 4: Construct an OPT Solution by Backtracking
- What is the time complexity?

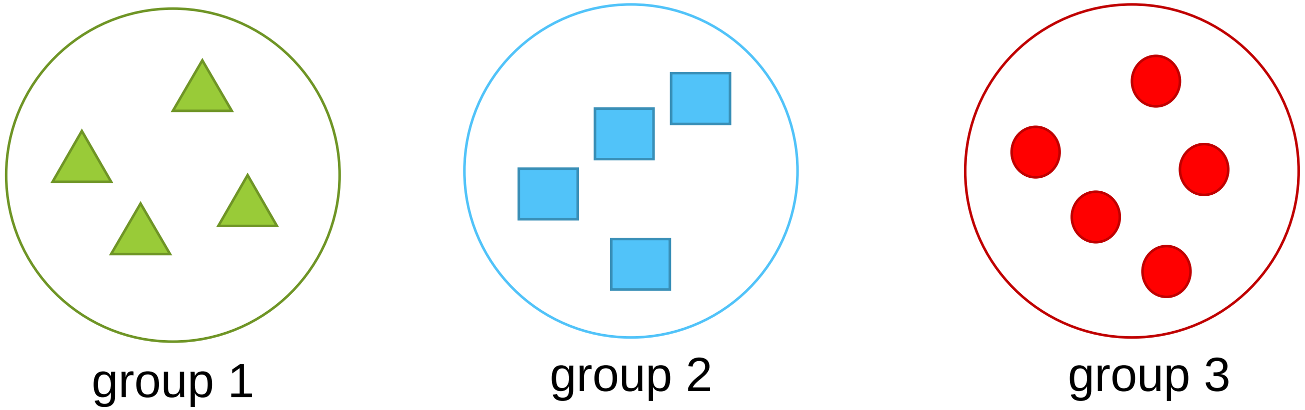
Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the maximum value for the knapsack with capacity of W

- Variants of knapsack problem
  - 0-1 Knapsack Problem: 每項物品只能拿一個
  - Unbounded Knapsack Problem: 每項物品可以拿多個
  - Multidimensional Knapsack Problem: 背包空間有限
  - **Multiple-Choice Knapsack Problem: 每一類物品最多拿一個**
  - Fractional Knapsack Problem: 物品可以只拿部分

Multiple-Choice Knapsack Problem

- Input: n items
  - $v_{i,j}$: value of j-th item in the group i
  - $w_{i,j}$: weight of j-th item in the group i
  - n_i: number of items in group i
  - n: total number of items ($\sum n_i$)
  - G: total number of groups
- Output: the maximum value for the knapsack with capacity of W, where **the item from each group can be selected at most once**



Step 1: Characterize an OPT Solution

Multiple-Choice Knapsack Problem
Input: n items with value $v_{i,j}$ and weighs $w_{i,j}$ (n_i: #items in group i, G: #groups)
Output: the max value within W capacity, where each group is chosen **at most once**

- Subproblems
  - `MC-KP(w)`: w capacity
  -  `MC-KP(i, w)`: w capacity for the first i groups — the constraint is for groups
  - `MC-KP(i, j, w)`: w capacity for the first j items from first i groups

Which one is more suitable for this problem?

Step 1: Characterize an OPT Solution

Multiple-Choice Knapsack Problem
Input: n items with value $v_{i,j}$ and weighs $w_{i,j}$ (n_i: #items in group i, G: #groups)
Output: the max value within W capacity, where each group is chosen **at most once**

- Subproblems
  - `MC-KP(i, w)`: multi-choice knapsack problem with w capacity for the first i groups
  - Goal: `MC-KP(G, W)`
- Optimal substructure: suppose OPT is an optimal solution to `MC-KP(i, w)`, for the group i, there are $n_i + 1$ cases:
  - Case 1: no item from i-th group in OPT
    - OPT is an optimal solution of `MC-KP(i - 1, w)`

    :
  - Case $j + 1$: j-th item from i-th group ($item_{i,j}$) in OPT
    - OPT\ $item_{i,j}$ is an optimal solution of `MC-KP(i - 1, w - w_{i,j})`

Step 2: Recursively Define the Value of an OPT Solution

Multiple-Choice Knapsack Problem
Input: n items with value $v_{i,j}$ and weighs $w_{i,j}$ (n_i: #items in group i, G: #groups)
Output: the max value within W capacity, where each group is chosen **at most once**

- Optimal substructure: suppose OPT is an optimal solution to `MC-KP(i, w)`, for the group i, there are $n_i + 1$ cases:
  - Case 1: no item from i-th group in OPT
    - OPT is an optimal solution of `MC-KP(i - 1, w)`

$$M_{i,w} = M_{i-1,w}$$

  - Case $j + 1$: j-th item from i-th group ($\text{item}_{i,j}$) in OPT
    - OPT\$\text{item}_{i,j}$ is an optimal solution of `MC-KP(i - 1, w - w_{i,j})`

$$M_{i,w} = v_{i,j} + M_{i-1,w-w_{i,j}}$$

- Recursively define the value

$$M_{i,w} = \begin{cases} 0 & \text{if } i = 0 \\ M_{i-1,w} & \text{if } w_{i,j} > w \text{ for all } j \\ \underbrace{\max_{1 \le j \le n_i}(v_{i,j} + M_{i-1,w-w_{i,j}}, M_{i-1,w})}_{n_i + 1} & \text{otherwise} \end{cases}$$

Step 3: Compute Value of an OPT Solution

Multiple-Choice Knapsack Problem
Input: n items with value $v_{i,j}$ and weighs $w_{i,j}$ (n_i: #items in group i, G: #groups)
Output: the max value within W capacity, where each group is chosen **at most once**

- Bottom-up method: solve smaller subproblems first

$$M_{i,w} = \begin{cases} 0 & \text{if } i = 0 \\ M_{i-1,w} & \text{if } w_{i,j} > w \text{ for all } j \\ \max_{1 \le j \le n_i}(v_{i,j} + M_{i-1,w-w_{i,j}}, M_{i-1,w}) & \text{otherwise} \end{cases}$$

i\w	0	1	2	3	…	w	…	W
0								
1								
2			$M_{i-1,w-w_{i,j}}$			$M_{i-1,w}$		
i						$\boxed{M_{i,w}}$		
n								

Step 3: Compute Value of an OPT Solution

Multiple-Choice Knapsack Problem

Input: n items with value $v_{i,j}$ and weighs $w_{i,j}$ (n_i: #items in group i, G: #groups)

Output: the max value within W capacity, where each group is chosen **at most once**

- Bottom-up method: solve smaller subproblems first

```
MC-KP(n, v, W)
  for w = 0 to W
    M[0, w] = 0
  for i = 1 to G // consider groups 1 to i
    for w = 0 to W // consider capacity = w
        M[i, w] = M[i - 1, w]
        for j = 1 to n_i // check j-th item in group i
          if(v_i,j + M[i - 1, w - w_i,j] > M[i, w])
            M[i, w] = v_i,j + M[i - 1, w - w_i,j]
  return M[G, W]
```

$$T(n) = \Theta(nW)$$

$$\sum_{i=1}^{G}\sum_{w=0}^{W}\sum_{j=1}^{n_i} c = c\sum_{w=0}^{W}\sum_{i=1}^{G}\sum_{j=1}^{n_i} 1 = c\sum_{w=0}^{W} n = cnW$$

Step 4: Construct an OPT Solution by Backtracking

```
MC-KP(n, v, W)
  for w = 0 to W
    M[0, w] = 0
for i = 1 to G // consider groups 1 to i
    for w = 0 to W // consider capacity = w
        M[i, w] = M[i - 1, w]
        for j = 1 to n_i // check items in group i
          if(v_i,j + M[i - 1, w - w_i,j] > M[i, w])
            M[i, w] = v_i,j + M[i - 1, w - w_i,j]
            B[i, w] = j
  return M[G, W], B[G, W]
```

$$T(n) = \Theta(nW)$$

Practice to write the pseudo code for `Find-Solution()`

$$T(n) = \Theta(G + W)$$

Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the maximum value for the knapsack with capacity of W

- Variants of knapsack problem
  - 0-1 Knapsack Problem: 每項物品只能拿一個
  - Unbounded Knapsack Problem: 每項物品可以拿多個
  - Multidimensional Knapsack Problem: 背包空間有限
  - Multiple-Choice Knapsack Problem: 每一類物品最多拿一個
  - **Fractional Knapsack Problem: 物品可以只拿部分**

Fractional Knapsack Problem

- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the maximum value for the knapsack with capacity of W, where we can take **any fraction of items**

- Dynamic programming algorithm should work

Can we do better?

- Choose maximal $\frac{v_i}{w_i}$ (類似CP值) first

"Greedy Algorithm"
Next topic!

Concluding Remarks

- "Dynamic Programming": solve many subproblems in polynomial time for which a naïve approach would take exponential time
- When to use DP
  - Whether subproblem solutions can combine into the original solution
  - When subproblems are <u>overlapping</u>
  - Whether the problem has <u>optimal substructure</u>
  - Common for <u>optimization</u> problem
- Two ways to avoid recomputation
  - Top-down with memoization
  - Bottom-up method
- Complexity analysis
  - Space for tabular filling
  - Size of the subproblem graph

Question?

Important announcement will be sent to
@ntu.edu.tw mailbox & post to the course website

Course Website: http://ada.miulab.tw
Email: ada-ta@csie.ntu.edu.tw