OPTIMAL BINARY SEARCH TREE <<DYNAMIC PROGRAMMING>>

Dynamic Programming (2)

Oct 11th, 2018

Algorithm Design and Analysis

YUN-NUNG (VIVIAN) CHEN HTTP://ADA.MIULAB.TW

國立臺灣大學 National Taiwan University

Slides credited from Hsueh-I Lu, Hsu-Chun Hsiao, & Michael Tsai

Announcement

- Mini-HW 4 released
  - Due on 10/18 (Thu) 14:20
- Homework 1 due a week later
  - A4 hardcopy submitted before the class starts
  - Softcopy submitted to NTU COOL before the deadline
- Homework 2 released
  - Due on 11/06 (Tue) 18:00 (3.5 weeks)
  - Submitted to NTU COOL only

Frequently check the website for the updated information!

Mini-HW 4

Consider a 0/1 Knapsack Problem where you have ***N*** objects to choose from

The weight and value of each object is listed in the table below

Weight	1	3	4	5	8	10	11
Value	3	7	10	12	17	19	21

1. Construct a DP table to fill knapsack with capacity ***W = 15*** (80%)
   (Your DP algorithm must run in ***O(N*W)*** time)

2. Briefly explain whether your algorithm can adapt to objects with non-integer weight (20%)

Homework 2

Homework #2

Due Time: 2018/11/06 (Tues.) 18:00
Contact TAs: `ada-ta@csie.ntu.edu.tw`

Instructions and Announcements

- There are **four programming problems** and **three hand-written problems**, and the homework set including bonus are worthy of 110 points. If you get more than 100 points, your score will still be counted as 100 points.
- **Programming.** The judge system is located at `https://ada18-judge.csie.org`. Please login and submit your code for the programming problems (i.e., those containing "Programming" in the problem title) by the deadline. NO LATE SUBMISSION IS ALLOWED.
- **Hand-written.** For other problems (also known as the "hand-written problems"), please turn in a **printed/written version** of your answers to the instructor at the beginning of the class on 2018/11/01, or put them in the box in front of R307 before the deadline. **Remember to print your name/student ID on the first page of your submitted answers.** In case that your homework is lost during the grading, you can also upload your homework to the NTU COOL system. NO LATE SUBMISSION IS ALLOWED.

Outline

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

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?

8 Review

Algorithm Design Paradigms

- Divide-and-Conquer
  - partition the problem into **independent** or **disjoint** subproblems
  - repeatedly solving the common subsubproblems

  → more work than necessary

- Dynamic Programming
  - partition the problem into **dependent** or **overlapping** subproblems
  - avoid recomputation
    - ✓ Top-down with memoization
    - ✓ Bottom-up method

Dynamic Programming Procedure

- DP procedure
  1. Characterize the structure of an optimal solution
  2. **Recursively** define the value of an optimal solution
  3. Compute the value of an optimal solution, typically in a **bottom-up** fashion
  4. Construct an optimal solution from computed information

- Two key properties of DP for optimization
  - Overlapping subproblems
  - Optimal substructure – an optimal solution can be constructed from optimal solutions to subproblems
    - ✓ Reduce search space (ignore non-optimal solutions)

11 DP#1: Rod Cutting

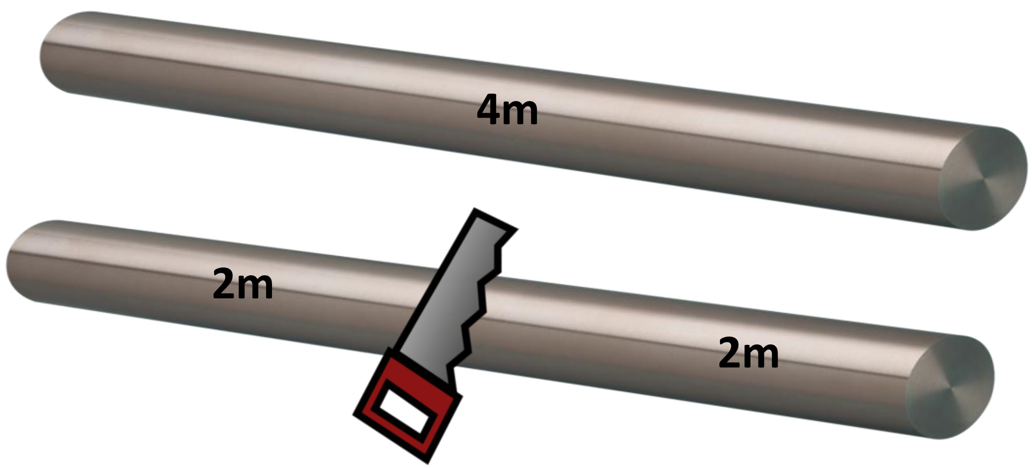
Textbook Chapter 15.1 – Rod Cutting

Rod Cutting Problem

- Input: a rod of length n and a table of prices p_i for $i = 1, \dots, n$

length i (m)	1	2	3	4	5
price p_i	1	5	8	9	10

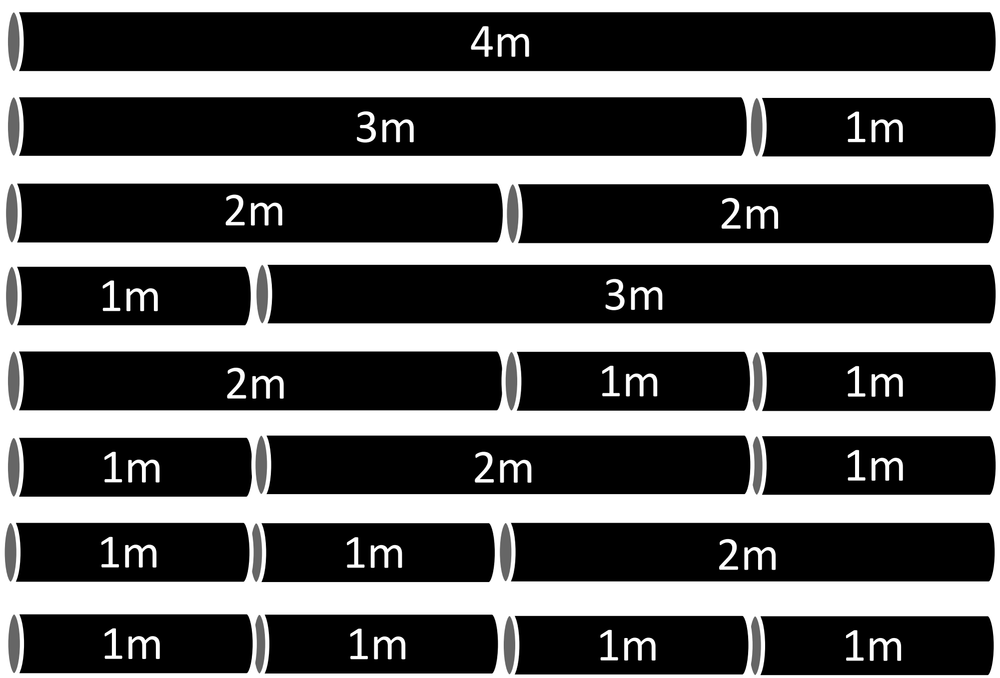
- Output: the maximum revenue r_n obtainable by cutting up the rod and selling the pieces



Brute-Force Algorithm

length i (m)	**1**	**2**	**3**	**4**	**5**
price p_i	1	5	8	9	10

- A rod with the length = 4

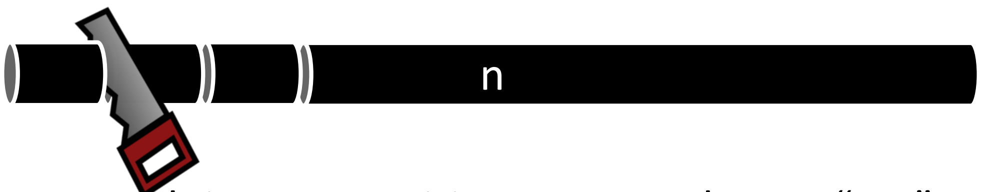


→ 9
→ 8 + 1 = 9
→ 5 + 5 = $\boxed{10}$
→ 1 + 8 = 9
→ 5 + 1 + 1 = 7
→ 1 + 5 + 1 = 7
→ 1 + 1 + 5 = 7
→ 1 + 1 + 1 + 1 = 4

Brute-Force Algorithm

length i (m)	1	2	3	4	5
price p_i	1	5	8	9	10

- A rod with the length = n

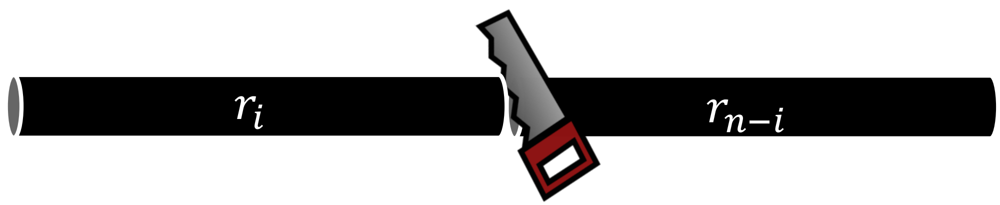


  - For each integer position, we can choose "cut" or "not cut"
  - There are $n - 1$ positions for consideration
- The total number of cutting results is $2^{n-1} = \Theta(2^{n-1})$

Recursive Thinking

r_n: the maximum revenue obtainable for a rod of length n

- We use a *recursive* function to solve the subproblems
- If we know the answer to the subproblem, can we get the answer to the original problem?



$$r_n = \max(\underbrace{p_n}_{\text{no cut}}, \underbrace{r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1}_{\text{cut at the i-th position (from left to right)}})$$

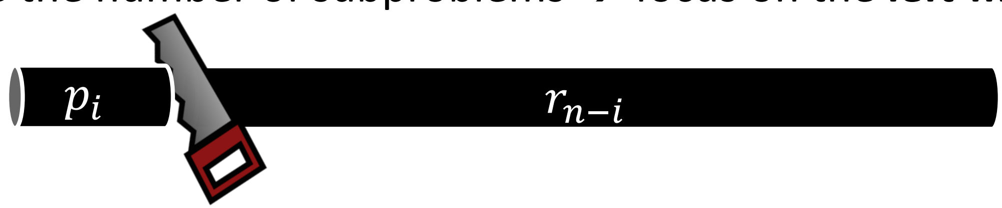
- Optimal substructure – an optimal solution can be constructed from optimal solutions to subproblems

Recursive Algorithms

- Version 1

$$r_n = \max(\underbrace{p_n}_{\text{no cut}}, \underbrace{r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1}_{\text{cut at the i-th position (from left to right)}})$$

- Version 2
  - try to reduce the number of subproblems → focus on the **left-most** cut



$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

left-most value

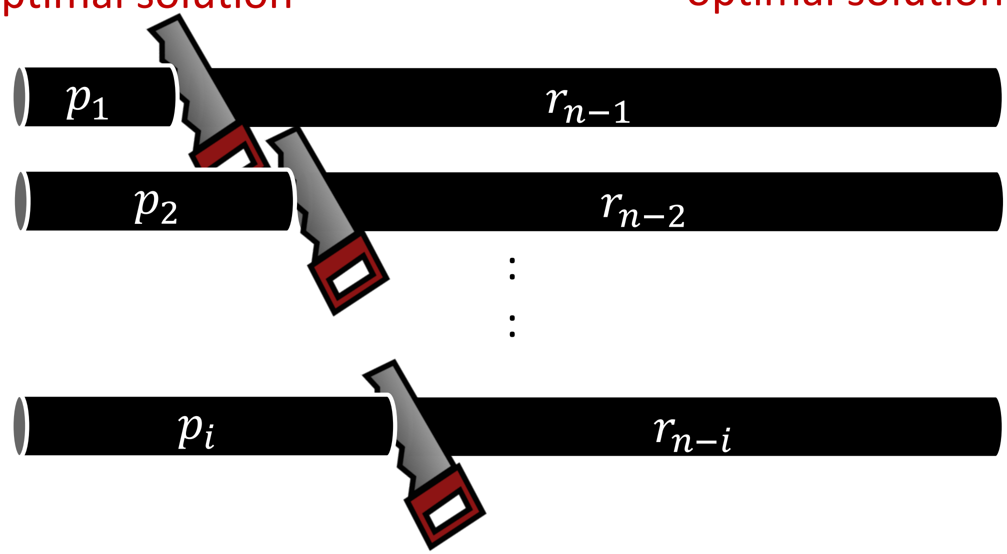
maximum value obtainable from the remaining part

Recursive Procedure

- Focus on the left-most cut
  - assume that we always cut **from left to right** → the **first cut**

optimal solution

optimal solution to subproblems



Rod cutting problem has optimal substructure

Naïve Recursion Algorithm

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

```
Cut-Rod(p, n)
  // base case
  if n == 0
    return 0
  // recursive case
  q = -∞
  for i = 1 to n
    q = max(q, p[i] + Cut-Rod(p, n - i))
  return q
```

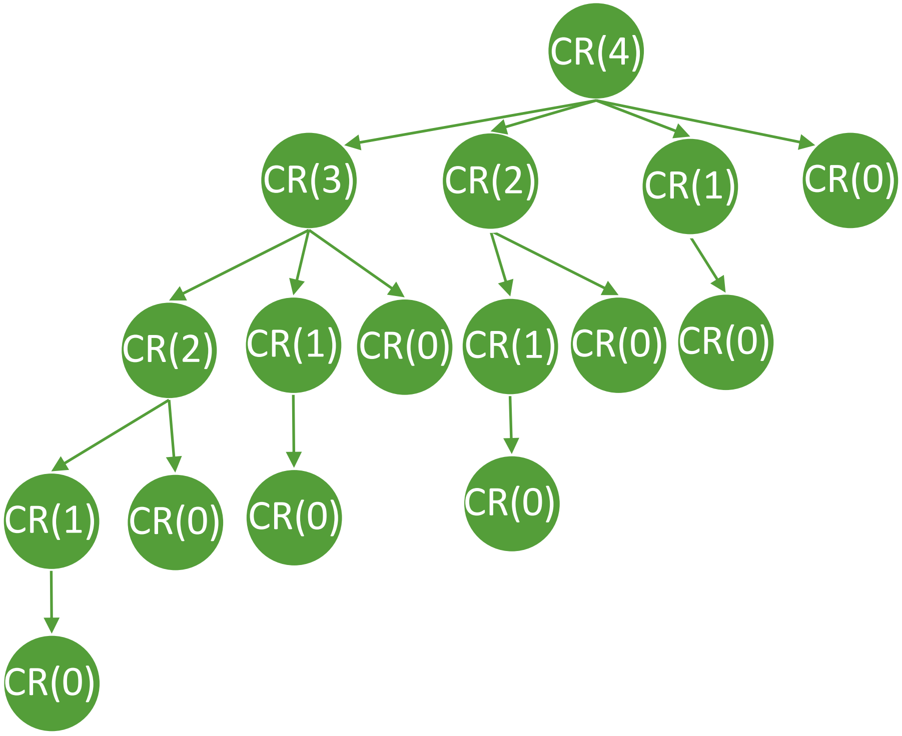
- $T(n)$ = time for running `Cut-Rod(p, n)`

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(1) + \sum_{i=0}^{n} T(n-i) & \text{if } n \ge 2 \end{cases} \Rightarrow T(n) = \Theta(2^n)$$

Naïve Recursion Algorithm

- Rod cutting problem

```
Cut-Rod(p, n)
  // base case
  if n == 0
    return 0
  // recursive case
  q = -∞
  for i = 1 to n
    q = max(q, p[i] + Cut-Rod(p, n - i))
  return q
```



Calling overlapping subproblems result in poor efficiency

Dynamic Programming

- Idea: use space for better time efficiency
- Rod cutting problem has **overlapping subproblems** and **optimal substructures** → can be solved by DP
- When the number of subproblems is polynomial, the time complexity is polynomial using DP
- DP algorithm
  - Top-down: solve overlapping subproblems recursively with memoization
  - Bottom-up: build up solutions to larger and larger subproblems

Dynamic Programming

- Top-Down with Memoization
  - Solve recursively and memo the subsolutions (跳著填表)
  - Suitable that **not all subproblems should be solved**

f(0)	f(1)	f(2)	…	f(n)

- Bottom-Up with Tabulation
  - Fill the table **from small to large**
  - Suitable that **each small problem should be solved**

f(0)	f(1)	f(2)	…	f(n)

Algorithm for Rod Cutting Problem

Top-Down with Memoization

```
Memoized-Cut-Rod(p, n)
  // initialize memo (an array r[] to keep max revenue)
  r[0] = 0
  for i = 1 to n                                              Θ(n)
    r[i] = -∞ // r[i] = max revenue for rod with length=i
  return Memorized-Cut-Rod-Aux(p, n, r)

Memoized-Cut-Rod-Aux(p, n, r)
  if r[n] >= 0                                                Θ(1)
    return r[n] // return the saved solution
  q = -∞
  for i = 1 to n                                              Θ(n^2)
    q = max(q, p[i] + Memoized-Cut-Rod-Aux(p, n-i, r))
  r[n] = q // update memo
  return q
```

- $T(n)$ = time for running `Memoized-Cut-Rod(p, n)` ➡ $T(n) = \Theta(n^2)$

Algorithm for Rod Cutting Problem

Bottom-Up with Tabulation

```
Bottom-Up-Cut-Rod(p, n)
  r[0] = 0
  for j = 1 to n // compute r[1], r[2], ... in order
    q = -∞
    for i = 1 to j
      q = max(q, p[i] + r[j - i])
    r[j] = q
  return r[n]
```

$\Theta(n^2)$

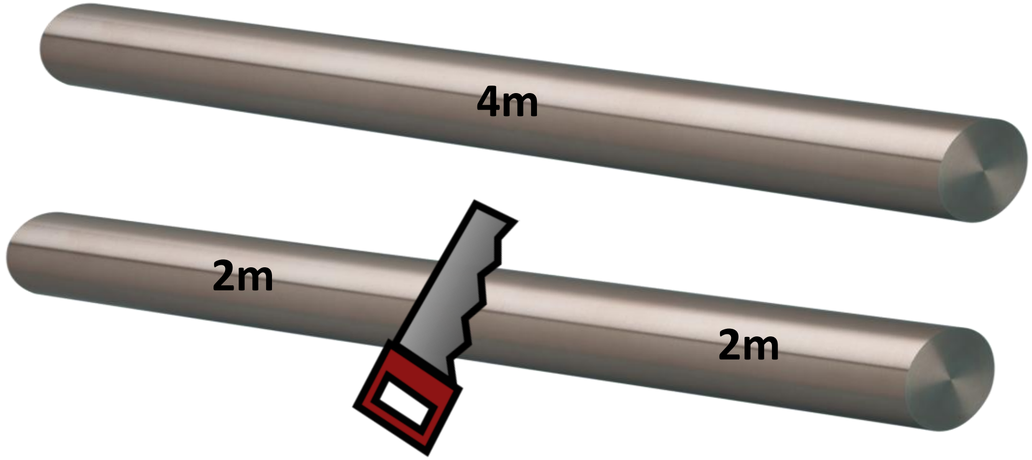
- $T(n)$ = time for running `Bottom-Up-Cut-Rod(p, n)` ➡ $T(n) = \Theta(n^2)$

Rod Cutting Problem

- Input: a rod of length n and a table of prices p_i for $i = 1, \dots, n$

length i (m)	1	2	3	4	5
price p_i	1	5	8	9	10

- Output: the maximum revenue r_n obtainable and **the list of cut pieces**



Algorithm for Rod Cutting Problem

Bottom-Up with Tabulation

- Add an array to keep the cutting positions **cut**

```
Extended-Bottom-Up-Cut-Rod(p, n)
  r[0] = 0
  for j = 1 to n //compute r[1], r[2], ... in order
  q = -∞
    for i = 1 to j
      if q < p[i] + r[j - i]
        q = p[i] + r[j - i]
        cut[j] = i // the best first cut for len j rod
    r[i] = q
  return r[n], cut
```

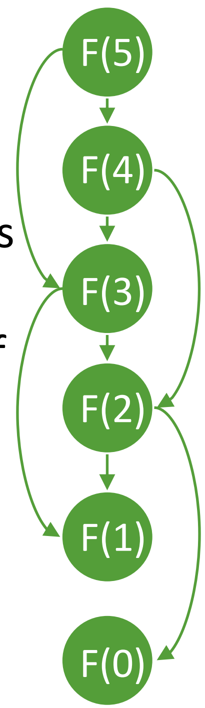
```
Print-Cut-Rod-Solution(p, n)
  (r, cut) = Extended-Bottom-up-Cut-Rod(p, n)
  while n > 0
    print cut[n]
    n = n - cut[n] // remove the first piece
```

Dynamic Programming

- Top-Down with Memoization

f(0)	f(1)	f(2)	…	f(n)

  - Better when some subproblems not be solved at all
  - Solve only the required parts of subproblems



- Bottom-Up with Tabulation

f(0)	f(1)	f(2)	…	f(n)

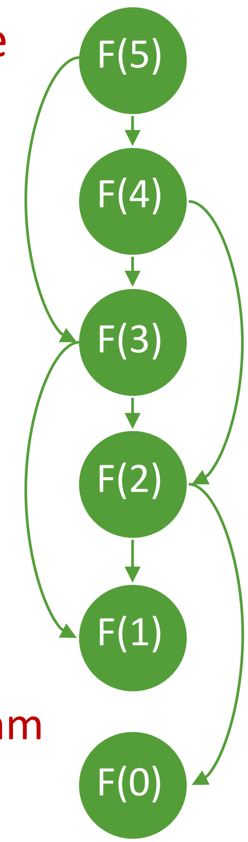
  - Better when all subproblems must be solved at least once
  - Typically outperform top-down method by a constant factor
    - No overhead for recursive calls
    - Less overhead for maintaining the table

Informal Running Time Analysis

- Approach 1: approximate via (#subproblems) * (#choices for each subproblem)
  - For rod cutting
    - #subproblems = n
    - #choices for each subproblem = O(n)
    - → T(n) is about $O(n^2)$
- Approach 2: approximate via subproblem graphs

Subproblem Graphs

- The size of the subproblem graph allows us to estimate the time complexity of the DP algorithm
- A graph illustrates the set of subproblems involved and how subproblems depend on another $G = (V, E)$ (E: edge, V: vertex)
  - $|V|$: #subproblems
    - A subproblem is run only once
  - $|E|$: sum of #subsubproblems are needed for each subproblem
  - Time complexity: linear to $O(|E| + |V|)$



Top-down: Depth First Search

Bottom-up: Reverse Topological Sort

Graph Algorithm (taught later)

Dynamic Programming Procedure

1. Characterize the structure of an optimal solution
   - ✓ Overlapping subproblems: revisit same subproblems
   - ✓ Optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems
2. Recursively define the value of an optimal solution
   - ✓ Express the solution of the original problem in terms of optimal solutions for subproblems
3. Compute the value of an optimal solution
   - ✓ typically in a bottom-up fashion
4. Construct an optimal solution from computed information
   - ✓ Step 3 and 4 may be combined

Revisit DP for Rod Cutting Problem

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. Construct an optimal solution from computed information

Step 1: Characterize an OPT Solution

Rod Cutting Problem
Input: a rod of length n and a table of prices p_i for $i = 1, \dots, n$
Output: the maximum revenue r_n obtainable

- Step 1-Q1: What can be the subproblems?
- Step 1-Q2: Does it exhibit optimal structure? (an optimal solution can be represented by the optimal solutions to subproblems)
  - Yes. → continue
  - No. → go to Step 1-Q1 or there is no DP solution for this problem

Step 1: Characterize an OPT Solution

Rod Cutting Problem
Input: a rod of length n and a table of prices p_i for $i = 1, \dots, n$
Output: the maximum revenue r_n obtainable

- Step 1-Q1: What can be the subproblems?
- Subproblems: `Cut-Rod(0)`, `Cut-Rod(1)`, …, `Cut-Rod(n-1)`
  - `Cut-Rod(i)`: rod cutting problem with length-i rod
  - Goal: `Cut-Rod(n)`
- Suppose we know the optimal solution to `Cut-Rod(i)`, there are i cases:
  - Case 1: the first segment in the solution has length 1
    從solution中拿掉一段長度為1的鐵條, 剩下的部分是`Cut-Rod(i-1)`的最佳解
  - Case 2: the first segment in the solution has length 2
    從solution中拿掉一段長度為2的鐵條, 剩下的部分是`Cut-Rod(i-2)`的最佳解

    :
  - Case i: the first segment in the solution has length i
    從solution中拿掉一段長度為i的鐵條, 剩下的部分是`Cut-Rod(0)`的最佳解

Step 1: Characterize an OPT Solution

Rod Cutting Problem
Input: a rod of length n and a table of prices p_i for $i = 1, \dots, n$
Output: the maximum revenue r_n obtainable

- Step 1-Q2: Does it exhibit optimal structure? (an optimal solution can be represented by the optimal solutions to subproblems)
- Yes. Prove by contradiction.

Step 2: Recursively Define the Value of an OPT Solution

Rod Cutting Problem
Input: a rod of length n and a table of prices p_i for $i = 1, \ldots, n$
Output: the maximum revenue r_n obtainable

- Suppose we know the optimal solution to `Cut-Rod(i)`, there are i cases:
  - Case 1: the first segment in the solution has length 1
    從solution中拿掉一段長度為1的鐵條, 剩下的部分是`Cut-Rod(i-1)`的最佳解 $r_i = p_1 + r_{i-1}$
  - Case 2: the first segment in the solution has length 2
    從solution中拿掉一段長度為2的鐵條, 剩下的部分是`Cut-Rod(i-2)`的最佳解 $r_i = p_2 + r_{i-2}$

    :
  - Case i: the first segment in the solution has length i
    從solution中拿掉一段長度為i的鐵條, 剩下的部分是`Cut-Rod(0)`的最佳解 $r_i = p_i + r_0$
- Recursively define the value

$$r_i = \begin{cases} 0 & \text{if } i = 0 \\ \max_{1 \le j \le i} (p_j + r_{i-j}) & \text{if } i \ge 1 \end{cases}$$

Step 3: Compute Value of an OPT Solution

Rod Cutting Problem
Input: a rod of length n and a table of prices p_i for $i = 1, \dots, n$
Output: the maximum revenue r_n obtainable

- Bottom-up method: solve smaller subproblems first

$$r_i = \begin{cases} 0 & \text{if } i = 0 \\ \max_{1 \le j \le i} \left(p_j + r_{i-j} \right) & \text{if } i \ge 1 \end{cases}$$

i	0	1	2	3	4	5	…	n
r[i]								

```
Bottom-Up-Cut-Rod(p, n)
  r[0] = 0
  for j = 1 to n // compute r[1], r[2], ... in order
    q = -∞
    for i = 1 to j
      q = max(q, p[i] + r[j - i])
    r[j] = q
  return r[n]
```

$$T(n) = \Theta(n^2)$$

Step 4: Construct an OPT Solution by Backtracking

length i	1	2	3	4	5
price p_i	1	5	8	9	10

Rod Cutting Problem
Input: a rod of length n and a table of prices p_i for $i = 1, \ldots, n$
Output: the maximum revenue r_n obtainable

- Bottom-up method: solve smaller subproblems first

$$r_i = \begin{cases} 0 & \text{if } i = 0 \\ \max_{1 \le j \le i} (p_j + r_{i-j}) & \text{if } i \ge 1 \end{cases}$$

i	0	1	2	3	4	5	…	n
r[i]	0	1	5	8	10			
cut[i]	0	1	2	3	2			

$\max(p_1 + r_0)$

$\max(p_1 + r_1, p_2 + r_0)$

$\max(p_1 + r_2, p_2 + r_1, p_3 + r_0)$

$\max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4 + r_0)$

Step 4: Construct an OPT Solution by Backtracking

```
Cut-Rod(p, n)
  r[0] = 0
  for j = 1 to n // compute r[1], r[2], ... in order
  q = -∞
    for i = 1 to j
      if q < p[i] + r[j - i]
        q = p[i] + r[j - i]
        cut[j] = i // the best first cut for len j rod
    r[i] = q
  return r[n], cut
```

$$T(n) = \Theta(n^2)$$

```
Print-Cut-Rod-Solution(p, n)
  (r, cut) = Cut-Rod(p, n)
  while n > 0
    print cut[n]
    n = n - cut[n] // remove the first piece
```

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

38 DP#2: Stamp Problem

Stamp Problem

- Input: the postage n and the stamps with values $v_1, v_2, \dots, v_k$

- Output: the minimum number of stamps to cover the postage

- The optimal solution S_n can be recursively defined as

A Recursive Algorithm

- The optimal solution S_n can be recursively defined as $1 + \min_i(S_{n-v_i})$

$$1 + \min(S_{n-3}, S_{n-5}, S_{n-7}, S_{n-12})$$

```
Stamp(v, n)
  r_min = ∞
  if n == 0 // base case
    return 0
  for i = 1 to k // recursive case
    r[i] = Stamp(v, n - v[i])
    if r[i] < r_min
      r_min = r[i]
  return r_min + 1
```

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

Step 1: Characterize an OPT Solution

Stamp Problem
Input: the postage n and the stamps with values $v_1, v_2, \ldots, v_k$
Output: the minimum number of stamps to cover the postage

- Subproblems
  - `S(i)`: the min #stamps with postage i
  - Goal: `S(n)`
- Optimal substructure: suppose we know the optimal solution to `S(i)`, there are k cases:
  - Case 1: there is a stamp with v_1 in OPT
    從solution中拿掉一張郵資為v_1的郵票, 剩下的部分是`S(i-v[1])`的最佳解
  - Case 2: there is a stamp with v_2 in OPT
    從solution中拿掉一張郵資為v_2的郵票, 剩下的部分是`S(i-v[2])`的最佳解

    :
  - Case k: there is a stamp with v_k in OPT
    從solution中拿掉一張郵資為v_k的郵票, 剩下的部分是`S(i-v[k])`的最佳解

Step 2: Recursively Define the Value of an OPT Solution

Stamp Problem
Input: the postage n and the stamps with values $v_1, v_2, \dots, v_k$
Output: the minimum number of stamps to cover the postage

- Suppose we know the optimal solution to `S(i)`, there are k cases:
  - Case 1: there is a stamp with v_1 in OPT
    從solution中拿掉一張郵資為v_1的郵票, 剩下的部分是`S(i-v[1])`的最佳解
    $S_i = 1 + S_{i-v_1}$
  - Case 2: there is a stamp with v_2 in OPT
    從solution中拿掉一張郵資為v_2的郵票, 剩下的部分是`S(i-v[2])`的最佳解
    $S_i = 1 + S_{i-v_2}$

    :
  - Case k: there is a stamp with v_k in OPT
    從solution中拿掉一張郵資為v_k的郵票, 剩下的部分是`S(i-v[k])`的最佳解
    $S_i = 1 + S_{i-v_k}$
- Recursively define the value

$$S_i = \begin{cases} 0 & \text{if } i = 0 \\ \min_{1 \le j \le k} \left(1 + S_{i-v_j}\right) & \text{if } i \ge 1 \end{cases}$$

Step 3: Compute Value of an OPT Solution

Stamp Problem
Input: the postage n and the stamps with values $v_1, v_2, \dots, v_k$
Output: the minimum number of stamps to cover the postage

- Bottom-up method: solve smaller subproblems first

$$S_i = \begin{cases} 0 & \text{if } i = 0 \\ \min_{1 \le j \le k} \left(1 + S_{i - v_j}\right) & \text{if } i \ge 1 \end{cases}$$

i	0	1	2	3	4	5	…	n
S[i]								

```
Stamp(v, n)
  S[0] = 0
  for i = 1 to n // compute r[1], r[2], ... in order
    r_min = ∞
    for j = 1 to k
      if S[i - v[j]] < r_min
        r_min = 1 + S[i - v[j]]
    S[i] = r_min
  return S[n]
```

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

Step 4: Construct an OPT Solution by Backtracking

```
Stamp(v, n)
  S[0] = 0
  for i = 1 to n
    r_min = ∞
    for j = 1 to k
      if S[i - v[j]] < r_min
        r_min = 1 + S[i - v[j]]
        B[i] = j // backtracking for stamp with v[j]
    S[i] = r_min
  return S[n], B
```

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

```
Print-Stamp-Selection(v, n)
  (S, B) = Stamp(v, n)
  while n > 0
    print B[n]
    n = n - v[B[n]]
```

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

45

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 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

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

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\\$\{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(w_i > w)
        M[i, w] = M[i-1, w]
      else
        M[i, w] = max(v_i + M[i-1, w-w_i], 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 - w_i
      S = S ∪ {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 $\log_2 W$

$= 2^m \qquad = m$

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

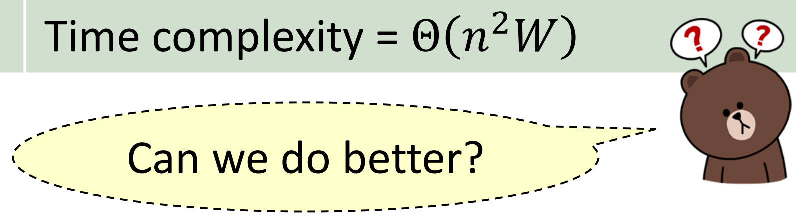
- Variants of knapsack problem
  - 0-1 Knapsack Problem: 每項物品只能拿一個
  - **Unbounded Knapsack Problem: 每項物品可以拿多個**
  - Multidimensional Knapsack Problem: 背包空間有限
  - Multiple-Choice Knapsack Problem: 每一類物品最多拿一個
  - Fractional Knapsack Problem: 物品可以只拿部分

Step 1: Characterize an OPT Solution

Unbounded Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i, each 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 = $\Theta(n^2W)$



Step 1: Characterize an OPT Solution

Unbounded Knapsack Problem
Input: n items where i-th item has value v_i and weighs w_i, each 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 - w1)`
  - Case 2: item 2 in OPT
    - Removing an item 2 from OPT is an optimal solution of `U-KP(w - w2)`

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

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, each 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 - w₁)`

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

- 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, each 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, each has **unlimited supplies**
Output: the max value within W capacity

- Bottom-up method: solve smaller subproblems first

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

i	w_i	v_i
1	1	4
2	2	9
3	4	17

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

$W = 5$

$$\max(4+0)$$
$$\max(4+4, 9+0)$$
$$\max(4+9, 9+4)$$
$$\max(4+13, 9+9, 17+0)$$
$$\max(4+18, 9+13, 17+4)$$

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, each 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(w_i <= w)
        tmp = v_i + M[w - w_i]
        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 $\boldsymbol{D}$**, 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\\$\{i\}$ is an optimal solution of `M-KP(i - 1, w - wi, d - di)`
  - 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 - w_i, d - d_i)`

    $$M_{i,w,d} = v_i + M_{i-1,w-w_i,d-d_i}$$
  - 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?

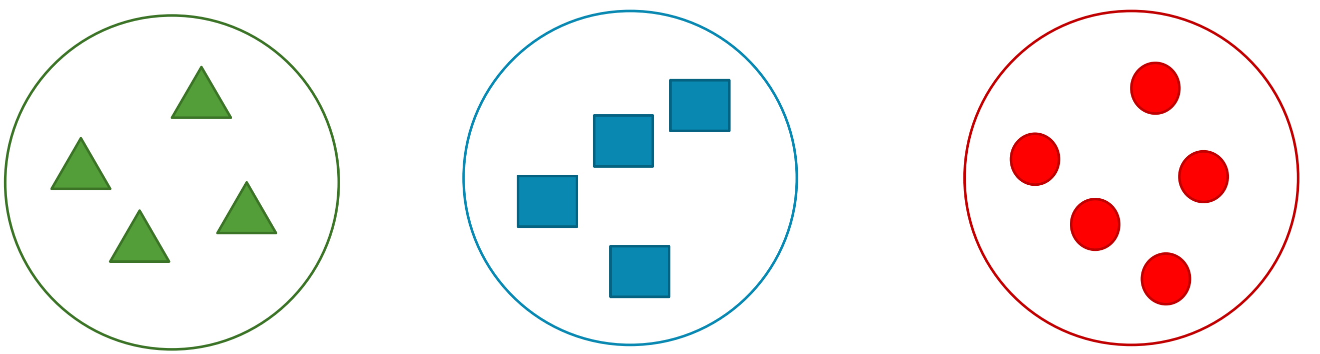
Knapsack Problem

- 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**



group 1 group 2 group 3

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

the constraint is for 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 ($\text{item}_{i,j}$) in OPT
    - OPT\\$\text{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 $\quad M_{i,w} = M_{i-1,w}$
    - 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 $\quad M_{i,w} = v_{i,j} + M_{i-1,w-w_{i,j}}$
    - OPT\item$_{i,j}$ is an optimal solution of `MC-KP(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!

77

To Be Continued…

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