Dynamic Programming

Algorithm Design and Analysis
Dynamic Programming (1)

http://ada.miulab.tw   slido: #ADA2021

Yun-Nung (Vivian) Chen

國立臺灣大學 National Taiwan University

Outline

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

Algorithm Design Strategy

- Do not focus on "specific algorithms"
- But "some strategies" to "design" algorithms

- First Skill: Divide-and-Conquer (各個擊破/分治法)
- Second Skill: Dynamic Programming (動態規劃)

Dynamic Programming

Textbook Chapter 15 – Dynamic Programming
Textbook Chapter 15.3 – Elements of dynamic programming

What is Dynamic Programming?

- Dynamic programming, like the divide-and-conquer method, solves problems by <u>combining the solutions to subproblems</u>
  - 用空間換取時間
  - 讓走過的留下痕跡
- “Dynamic”: time-varying
- “Programming”: a *tabular* method

Dynamic Programming: planning over time

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

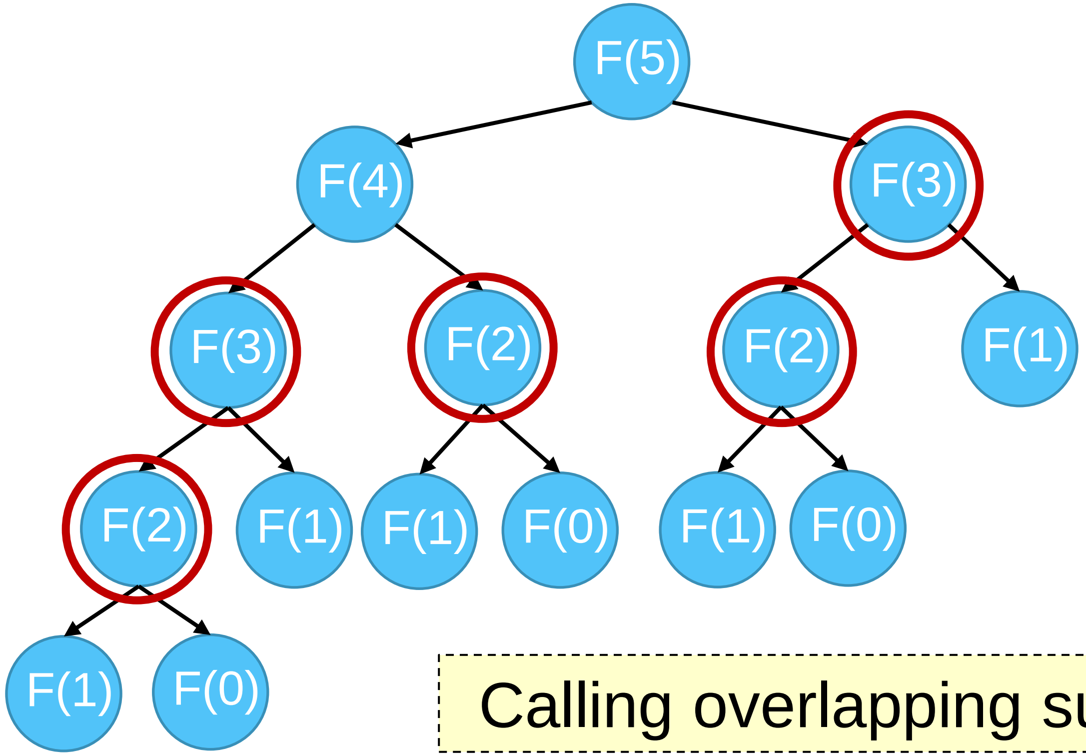
Dynamic Programming Procedure

- Apply four steps
  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

Rethink Fibonacci Sequence

- Fibonacci sequence (費波那契數列)
  - Base case: F(0) = F(1) = 1
  - Recursive case: F(n) = F(n-1) + F(n-2)

```
Fibonacci(n)
  if n < 2 // base case
    return 1
  // recursive case
  return Fibonacci(n-1)+Fibonacci(n-2)
```



✓F(3) was computed twice

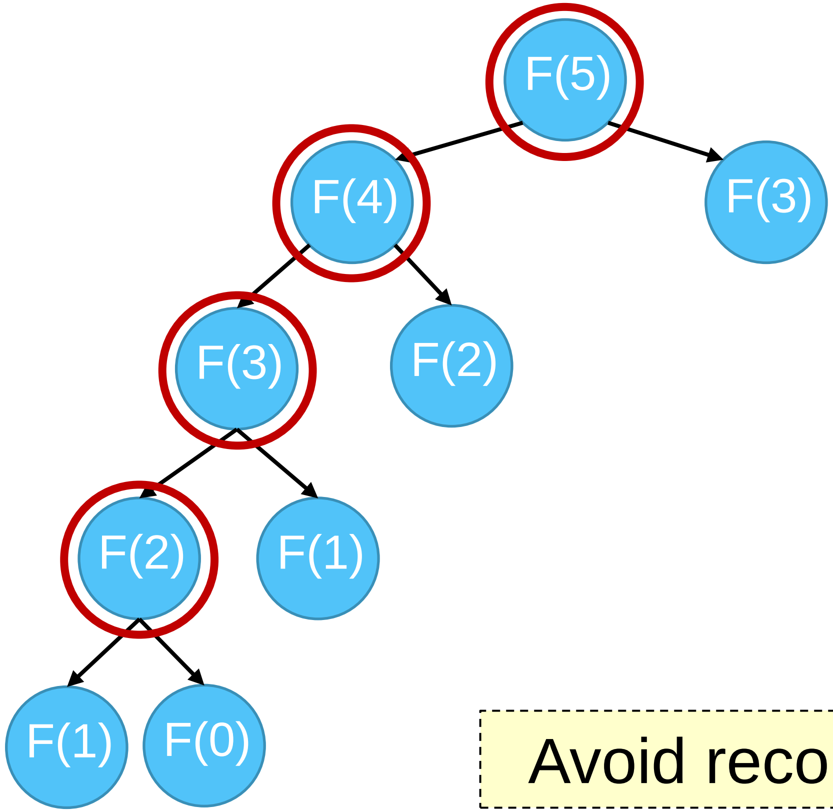
✓F(2) was computed 3 times

$$T(n) = O(2^n)$$

Calling overlapping subproblems result in poor efficiency

Fibonacci Sequence
Top-Down with Memoization

- Solve the overlapping subproblems recursively with memoization
  - Check the memo before making the calls



n	0	1	2	3	4	5
F(n)	1	1	2	3	5	8

Avoid recomputation of the same subproblems using memo

Fibonacci Sequence

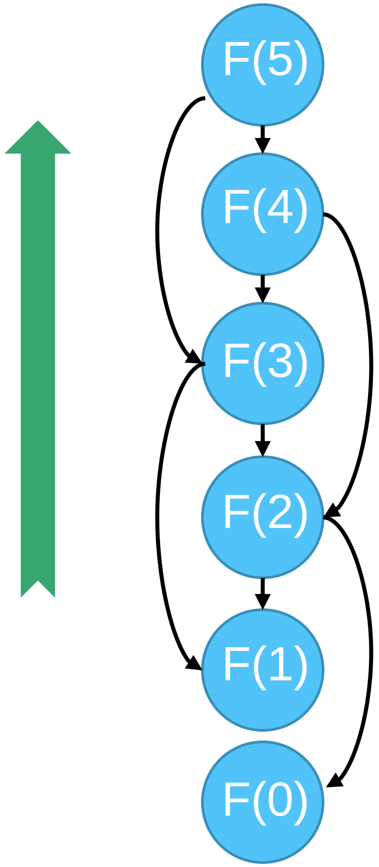
Top-Down with Memoization

```
Memoized-Fibonacci(n)
  // initialize memo (array a[])
  a[0] = 1
  a[1] = 1
  for i = 2 to n
    a[i] = 0
  return Memoized-Fibonacci-Aux(n, a)

Memoized-Fibonacci-Aux(n, a)
  if a[n] > 0
    return a[n]
  // save the result to avoid recomputation
  a[n] = Memoized-Fibonacci-Aux(n-1, a) + Memoized-Fibonacci-Aux(n-2, a)
  return a[n]
```

Fibonacci Sequence
Bottom-Up Method

- Building up solutions to larger and larger subproblems




```
Bottom-Up-Fibonacci(n)
  if n < 2
    return 1
  a[0] = 1
  a[1] = 1
  for i = 2 … n
    a[i] = a[i-1] + a[i-2]
  return a[n]
```

Avoid recomputation of the same subproblems

Optimization Problem

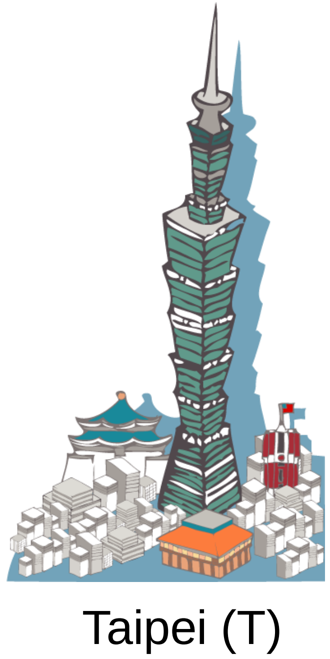
- Principle of Optimality
  - Any subpolicy of an optimum policy must itself be an optimum policy with regard to the initial and terminal states of the subpolicy
- Two key properties of DP for optimization
  - Overlapping subproblems
  - Optimal substructure – an optimal solution can be constructed from optimal solutions to subproblems
    - ✓ Reduce search space (ignore non-optimal solutions)

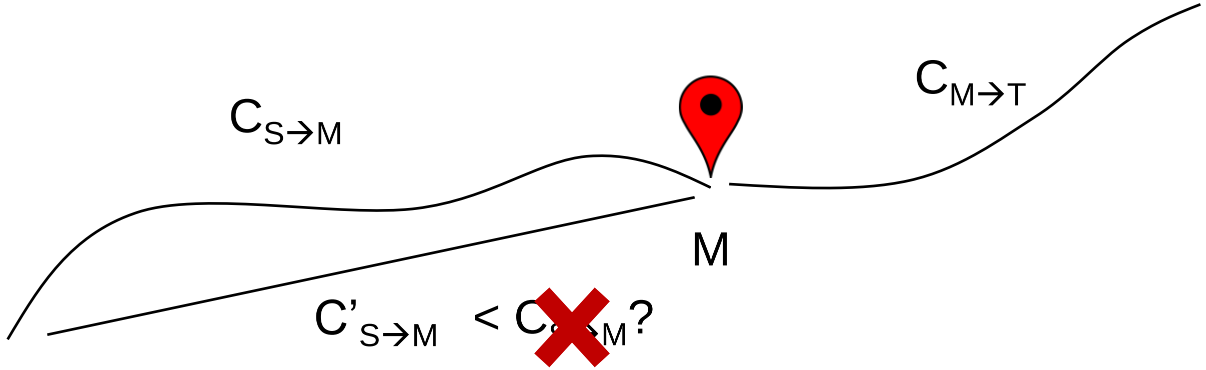
If the optimal substructure (principle of optimality) does not hold, then it is incorrect to use DP

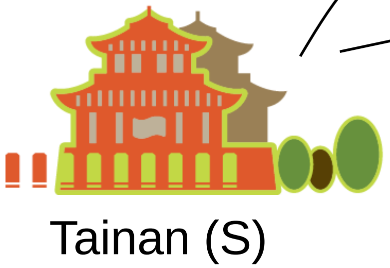
Optimal Substructure Example

- Shortest Path Problem
  - Input: a graph where the edges have positive costs
  - Output: a path from S to T with the smallest cost

The path costing $C_{S \to M} + C_{M \to T}$ is the shortest path from S to T
→ The path with the cost $C_{S \to M}$ must be a shortest path from S to M







Proof by "Cut-and-Paste" argument (proof by contradiction):
Suppose that it exists a path with smaller cost $C'_{S \to M}$, then we can "cut" $C_{S \to M}$ and "paste" $C'_{S \to M}$ to make the original cost smaller

DP#1: Rod Cutting

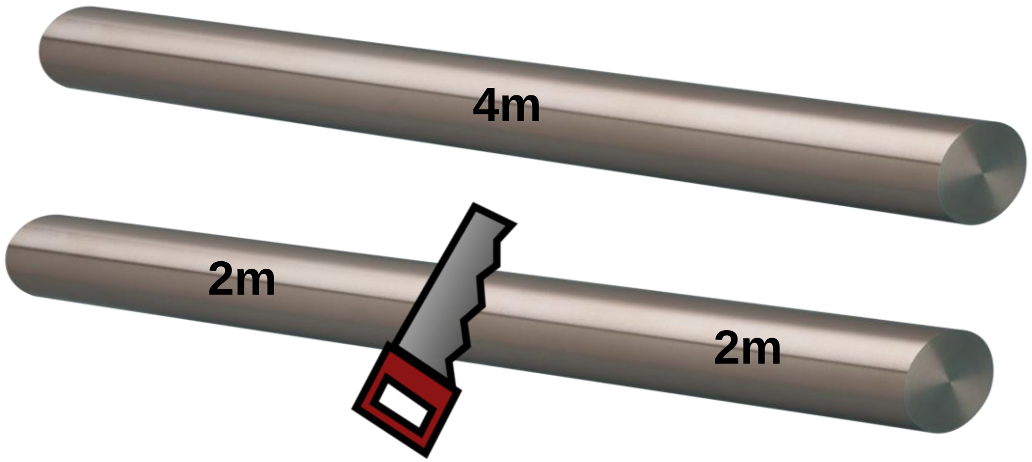
Textbook Chapter 15.1 – Rod Cutting

Rod Cutting Problem

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

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

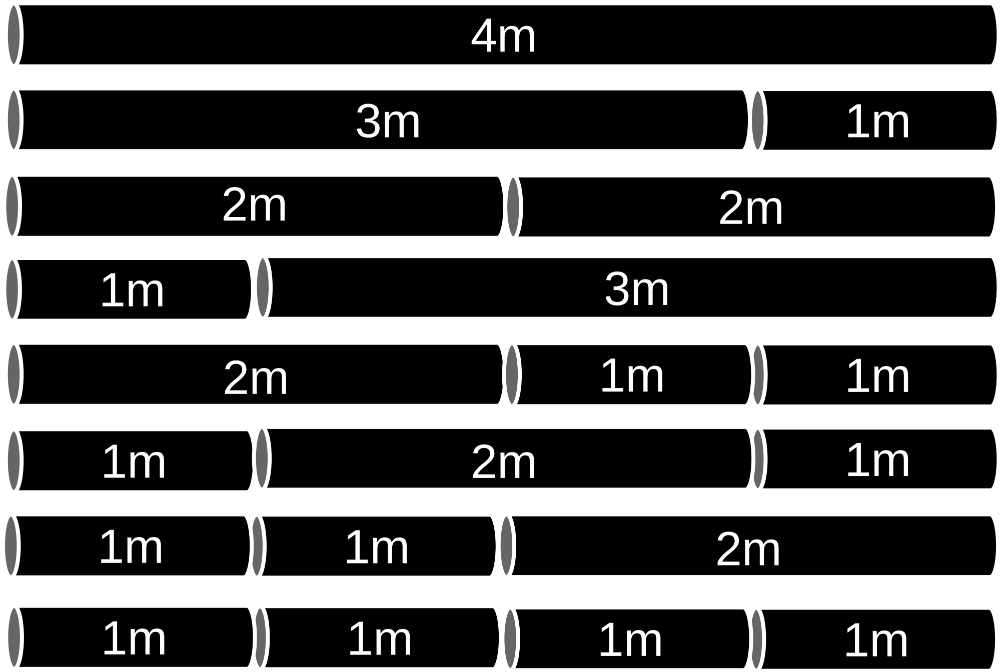
- Output: the maximum revenue r_n obtainable by cutting up the rod and selling the pieces



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

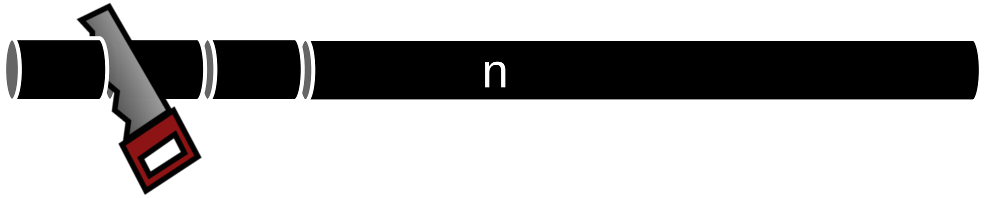
→ 1 + 1 + 5 = 7

→ 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 $2^{n-1} = \Theta\left(2^{n-1}\right)$

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?

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

- 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

$$r_n = \max_{1 \le i \le n} (\underbrace{p_i}_{\text{left-most value}} + \underbrace{r_{n-i}}_{\text{maximum value obtainable from the remaining part}})$$

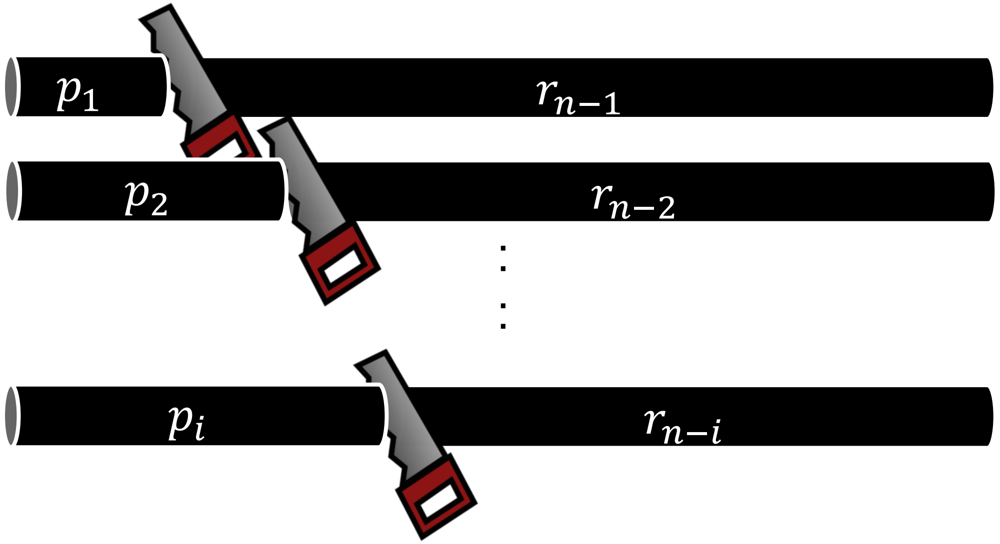
Recursive Procedure

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

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

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
```

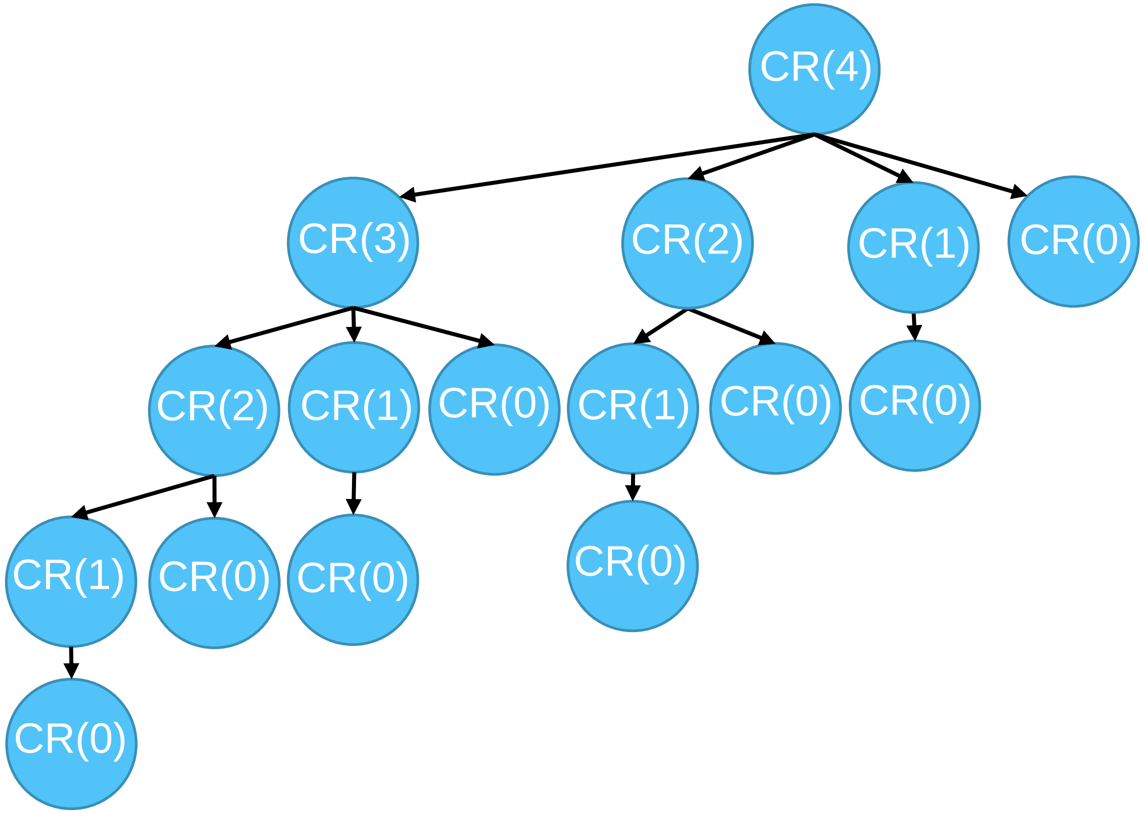
- $T(n)$ = time for running `Cut-Rod(p, n)`

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

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

$\Theta(n)$

$\Theta(1)$

$\Theta(n^2)$

- $T(n)$ = 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]
```

$\Theta(n^2)$

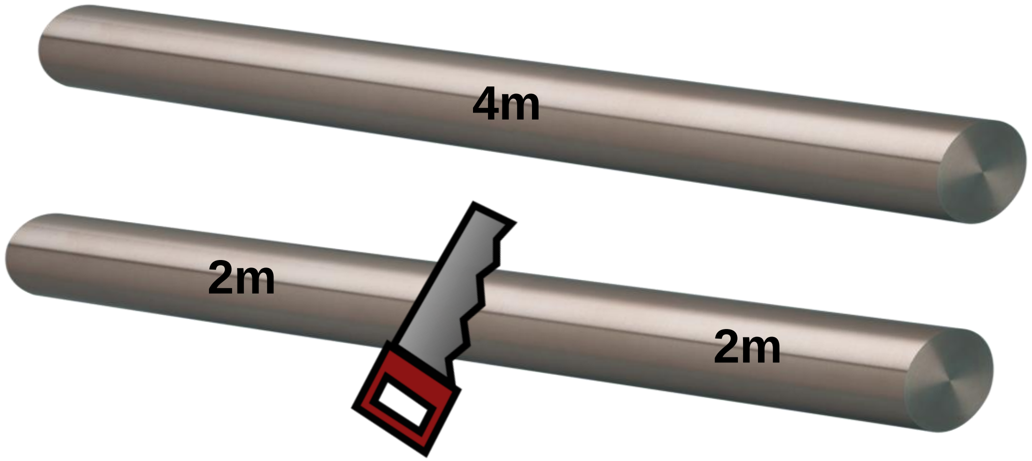
- $T(n)$ = time for running `Bottom-Up-Cut-Rod(p, n)` ➡ $T(n) = \Theta(n^2)$

Rod Cutting Problem

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

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
```

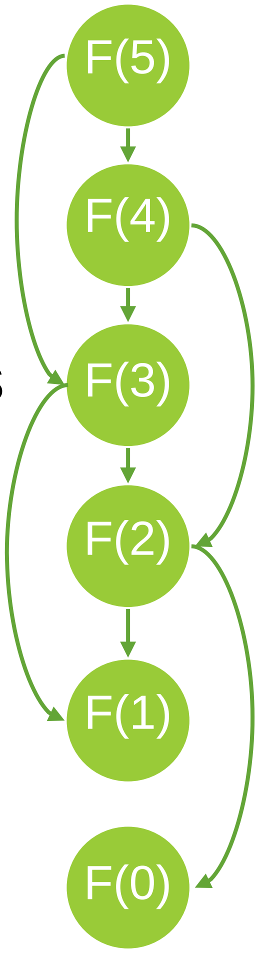
```
Print-Cut-Rod-Solution(p, n)
  (r, cut) = Extended-Bottom-up-Cut-Rod(p, n)
  while n > 0
    print cut[n]
    n = n - cut[n] // remove the first piece
```

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 <u>required</u> parts of subproblems



- Bottom-Up with Tabulation

f(0)	f(1)	f(2)	…	f(n)

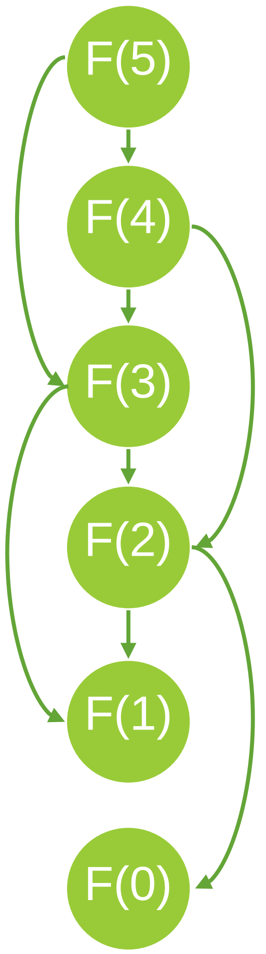
  - Better when all subproblems must be solved at least once
  - Typically outperform top-down method by a constant factor
    - No overhead for recursive calls
    - Less overhead for maintaining the table

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 $i = 1, \ldots, n$
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 $i = 1, \ldots, n$
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 $i = 1, \ldots, n$
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

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

i	0	1	2	3	4	5	…	n
r[i]								$\boxed{}$

```
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 $i = 1, \dots, n$
Output: the maximum revenue r_n obtainable

- Bottom-up method: solve smaller subproblems first

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

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
```

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

DP#2: Stamp Problem

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

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

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

Step 3: Compute Value of an OPT Solution

Stamp Problem
Input: the postage n and the stamps with values $v_1, v_2, \ldots, v_k$
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]]
```

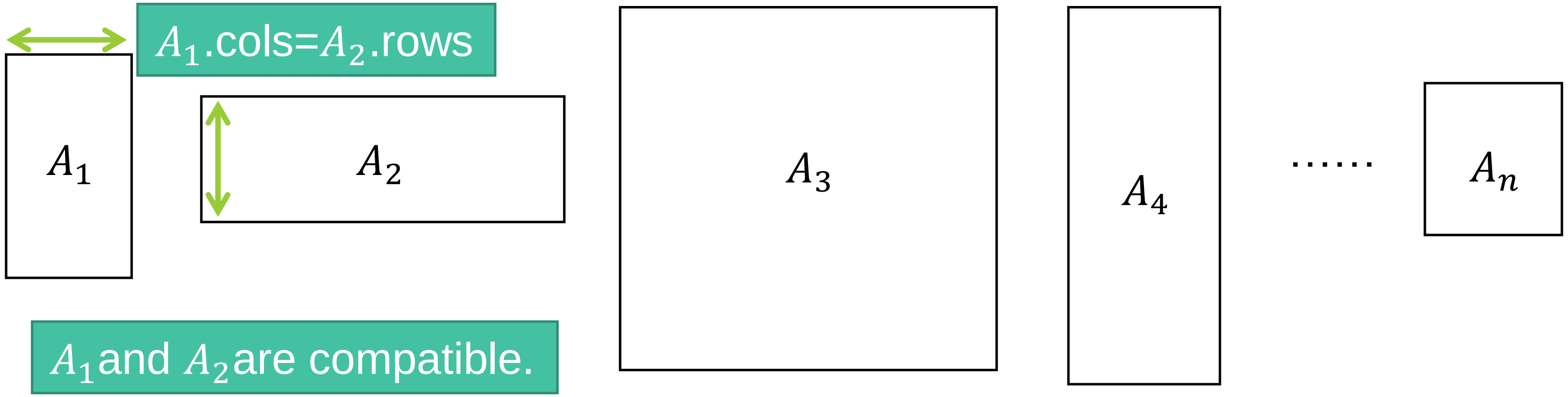
$$T(n) = \Theta(n)$$

DP#3: Matrix-Chain Multiplication

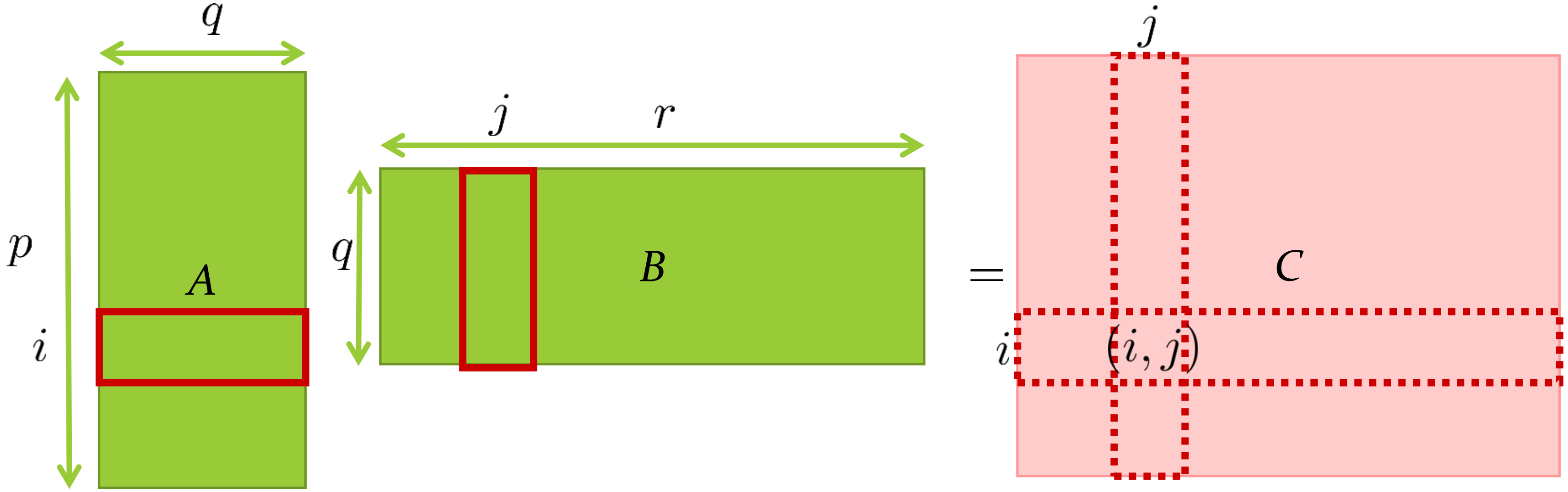
Textbook Chapter 15.2 – Matrix-chain multiplication

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



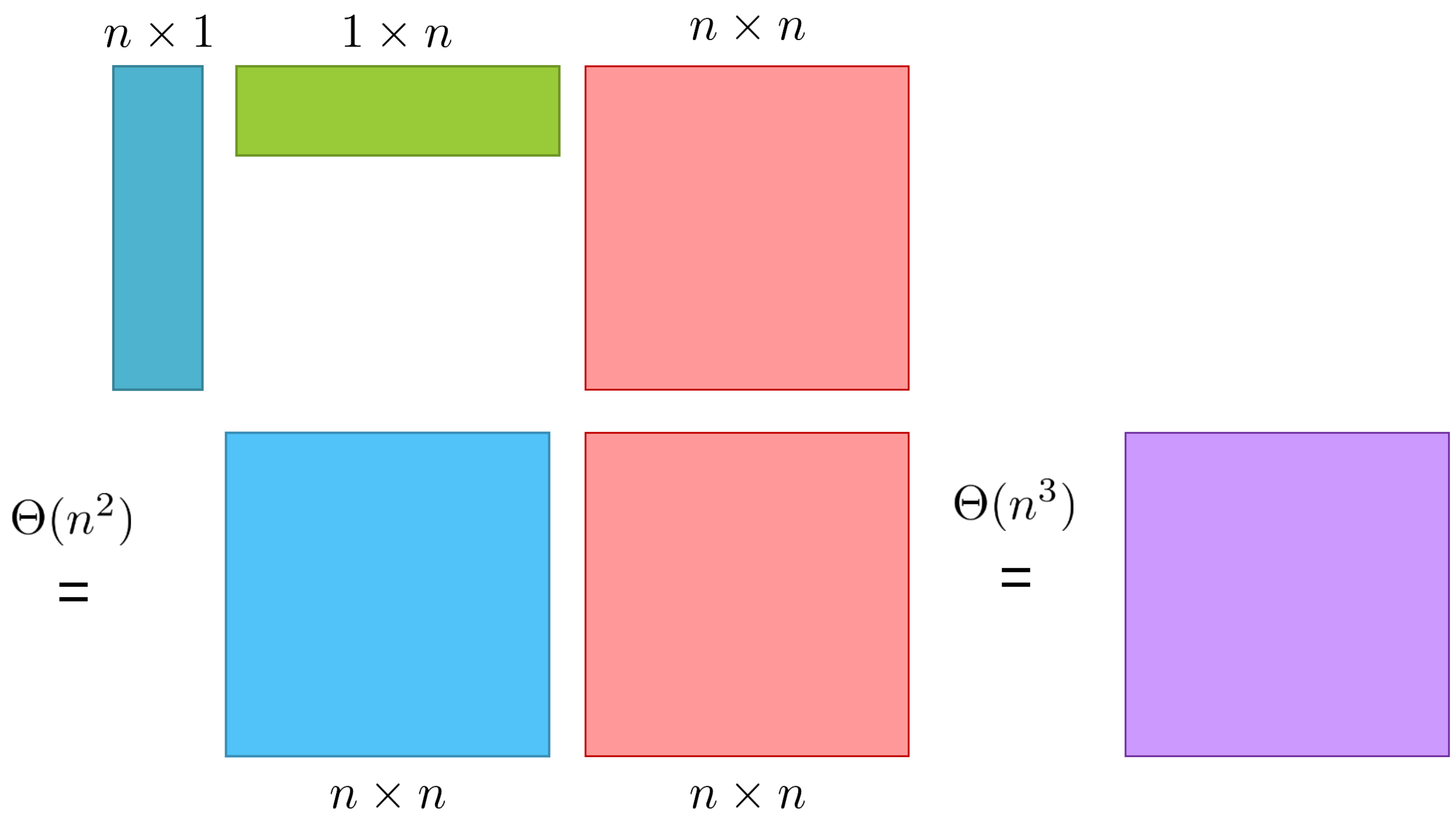
$$C(i,j) = \sum_{k=1}^{n} A(i,q) \cdot B(k,j)$$

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

➡ $\Theta(q)\Theta(pr) = \Theta(pqr)$

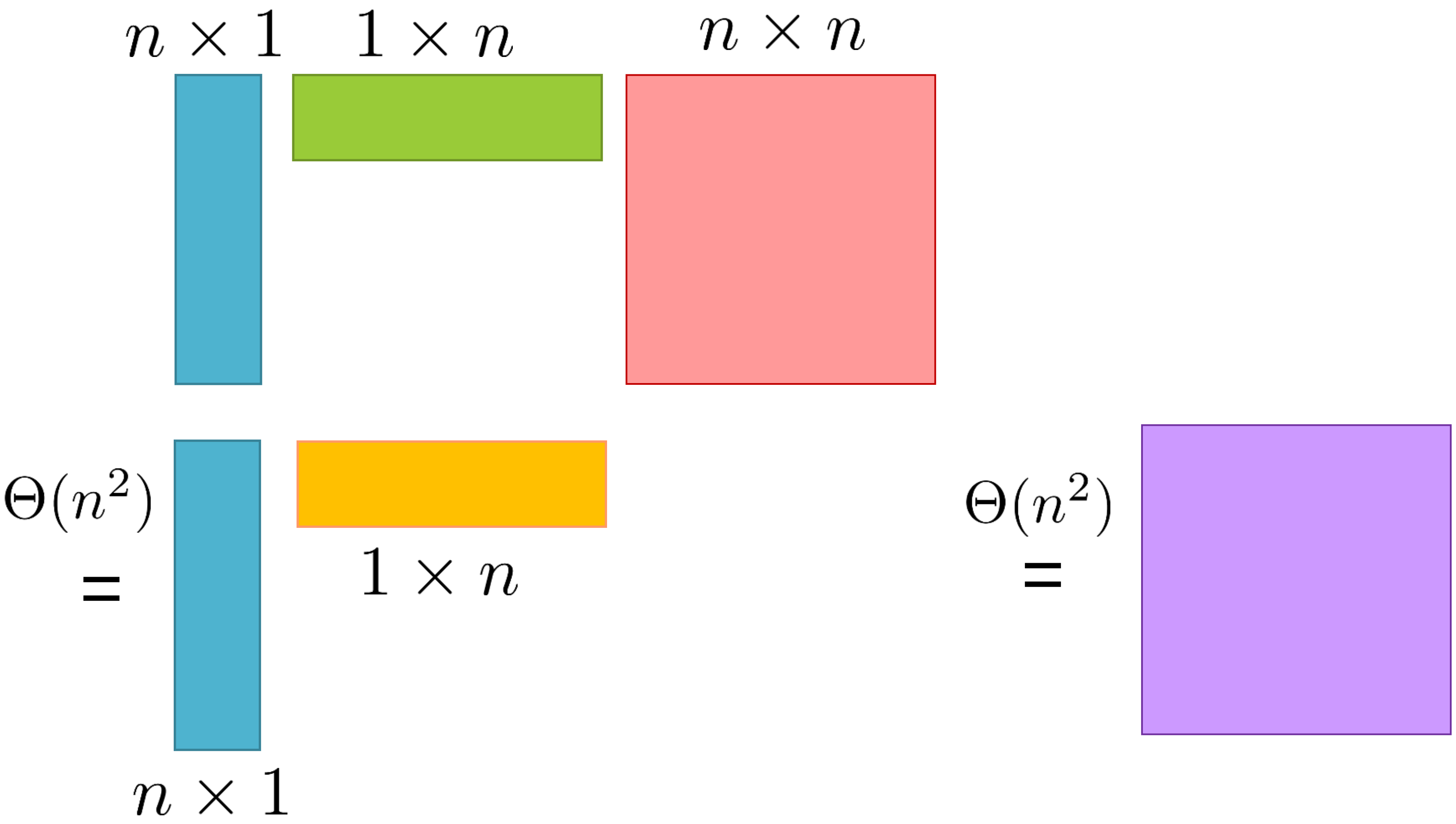
Matrix multiplication is associative: $A(BC) = (AB)C$. The time required by obtaining $A \times B \times C$ could be affected by which two matrices multiply first .

Example



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

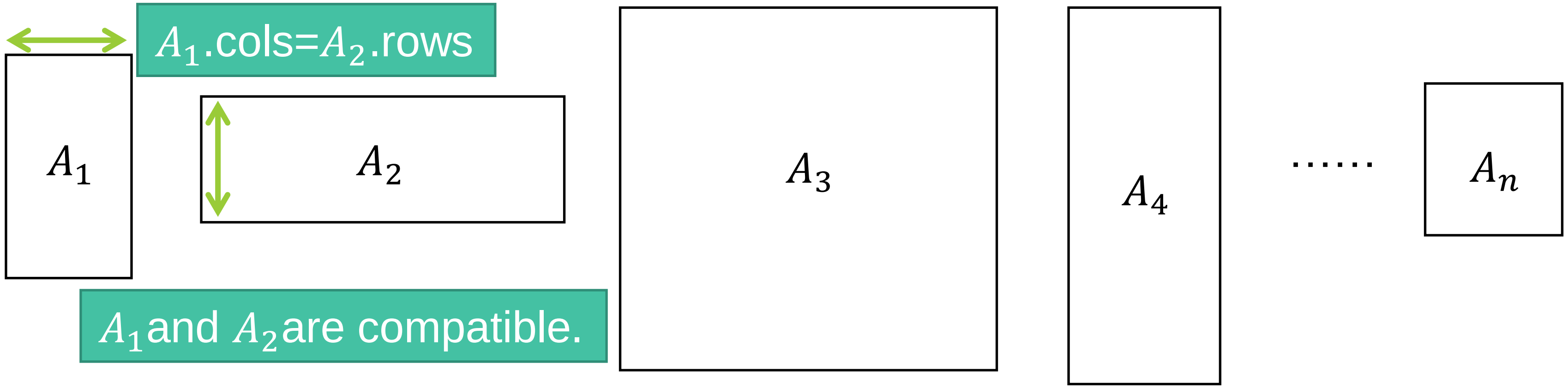
Example



- 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:

$$i \leq k < j$$

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

  - 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

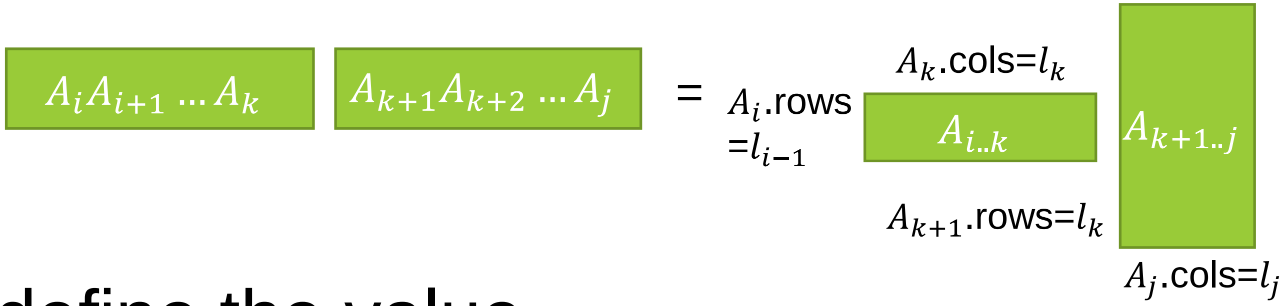
Step 2: Recursively Define the Value of an OPT Solution

Matrix-Chain Multiplication Problem
Input: a sequence of integers $l_0, l_1, \ldots, l_n$ 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

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

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



- 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 over the columns, i beside the 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						

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

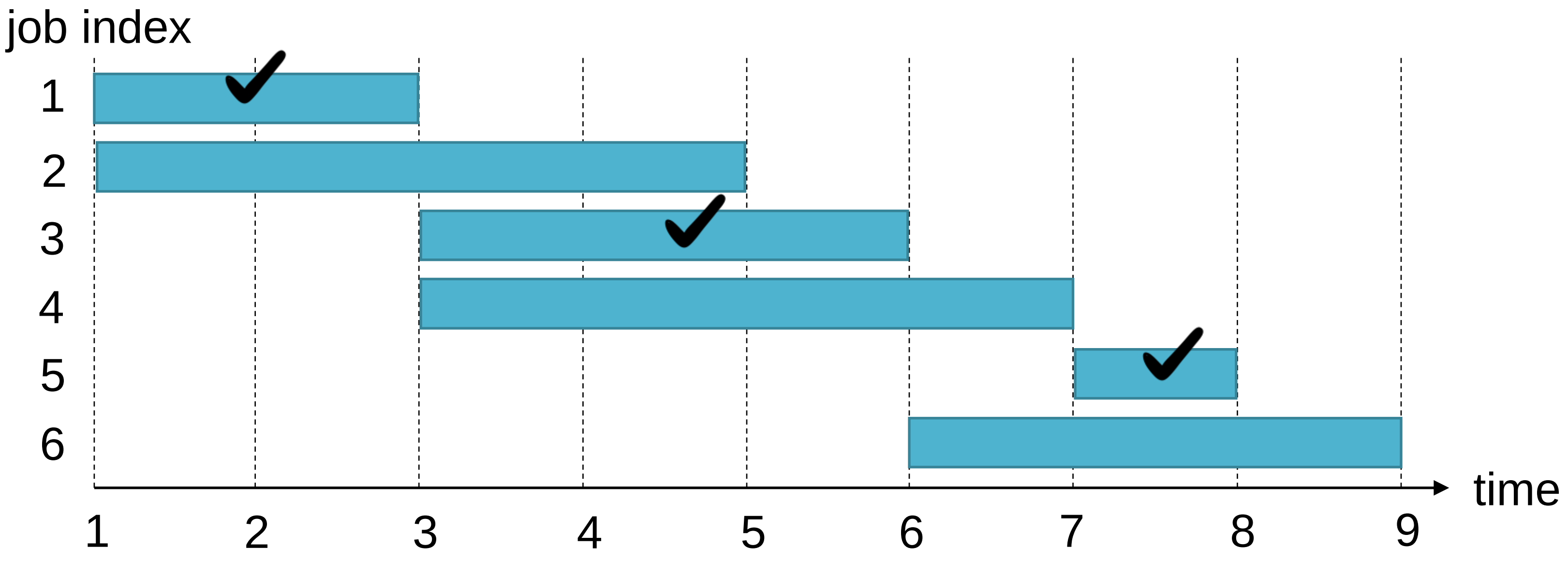
DP#4: Weighted Interval Scheduling

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

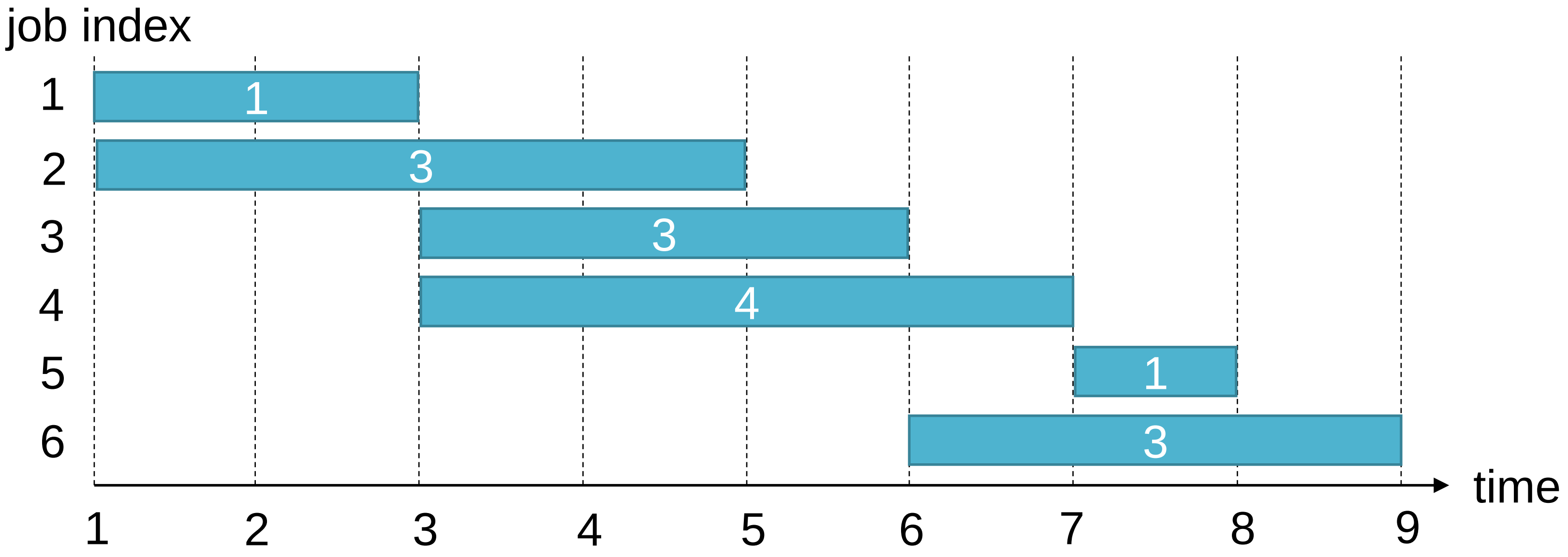
"Greedy Algorithm"
Next topic!



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

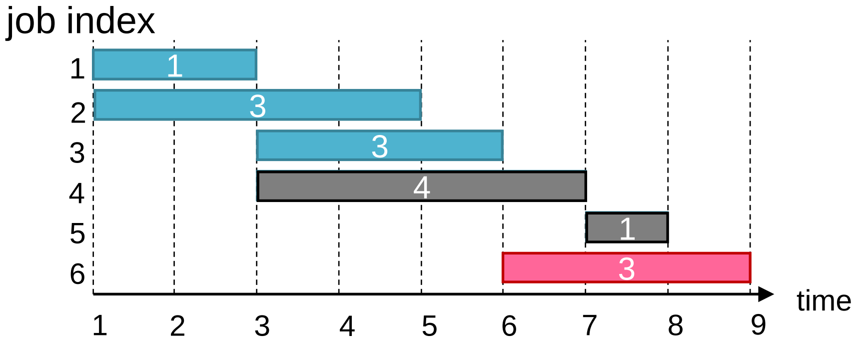
Assume that the requests are sorted in non-decreasing order ($f_i \leq f_j$ when $i < j$)
$p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
e.g. $p(1) = 0$, $p(2) = 0$, $p(3) = 1$, $p(4) = 1$, $p(5) = 4$, $p(6) = 3$



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

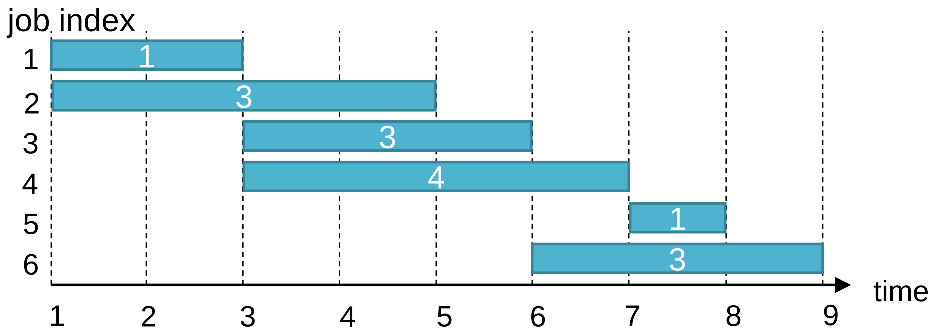
Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the maximum total value obtainable from compatible

- 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
```

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

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