Dynamic Programming - I

CSIE 2136 Algorithm Design and Analysis, Fall 2018

https://cool.ntu.edu.tw/courses/61

Hsu-Chun Hsiao

Announcement

Mini-hw3 due on 10/11 (Thu.) 14:20

Where to find TA hours

- https://www.csie.ntu.edu.tw/~hchsiao/courses/ada18.html

General guidelines on homework

暖身運動:
Hats with Consecutive Numbers

http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15251-s09/Site/Materials/Lectures/lecture01.pdf

Agenda

What is dynamic programming?

Examples

- Rod cutting problem (鐵條切割)
- Weighted interval scheduling (加權區間調度)
- Knapsack Problem (背包問題)
- Sequence Alignment Problem (序列比對)

Dynamic Programming (DP)

"Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. "

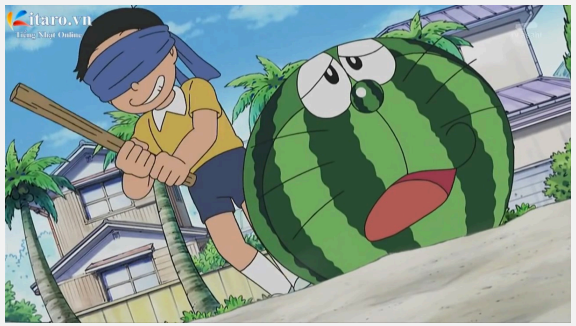
"In mathematics, computer science, economics, and bioinformatics, dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems." – Wikipedia

Algorithm design paradigms

Divide and Conquer (DC) **Dynamic Programming (DP)**

把一個問題分解成數個性質相同的小問題。

- Blindly recompute overlapping subproblems
- Work best when subproblems are **independent**, or **disjoint**



- Solve each subproblem at most once
- Applicable when subproblems are **dependent,** or **overlapping**
- Two equivalent ways to avoid recomputation
  - Top-down with memoization

    Memo=備忘錄
  - Bottom-up method

    按部就班 由小到大逐一解決

Why the name “dynamic programming”?

Richard Bellman coined this impressive yet distracting term

Bellman: “The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word ‘research’…”

硬是要解讀的話…

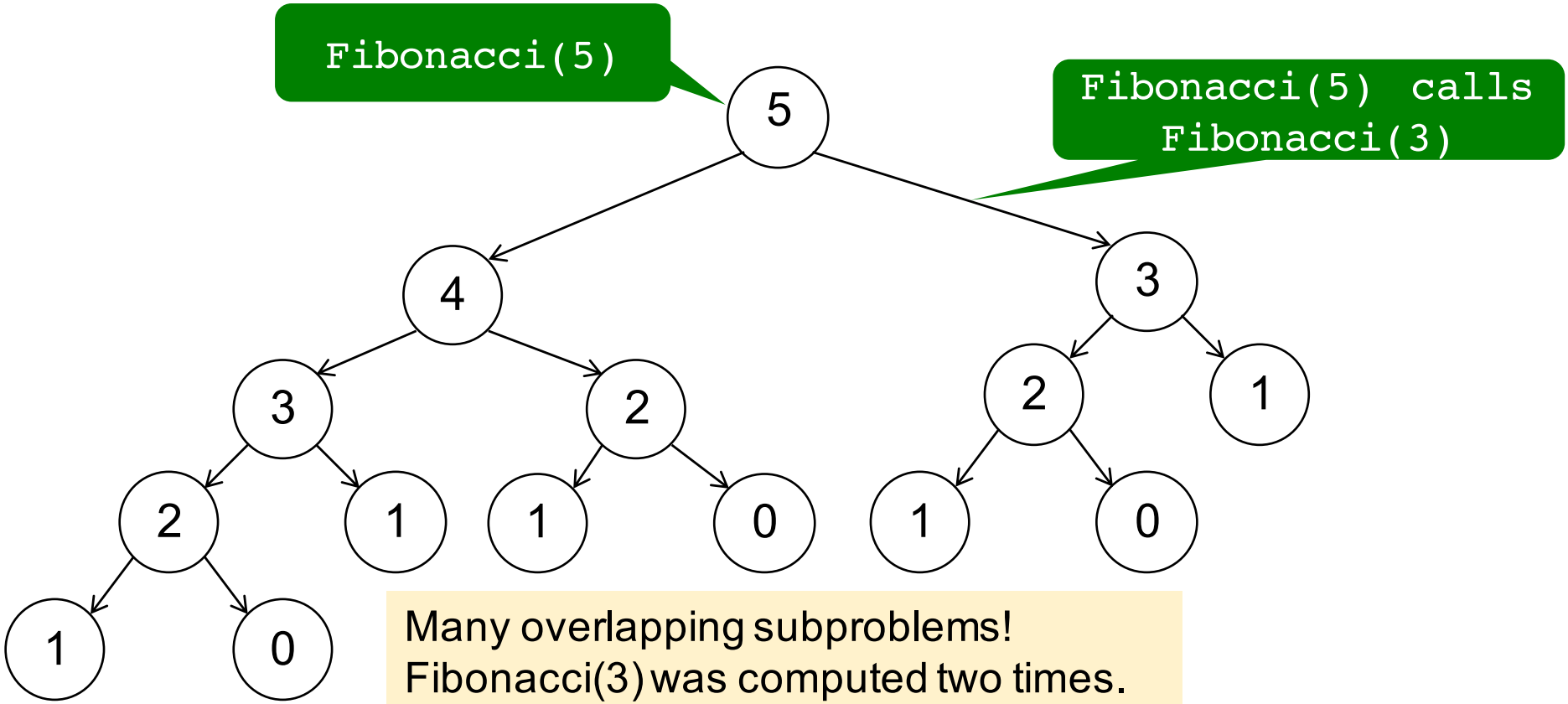
Dynamic = time-varying

Programming = “refers to a tabular method, not to writing computer code”

Dynamic programming = planning over time

還記得費波那契數列嗎

```
F(n):
    if n < 2
        return 1
    return F(n-1) + F(n-2)
```

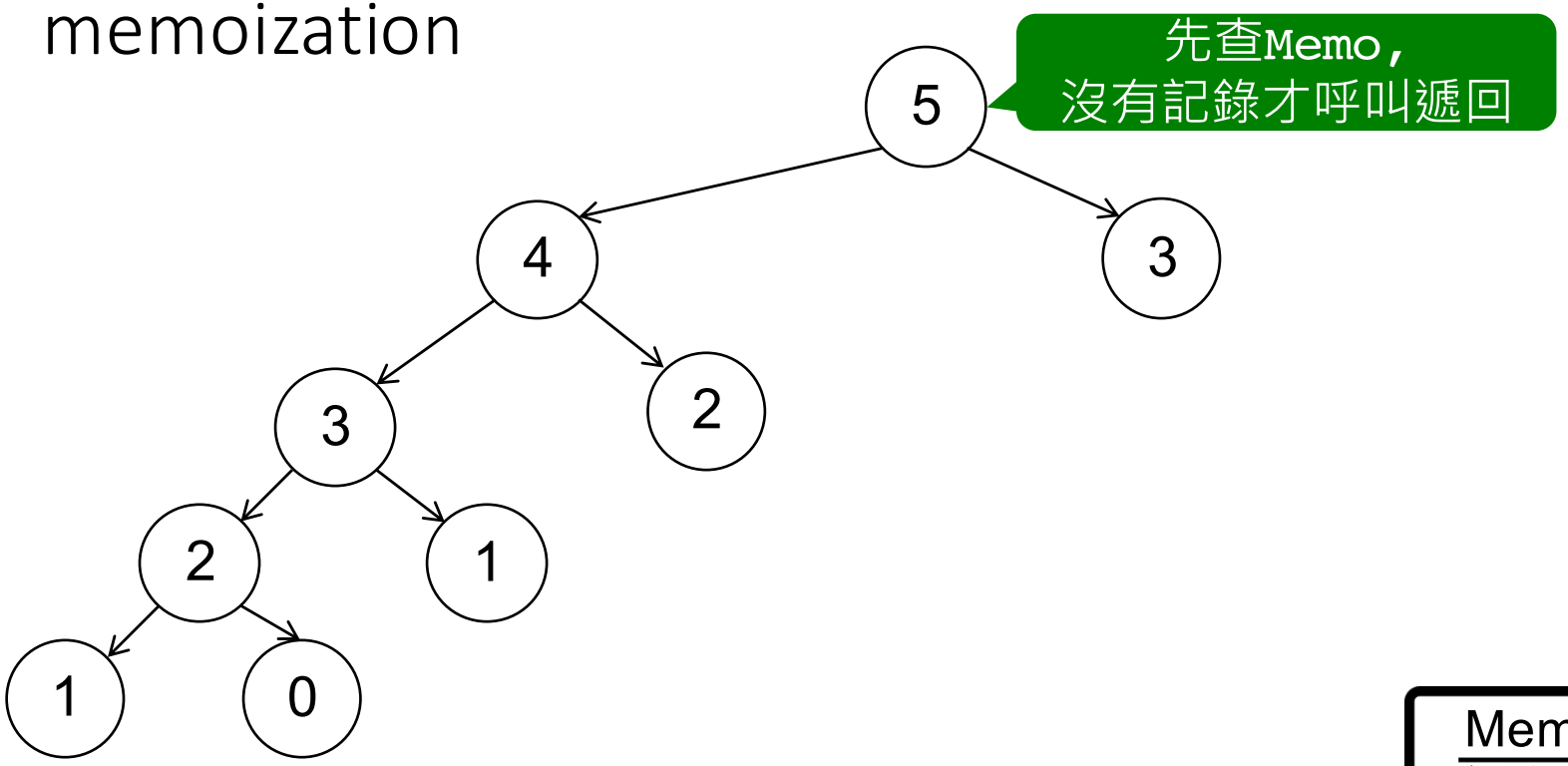


Many overlapping subproblems!
Fibonacci(3) was computed two times.
Fibonacci(2) was computed three times.

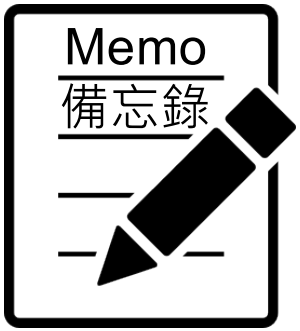
Fibonacci sequence

Top-down with memoization

Solve overlapping subproblems recursively with memoization

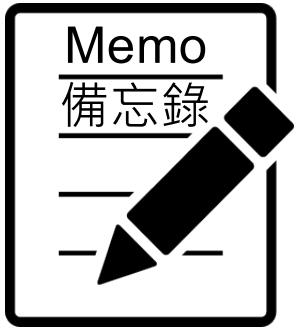


n	0	1	2	3	4	5
F(n)	1	1	2	3	5	8



Fibonacci sequence

Top-down with memoization



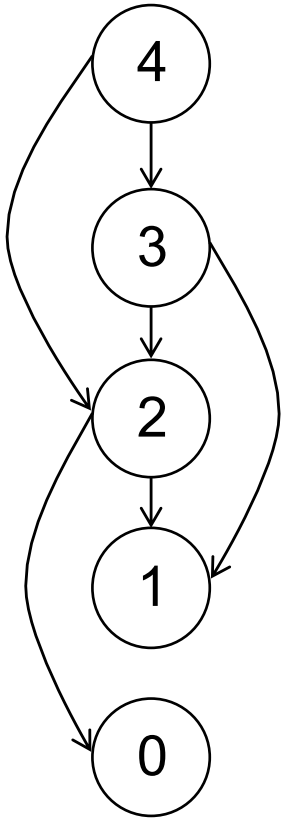

```
Memo-Fibonacci(n):
    //initialize memo (array a[])
    a[0] <- 1
    a[1] <- 1
    for i = 2 to n
        a[i] ← 0
    return Memo-Fibonacci-Aux(n,a)

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

Fibonacci sequence

Bottom-up method

Build up solutions to larger and larger subproblems
(按部就班　由小到大逐一解決)




```
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]
```

DP and optimization problems

Dynamic programming are often applied to solving optimization problems (最佳化問題)

- 從問題的多個解之中，選出 “最佳”的
- 最佳的解可能有很多個，找出一個就好了

Examples of optimization problems

- 從兩個字串中，找出最長的共同子字串
- 給一個背包和一堆物品，找出背包最多能裝多少物品
- …

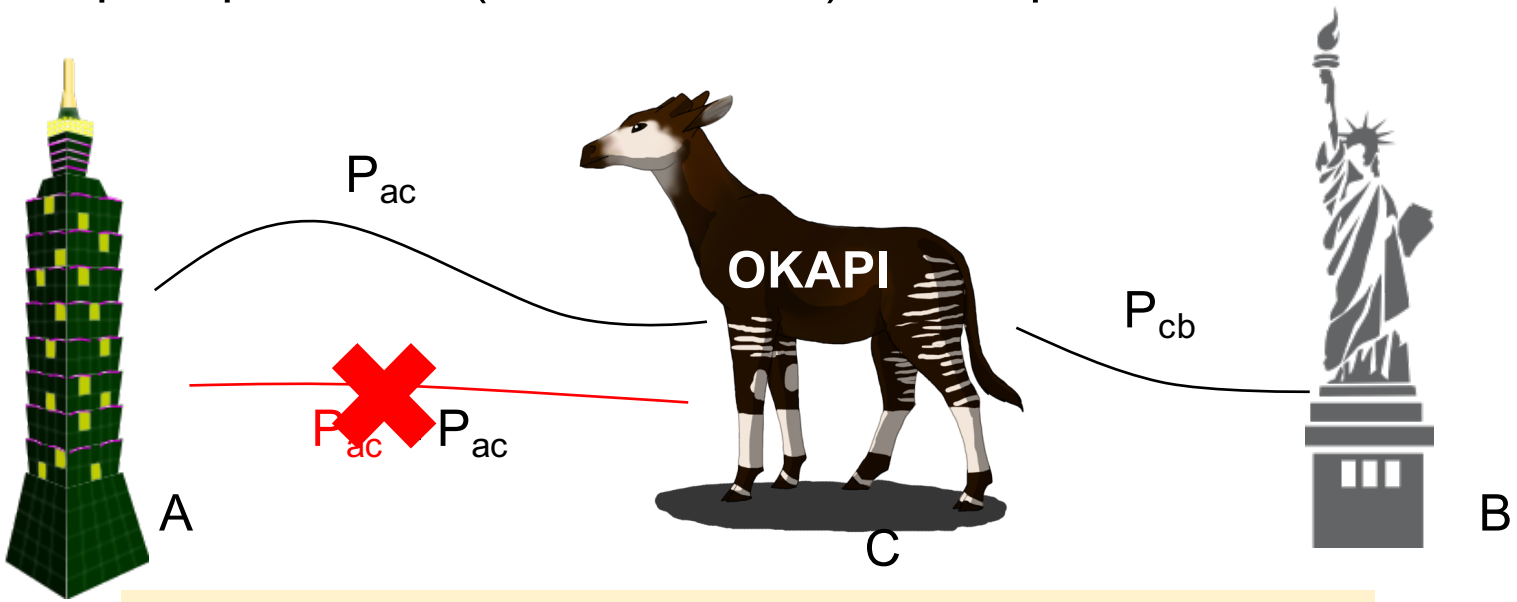
DP and optimization problems

To apply DP, an optimization problem must exhibit two key properties:

- **Overlapping subproblems**
- **Optimal substructure** – an optimal solution can be constructed from optimal solutions to subproblems
  - Reduce search space, as we don't need to consider non-optimal solutions to a subproblem

Optimal substructure

Shortest path problems (最短路徑問題) have optimal substructure



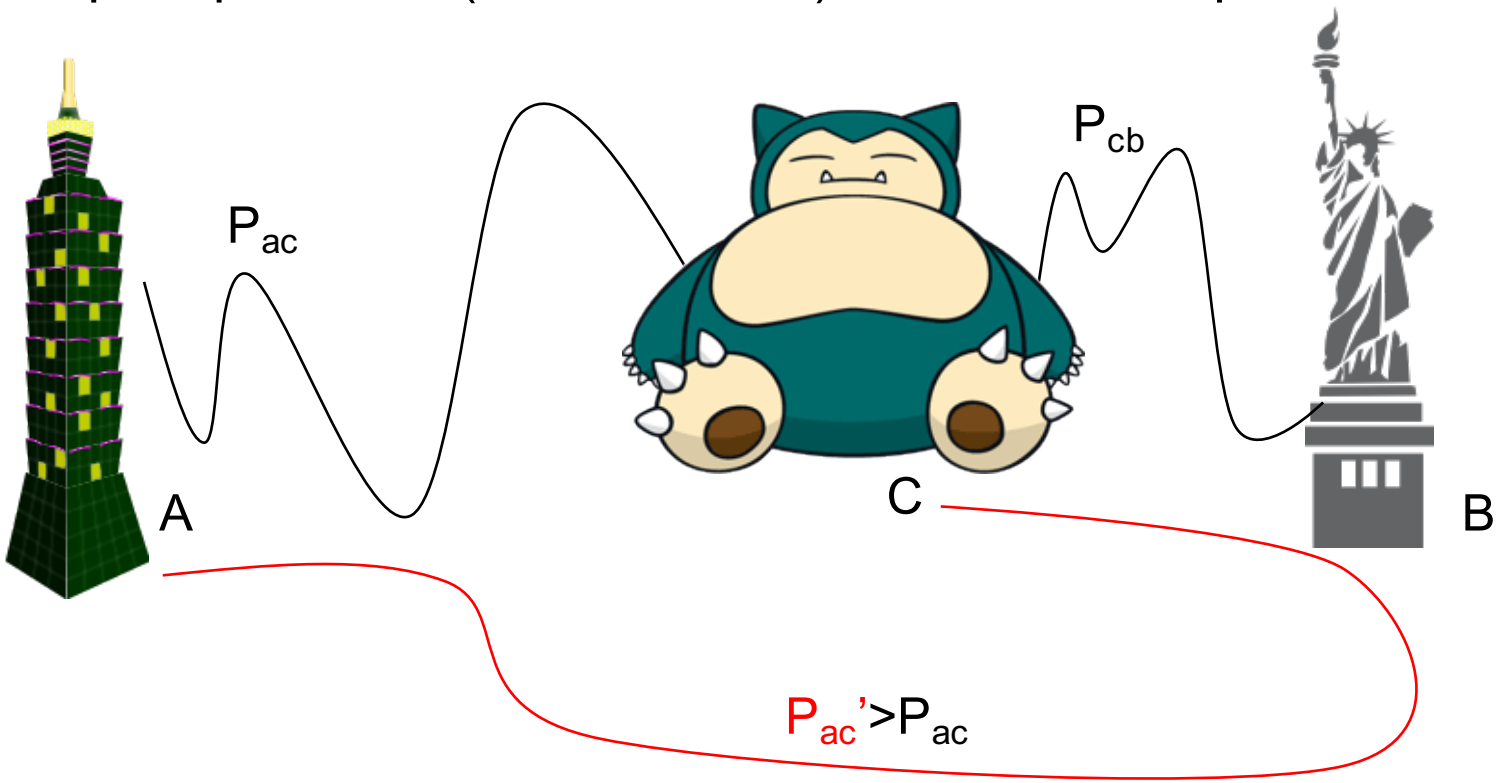
Path P_{ac}+P_{cb} is a **shortest** path between A and B
$\Rightarrow$ Then P_{ac} must be a shortest path between A and C

Proof by **"Cut-and-Paste" argument** (a sort of proof by contradiction):

Suppose there exists another path P_{ac}' that is shorter than P_{ac}
We can "cut" P_{ac} and "paste" P_{ac}' to form a shorter path P_{ac}' +P_{cb} between A and B, which contradicts that P_{ac}+P_{cb} is a shortest path between A and B.

Optimal substructure

Longest path problems (最長路徑問題) **do not have** optimal substructure



Path P_{ac}+P_{cb} is a **longest loop-free** path between A and B
$\Rightarrow$ P_{ac} may not be a longest loop-free path between A and C

Rod cutting
(鐵條切割問題)

鐵條切割廠

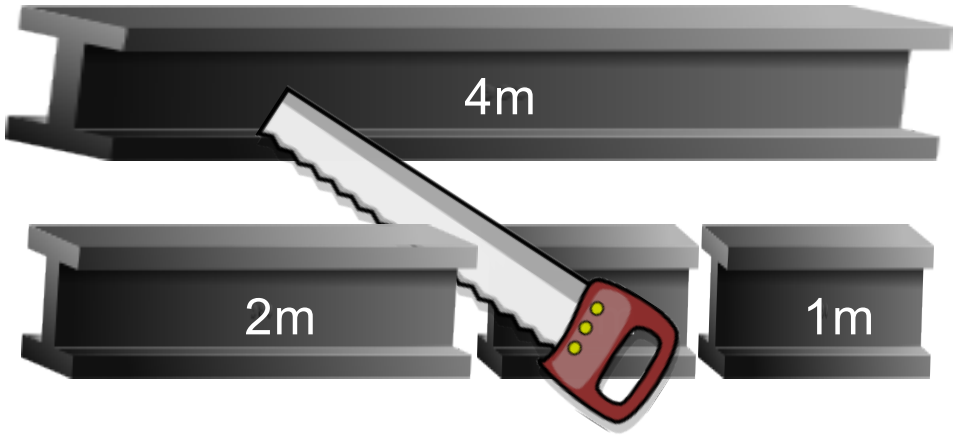
笑話顧問最近回到老家經營鐵條切割廠，該工廠購買長度為4公尺的鐵條，希望能切割成小鐵條後售出

假設切割的成本可忽略，要如何切割才能最大化利潤？

目前的市場價格如下：

Length (m)	1	2	3	4
Price (k)	1	5	8	9

E.g.,



要如何切總賣價才最高?

Length (m)	1	2	3	4
Price (k)	1	5	8	9

暴力解法：考慮所有$2^{4-1}=8$種切法

切法	售價
4	9
1, 3	1 + 8 = 9
2, 2	5 + 5 = 10
3, 1	8 + 1 = 9
1, 1, 2	1 + 1 + 5 = 7
1, 2, 1	1 + 5 + 1 = 7
2, 1, 1	5 + 1 + 1 = 7
1, 1, 1, 1	1 + 1 + 1 + 1 = 4

切成2m, 2m賺最多

But brute-force = $O(2^{n-1})$ ☹

Rod cutting problem
鐵條切割問題

Given a rod of length `n`

Given a price table `p`, p_i = price of length-i rod

Determine the maximum revenue `M[n]` obtainable by cutting up the rod and selling the pieces

Design an algorithm `CR(p,n)` to discover an optimal solution and its value `M[n]`

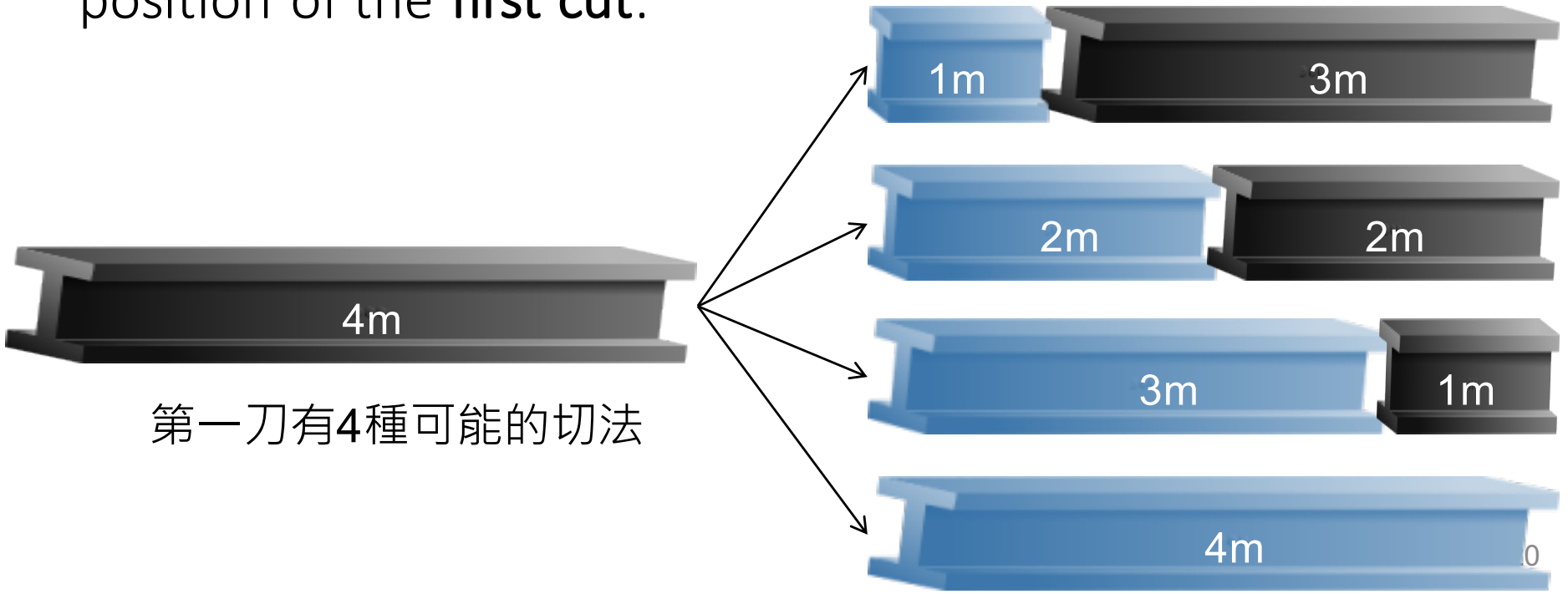
- For simplicity, let's use `CR(n)` instead of `CR(p,n)`

Recursive procedure: n choices for a cut

M[i] = the value of an optimal solution to CR(i)

- Let's define M[n] using recursive calls
- n = the length of the rod
- p = price array $p_1, p_2, \ldots, p_n$ (p_i = price of length-i rod)

Suppose we always cut **from left to right**. Consider the position of the **first cut**:



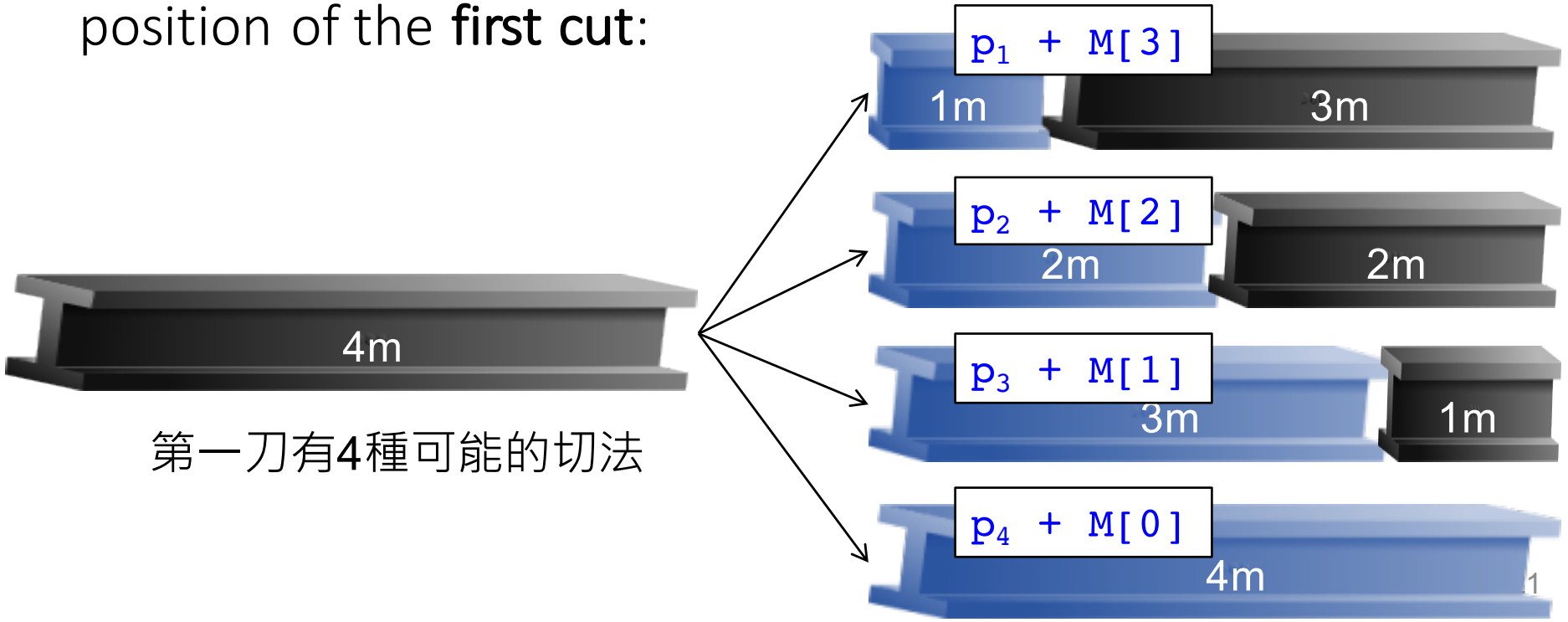
第一刀有4種可能的切法

Recursive procedure

M[i] = the value of an optimal solution to CR(i)

- Let's define M[n] using recursive calls
- n = the length of the rod
- p = price array $p_1, p_2, \ldots, p_n$ (p_i = price of length-i rod)

Suppose we always cut **from left to right**. Consider the position of the **first cut**:



第一刀有4種可能的切法

Recursive procedure

M[i] = the value of an optimal solution to CR(i)

- Let's define M[n] using recursive calls
- n = the length of the rod
- p = price array $p_1, p_2, \ldots, p_n$ (p_i = price of length-i rod)

Suppose we always cut **from left to right**. Consider the position of the **first cut**:

Suppose rod cutting problem has optimal substructure (we will prove this later):

$$M[4] = \max\{p_1 + M[3],\ p_2 + M[2],\ p_3 + M[1],\ p_4 + M[0]\}$$

Generalization to n:

$$M[n] = \max_{1 \le i \le n}\{p_i + M[n-i]\}$$

Optimal solution — Optimal solutions to subproblems

Naïve recursion

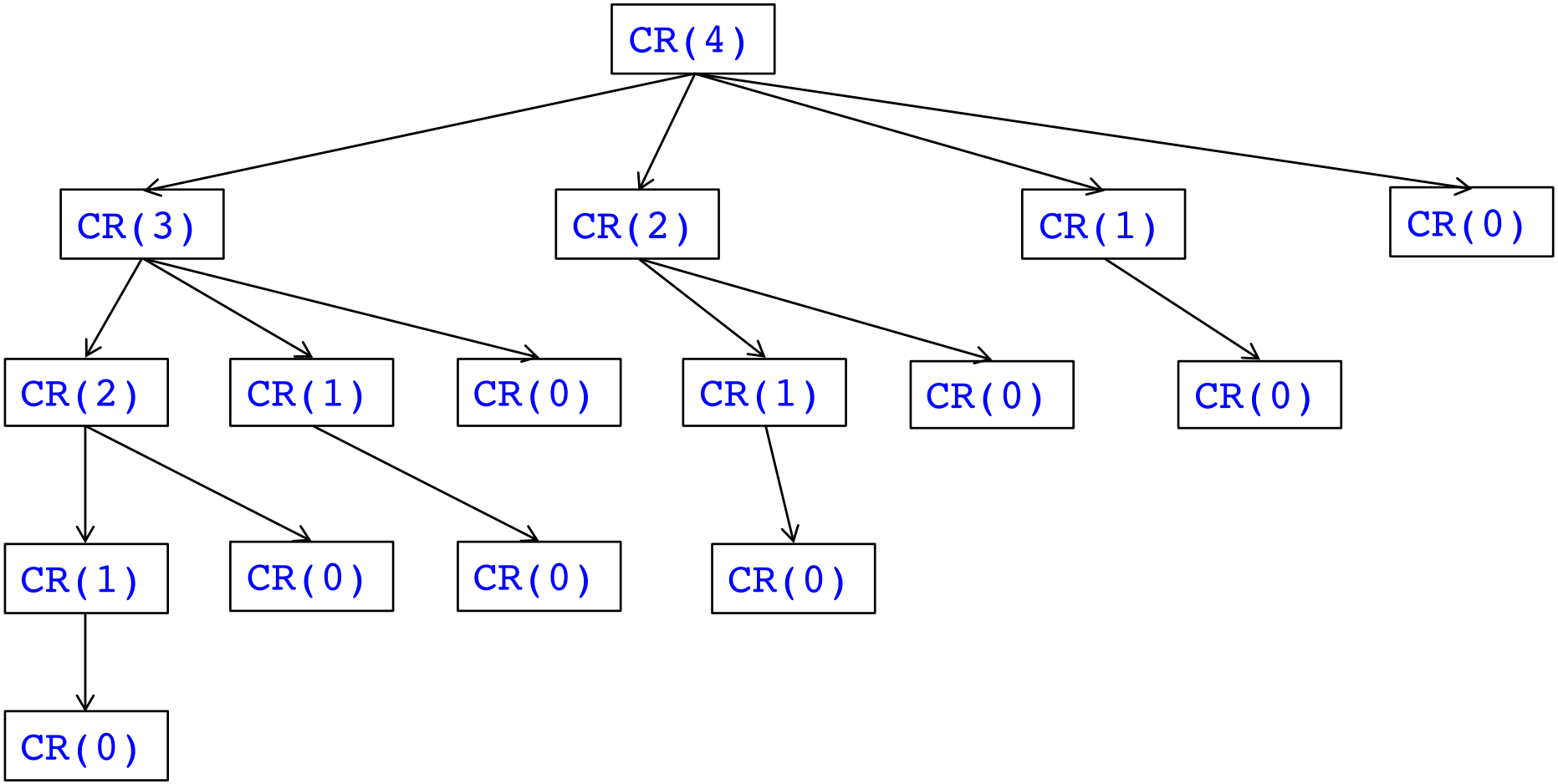
M[i] = the value of an optimal solution to CR(i)

$M[n] = \max_{1 \le i \le n}\{p_i + M[n-i]\}$

```
CR(n):
    //base case
    if n = 0
        return 0
    //recursive case
    q ← -∞
    For i = 1 to n
        q ← max(q, p_i + CR(n-i))
    return q
```

Running time: $T(n) \approx \Sigma_i T(n-i)$ ➔ $T(n) = O(2^n)$

Rod cutting
Naïve recursion



Dynamic Programming

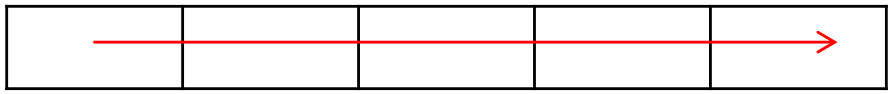
Rod cutting has **overlapping subproblems** and **optimal substructure** => can be solved by DP

Recall that there two equivalent ways to implement DP, both solve a subproblem at most once

- Top-down: solve overlapping subproblems recursively with memoization Memo=備忘錄
- Bottom-up: build up solutions to larger and larger subproblems 按部就班 由小到大逐一解決

Bottom-up with tabulation

按問題大小順序填表
（小問題要先解決）

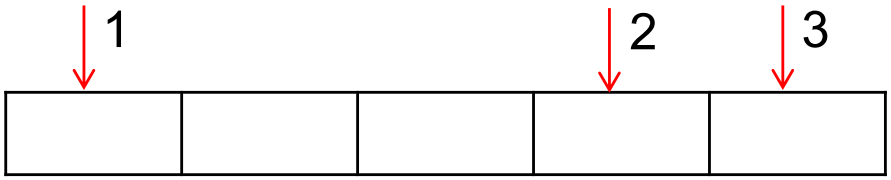


適合用於每個小問題都得解決的情況

Top-down with memoization

用遞迴解，把小問題的解答記在備忘錄裡

可看成是跳著填表



適合用於不需要解決所有的小問題的情況

Rod cutting

Top-down with memorization

```
Memorized-CR(n):
//initialize memo (an array M[] to keep max revenue)
    M[0] ← 0
    for i = 1 to n
        M[i] ← -∞ //M[i] = max revenue for length-i rod
    return Memorized-CR-Aux(n,M)

Memorized-CR-Aux(n,M):
    if M[n] ≥ 0
        return M[n] //return the saved solution
    q ← -∞
    for i = 1 to n
        q ← max(q, p_i + Memorized-CR-Aux(n-i,M))
    M[n] ← q //update memo
    return q
```

Running time: $T(n) = \Theta(n^2)$

Rod cutting

Bottom-up method

- M[j] = maximum revenue when cutting length-j rod
- Because of the natural ordering of the subproblems, we can solve subproblems of sizes j = 0, 1, 2, … in that order

```
Bottom-Up-CR(n):
M[0] ← 0
for j = 1 to n //compute M[1], M[2], … in order
    q ← -∞
    for i = 1 to j
        q ← max(q, p_i + M[j-i])
    M[j] ← q
return M[n]
```

Running time: $T(n) = \Theta(n^2)$

Top-down vs. bottom-up

Bottom-up method

- When all subproblems must be solved at least once 

- Typically outperforms top-down one by a constant factor
- No overhead for recursion and less overhead for maintaining the table

Top-down with memoization

- When some subproblems in the subproblem space need not be solved at all 

- Solving only those subproblems that are required

Reconstructing a solution

In addition to the maximum revenue, we also want to know **a list of piece sizes**

```
Extended-Bottom-Up-CR(n):
    M[0] ← 0
    for j = 1 to n //compute M[1], M[2], … in order
        q ← -∞
        for i = 1 to j
            if q < p_i + M[j-i]
                q ← p_i + M[j-i]
                cut[j] ← i // the best first cut for len j rod
        M[j] ← q
    return M[n], cut

Print-CR-Solution(n):
    (m, cut) ← Extended-Bottom-up-CR(n)
    While n > 0
        print cut[n]
        n ← n – cut[n] //remove the first piece
```

Informal running time analysis

Approach 1: approximate via (# of subproblems) * (# of choices for each subproblems)

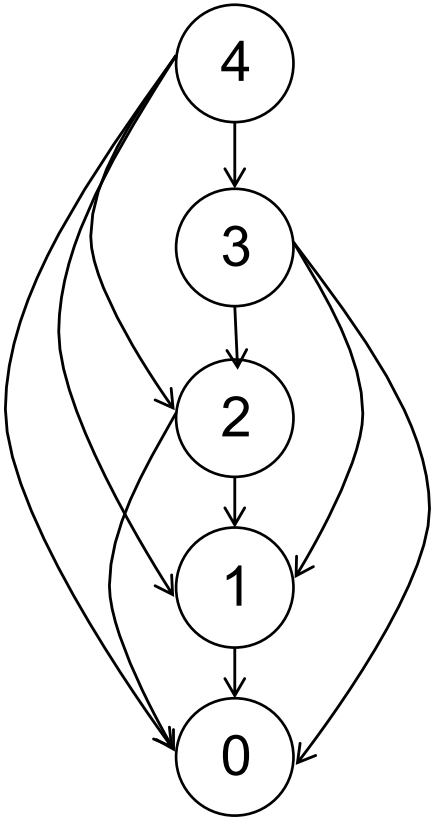
- In rod cutting,
  - # of subproblems = n
  - # of choices for each subproblem = O(n)
  - => T(n) is about $O(n^2)$

Approach 2: approximate via subproblem graphs

Subproblem graphs

A graph illustrates the set of subproblems involved and how subproblems depend on one another

E.g., in rod cutting with n = 4,



Running time estimation

- |V| = # of subproblems
- |E| = sum of # of choices for each subproblem
- T(n) is about O(|V| + |E|)

DP and graph traversal strategies

Bottom-up method: Reverse Topological Ordering

Top-down method: Depth First Search

Dynamic programming: 4 steps

1. Characterize the structure of an optimal solution
   - **Overlapping subproblems:** revisits same subproblem repeatedly
   - **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 smaller problems
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 Step 4 may be combined

Dynamic programming: 4 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
4. Construct an optimal solution from computed information

Let's revisit the rod cutting problem with these 4 steps in mind

Step 1: Characterize an optimal solution

Rod cutting problem (鐵條切割問題): determine the **maximum revenue** obtainable by cutting up the rod and selling the pieces

- n = length of the rod, p_i = price of length-i rod

In this step, we need to answer two questions:

Step1-Q1: What can be the subproblems?

Step1-Q2: Does it exhibit optimal substructure?

- Can an optimal solution be represented by the optimal solutions to the subproblems?
- If we cannot find optimal substructure, either we have to go back to Step1-Q1 or there is no DP solution to this problem.

Step 1: Characterize an optimal solution

Rod cutting problem (鐵條切割問題): determine the **maximum revenue** obtainable by cutting up the rod and selling the pieces

- n = length of the rod, p_i = price of length-i rod

Subproblems: CR(0), CR(1), …, CR(n-1)

- CR(i): rod cutting problem with length-i rod
- Our goal: CR(n)

Suppose we know an optimal solution, OPT, to CR(i)

There are i possibilities:

- Case 1: the first segment in OPT has length 1
  - 從OPT拿掉一段長度為1的鐵條，剩下的部分是CR(i-1)的最佳解
- Case 2: the first segment in OPT has length 2
  - 從OPT拿掉一段長度為2的鐵條，剩下的部分是CR(i-2)的最佳解

…..

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

Need to justify this

Proof of optimal substructure

- **Optimal substructure**: an optimal solution can be constructed from optimal solutions to subproblems
  - Proof by contradiction (specifically, a "cut-and-paste" argument)

Proof of case x: the first segment in OPT has length x

Goal: 從OPT拿掉一段長度為x的鐵條，證明剩下的部分是CR(i-x)的最佳解

Suppose OPT is optimal to CR(i) but OPT\{x} is not optimal to CR(i-x)

- => there exist an optimal solution OPT' to CR(i-x) such that the value of OPT' is higher than it of OPT\{x}
- => OPT' ∪ {x} is a better solution to CR(i) than OPT
- => Contradiction!

Step 2: Recursively define the value of an optimal solution

Case 1: the first segment in OPT has length 1

$M[i] = p_1 + M[i-1]$

- 從OPT拿掉一段長度為1的鐵條，剩下的部分是CR(i-1)的最佳解

Case 2: the first segment in OPT has length 2

$M[i] = p_2 + M[i-2]$

- 從OPT拿掉一段長度為2的鐵條，剩下的部分是CR(i-2)的最佳解

.....

Case i: the first segment in OPT has length i

$M[i] = p_i + M[0]$

- 從OPT拿掉一段長度為i的鐵條，剩下的部分是CR(0)的最佳解

M[i] = the value of an optimal solution to CR(i)

用遞迴表示最佳解的值：

$$M[i] = \begin{cases} 0, & \text{if } i=0 \text{ (base case)} \\ \max_{1 \le j \le i}\{p_j + M[i-j]\}, & \text{otherwise} \end{cases}$$

Step 3: Compute value of an optimal solution

- **M[i] = the value of an optimal solution to CR(i)**

$$M[i] = \begin{cases} 0, & \text{if } i=0 \text{ (base case)} \\ \max_{1 \le j \le i}\{p_j + M[i-j]\}, & \text{otherwise} \end{cases}$$

- Let's try the bottom-up method
- Solve "smaller" subproblems first

i	0	1	2	3	4
M[i]	0	1	5	8	10

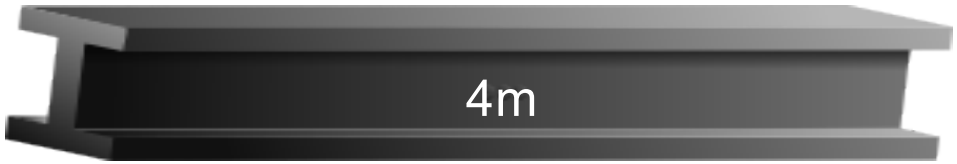
Our goal

$\max\{p_1+M[0]\}$

$\max\{p_1+M[1], p_2+M[0]\}$

$\max\{p_1+M[2], p_2+M[1], p_3+M[0]\}$

$\max\{p_1+M[3], p_2+M[2], p_3+M[1], p_4+M[0]\}$



Length (m)	1	2	3	4
Price (k)	1	5	8	9

Step 3: Compute value of an optimal solution

- **M[i] = the value of an optimal solution to CR(i)**

$$M[i] = \begin{cases} 0, & \text{if } i=0 \text{ (base case)} \\ \max_{1 \le j \le i}\{p_j + M[i-j]\}, & \text{otherwise} \end{cases}$$

- Let's try the bottom-up method
- Solve "smaller" subproblems first

i	0	1	2	3	…	n
M[i]						

Our goal

small → large

Running time: $\Theta(n^2)$

```
CR(n):
    for i = 1 to n
        M[i] ← 0
    for i = 1 to n //compute M[1], M[2], … in order
        for j = 1 to i
            M[i] ← max(M[i], p_j + M[i-j])
    return M[n]
```

Step 4: Construct an optimal solution

- M[i] = the value of an optimal solution to CR(i)

$$M[i] = \begin{cases} 0, & \text{if } i=0 \text{ (base case)} \\ \max_{1 \le j \le i}\{p_j + M[i-j]\}, & \text{otherwise} \end{cases}$$

- **Keep one more array:**

  cut[i] = first cut made to length-i rod in CR(i)

i	0	1	2	3	4
M[i]	0	1	5	8	10
cut[i]	0	1	2	3	2

Our goal

$\max\{p_1+M[0]\}$

$\max\{p_1+M[1], p_2+M[0]\}$

$\max\{p_1+M[2], p_2+M[1], p_3+M[0]\}$

$\max\{p_1+M[3], p_2+M[2], p_3+M[1], p_4+M[0]\}$

Length (m)	1	2	3	4
Price (k)	1	5	8	9

Step 4: Construct an optimal solution

```
Input: p[1…n]
CR(n):
    M[0] ← 0
    for j = 1 to n //compute M[1], M[2], … in order
        q ← -∞
        for i = 1 to j
            if q < p_i + M[j-i]
                q ← p_i + M[j-i]
                cut[j] ← i // the best first cut for len j rod
        M[i] ← q
    return M[n], cut
```

```
Print-CR-Sol(n):
   (M, cut) ← CR(n)
    While n > 0
        print cut[n]
        n ← n – cut[n] //remove the first piece
```

Weighted Interval Scheduling (加權區間調度)

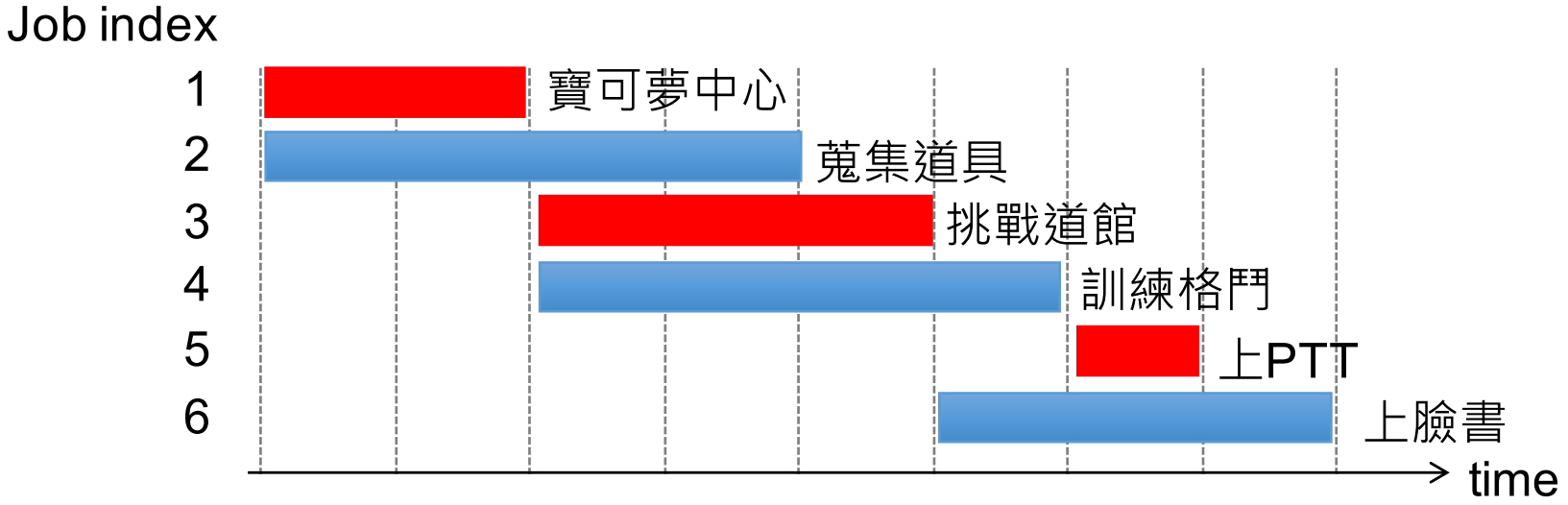
Interval scheduling (區間調度)

Given a set of job requests with start times and finish times, find the maximum number of compatible jobs

- E.g., 寶可夢訓練師也是很忙，一天到底可以做多少事情呢？

Interval scheduling can be solved using an earliest-finish-time-first greedy algorithm in O(n) time

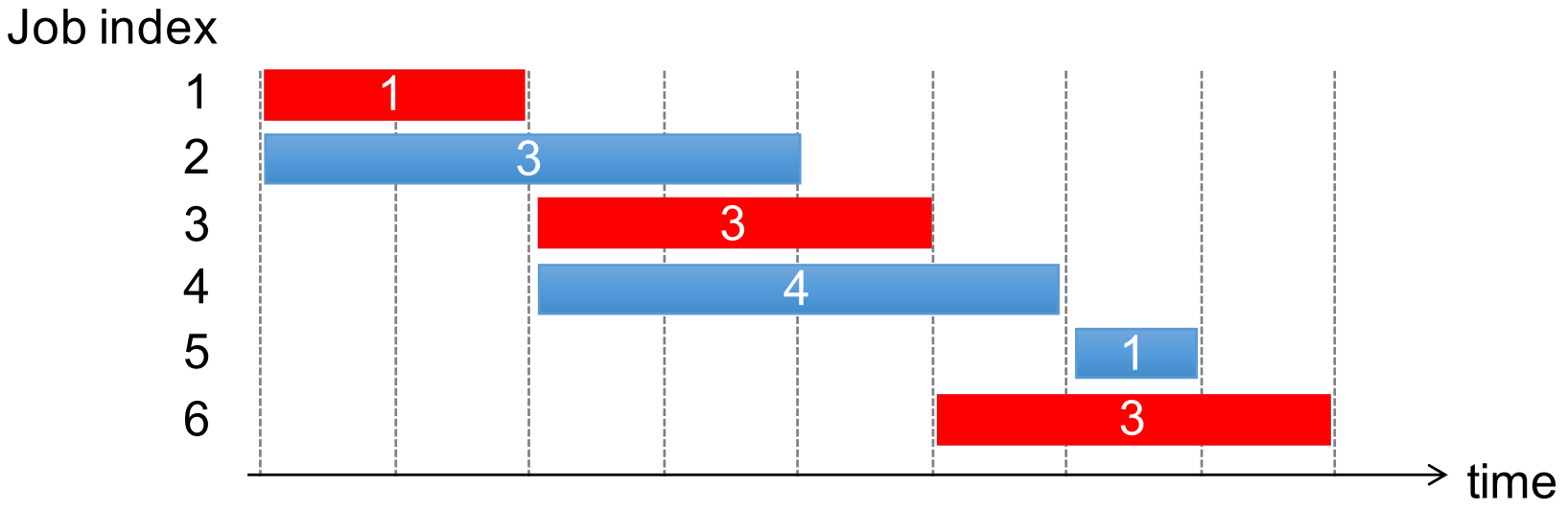
- 短視近利有時也會走運！



Weighted interval scheduling (加權區間調度)

Given a set of job requests with start times, finish times and values, find a compatible subset of maximum total value

- E.g., 給定每個行程的時間和經驗值，最多可以獲得多少經驗值？

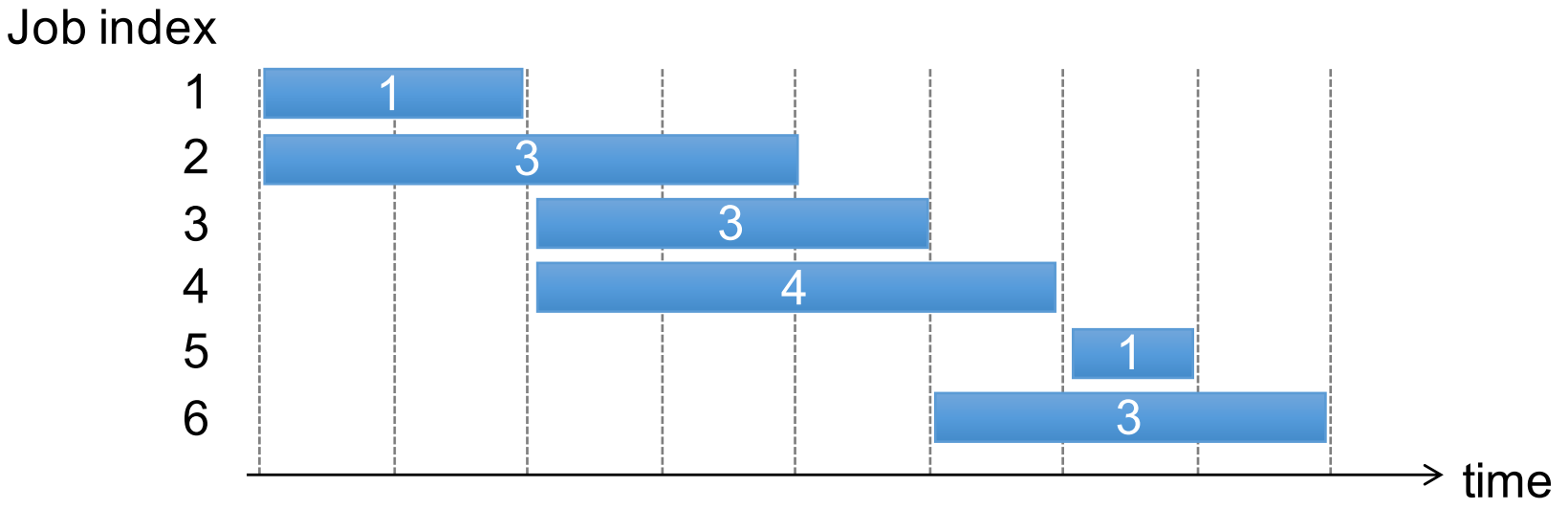


Weighted interval scheduling

Given n job requests, find a compatible subset of **maximum total value**

- Suppose requests are sorted in order of nondecreasing finish time
- v_i = values of job i
- f_i = finish time of job i, $f_1 \leq f_2 \leq \ldots \leq f_n$
- Let p(j) = largest index i < j such that job i and j are compatible
  - E.g., p(5) = 4, p(3) = 1, p(1) = 0

Brute-force = $O(2^n)$ ☹



Dynamic programming: 4 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
4. Construct an optimal solution from computed information

Step 1: Characterize an optimal solution

Weighted Interval Scheduling (加權區間調度): Given n job requests, find a compatible subset of **maximum total value**

- v_i = values of job i, f_i = finish time of job i, $f_1 \leq f_2 \leq \ldots \leq f_n$
- Let p(j) = largest index i < j such that job i and j are compatible

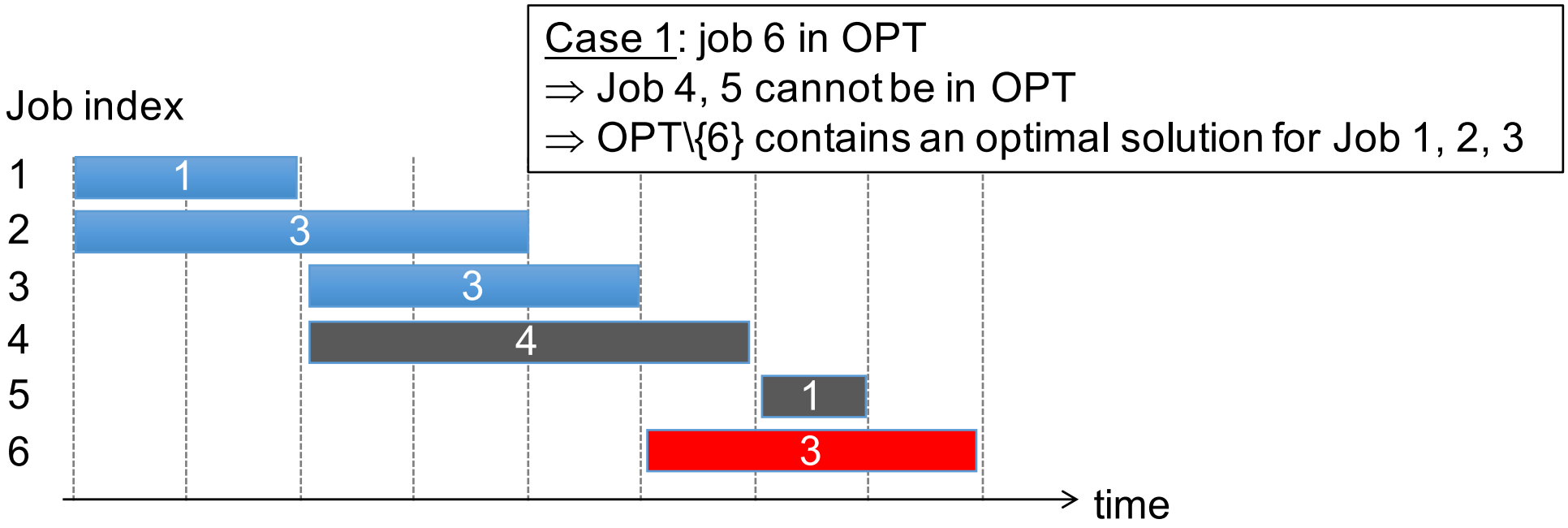
Subproblems: WIS(0), WIS(1), …, WIS(n-1)

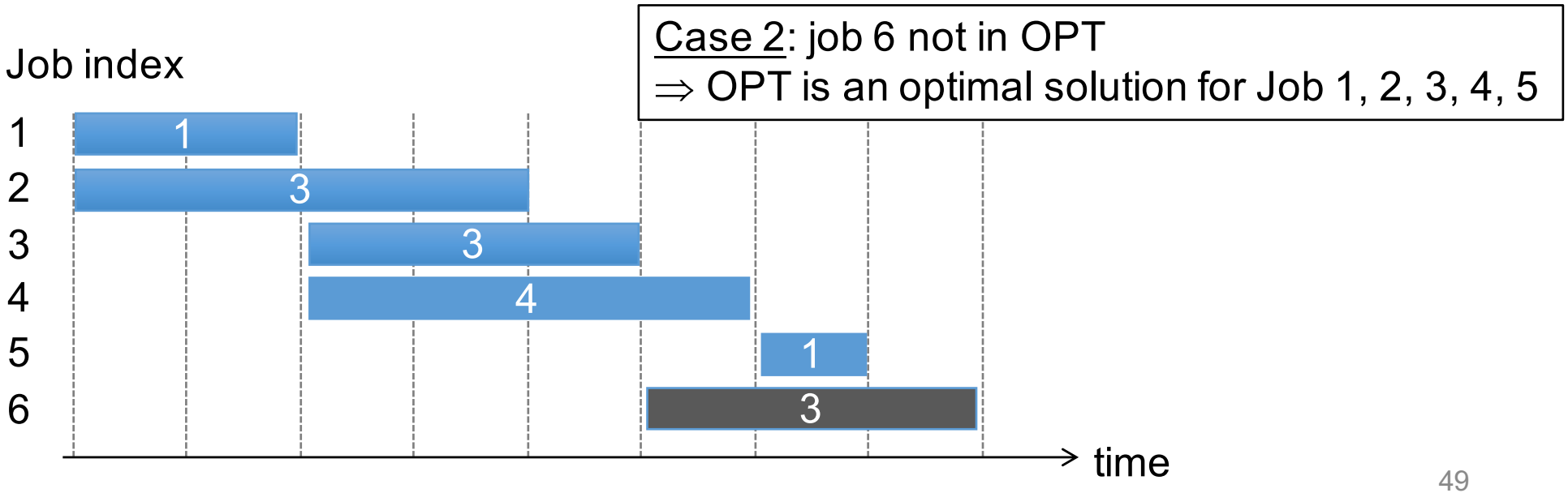
- WIS(i): weighted interval scheduling for the first i jobs
- Our goal: WIS(n)

Suppose we know one optimal solution, OPT, to WIS(i)

There are two possibilities:

- Case 1: job i in OPT
  - OPT\{i} is an optimal solution to WIS(p(i))
- Case 2: job i not in OPT
  - OPT is an optimal solution to WIS(i-1)


Case 1: job 6 in OPT
⇒ Job 4, 5 cannot be in OPT
⇒ OPT\{6} contains an optimal solution for Job 1, 2, 3
Job index
1
2
3
4
5
6
1
3
3
4
1
3
time


Case 2: job 6 not in OPT
⇒ OPT is an optimal solution for Job 1, 2, 3, 4, 5
Job index
1
2
3
4
5
6
1
3
3
4
1
3
time

Proof of optimal substructure

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

- Proof by contradiction (specifically, a "cut-and-paste" argument)

Proof of case 1: job i in OPT

Goal: 證明OPT\{i} is an optimal solution to WIS(p(i))

Suppose OPT is optimal to WIS(i) but OPT\{i} is not optimal to WIS(p(i))

- => there exist an optimal solution OPT' to WIS(p(i)) such that the value of OPT' is higher than it of OPT\{i}
- => OPT' ∪ {i} is a better solution to WIS(i) than OPT
- => Contradiction!

Prove case 2 by yourself

Step 2: Recursively define the value of an optimal solution

Case 1: job i in OPT

$M[i] = v_i + M[p(i)]$

- OPT\{i} is an optimal solution to WIS(p(i))

Case 2: job i not in OPT

$M[i] = M[i-1]$

- OPT is an optimal solution to WIS(i-1)

M[i] = the value of an optimal solution to WIS(i)

用遞迴表示最佳解的值：

$$M[i] = \begin{cases} 0, & \text{if } i=0 \text{ (base case)} \\ \max\{v_i + M[p(i)], M[i-1]\}, & \text{otherwise} \end{cases}$$

Step 3: Compute value of an optimal solution

M[i] = the value of an optimal solution to WIS(i)

$$M[i] = \begin{cases} 0, & \text{if } i=0 \text{ (base case)} \\ \max\{v_i + M[p(i)], M[i-1]\}, & \text{otherwise} \end{cases}$$

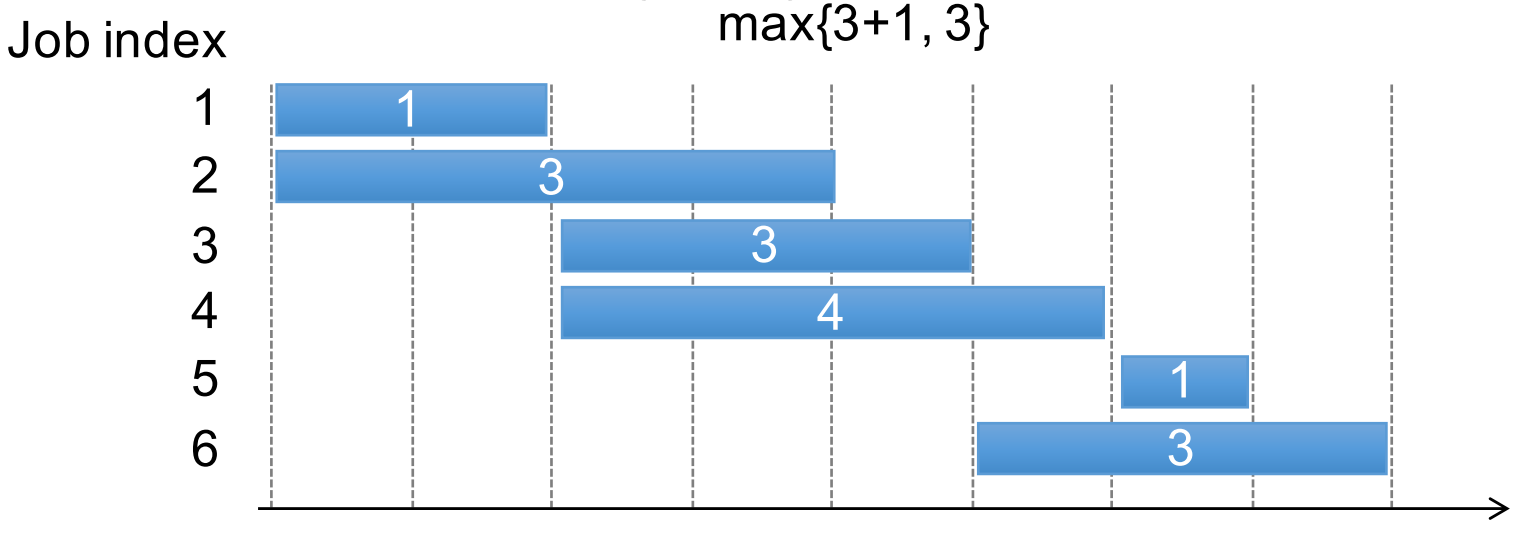
i	0	1	2	3	4	5	6
M[i]	0	1	3	4	5	6	7

Our goal

max{1+0, 0}

max{3+0, 1}

max{3+1, 3}



Step 3: Compute value of an optimal solution

M[i] = the value of an optimal solution to WIS(i)

$$M[i] = \begin{cases} 0, & \text{if } i=0 \text{ (base case)} \\ \max\{v_i + M[p(i)], M[i-1]\}, & \text{otherwise} \end{cases}$$

i	0	1	2	…	n
M[i]					

Our goal

small → large

```
Input: n, s[1..n], f[1..n], v[1..n], p[1..n], f1 ≤ f2 ≤ …≤ fn

WIS(n):
//M[i] = maximum value for scheduling jobs 1, 2, …, i
    M[0] <- 0
    for i = 1 to n
        M[i] <- max(v[i] + M[p[i]], M[i-1])
    Return M[n]
```

Running time = Θ(n)

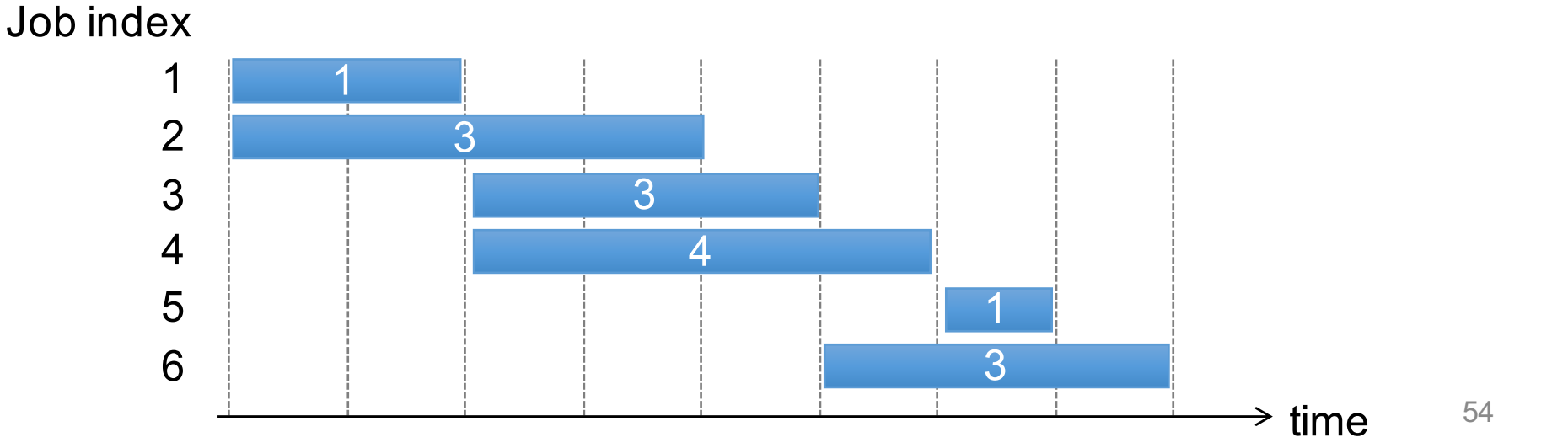
Step 4: Construct an optimal solution

M[i] = the value of an optimal solution to WIS(i)

$$M[i] = \begin{cases} 0, & \text{if } i=0 \text{ (base case)} \\ \max\{v_i + M[p(i)], M[i-1]\}, & \text{otherwise} \end{cases}$$

i	0	1	2	3	4	5	6
M[i]	0	1	3	4	5	6	7

Our goal



Step 4: Construct an optimal solution from computed information

Make a second pass to find the solution

```
Input: n, s[1..n], f[1..n], v[1..n], p[1..n], f_1 ≤ f_2 ≤ …≤ f_n

WIS(n):
    M[0] <- 0
    for i = 1 to n
        M[i] <- max(v[i] + M[p[i]], M[i-1])
    Return M[n]

Find-Solution(n):
    if n = 0
        return {} //empty set
    else if (M[n] > M[n-1])
        return {n} U Find-Solution(p[n])
    else
        return Find-Solution(n-1)
```

Running time = O(n)
Because # of recursive calls ≤ n

Informal running time analysis

Approach 1: approximate via (# of subproblems) * (# of choices for each subproblems)

- # of subproblems = n
- # of choices for each subproblem = 2
- => T(n) is about O(n)

Approach 2: approximate via subproblem graphs

- Practice: drawing a subproblem graph by yourself

Defining subproblems

Arguably the most difficult step in dynamic programming

Unlike divide and conquer, often time it's unclear how to define subproblems

Imagine intervals are not sorted by finished time in WIS

- Can the same definition of subproblems lead to optimal substructure?