Dynamic Programming

Algorithm Design and Analysis
Dynamic Programming (1)

http://ada.miulab.tw

Yun-Nung (Vivian) Chen

國立臺灣大學 National Taiwan University

Announcement

- Mini-HW 3 released
  - Due on 10/10 (Thu) 14:20
  - Online submission
- Homework 1 released
  - Due on 10/17 (Thur) 17:20 (2 weeks left)
  - Writing: print out the A4 hard copy and submit to NTU COOL
  - Programming: submit to Online Judge – http://ada-judge.csie.ntu.edu.tw

Outline

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

動腦一下 – 囚犯問題

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

猜測規則

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

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

有沒有更多人可以存活的好策略?

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 combining the solutions to subproblems
  - 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

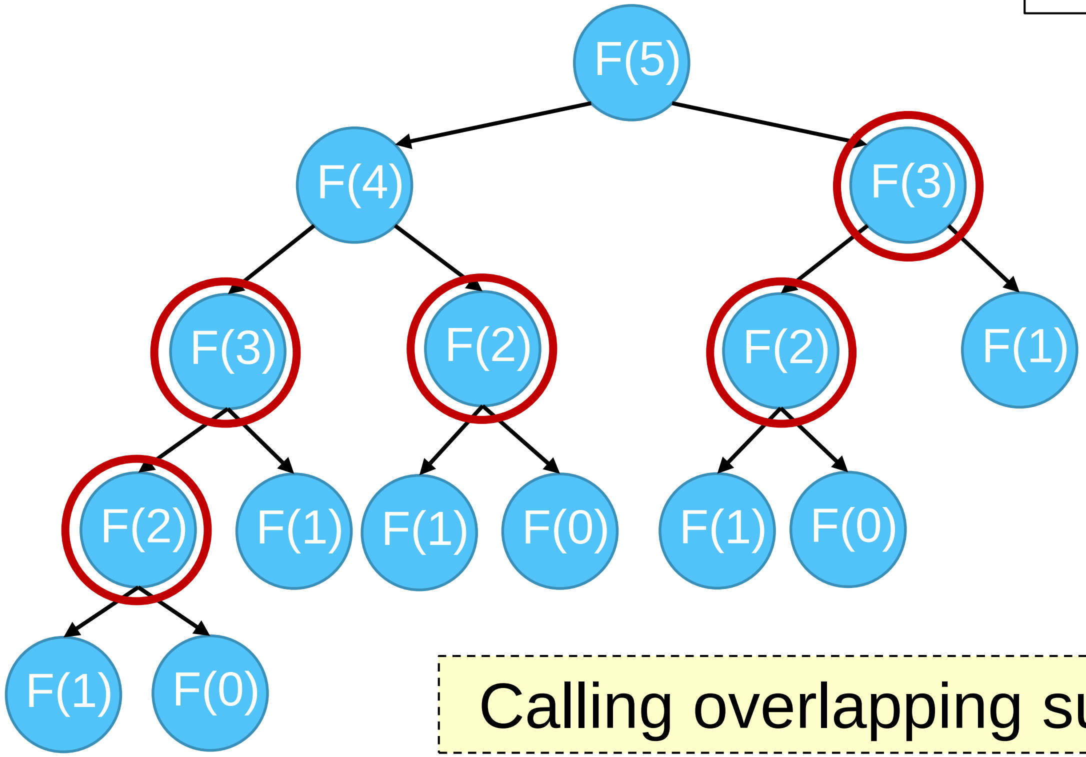
Dynamic Programming Procedure

- 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

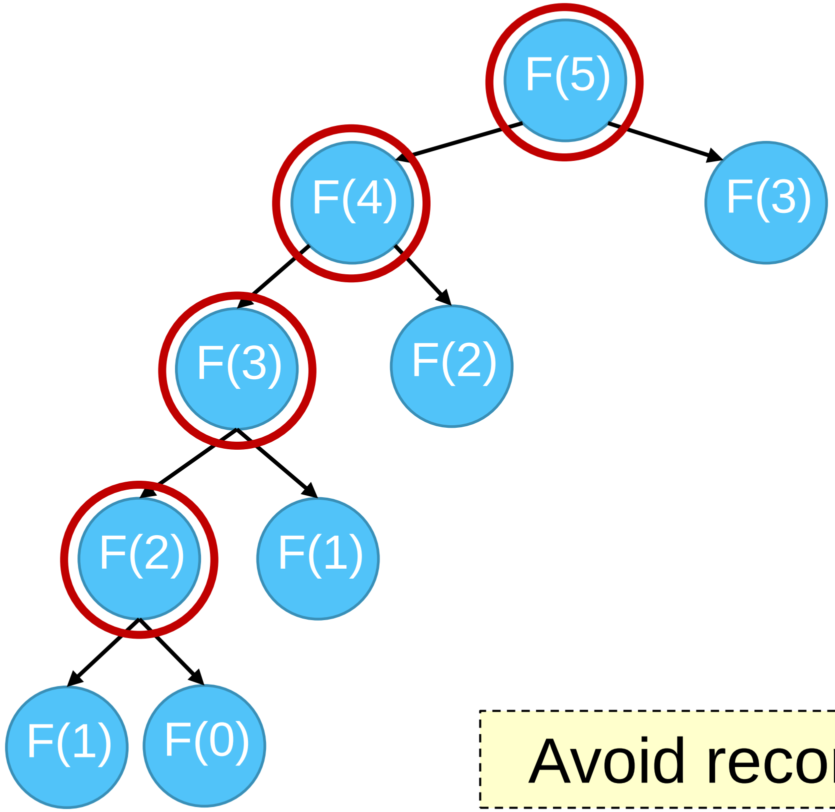
✓F(2) was computed 3 times

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

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

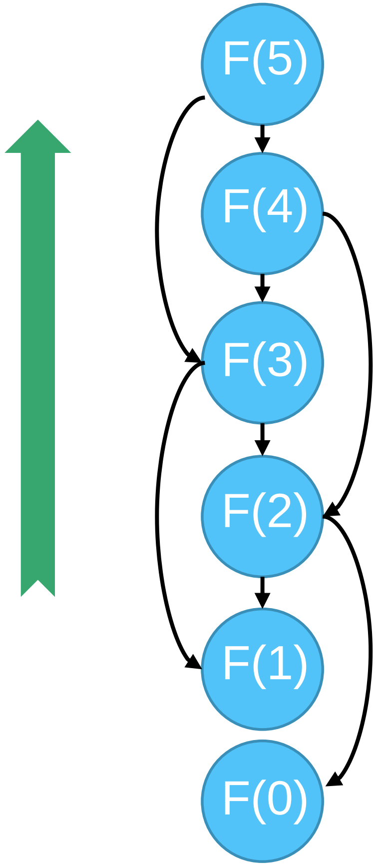
Top-Down with Memoization

```
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

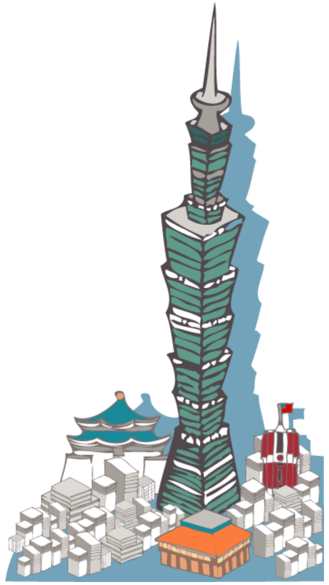
- Principle of Optimality
  - Any subpolicy of an optimum policy must itself be an optimum policy with regard to the initial and terminal states of the subpolicy
- Two key properties of DP for optimization
  - Overlapping subproblems
  - Optimal substructure – an optimal solution can be constructed from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

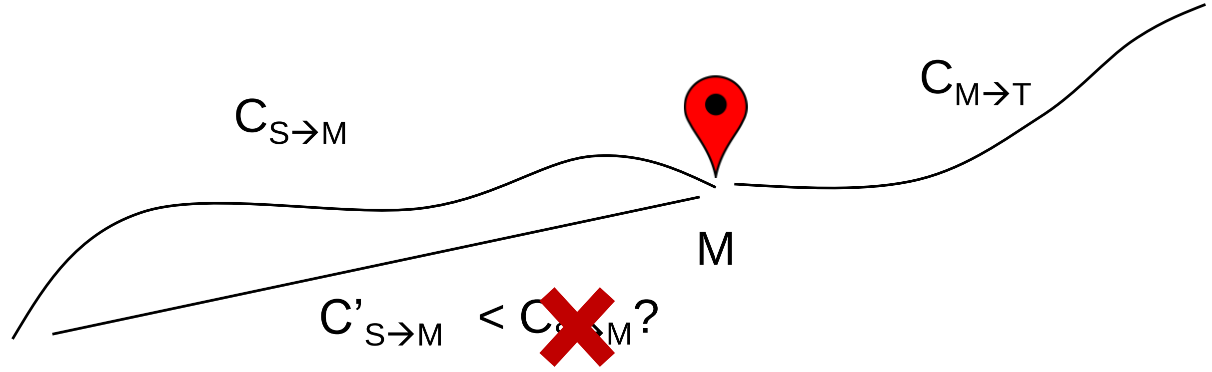
If the optimal substructure (principle of optimality) does not hold, then it is incorrect to use DP

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 $C_{S→M}$+ $C_{M→T}$ is the shortest path from S to T
→ The path with the cost $C_{S→M}$ 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 $C'_{S→M}$, then we can "cut" $C_{S→M}$ and "paste" $C'_{S→M}$ to make the original cost smaller

DP#1: Rod Cutting

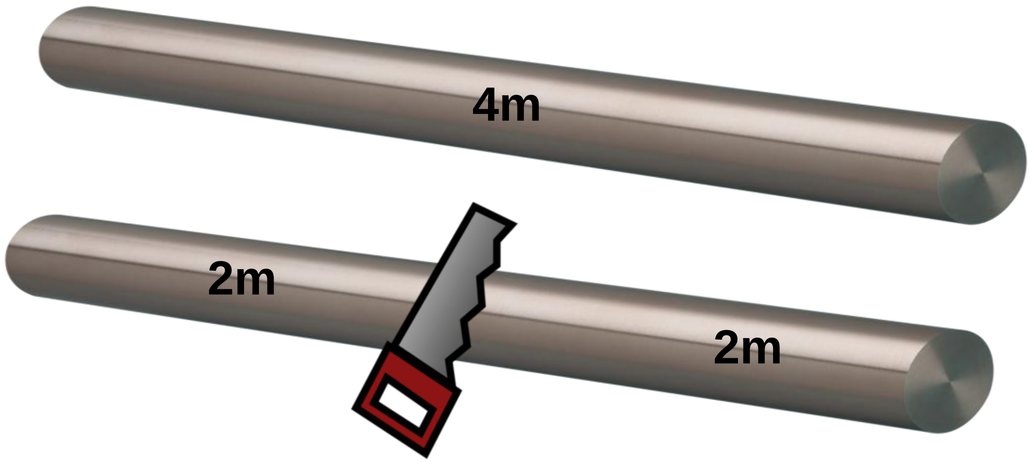
Textbook Chapter 15.1 – Rod Cutting

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

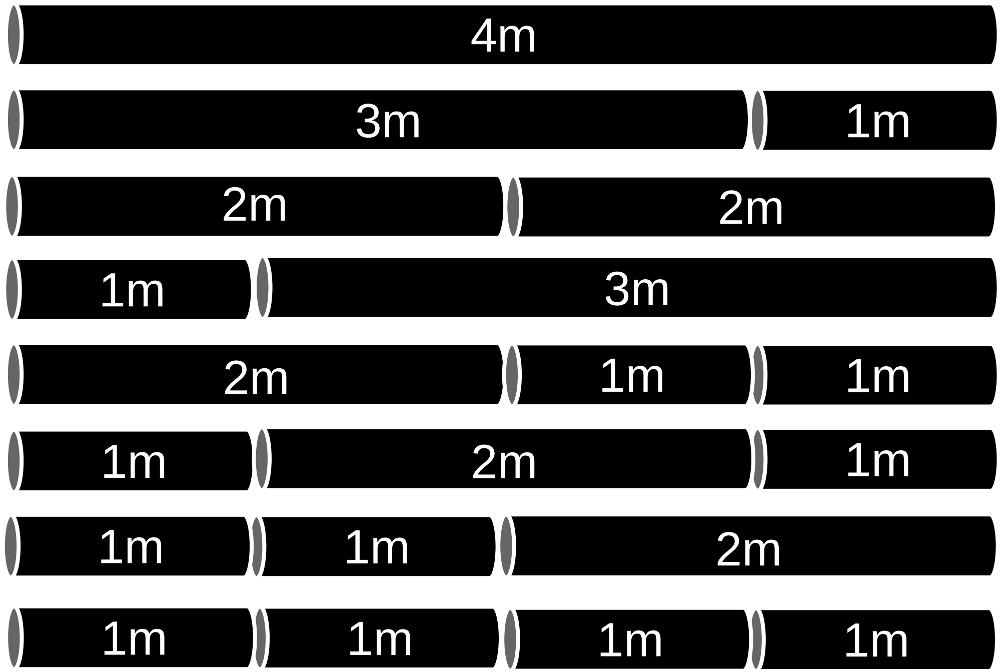
- Output: the maximum revenue r_n obtainable by cutting up the rod and selling the pieces



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

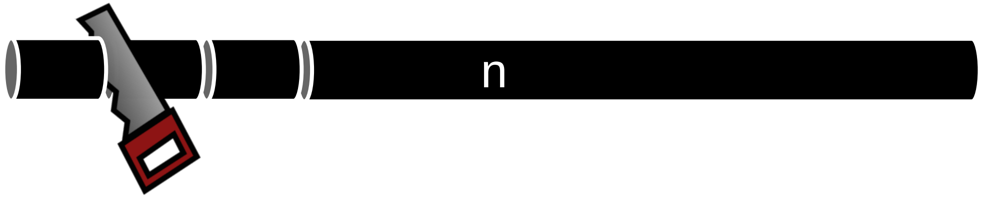


→ 9
→ 8 + 1 = 9
→ 5 + 5 = 10
→ 1 + 8 = 9
→ 5 + 1 + 1 = 7
→ 1 + 5 + 1 = 7
→ 1 + 1 + 5 = 7
→ 1 + 1 + 1 + 1 = 4

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?

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

no cut

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

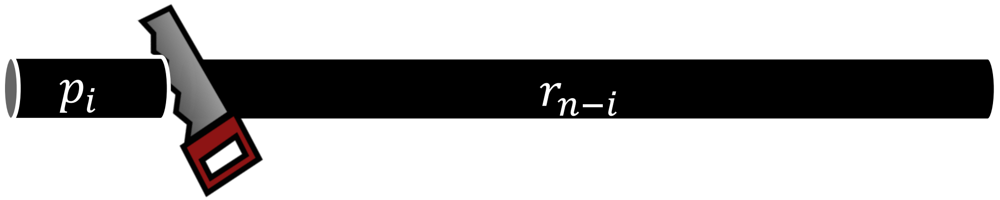
- Version 1

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

no cut

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

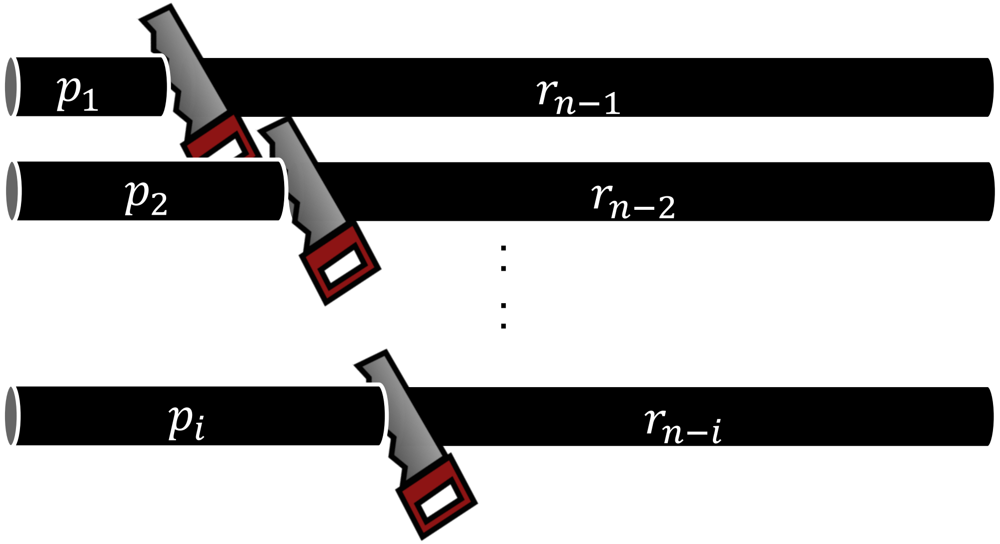
maximum value obtainable from the remaining part

Recursive Procedure

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

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

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
```

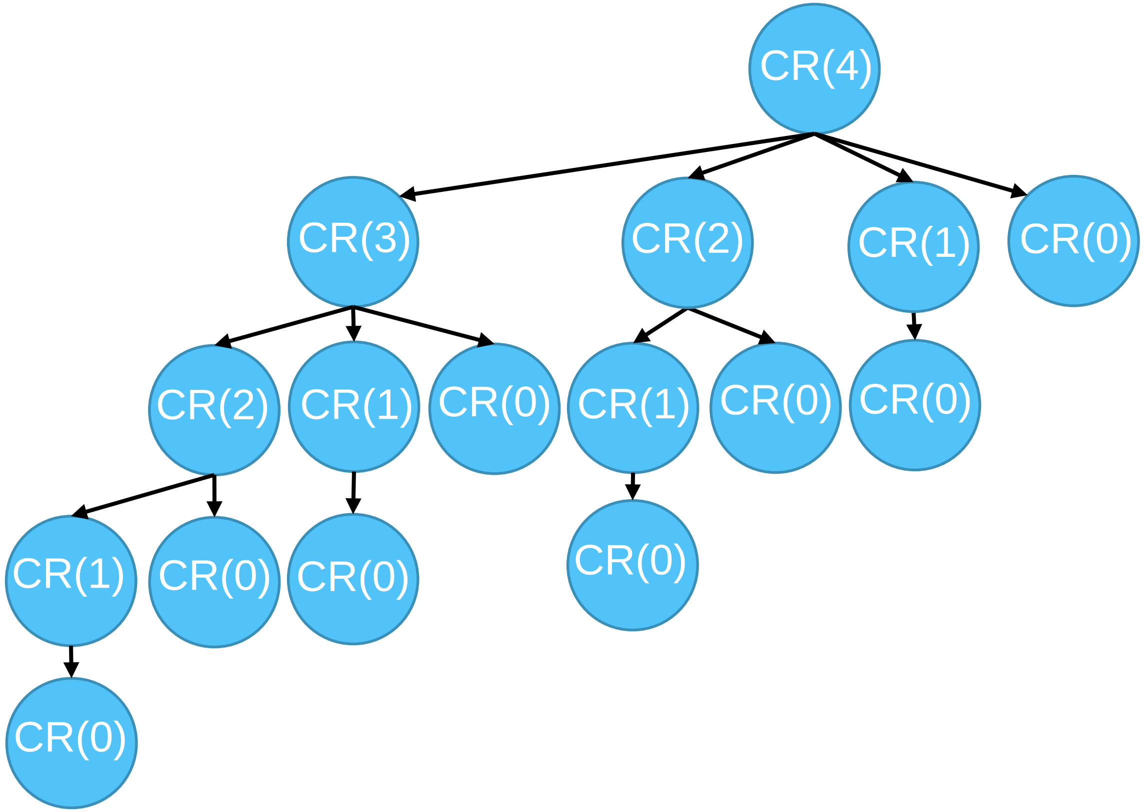
- $T(n)$ = time for running `Cut-Rod(p, n)`

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

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]
```

$\Theta(n^2)$

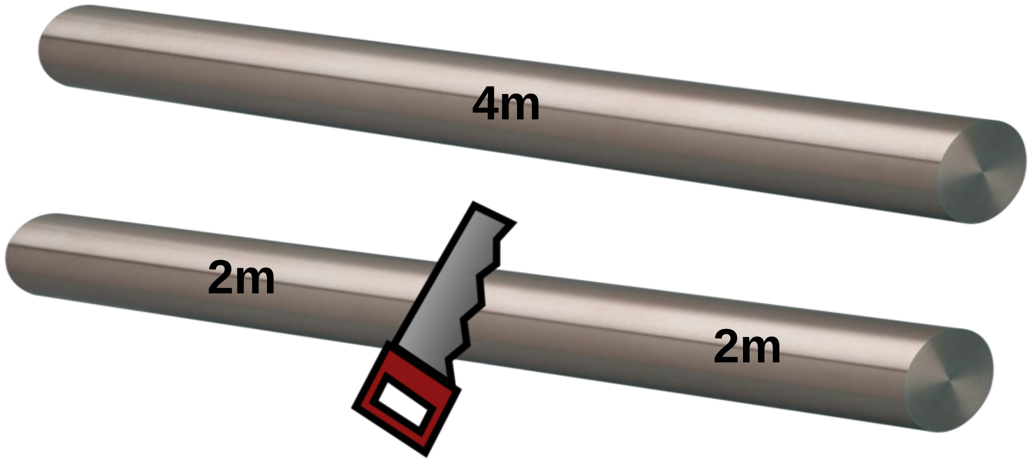
- $T(n)$ = time for running `Bottom-Up-Cut-Rod(p, n)` ➡ $T(n) = \Theta(n^2)$

Rod Cutting Problem

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

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
```

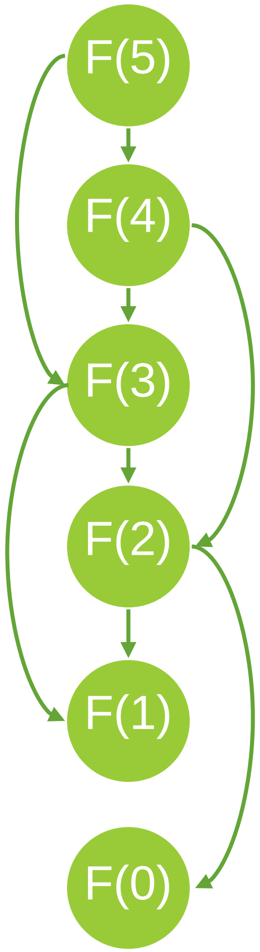
```
Print-Cut-Rod-Solution(p, n)
  (r, cut) = Extended-Bottom-up-Cut-Rod(p, n)
  while n > 0
    print cut[n]
    n = n - cut[n] // remove the first piece
```

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 <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

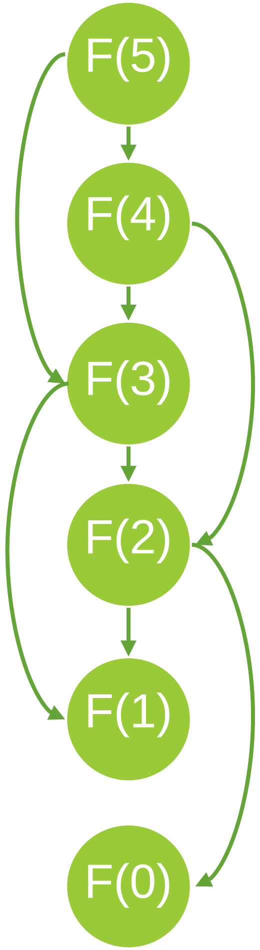
Subproblem Graphs

- 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 $i = 1, \dots, n$
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 $i = 1, \ldots, n$
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]								

```
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 $v_1, v_2, \ldots, v_k$
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 $v_1, v_2, \dots, v_k$
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]]
```

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

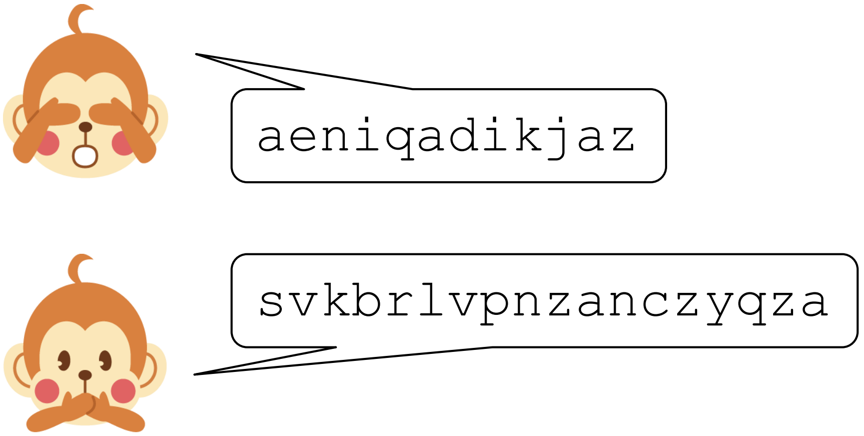
DP#3: Sequence Alignment

Textbook Chapter 15.4 – Longest common subsequence

Textbook Problem 15-5 – Edit distance

Monkey Speech Recognition

- 猴子們各自講話，經過語音辨識系統後，哪一支猴子發出最接近英文字”banana”的語音為優勝者
- How to evaluate the similarity between two sequences?



banana

Longest Common Subsequence (LCS)

- Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$
  $Y = \langle y_1, y_2, \cdots, y_n \rangle$
- Output: <u>longest common subsequence</u> of two sequences
  - The maximum-length sequence of characters that appear left-to-right (but not necessarily a continuous string) in both sequences

```
X = banana            X = banana
Y = aeniqadikjaz      Y = svkbrlvpnzanczyqza

4  X → ba-n--an---a-     X → ---ba---n-an-----a  5
   Y → -aeniqadikjaz     Y → svkbrlvpnzanczyqza
```

The **infinite monkey theorem**: a monkey hitting keys at random for an infinite amount of time will almost surely type a given text

Edit Distance

- Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$
  $Y = \langle y_1, y_2, \cdots, y_n \rangle$
- Output: <u>the minimum cost of transformation</u> from *X* to *Y*
  - Quantifier of the dissimilarity of two strings

X = `banana`
Y = `aeniqadikjaz`

9

$X \rightarrow$ `ba-n--an---a-`
$Y \rightarrow$ `-aeniqadikjaz`

1 deletion, 7 insertions, 1 substitution

X = `banana`
Y = `svkbrlvpnzanczyqza`

13

$X \rightarrow$ `---ba---n-an------a`
$Y \rightarrow$ `svkbrlvpnzanczyqza`

12 insertions, 1 substitution

Sequence Alignment Problem

- Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$
  $Y = \langle y_1, y_2, \cdots, y_n \rangle$
- Output: <u>the minimal cost</u> $M_{m,n}$ for aligning two sequences
  - $\text{Cost} = \#\text{insertions} \times C_{\text{INS}} + \#\text{deletions} \times C_{\text{DEL}} + \#\text{substitutions} \times C_{p,q}$

Step 1: Characterize an OPT Solution

Sequence Alignment Problem
Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$ $Y = \langle y_1, y_2, \cdots, y_n \rangle$
Output: the minimal cost $M_{m,n}$ for aligning two sequences

- Subproblems
  - `SA(i, j)`: sequence alignment between prefix strings $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$
  - Goal: `SA(m, n)`
- Optimal substructure: suppose OPT is an optimal solution to `SA(i, j)`, there are 3 cases:
  - Case 1: x_i and y_j are aligned in OPT (match or substitution)
    - OPT/$\{x_i, , y_j\}$ is an optimal solution of `SA(i-1, j-1)`
  - Case 2: x_i is aligned with a gap in OPT (deletion)
    - OPT is an optimal solution of `SA(i-1, j)`
  - Case 3: y_j is aligned with a gap in OPT (insertion)
    - OPT is an optimal solution of `SA(i, j-1)`

Step 2: Recursively Define the Value of an OPT Solution

Sequence Alignment Problem
Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$ $Y = \langle y_1, y_2, \cdots, y_n \rangle$
Output: the minimal cost $M_{m,n}$ for aligning two sequences

- Suppose OPT is an optimal solution to `SA(i, j)`, there are 3 cases:
  - Case 1: x_i and y_j are aligned in OPT (match or substitution)
    - OPT/$\{x_i, , y_j\}$ is an optimal solution of `SA(i-1, j-1)` $M_{i,j} = M_{i-1,j-1} + C_{x_i,y_j}$
  - Case 2: x_i is aligned with a gap in OPT (deletion)
    - OPT is an optimal solution of `SA(i-1, j)` $M_{i,j} = M_{i-1,j} + C_{\text{DEL}}$
  - Case 3: y_j is aligned with a gap in OPT (insertion)
    - OPT is an optimal solution of `SA(i, j-1)` $M_{i,j} = M_{i,j-1} + C_{\text{INS}}$
- Recursively define the value

$$M_{i,j} = \begin{cases} jC_{\text{INS}} & \text{if } i = 0 \\ iC_{\text{DEL}} & \text{if } j = 0 \\ \min(M_{i-1,j-1} + C_{x_i,y_j}, M_{i-1,j} + C_{\text{DEL}}, M_{i,j-1} + C_{\text{INS}}) & \text{otherwise} \end{cases}$$

Step 3: Compute Value of an OPT Solution

Sequence Alignment Problem
Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$ $Y = \langle y_1, y_2, \cdots, y_n \rangle$
Output: the minimal cost $M_{m,n}$ for aligning two sequences

- Bottom-up method: solve smaller subproblems first

$$M_{i,j} = \begin{cases} jC_{\text{INS}} & \text{if } i = 0 \\ iC_{\text{DEL}} & \text{if } j = 0 \\ \min(M_{i-1,j-1} + C_{x_i,y_j}, M_{i-1,j} + C_{\text{DEL}}, M_{i,j-1} + C_{\text{INS}}) & \text{otherwise} \end{cases}$$

X\Y	0	1	2	3	4	5	…	n
0								
1								
:								
m								

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

Step 3: Compute Value of an OPT Solution

Sequence Alignment Problem
Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$ $Y = \langle y_1, y_2, \cdots, y_n \rangle$
Output: the minimal cost $M_{m,n}$ for aligning two sequences

- Bottom-up method: solve smaller subproblems first

$$M_{i,j} = \begin{cases} jC_{\text{INS}} & \text{if } i = 0 \\ iC_{\text{DEL}} & \text{if } j = 0 \\ \min(M_{i-1,j-1} + C_{x_i,y_j}, M_{i-1,j} + C_{\text{DEL}}, M_{i,j-1} + C_{\text{INS}}) & \text{otherwise} \end{cases}$$

$$C_{\text{DEL}} = 4, C_{\text{INS}} = 4$$
$$C_{p,q} = 7, \text{if } p \neq q$$

			a	e	n	i	q	a	d	i	k	j	a	z
	X\Y	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	4	8	12	16	20	24	28	32	36	40	44	48
b	1	4	7	11	15	19	23	27	31	35	39	43	47	51
a	2	8	4	8	12	16	20	23	27	31	35	39	43	47
n	3	12	8	12	8	12	16	20	24	28	32	36	40	44
a	4	16	12	15	12	15	19	16	20	24	28	32	36	40
n	5	20	16	19	15	19	22	20	23	27	31	35	39	43
a	6	24	20	23	19	22	26	22	26	30	34	38	35	39

Step 3: Compute Value of an OPT Solution

Sequence Alignment Problem
Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$ $Y = \langle y_1, y_2, \cdots, y_n \rangle$
Output: the minimal cost $M_{m,n}$ for aligning two sequences

- Bottom-up method: solve smaller subproblems first

$$M_{i,j} = \begin{cases} jC_{\text{INS}} & \text{if } i = 0 \\ iC_{\text{DEL}} & \text{if } j = 0 \\ \min(M_{i-1,j-1} + C_{x_i,y_j}, M_{i-1,j} + C_{\text{DEL}}, M_{i,j-1} + C_{\text{INS}}) & \text{otherwise} \end{cases}$$

```
Seq-Align(X, Y, C_DEL, C_INS, C_p,q)
  for j = 0 to n
    M[0][j] = j * C_INS // |X|=0, cost=|Y|*penalty
  for i = 1 to m
    M[i][0] = i * C_DEL // |Y|=0, cost=|X|*penalty
  for i = 1 to m
    for j = 1 to n
      M[i][j] = min(M[i-1][j-1]+C_xi,yi, M[i-1][j]+C_DEL, M[i][j-1]+C_INS)
  return M[m][n]
```

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

Step 4: Construct an OPT Solution by Backtracking

Sequence Alignment Problem
Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$ $Y = \langle y_1, y_2, \cdots, y_n \rangle$
Output: the minimal cost $M_{m,n}$ for aligning two sequences

- Bottom-up method: solve smaller subproblems first

$$M_{i,j} = \begin{cases} jC_{\text{INS}} & \text{if } i = 0 \\ iC_{\text{DEL}} & \text{if } j = 0 \\ \min(M_{i-1,j-1} + C_{x_i,y_j}, M_{i-1,j} + C_{\text{DEL}}, M_{i,j-1} + C_{\text{INS}}) & \text{otherwise} \end{cases}$$

$$C_{\text{DEL}} = 4, C_{\text{INS}} = 4$$

$$C_{p,q} = 7, \text{if } p \neq q$$

			a	e	n	i	q	a	d	i	k	j	a	z
	X\Y	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	4	8	12	16	20	24	28	32	36	40	44	48
b	1	4	7	11	15	19	23	27	31	35	39	43	47	51
a	2	8	4	8	12	16	20	23	27	31	35	39	43	47
n	3	12	8	12	8	12	16	20	24	28	32	36	40	44
a	4	16	12	15	12	15	19	16	20	24	28	32	36	40
n	5	20	16	19	15	19	22	20	23	27	31	35	39	43
a	6	24	20	23	19	22	26	22	26	30	34	38	35	39

Step 4: Construct an OPT Solution by Backtracking

Sequence Alignment Problem
Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$ $Y = \langle y_1, y_2, \cdots, y_n \rangle$
Output: the minimal cost $M_{m,n}$ for aligning two sequences

- Bottom-up method: solve smaller subproblems first

$$M_{i,j} = \begin{cases} jC_{\text{INS}} & \text{if } i = 0 \\ iC_{\text{DEL}} & \text{if } j = 0 \\ \min(M_{i-1,j-1} + C_{x_i,y_j}, M_{i-1,j} + C_{\text{DEL}}, M_{i,j-1} + C_{\text{INS}}) & \text{otherwise} \end{cases}$$

```
Find-Solution(M)
  if m = 0 or n = 0
    return {}
  v = min(M[m-1][n-1] + C_{xm,yn}, M[m-1][n] + C_DEL, M[m][n-1] + C_INS)
  if v = M[m-1][n] + C_DEL // ↑: deletion
    return Find-Solution(m-1, n)
  if v = M[m][n-1] + C_INS // ←:insertion
    return Find-Solution(m, n-1)
return {(m, n)} ∪ Find-Solution(m-1, n-1) // ↖: match/substitution
```

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

Step 4: Construct an OPT Solution by Backtracking

```
Seq-Align(X, Y, C_DEL, C_INS, C_p,q)
  for j = 0 to n
    M[0][j] = j * C_INS // |X|=0, cost=|Y|*penalty
  for i = 1 to m
    M[i][0] = i * C_DEL // |Y|=0, cost=|X|*penalty
  for i = 1 to m
    for j = 1 to n
      M[i][j] = min(M[i-1][j-1]+C_xi,yi, M[i-1][j]+C_DEL, M[i][j-1]+C_INS)
  return M[m][n]
```

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

```
Find-Solution(M)
  if m = 0 or n = 0
    return {}
  v = min(M[m-1][n-1] + C_xm,yn, M[m-1][n] + C_DEL, M[m][n-1] + C_INS)
  if v = M[m-1][n] + C_DEL // ↑: deletion
    return Find-Solution(m-1, n)
  if v = M[m][n-1] + C_INS // ←:insertion
    return Find-Solution(m, n-1)
return {(m, n)} ∪ Find-Solution(m-1, n-1) // ↖: match/substitution
```

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

Space Complexity

- Space complexity

X\Y	0	1	2	3	4	5	…	n
0								
1								
:								
m								

$\rightarrow \Theta(mn)$

- If only keeping the most recent two rows: `Space-Seq-Align(X, Y)`

X\Y	0	1	2	3	…	j	…	n
i - 1								
i								

$\rightarrow \Theta(n)$

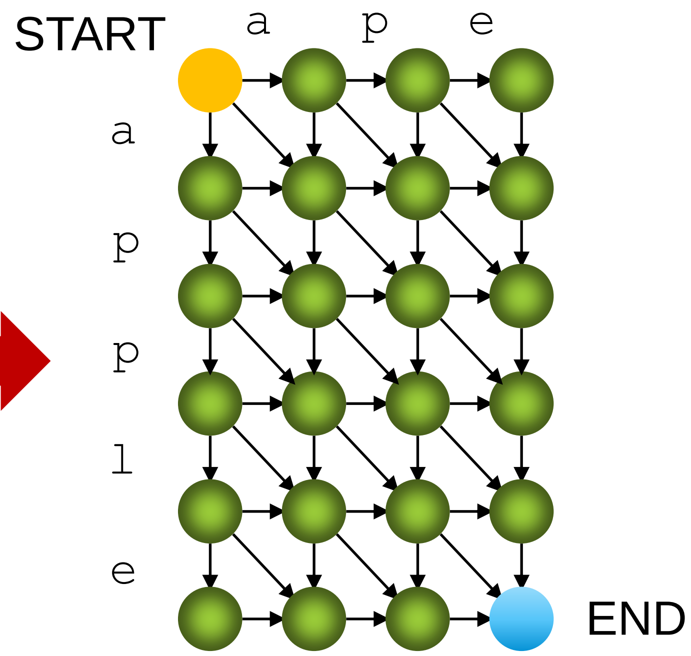
The optimal value can be computed, but the solution cannot be reconstructed

Space-Efficient Solution

Divide-and-Conquer
+
Dynamic
Programming

- Problem: find the min-cost alignment → find the shortest path

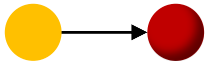
	X\Y	0	1	2	3
			a	p	e
	0	0	4	8	12
a	1	4	7	11	15
p	2	8	4	8	12
p	3	12	8	12	8
l	4	16	12	15	12
e	5	20	16	19	15

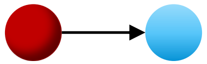


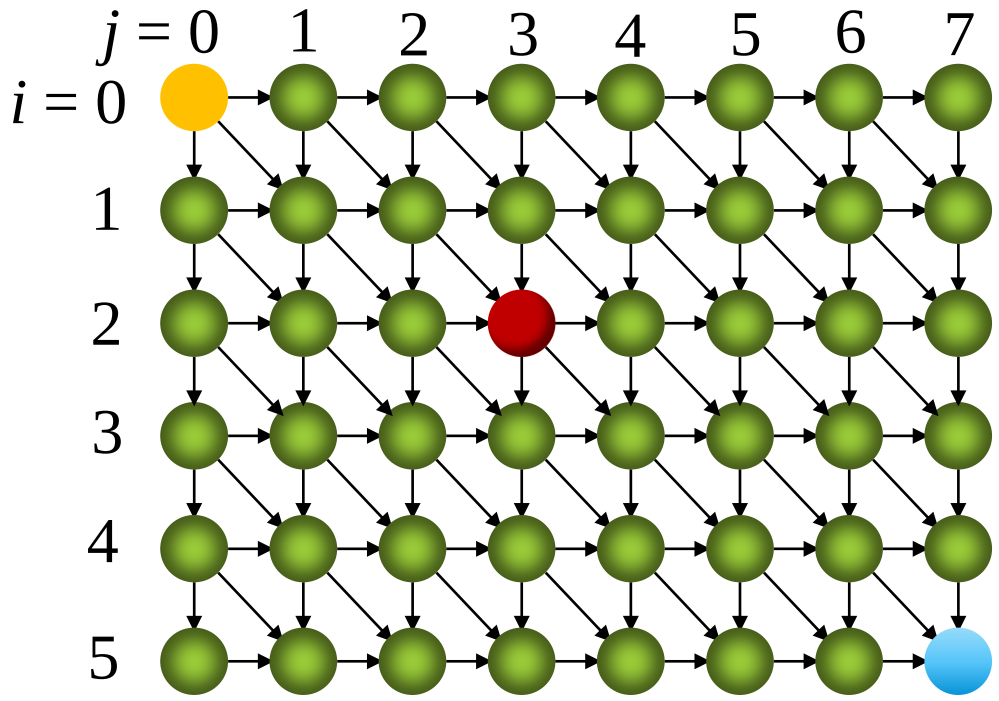
→ distance = C_{INS}
↓ distance = C_{DEL}
↘ distance = $C_{u,v}$ for edge (u, v)

Shortest Path in Graph

- Each edge has a length/cost
- $F(i,j)$: length of the shortest path from $(0,0)$ to (i,j) (START → (i,j))
- $B(i,j)$: length of the shortest path from (i,j) to (m,n) ((i,j) → END)
- $F(m,n) = B(0,0)$

$F(2,3)$ = distance of the shortest path 

$B(2,3)$ = distance of the shortest path 



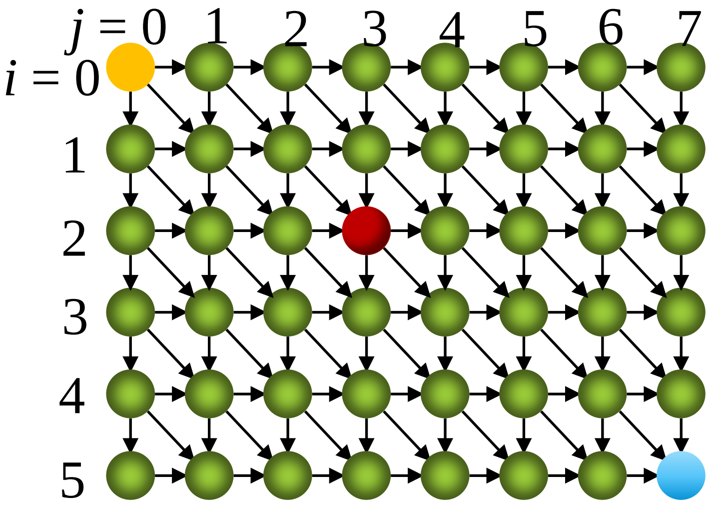
Recursive Equation

- Each edge has a length/cost
- $F(i,j)$: length of the shortest path from $(0,0)$ to (i,j) (START → (i,j))
- $B(i,j)$: length of the shortest path from (i,j) to (m,n) ((i,j) → END)
- Forward formulation

$$F_{i,j} = \begin{cases} jC_{\text{INS}} & \text{if } i = 0 \\ iC_{\text{DEL}} & \text{if } j = 0 \\ \min(F_{i-1,j-1} + C_{x_i,y_j}, F_{i-1,j} + C_{\text{DEL}}, F_{i,j-1} + C_{\text{INS}}) & \text{otherwise} \end{cases}$$

- Backward formulation

$$B_{i,j} = \begin{cases} (n-j)C_{\text{INS}} & \text{if } i = 0 \\ (m-i)C_{\text{DEL}} & \text{if } j = 0 \\ \min(B_{i+1,j+1} + C_{x_i,y_j}, B_{i+1,j} + C_{\text{DEL}}, B_{i,j+1} + C_{\text{INS}}) & \text{otherwise} \end{cases}$$

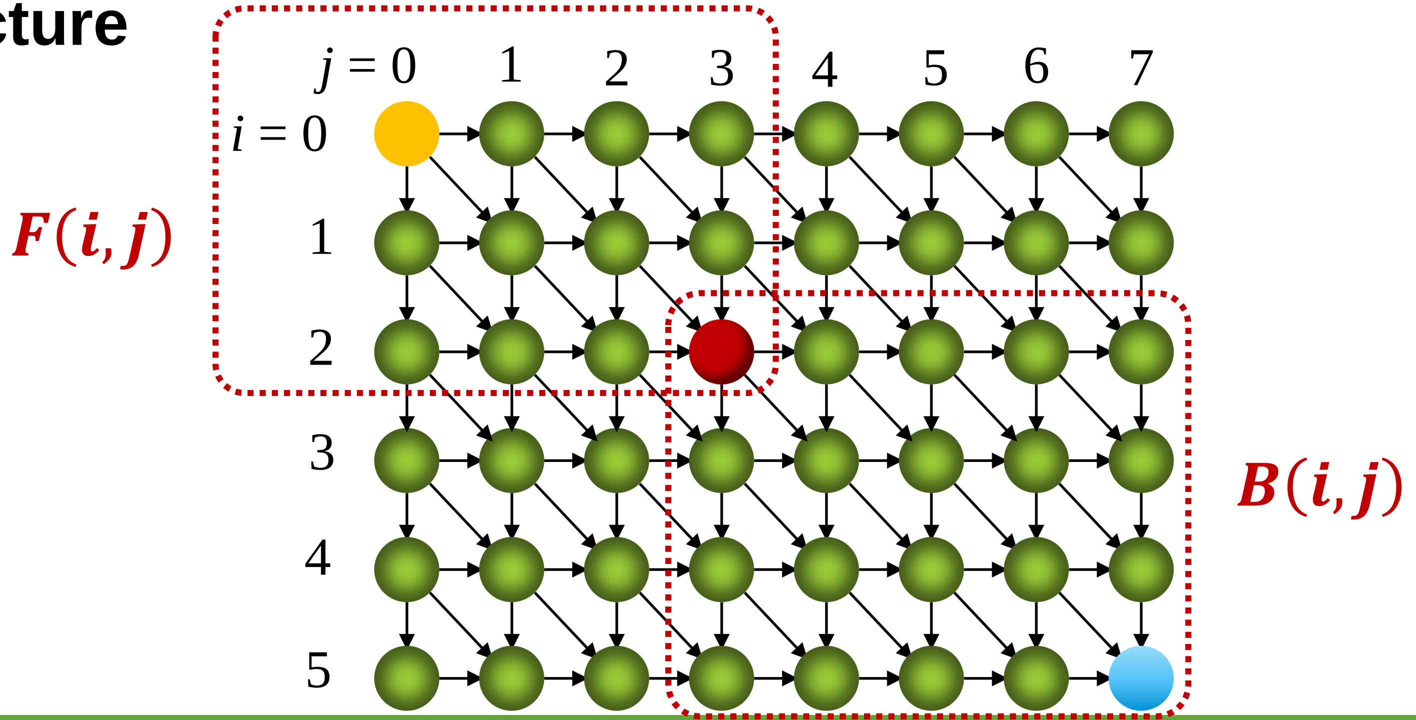


Shortest Path Problem

$F(i, j)$: length of the shortest path from $(0,0)$ to (i, j)
$B(i, j)$: length of the shortest path from (i, j) to (m, n)

- Observation 1: the length of the shortest path from $(0,0)$ to (m, n) that passes through (i, j) is $F(i, j) + B(i, j)$

  → **optimal substructure**

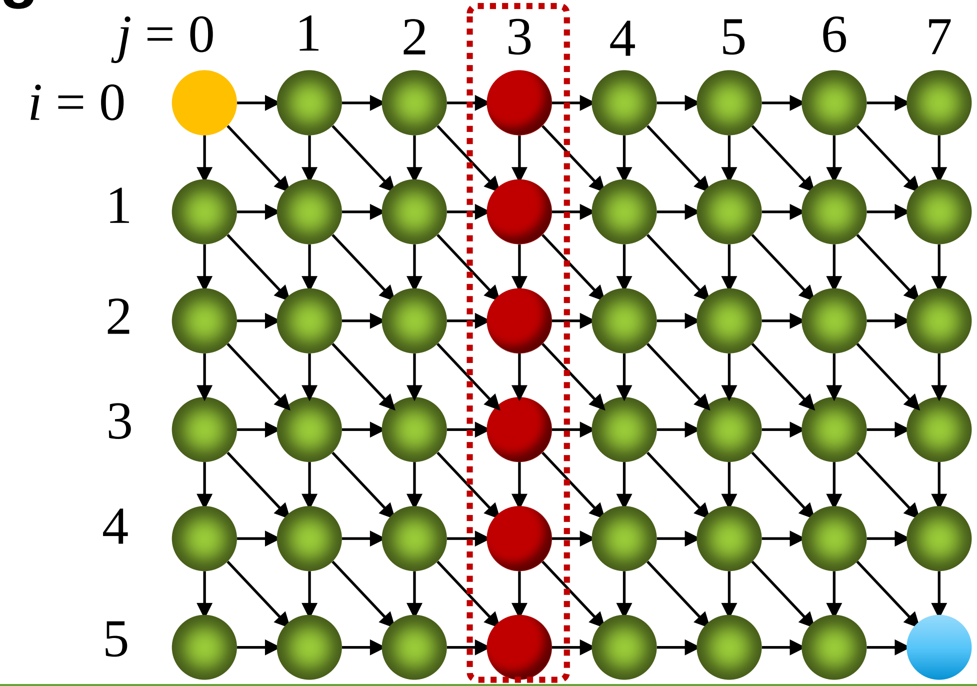


Shortest Path Problem

$F(i,j)$: length of the shortest path from $(0{,}0)$ to (i,j)
$B(i,j)$: length of the shortest path from (i,j) to (m,n)

- Observation 2: for any v in $\{0,\dots,n\}$, there exists a u s.t. the shortest path between $(0{,}0)$ and (m,n) goes through (u,v)

→ **the shortest path must go across a vertical cut**



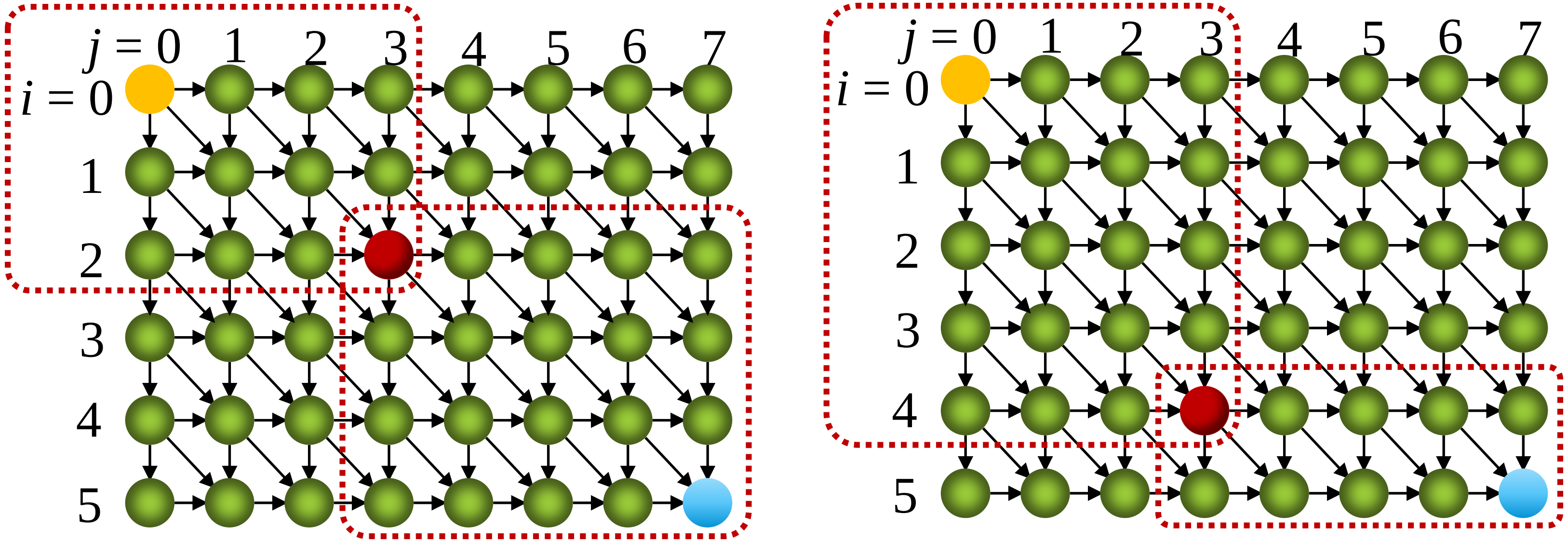
Shortest Path Problem

$F(i,j)$: length of the shortest path from $(0,0)$ to (i,j)
$B(i,j)$: length of the shortest path from (i,j) to (m,n)

- Observation 1+2:

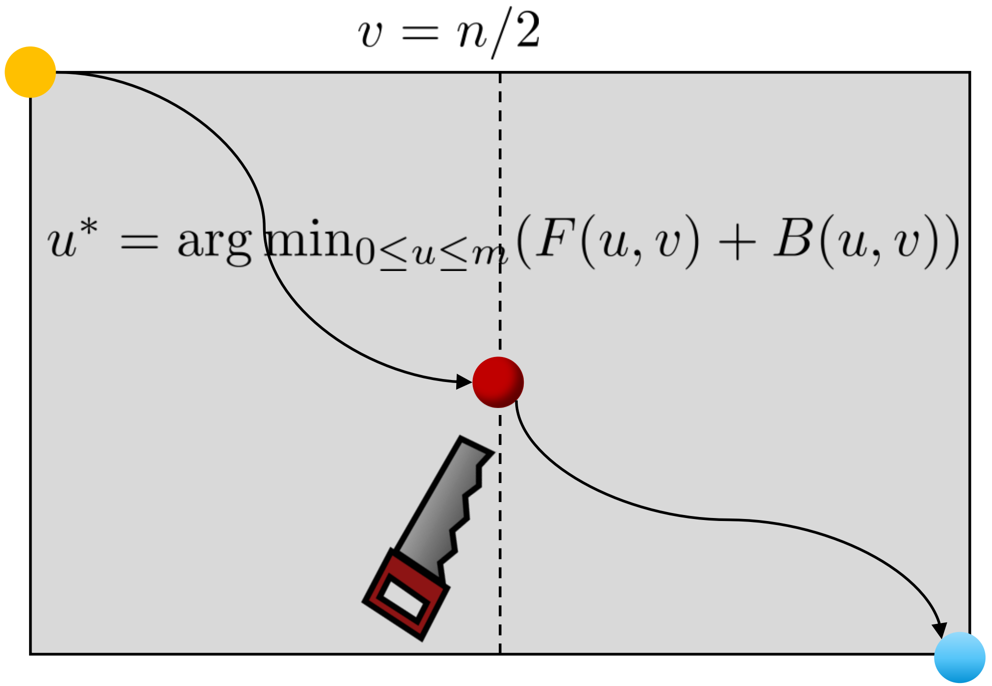
$$F(m,n) = \min\left(F(0,v) + B(0,v), F(1,v) + B(1,v), \cdots, F(m,v) + B(m,v)\right)$$

$$F(m,n) = \min_{0 \le u \le m} F(u,v) + B(u,v) \forall v$$



Divide-and-Conquer Algorithm

- Goal: finds optimal solution



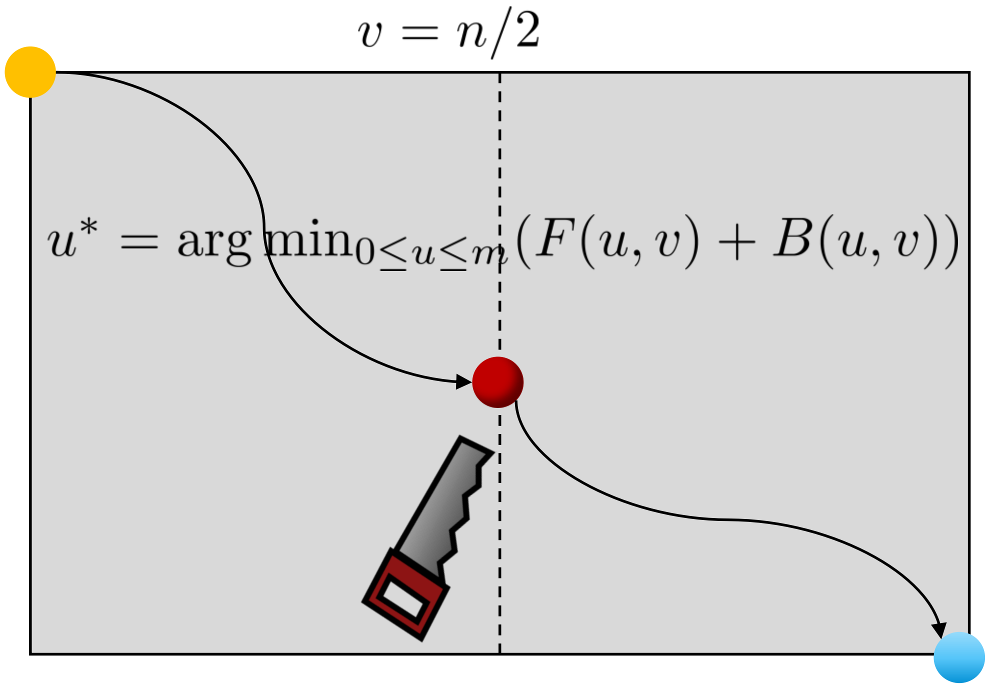
How to find the value of u^*?

- Idea: utilize sequence alignment algo.
  - Call Space-Seq-Align(X,Y[1:v]) to find $F(0,v), F(1,v), \dots, F(m,v)$
    $\Theta(m \times \frac{n}{2})$
  - Call Back-Space-Seq-Align(X,Y[v+1:n]) to find $B(0,v), B(1,v), \dots, B(m,v)$
    $\Theta(m \times \frac{n}{2})$
  - Let u be the index minimizing $F(u,v) + B(u,v)$
    $\Theta(m)$

Divide-and-Conquer Algorithm

- Goal: finds optimal solution – `DC-Align(X, Y)`

Space Complexity: $O(m+n)$



- $T(m,n)$ = time for running `DC-Align(X, Y)` with $|X| = m, |Y| = n$

1. Divide
2. Conquer
3. Combine

- Divide the sequence of size n into 2 subsequences
  - Find u to minimize $F(u,v) + B(u,v)$
- Recursive case ($n > 1$) $\Theta(mn)$
  - `prefix` $T(u, \frac{n}{2})$
    `= DC-Align(X[1:u], Y[1:v])`
  - `suffix` $T(m-u, \frac{n}{2})$
    `= DC-Align(X[u+1:m], Y[v+1:n])`
- Base case ($n = 1$)
  - Return `Seq-Align(X, Y)` $\Theta(m)$
- Return `prefix + suffix` $\Theta(1)$

$$T(m,n) = \begin{cases} O(m) & \text{if } n = 1 \\ T(u, n/2) + T(m-u, n/2) + O(mn) & \text{if } n \ge 2 \end{cases} \Rightarrow T(m,n) = O(mn)$$

Time Complexity Analysis

- Theorem

$$T(m,n) = \begin{cases} O(m) & \text{if } n = 1 \\ T(u, n/2) + T(m-u, n/2) + O(mn) & \text{if } n \geq 2 \end{cases} \Rightarrow T(m,n) = O(mn)$$

- Proof
  - There exists positive constants a, b s.t. all

$$T(m,n) \leq \begin{cases} a \cdot m & \text{if } n = 1 \\ T(u, n/2) + T(m-u, n/2) + b \cdot mn & \text{if } n \geq 2 \end{cases}$$

  - Use induction to prove $T(m,n) \leq kmn$

Practice to check the initial condition

$$T(m,n) \leq T(u, \tfrac{n}{2}) + T(m-u, \tfrac{n}{2}) + b \cdot mn$$

Inductive hypothesis

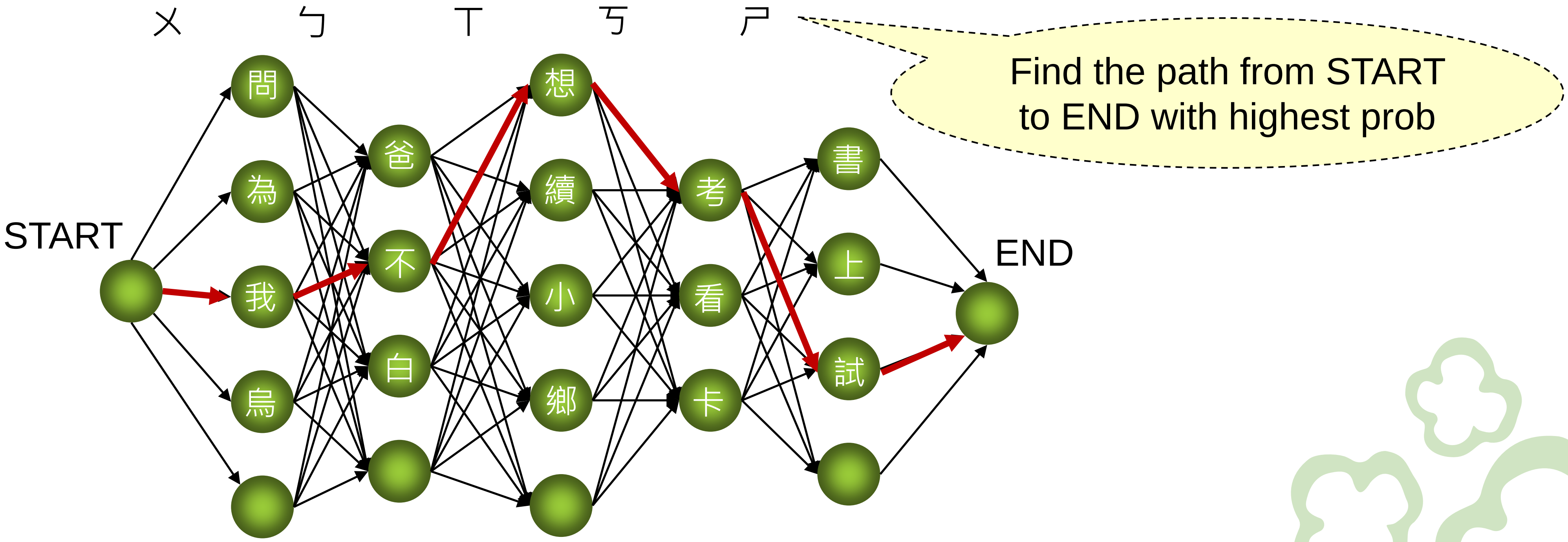
$$\leq ku\tfrac{n}{2} + k(m-u)\tfrac{n}{2} + b \cdot mn$$

$$\leq (\tfrac{k}{2} + b)mn$$

$$\leq kmn \text{ when } k \geq 2b$$

Extension: 注音文 Recognition

- Given a graph $G = (V, E)$, each edge $(u, v) \in E$ has an associated non-negative probability $p(u, v)$ of traversing the edge (u, v) and producing the corresponding character. Find the <u>most probable path</u> with the label $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$.

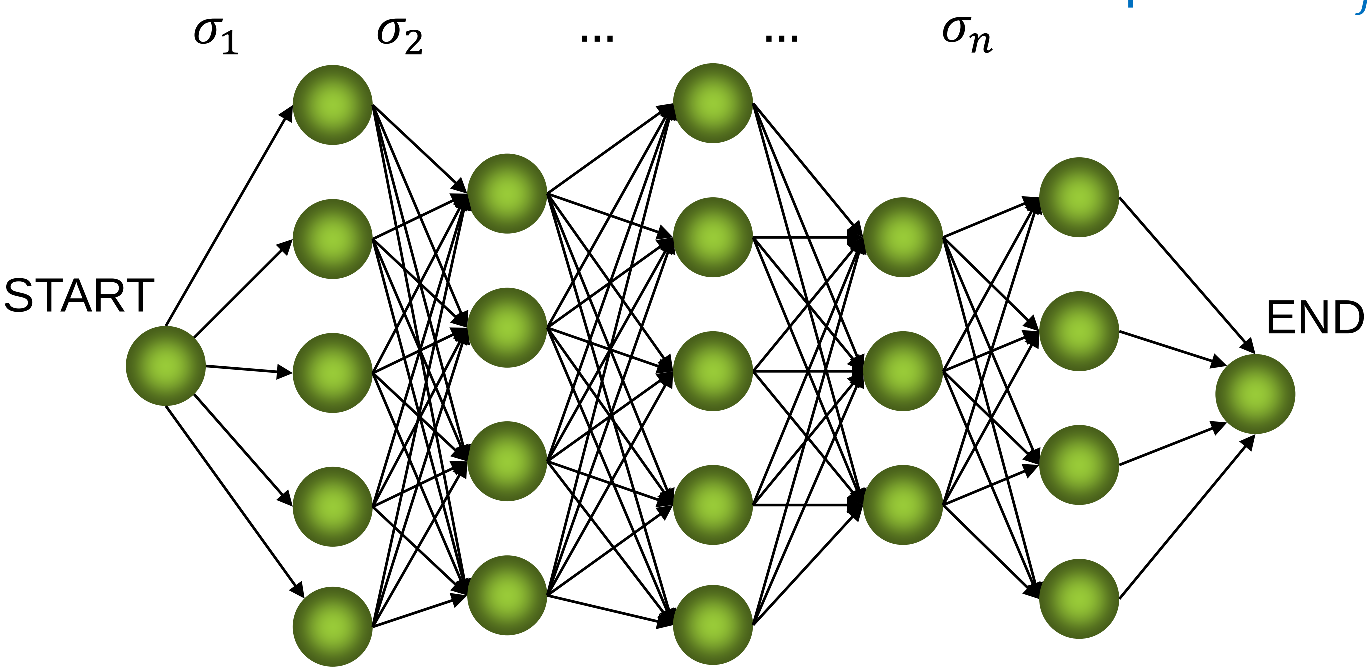


Viterbi Algorithm

produce σ_1

$$P_{i,j} = \begin{cases} p(\text{START}, v) & \text{if } j = 1 \\ \max_k (P_{k,j-1} \times p(u, v)) & \text{otherwise} \end{cases}$$

produce σ_j



V: vocabulary size

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

Viterbi has been applied to many AI applications, e.g. speech recognition

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