Algorithm Design and Analysis
Dynamic Programming (2)

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?

囚犯問題中最高存活人數的期望值為何？

ⓘ Start presenting to display the poll results on this slide.

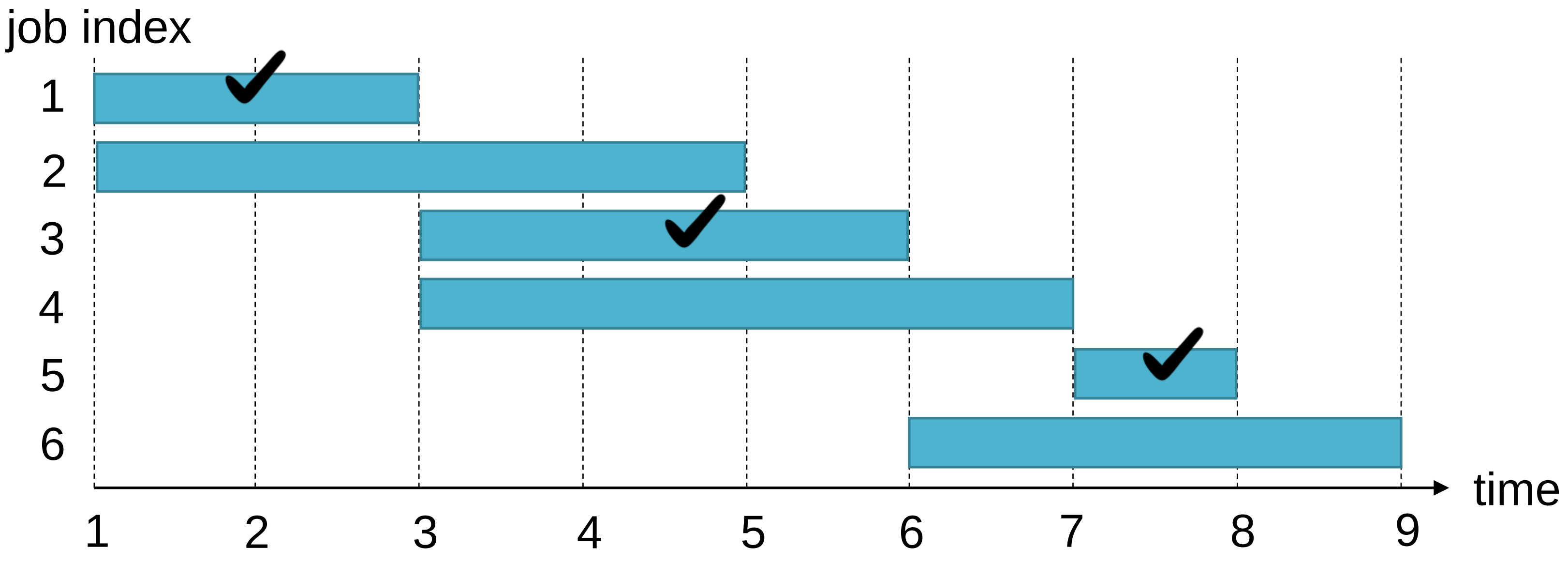
DP#4: Weighted Interval Scheduling

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

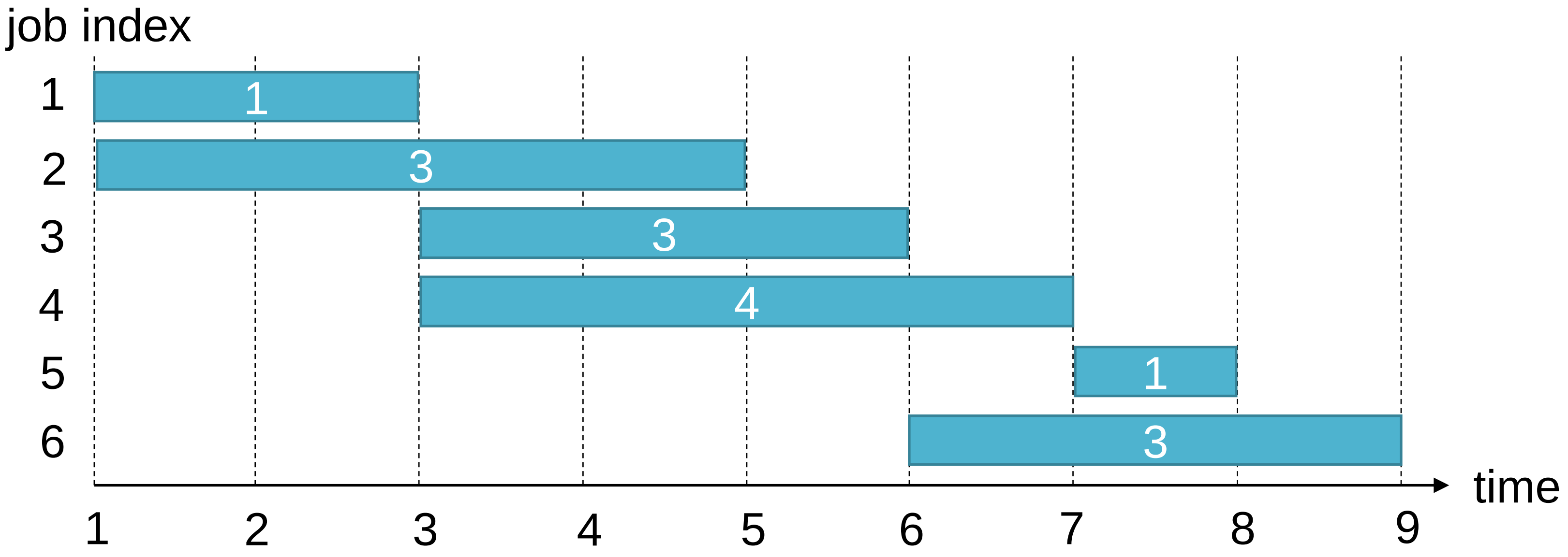
"Greedy Algorithm"
Next topic!



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

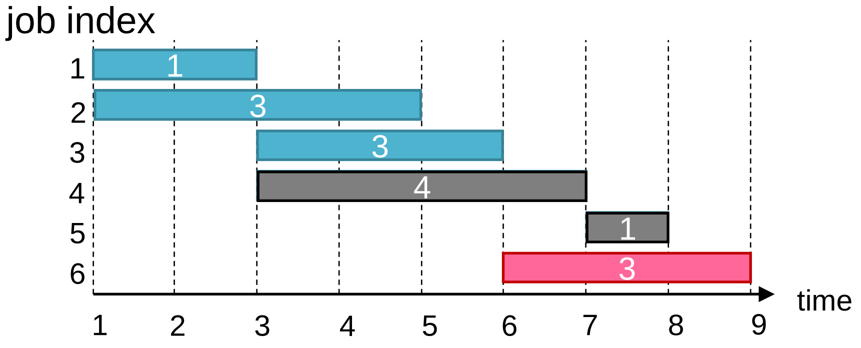
Assume that the requests are sorted in non-decreasing order ($f_i \le f_j$ when $i < j$)
$p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
e.g. $p(1) = 0$, $p(2) = 0$, $p(3) = 1$, $p(4) = 1$, $p(5) = 4$, $p(6) = 3$



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]								$\boxed{}$

```
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

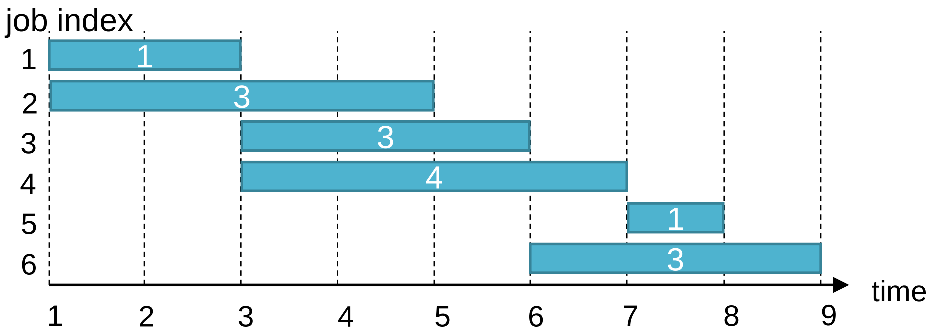
Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the maximum total value obtainable from compatible

- 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]	0	1	3	4	5	6	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
```

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

slido

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.

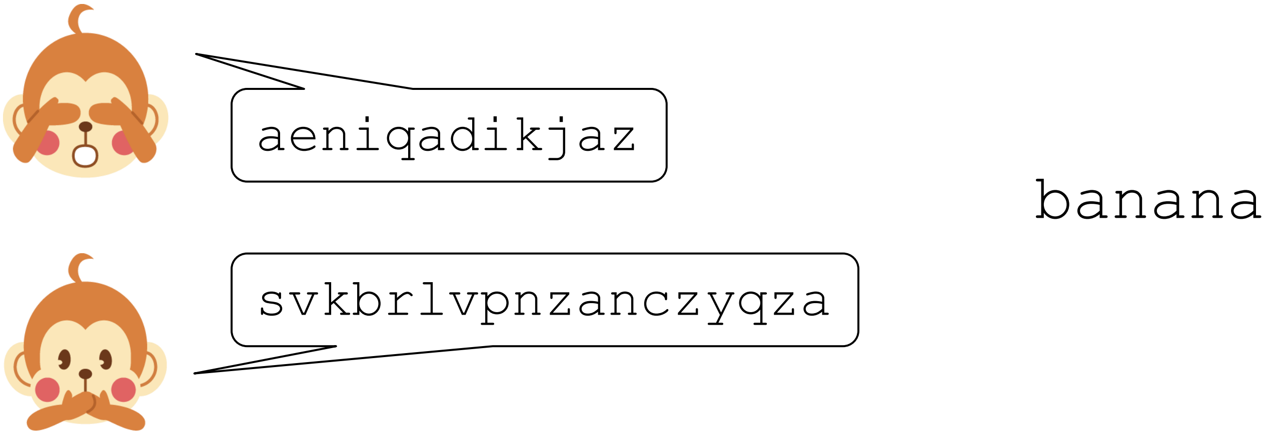
DP#5: Sequence Alignment

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?



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: longest common subsequence 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

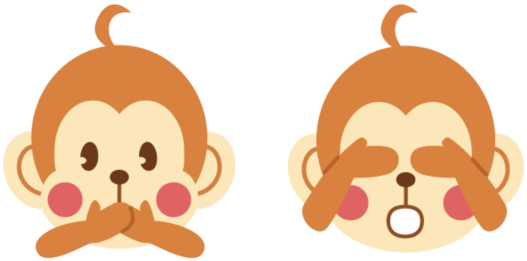
$X \rightarrow$ ---ba---n-an------a
$Y \rightarrow$ svkbrlvpnzanczyqza

13

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 $x_1, \dots, x_i$ and $y_1, \dots, y_j$
  - 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								$\boxed{}$

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

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

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

```
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								

$\rightarrow \Theta(n)$

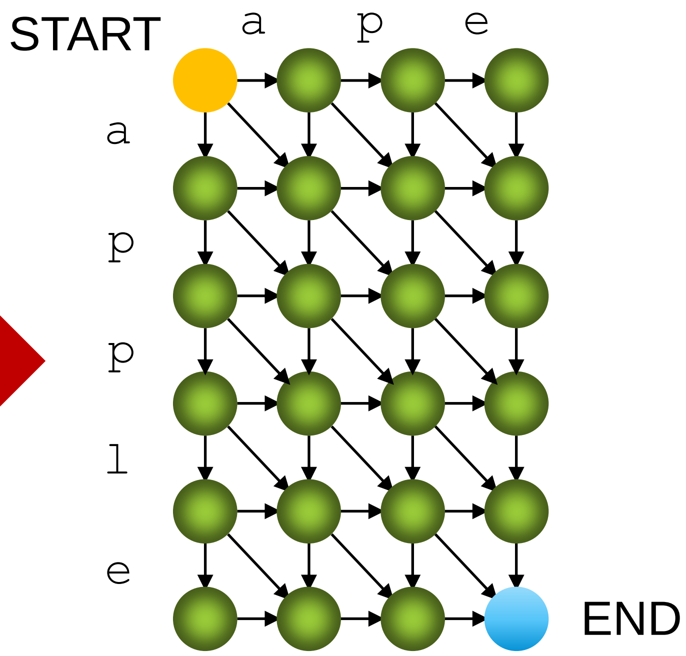
The optimal value can be computed, but the solution cannot be reconstructed

Space-Efficient Solution

Divide-and-Conquer
+
Dynamic Programming

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

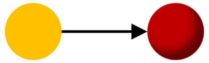
		a	p	e
X\Y	0	1	2	3
0	0	4	8	12
a 1	4	7	11	15
p 2	8	4	8	12
p 3	12	8	12	8
l 4	16	12	15	12
e 5	20	16	19	15

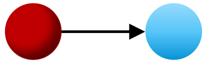


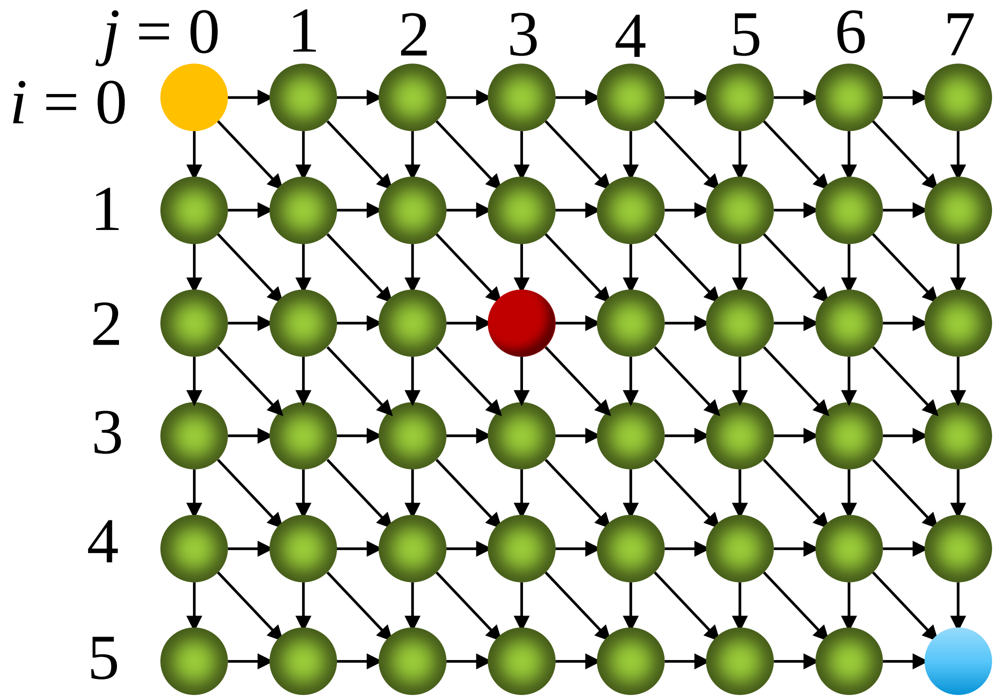
→ distance = C_{INS}
↓ distance = C_{DEL}
↘ distance = $C_{u,v}$ for edge (u, v)

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 



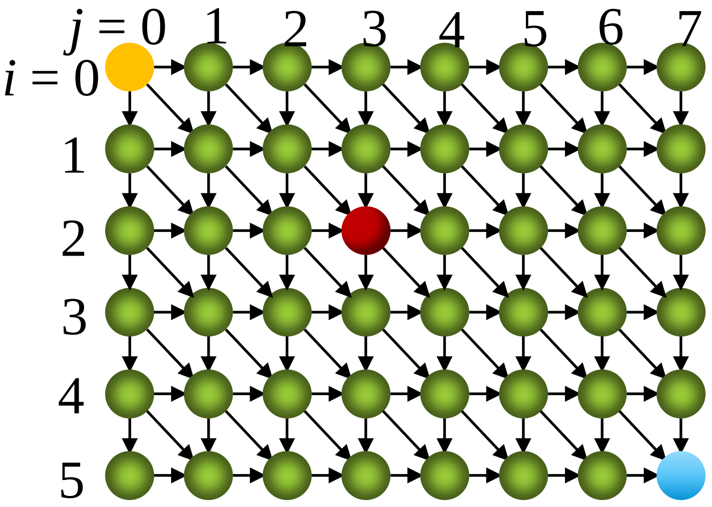
Recursive Equation

- 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

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

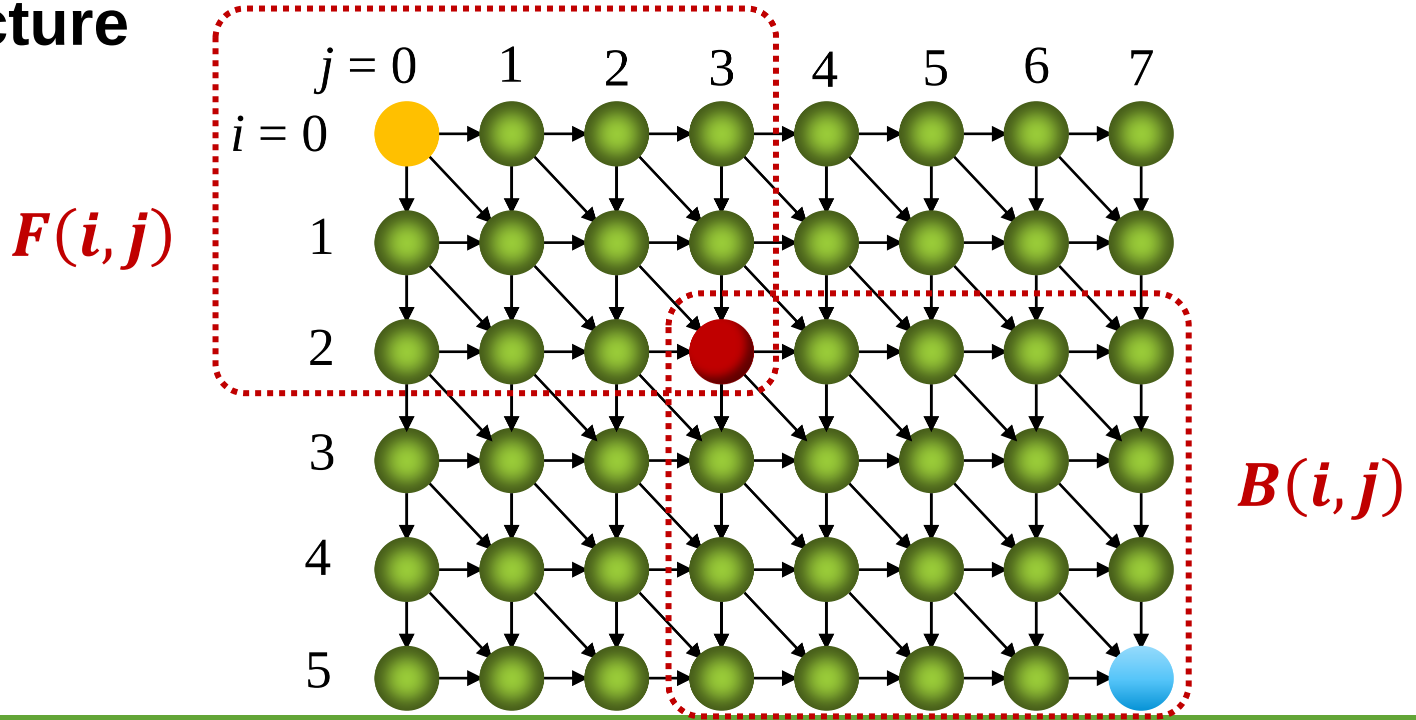


Shortest Path Problem

$F(i,j)$: 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**

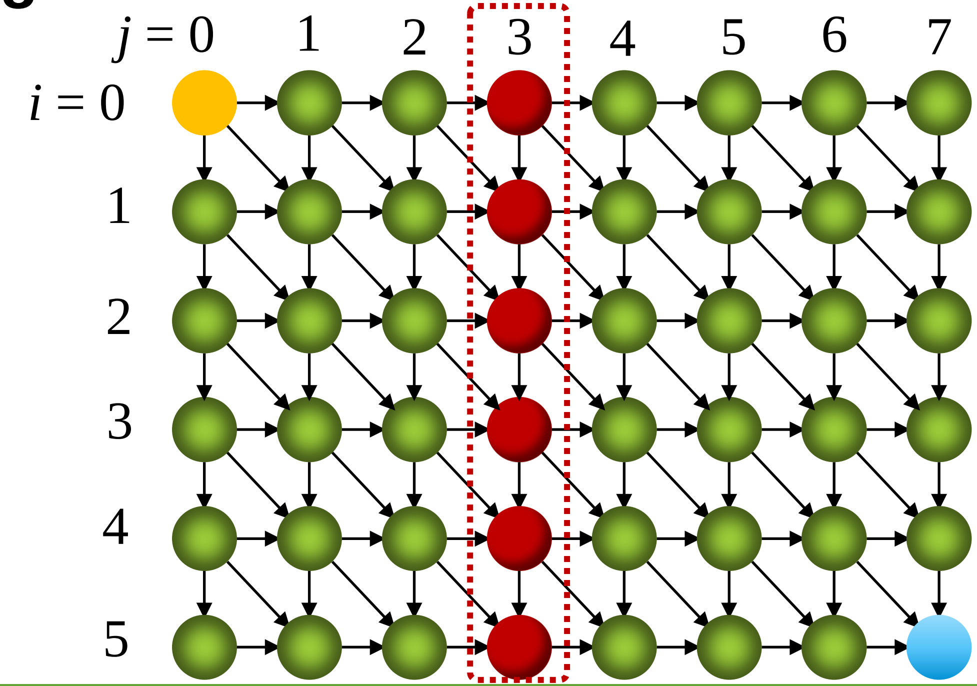


Shortest Path Problem

$F(i,j)$: 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**



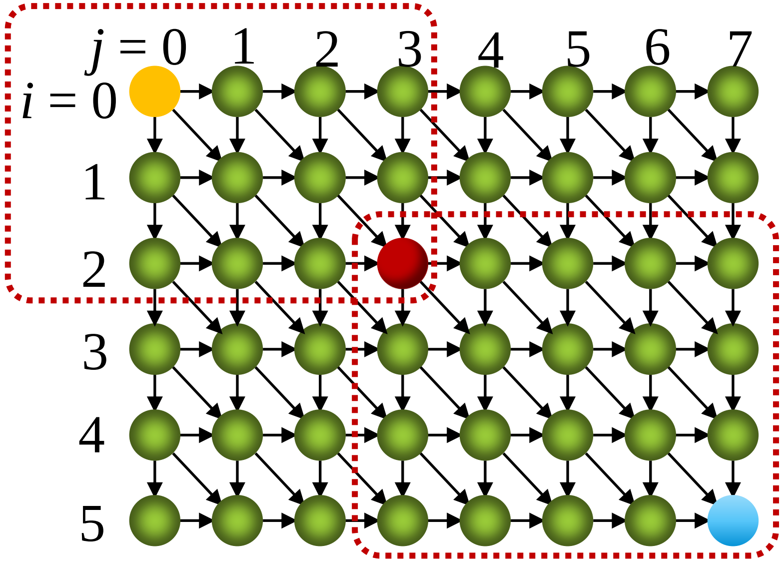
Shortest Path Problem

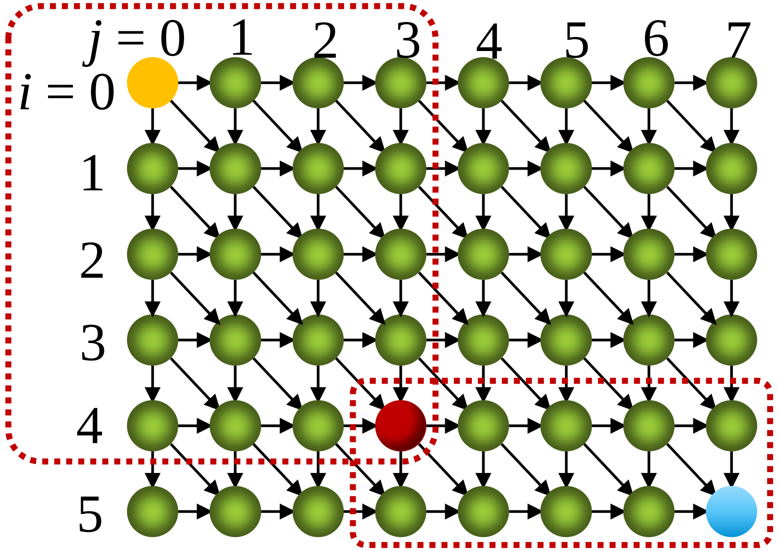
$F(i,j)$: length of the shortest path from $(0{,}0)$ to (i,j)
$B(i,j)$: length of the shortest path from (i,j) to (m,n)

- Observation 1+2:

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

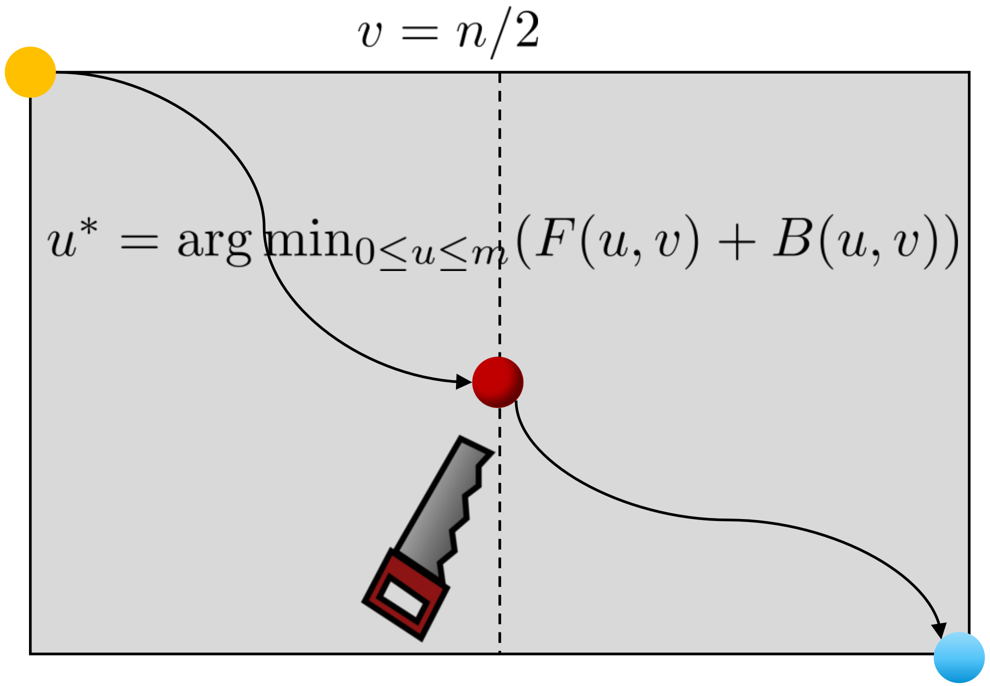
$$F(m,n) = \min_{0 \le u \le m} F(u,v) + B(u,v) \forall v$$





Divide-and-Conquer Algorithm

- Goal: finds optimal solution



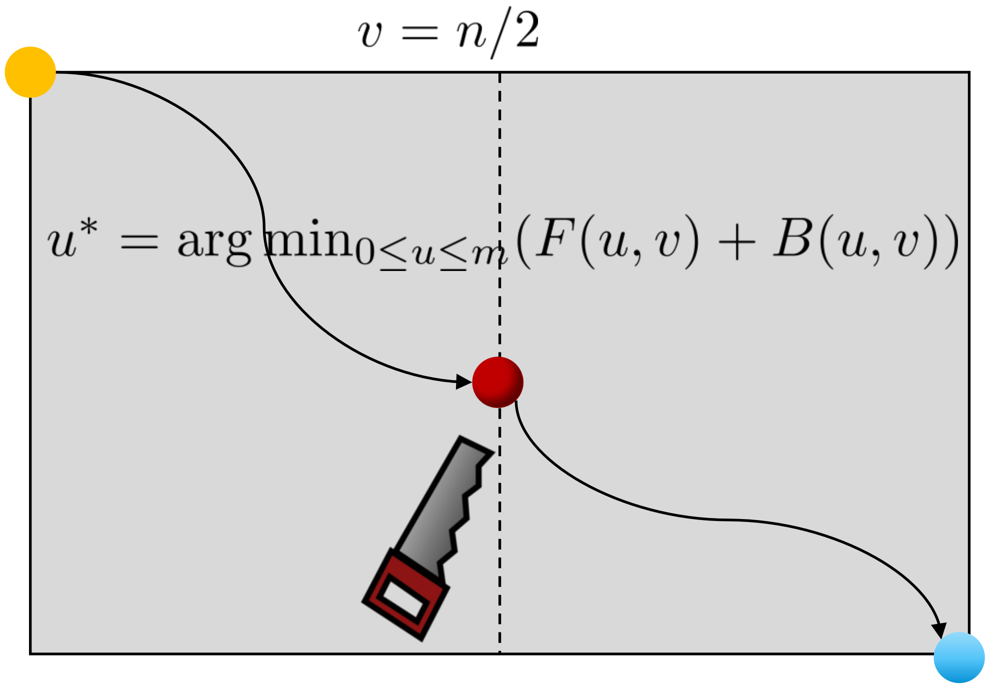
How to find the value of u^*?

- Idea: utilize sequence alignment algo.
  - Call `Space-Seq-Align(X,Y[1:v])` to find $F(0,v), F(1,v), \ldots, F(m,v)$
    $\Theta(m \times \frac{n}{2})$
  - Call `Back-Space-Seq-Align(X,Y[v+1:n])` to find $B(0,v), B(1,v), \ldots, B(m,v)$
    $\Theta(m \times \frac{n}{2})$
  - 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 $T(m,n) \leq kmn$

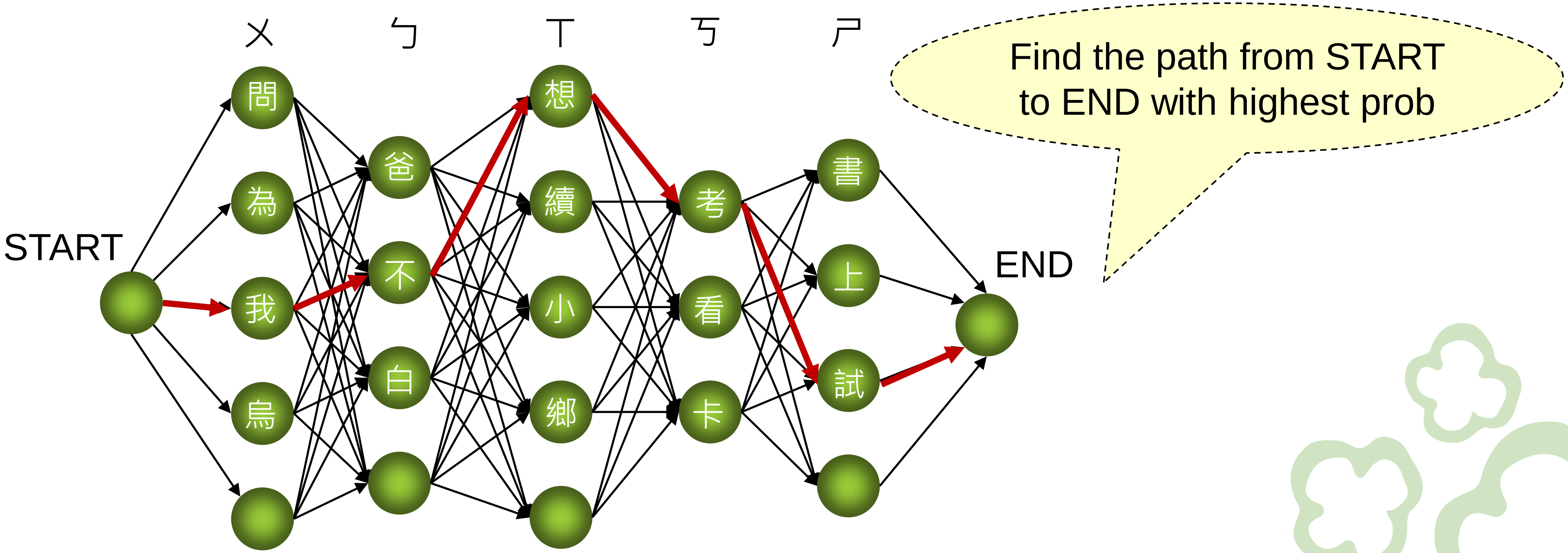
Practice to check the initial condition

$$\begin{aligned} T(m,n) &\leq T(u, \tfrac{n}{2}) + T(m-u, \tfrac{n}{2}) + b \cdot mn \\ &\leq ku\tfrac{n}{2} + k(m-u)\tfrac{n}{2} + b \cdot mn \\ &\leq (\tfrac{k}{2} + b)mn \\ &\leq kmn \text{ when } k \geq 2b \end{aligned}$$

Inductive hypothesis

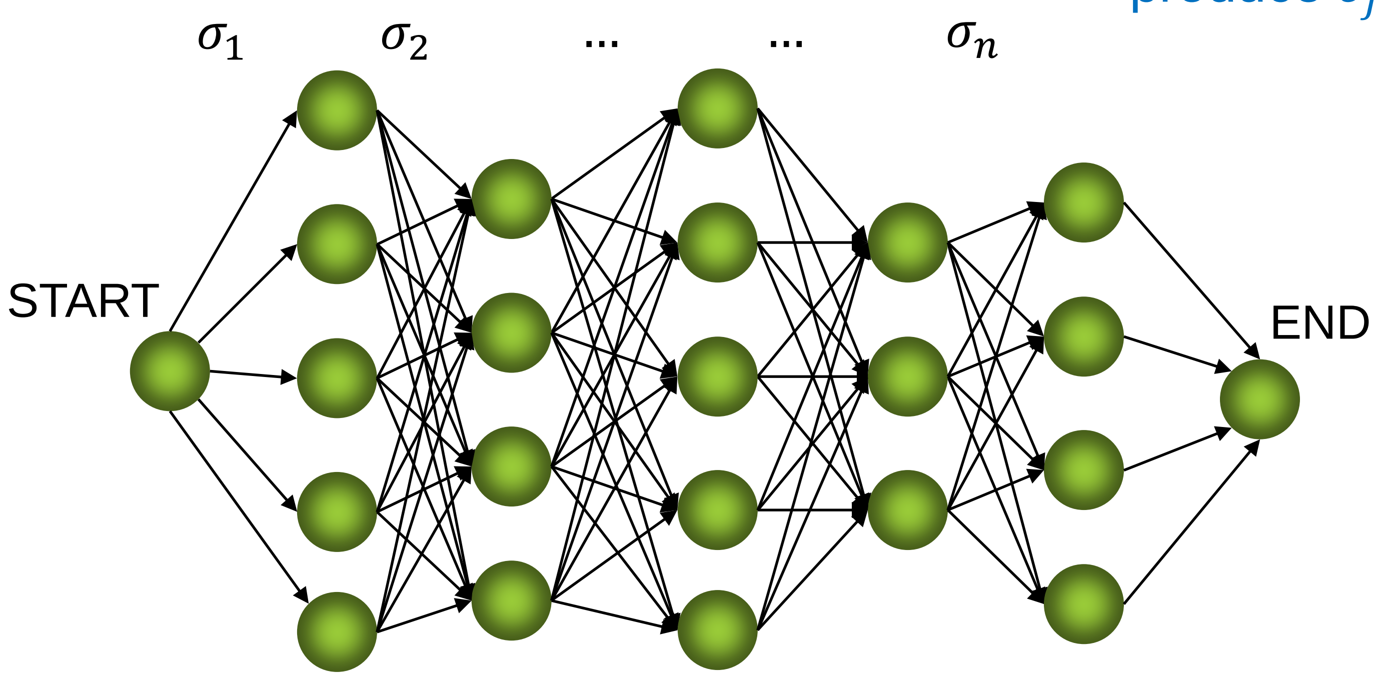
Extension: 注音文 Recognition

- 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 $s = \langle \sigma_1, \sigma_2, \dots, \sigma_n \rangle$.



Viterbi Algorithm

$$P_{i,j} = \begin{cases} \overbrace{p(\text{START}, v)}^{\text{produce } \sigma_1} & \text{if } j = 1 \\ \max_k (P_{k,j-1} \times \underbrace{p(u, v)}_{\text{produce } \sigma_j}) & \text{otherwise} \end{cases}$$



V: vocabulary size

➡ $T(n) = \Theta(Vn)$

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

slido

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.

DP#6: Knapsack (背包問題)

Textbook Exercise 16.2-2

Knapsack Problem

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

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

Knapsack Problem

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

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

Step 1: Characterize an OPT Solution

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

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

Step 2: Recursively Define the Value of an OPT Solution

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

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

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

  - Case 2: item i not in OPT
    - OPT is an optimal solution of `ZO-KP(i - 1, w)`

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

- Recursively define the value

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

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

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

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

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

$W = 5$

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

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

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

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

Step 4: Construct an OPT Solution by Backtracking

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

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

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

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

Knapsack Problem

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

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

Step 1: Characterize an OPT Solution

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

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

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

Can we do better?

Step 1: Characterize an OPT Solution

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

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

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

Step 2: Recursively Define the Value of an OPT Solution

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

- Optimal substructure: suppose OPT is an optimal solution to `U-KP(w)`, there are n cases:
  - Case i: item i in OPT
    - Removing an item i from OPT is an optimal solution of `U-KP(w - w₁)` $M_w = v_i + M_{w-w_i}$
- Recursively define the value

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

只考慮背包還裝的下的情形

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

w	0	1	2	3	4	5	…	W
M[w]								

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

$W = 5$

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

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

$\max(4+0)$

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

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

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

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

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

$W = 5$

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

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

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

Step 4: Construct an OPT Solution by Backtracking

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

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

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

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

Knapsack Problem

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

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

Step 1: Characterize an OPT Solution

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

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

Step 2: Recursively Define the Value of an OPT Solution

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

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

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

  - Case 2: item i not in OPT
    - OPT is an optimal solution of `M-KP(i - 1, w, d)`

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

- Recursively define the value

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

Exercise

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

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

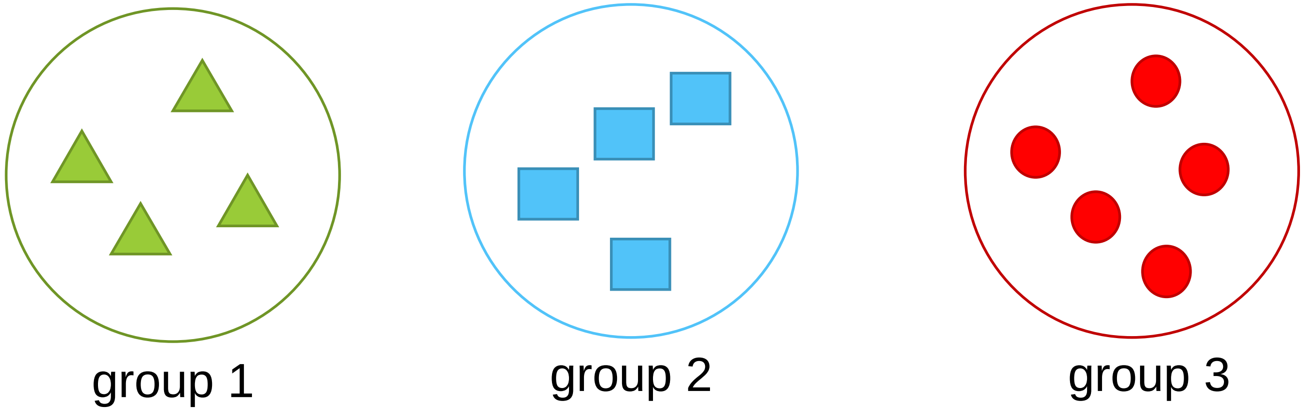
Knapsack Problem

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

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

Multiple-Choice Knapsack Problem

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



Step 1: Characterize an OPT Solution

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

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

Which one is more suitable for this problem?

Which one is more suitable for the multiple-choice knapsack problem?

ⓘ Start presenting to display the poll results on this slide.

Step 1: Characterize an OPT Solution

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

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

Which one is more suitable for this problem?

Step 1: Characterize an OPT Solution

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

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

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

Step 2: Recursively Define the Value of an OPT Solution

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

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

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

    $M_{i,w} = v_{i,j} + M_{i-1,w-w_{i,j}}$
- Recursively define the value

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

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

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

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

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

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

Step 4: Construct an OPT Solution by Backtracking

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

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

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

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

Knapsack Problem

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

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

Fractional Knapsack Problem

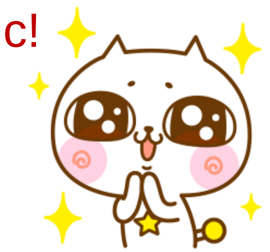
- Input: n items where i-th item has value v_i and weighs w_i (v_i and w_i are positive integers)
- Output: the maximum value for the knapsack with capacity of W, where we can take **any fraction of items**

- Dynamic programming algorithm should work

Can we do better?

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

"Greedy Algorithm"
Next topic!



Pseudo-Polynomial

Pseudo-Polynomial Time

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

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

https://youtu.be/9oI7fg-MIpE

Time Complexity Definition

- Time complexity is in measure the time an algorithm takes to run as a function of
  - ✔ the length of the input in bits
  - ✘ the value of the input

```
function(n)
  for i = 1 to n
    print i
```

$$O(n)$$
$$= O\left(2^{\text{bits in } n}\right)$$
$$= O(2^m)$$

- n is a value

$n = 4 = \underbrace{100}_{3}$

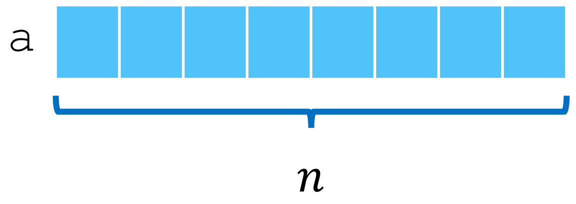
$n = 8 = \underbrace{1000}_{4}$

$n = 16 = \underbrace{10000}_{5}$

```
function(a)
  for i = 1 to n
    print i
```

$$O(n)$$

- a is an array



a = $\underbrace{\texttt{[0, 1, 1]}}_{3}$

a = $\underbrace{\texttt{[0, 1, 0, 1]}}_{4}$

a = $\underbrace{\texttt{[0, 1, 0, 1, 1]}}_{5}$

Time Complexity Definition

- Time complexity is in measure the time an algorithm takes to run as a function of
  - ✔ the length of the input in bits
  - ✘ the value of the input

```
function(n)
  for i = 1 to n
    print i
```

- `n` is a value

$$O(n) = O(2^{\text{bits in } n}) = O(2^m)$$

```
function(a)
  for i = 1 to n
    print i
```

- `a` is an array

$$O(n)$$

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

$$= \Theta(n2^{\text{bits in } W}) = O(n2^m)$$

= exponential in the length of the input
= polynomial in the numeric value
= pseudo-polynomial in input size

Concluding Remarks

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

slido

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.

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