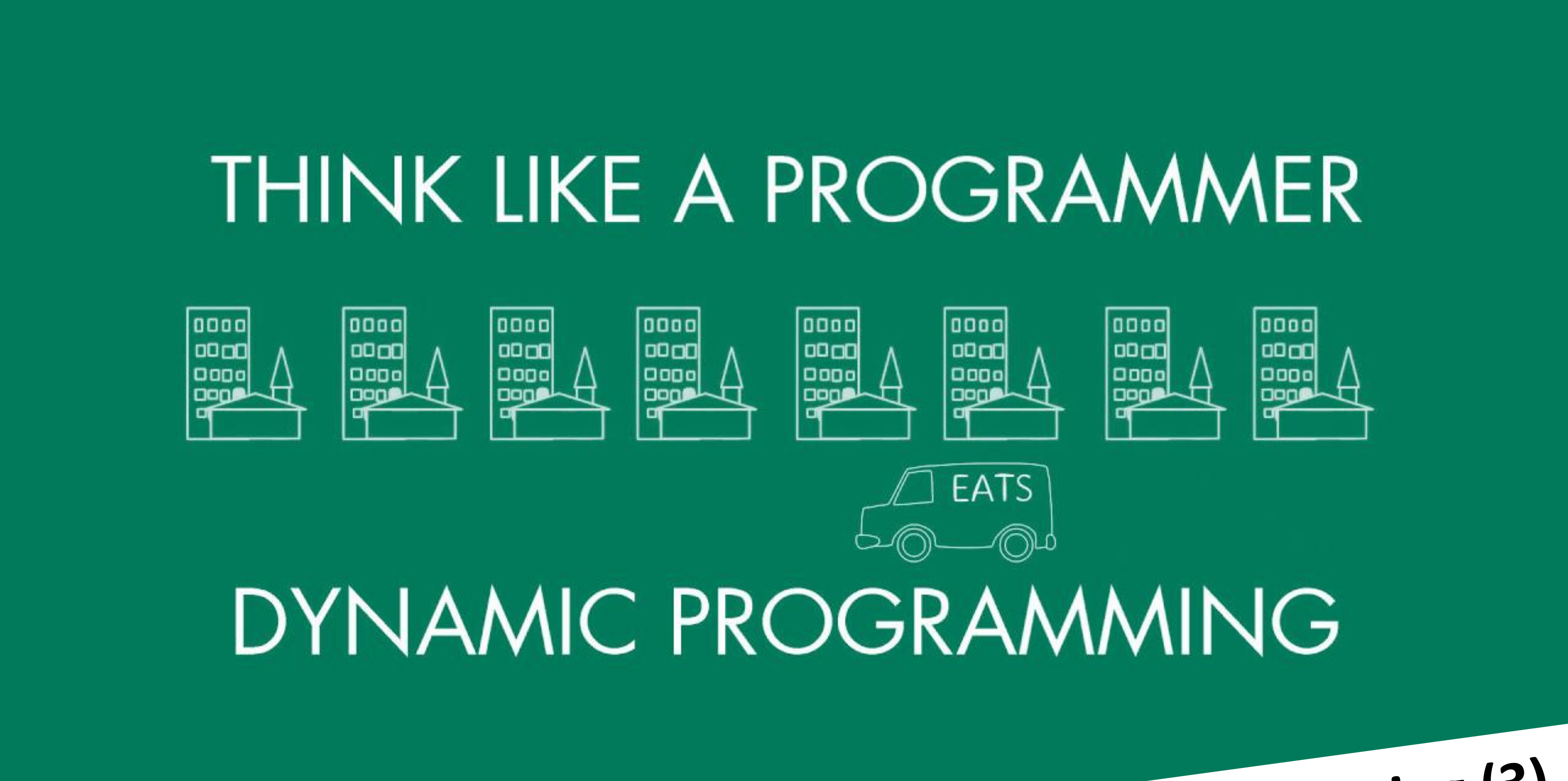


Dynamic Programming (3)

Oct 18th, 2018

Algorithm Design and Analysis

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

國立臺灣大學 National Taiwan University

Slides credited from Hsu-Chun Hsiao

Announcement

- Homework 1 due
- Mini-HW 5 released
  - Due on 10/25 (Thu) 14:20
- Homework 2 released
  - Due on 11/06 (Tue) 18:00 (3.5 weeks)
  - A4 hardcopy submitted to a box @R307
  - Softcopy submitted to NTU COOL before the deadline

Frequently check the website for the updated information!

Mini-HW 5

Consider the classical Sequence Alignment problem. In this mini HW, we want to use ***dynamic programming*** to find the minimal cost for aligning two sequence. The cost of deletion, insertion and substitution are 1, 1, 2 respectively.

(1) Consider two strings s1="ABCADB" and s2="CABDAB". Please fill the DP table below and tell me the distance between s1 and s2. For example, the 1 in the table means the edit distance of "AB" and "CAB" is 1 (50%)

S1 \ S2	#	C	A	B	D	A	B
#	0		2				
A							
B				1			
C							
A							
D	5						
B							

(2) Explain how you fill the DP table. (50%)

(You can briefly explain or write down the math equation of recurrence relation)

Outline

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

動腦一下 – 囚犯問題

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

猜測規則

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

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

有沒有更多人可以存活的好策略?

Vote for Your Answer

Rapidpoll

囚犯問題中最高的存活人數期望值為何？

SWALLOW0130

75 ~ 79	80~89	90~95	96~100
0%	0%	0%	0%

0 vote

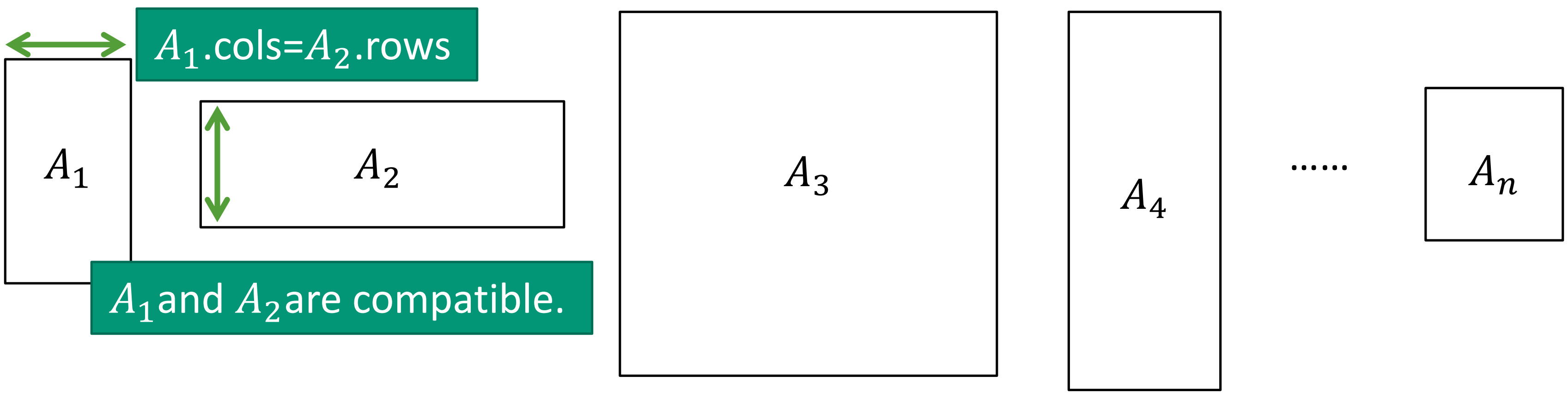
8

DP#4: Matrix-Chain Multiplication

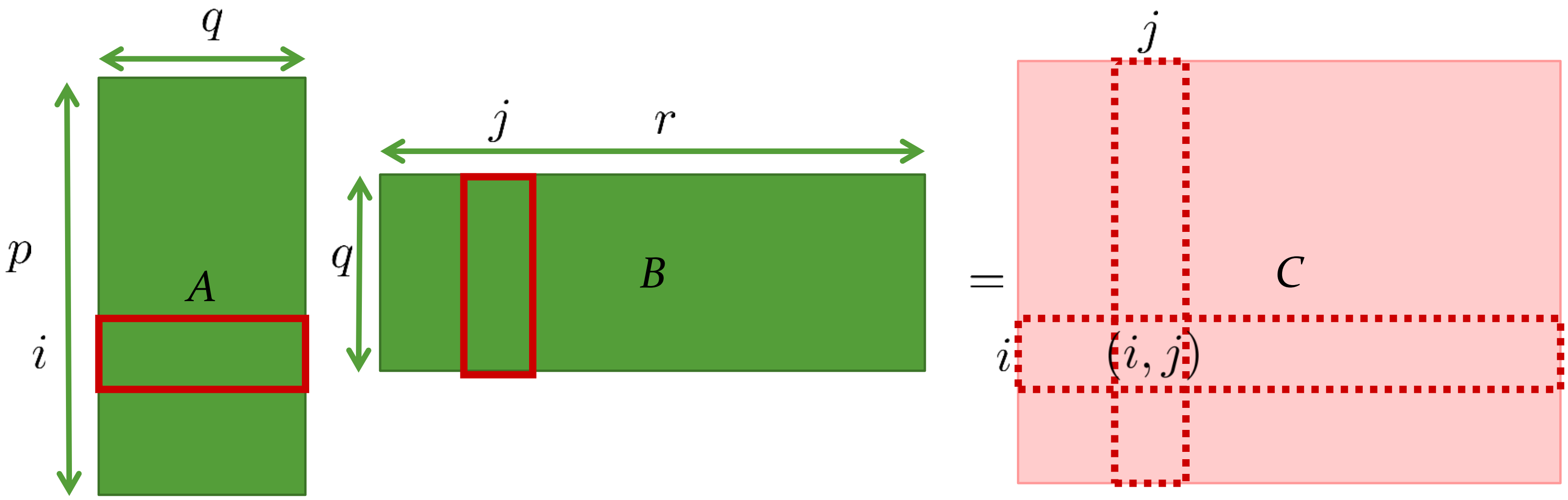
Textbook Chapter 15.2 – Matrix-chain multiplication

Matrix-Chain Multiplication

- Input: a sequence of *n* matrices $\langle A_1, \ldots, A_n \rangle$
- Output: the product of $A_1 A_2 \ldots A_n$



Observation



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

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

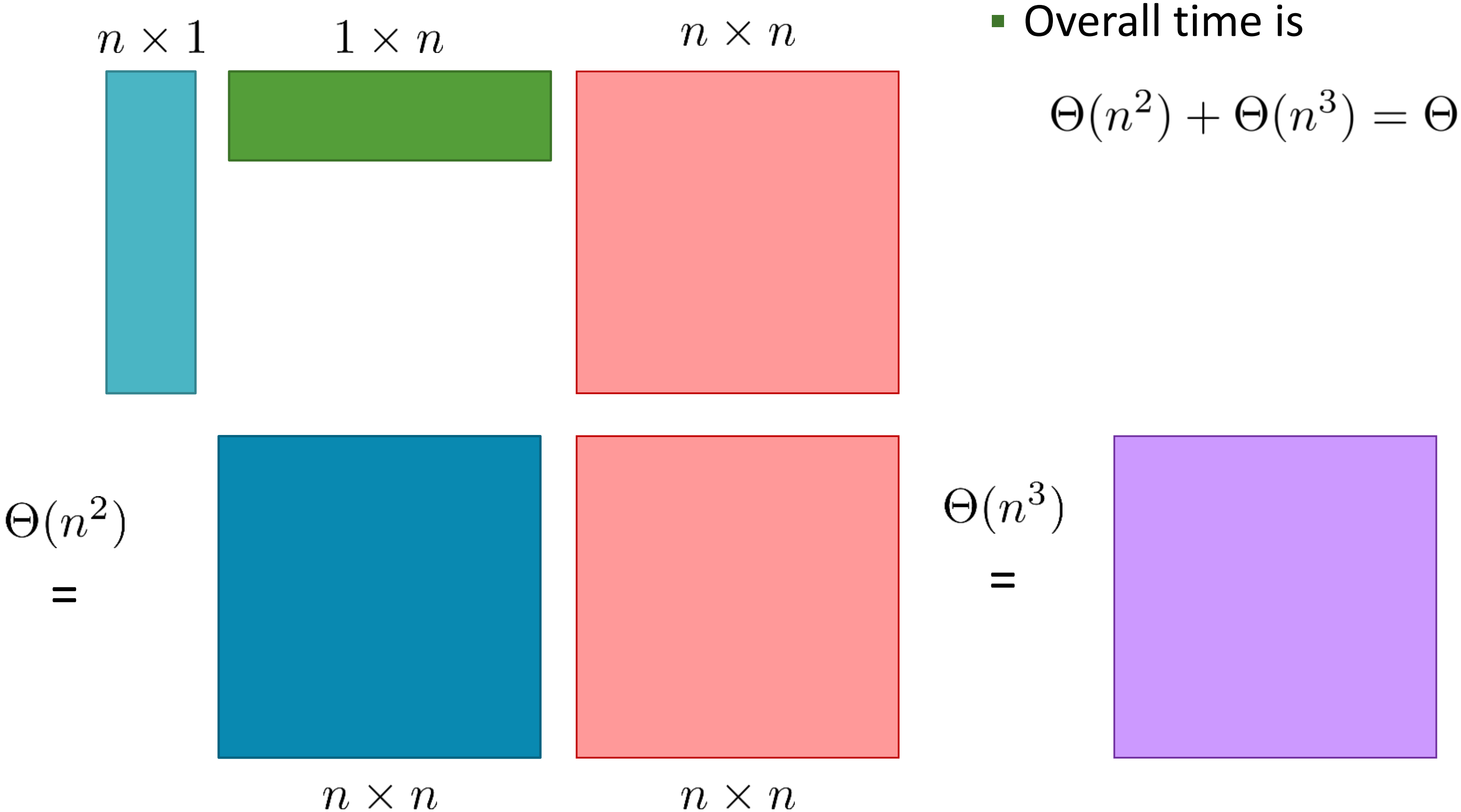
$\Rightarrow \Theta(q)\Theta(pr) = \Theta(pqr)$

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

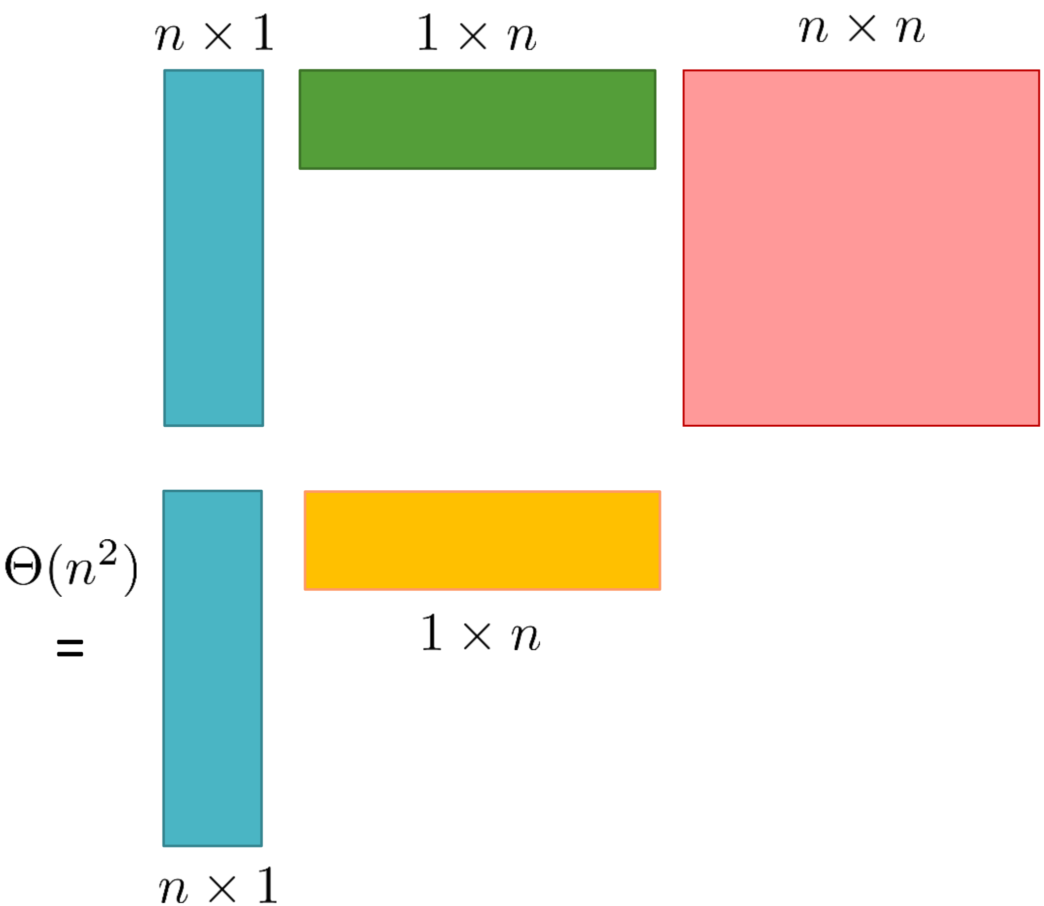
Example

- Overall time is

$$\Theta(n^2) + \Theta(n^3) = \Theta(n^3)$$

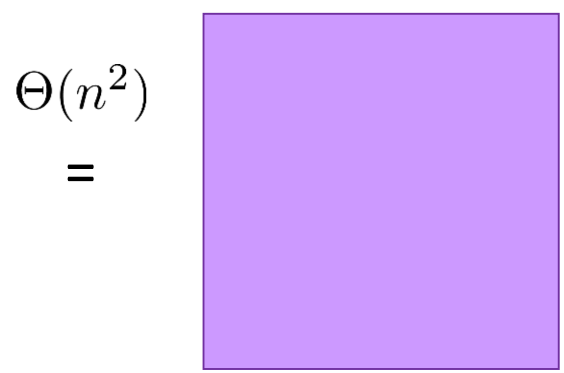


Example



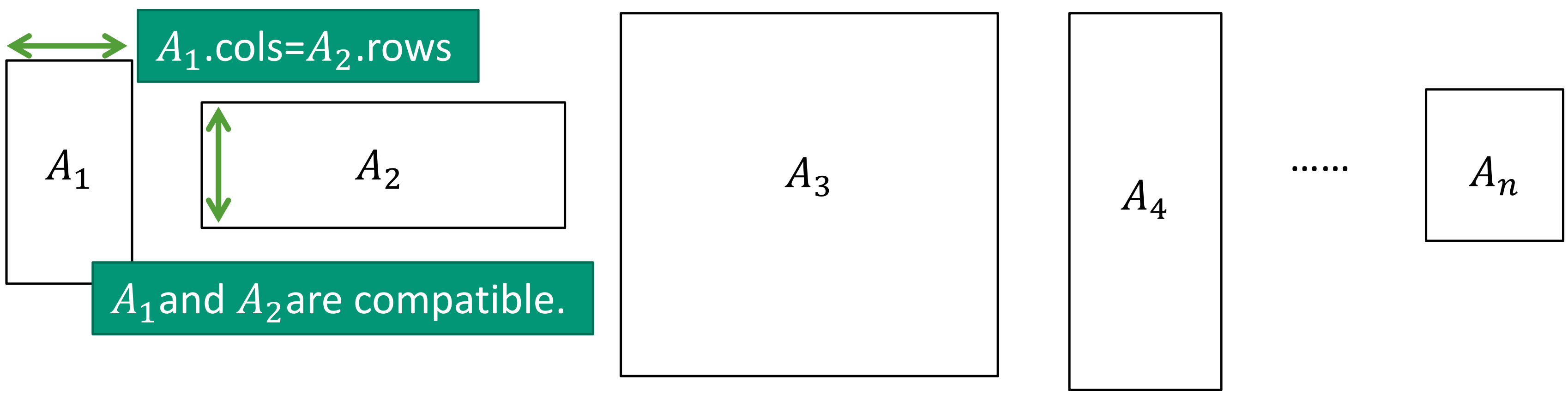
- Overall time is

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



Matrix-Chain Multiplication Problem

- Input: a sequence of integers $l_0, l_1, \dots, l_n$
  - l_{i-1} is the number of rows of matrix A_i
  - l_i is the number of columns of matrix A_i
- Output: a <u>order</u> of performing $n-1$ matrix multiplications in the minimum number of operations to obtain the product of $A_1 A_2 \dots A_n$



Do not need to compute the result but find the fast way to get the result!
(computing "how to fast compute" takes less time than "computing via a bad way")

Brute-Force Naïve Algorithm

- P_n: how many ways for n matrices to be multiplied

$$P_n = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P_k P_{n-k} & \text{if } n \geq 2 \end{cases}$$

$$(A_1 A_2 \cdots A_k)\,(A_{k+1} A_{k+2} \cdots A_n)$$

- The solution of P_n is Catalan numbers, $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$, or is also $\Omega(2^n)$ Exercise 15.2-3

Step 1: Characterize an OPT Solution

Matrix-Chain Multiplication Problem
Input: a sequence of integers $l_0, l_1, \ldots, l_n$ indicating the dimensionality of A_i
Output: a order of matrix multiplications with the minimum number of operations

- Subproblems
  - `M(i, j)`: the min #operations for obtaining the product of $A_i \ldots A_j$
  - Goal: `M(1, n)`
- Optimal substructure: suppose we know the OPT to `M(i, j)`, there are k cases:

$$i \le k < j$$

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

  - Case k: there is a cut right after A_k in OPT

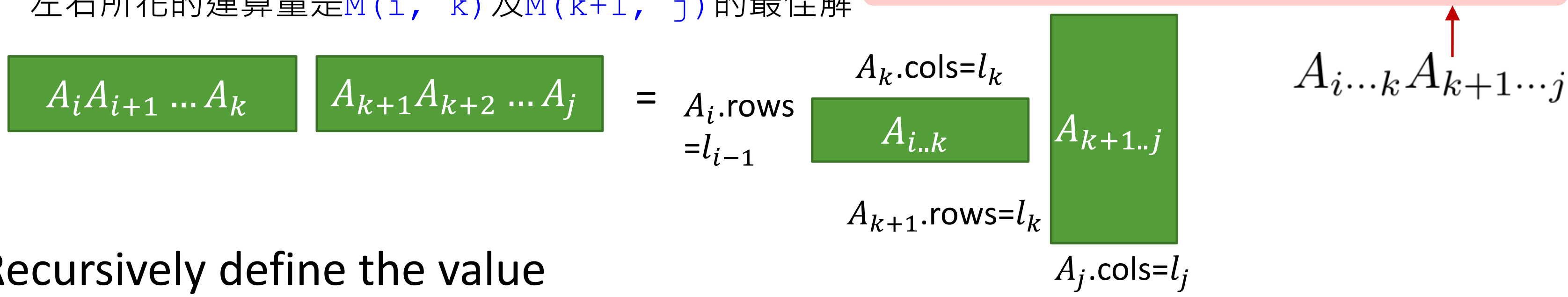
左右所花的運算量是`M(i, k)`及`M(k+1, j)`的最佳解

Step 2: Recursively Define the Value of an OPT Solution

Matrix-Chain Multiplication Problem
Input: a sequence of integers $l_0, l_1, \ldots, l_n$ indicating the dimensionality of A_i
Output: a order of matrix multiplications with the minimum number of operations

- Suppose we know the optimal solution to `M(i, j)`, there are k cases:
  - Case k: there is a cut right after A_k in OPT
    左右所花的運算量是`M(i, k)`及`M(k+1, j)`的最佳解

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



- Recursively define the value

$$M_{i,j} = \begin{cases} 0 & i \geq j \\ \min_{i \leq k < j} \left(M_{i,k} + M_{k+1,j} + l_{i-1} l_k l_j \right) & i < j \end{cases}$$

Step 3: Compute Value of an OPT Solution

Matrix-Chain Multiplication Problem
Input: a sequence of integers $l_0, l_1, \dots, l_n$ indicating the dimensionality of A_i
Output: a order of matrix multiplications with the minimum number of operations

- Bottom-up method: solve smaller subproblems first

$$M_{i,j} = \begin{cases} 0 & i \geq j \\ \min_{i \leq k < j} \left(M_{i,k} + M_{k+1,j} + l_{i-1} l_k l_j \right) & i < j \end{cases}$$

- How many subproblems to solve
  - #combination of the values i and j s.t. $1 \leq i \leq j \leq n$

$$T(n) = \underbrace{C_2^n}_{i \neq j} + \underbrace{n}_{i = j} = \Theta(n^2)$$

Step 3: Compute Value of an OPT Solution

```
Matrix-Chain(n, l)
  initialize two tables M[1..n][1..n] and B[1..n-1][2..n]
  for i = 1 to n
    M[i][i] = 0 // boundary case
  for p = 2 to n // p is the chain length
    for i = 1 to n − p + 1 // all i, j combinations
      j = i + p − 1
      M[i][j] = ∞
      for k = i to j − 1 // find the best k
        q = M[i][k] + M[k + 1][j] + l[i - 1] * l[k] * l[j]
        if q < M[i][j]
          M[i][j] = q
return M
```

$$T(n) = \Theta(n^3)$$

Dynamic Programming Illustration

How to decide the order of the matrix multiplication?

j

i

$M_{i,j}$	*1*	*2*	*3*	*4*	*5*	*6*	*…*	*n*
1	0							
2		0						
3			0					
4				0				
5					0			
6						0		
:							0	
n								0

Step 4: Construct an OPT Solution by Backtracking

```
Matrix-Chain(n, l)
  initialize two tables M[1..n][1..n] and B[1..n-1][2..n]
  for i = 1 to n
    M[i][i] = 0 // boundary case
  for p = 2 to n // p is the chain length
    for i = 1 to n - p + 1 // all i, j combinations
      j = i + p - 1
      M[i][j] = ∞
      for k = i to j - 1 // find the best k
        q = M[i][k] + M[k + 1][j] + l[i - 1] * l[k] * l[j]
        if q < M[i][j]
          M[i][j] = q
          B[i][j] = k // backtracking
  return M and B
```

$$T(n) = \Theta(n^3)$$

```
Print-Optimal-Parens(B, i, j)
  if i == j
    print A_i
  else
    print "("
    Print-Optimal-Parens(B, i, B[i][j])
    Print-Optimal-Parens(B, B[i][j] + 1, j)
    print ")"
```

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

Exercise

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 x 35	35 x 15	15 x 5	5 x 10	10 x 20	20 x 25

j

$M_{i,j}$	*1*	*2*	*3*	*4*	*5*	*6*
1	0	15,750	7,875	9,375	11,875	15,125
2		0	2,625	4,375	7,125	10,500
3			0	750	2,500	53,75
4				0	1,000	3,500
5					0	5,000
6						0

i

j

$B_{i,j}$	*1*	*2*	*3*	*4*	*5*	*6*
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

i

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

22

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?

aeniqadikjaz

banana

svkbrlvpnzanczyqza

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
Y = aeniqadikjaz

4

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

X = banana
Y = svkbrlvpnzanczyqza

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

5

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

9

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

1 deletion, 7 insertions, 1 substitution

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: the minimal cost $M_{m,n}$ for aligning two sequences
  - Cost $=$ #insertions $\times\, C_{\text{INS}}$ + #deletions $\times\, C_{\text{DEL}}$ + #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
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
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
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
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
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								

$\Rightarrow \Theta(n)$

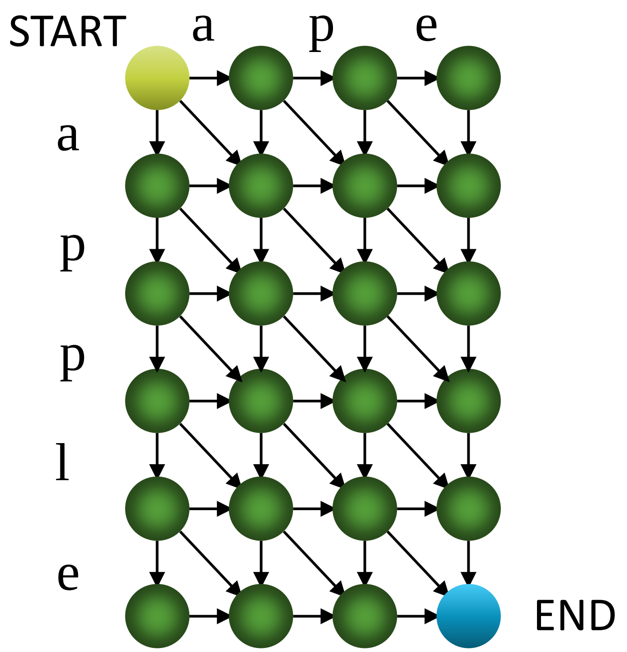
The optimal value can be computed, but the solution cannot be reconstructed

Space-Efficient Solution

Divide-and-Conquer
+
Dynamic Programming

- Problem: find the <u>min-cost alignment</u> → find the <u>shortest path</u>

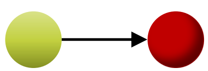
			a	p	e
	X\Y	**0**	**1**	**2**	**3**
	0	0	4	8	12
a	**1**	4	7	11	15
p	**2**	8	4	8	12
p	**3**	12	8	12	8
l	**4**	16	12	15	12
e	**5**	20	16	19	15

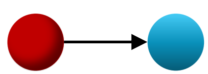


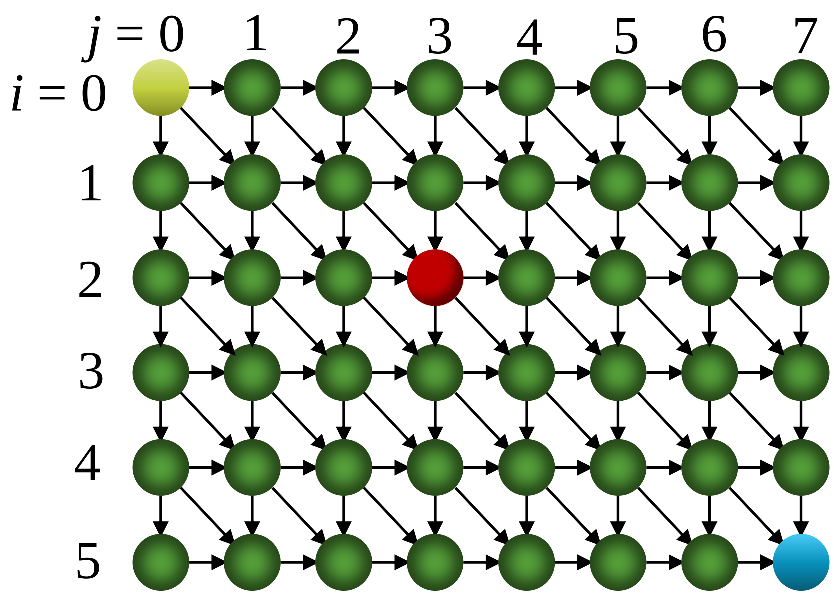
→ distance = C_{INS}
↓ distance = C_{DEL}
↘ distance = $C_{u,v}$ for edge (u, v)

Shortest Path in Graph

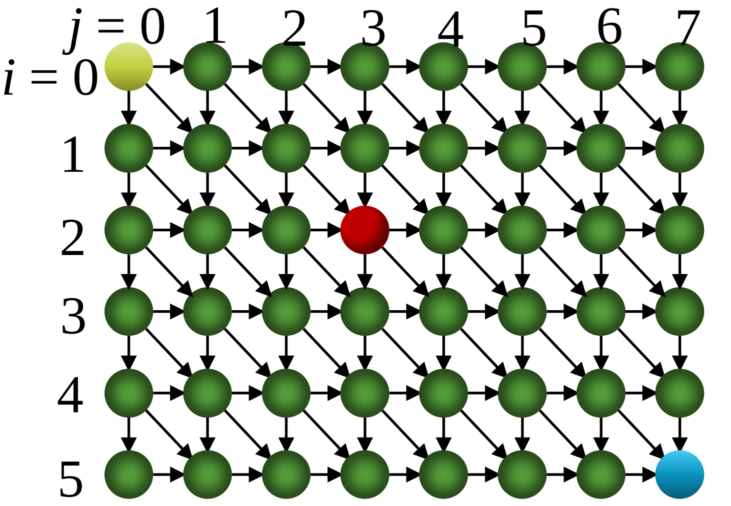
- Each edge has a length/cost
- $F(i,j)$: length of the shortest path from $(0,0)$ to (i,j) (START → (i,j))
- $B(i,j)$: length of the shortest path from (i,j) to (m,n) ((i,j) → END)
- $F(m,n) = B(0,0)$

$F(2,3)$ = 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

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

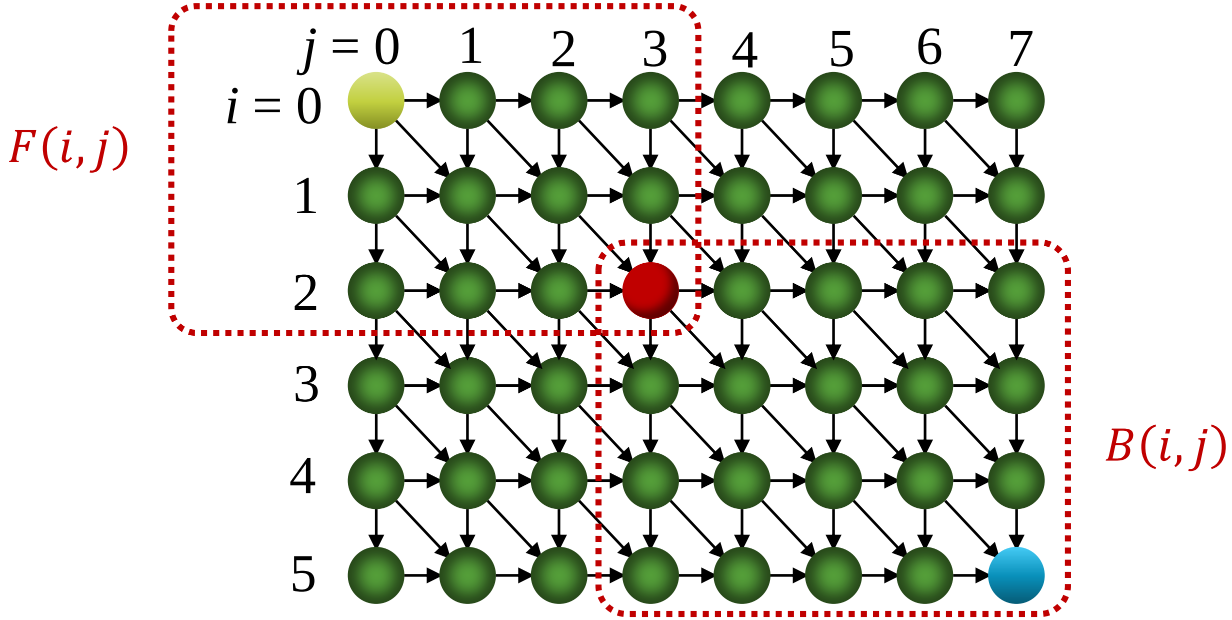
- Backward formulation

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

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)

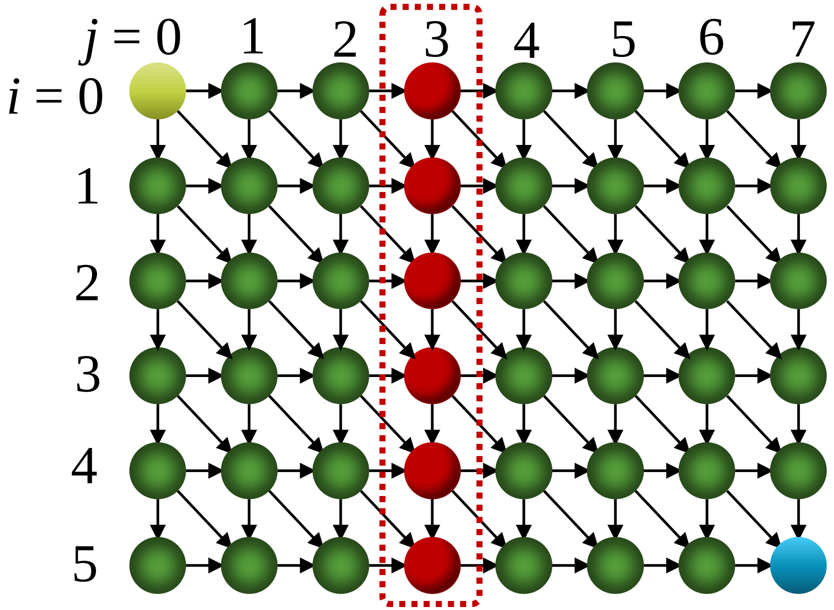
- Observation 1: the length of the shortest path from $(0,0)$ to (m,n) that passes through (i,j) is $F(i,j) + B(i,j)$
  → **optimal substructure**



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 $\{0, \ldots, n\}$, 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**



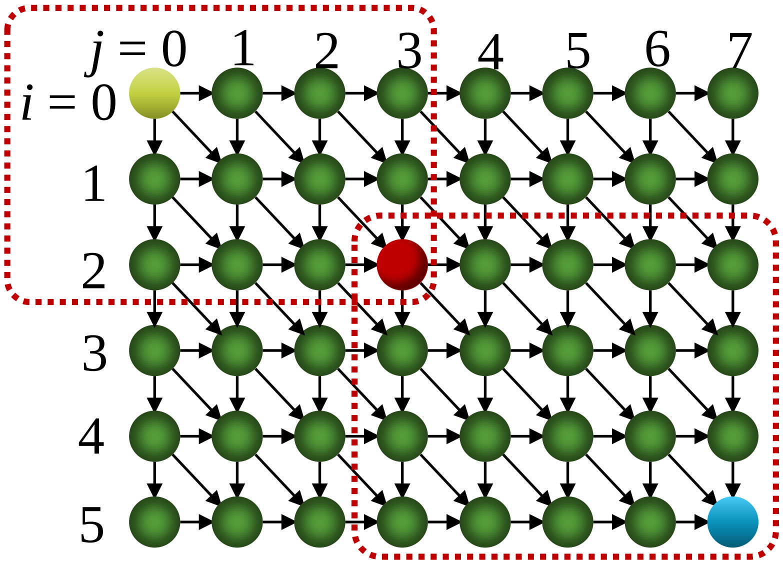
Shortest Path Problem

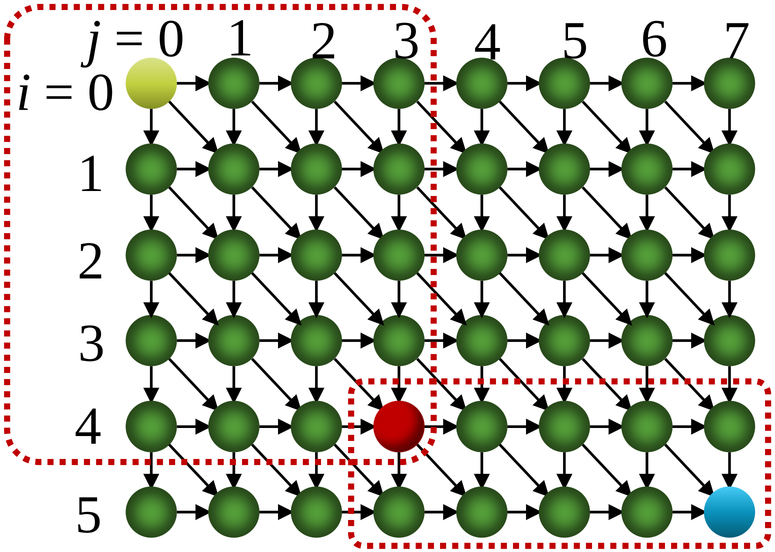
$F(i, j)$: length of the shortest path from $(0,0)$ to (i, j)
$B(i, j)$: length of the shortest path from (i, j) to (m, n)

- <u>Observation 1+2</u>:

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

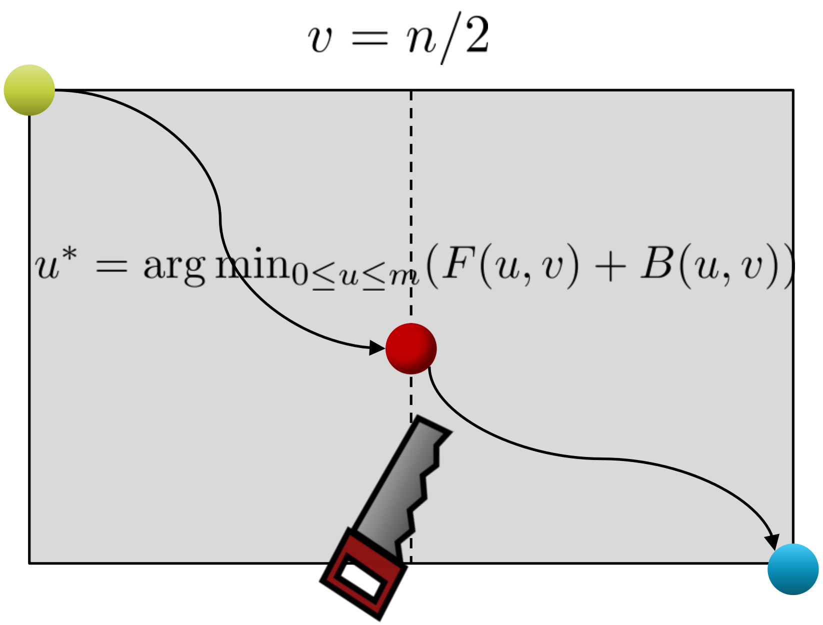
$$F(m,n) = \min_{0 \le u \le m} F(u,v) + B(u,v) \forall v$$





Divide-and-Conquer Algorithm

- Goal: finds optimal solution



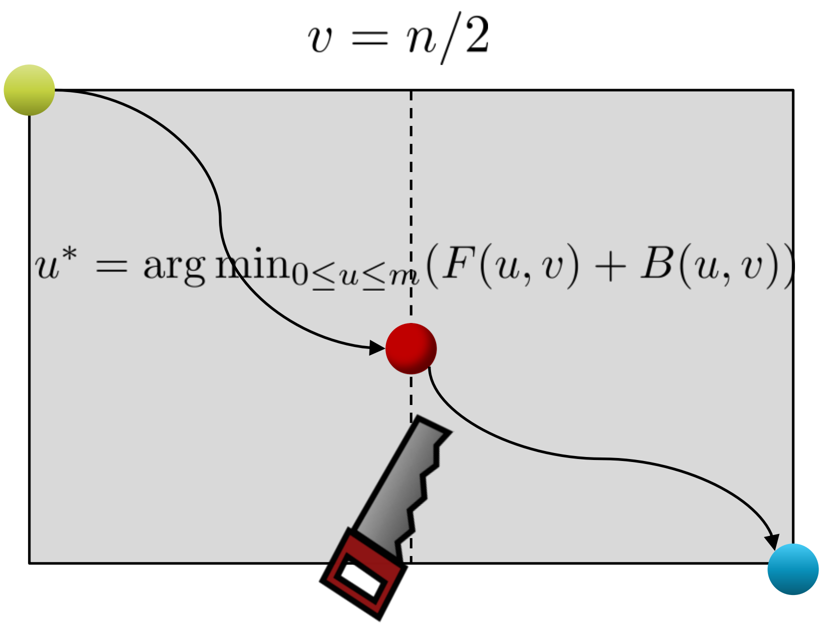
How to find the value of u^*?

- 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: $O(m+n)$



1. Divide
2. Conquer
3. Combine

- Divide the sequence of size *n* into 2 subsequences
  - Find u to minimize $F(u,v)+B(u,v)$ $\quad \Theta(mn)$
- Recursive case ($n>1$)
  - `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)` $\quad \Theta(m)$
- Return `prefix + suffix` $\quad \Theta(1)$

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

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

Time Complexity Analysis

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

- 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

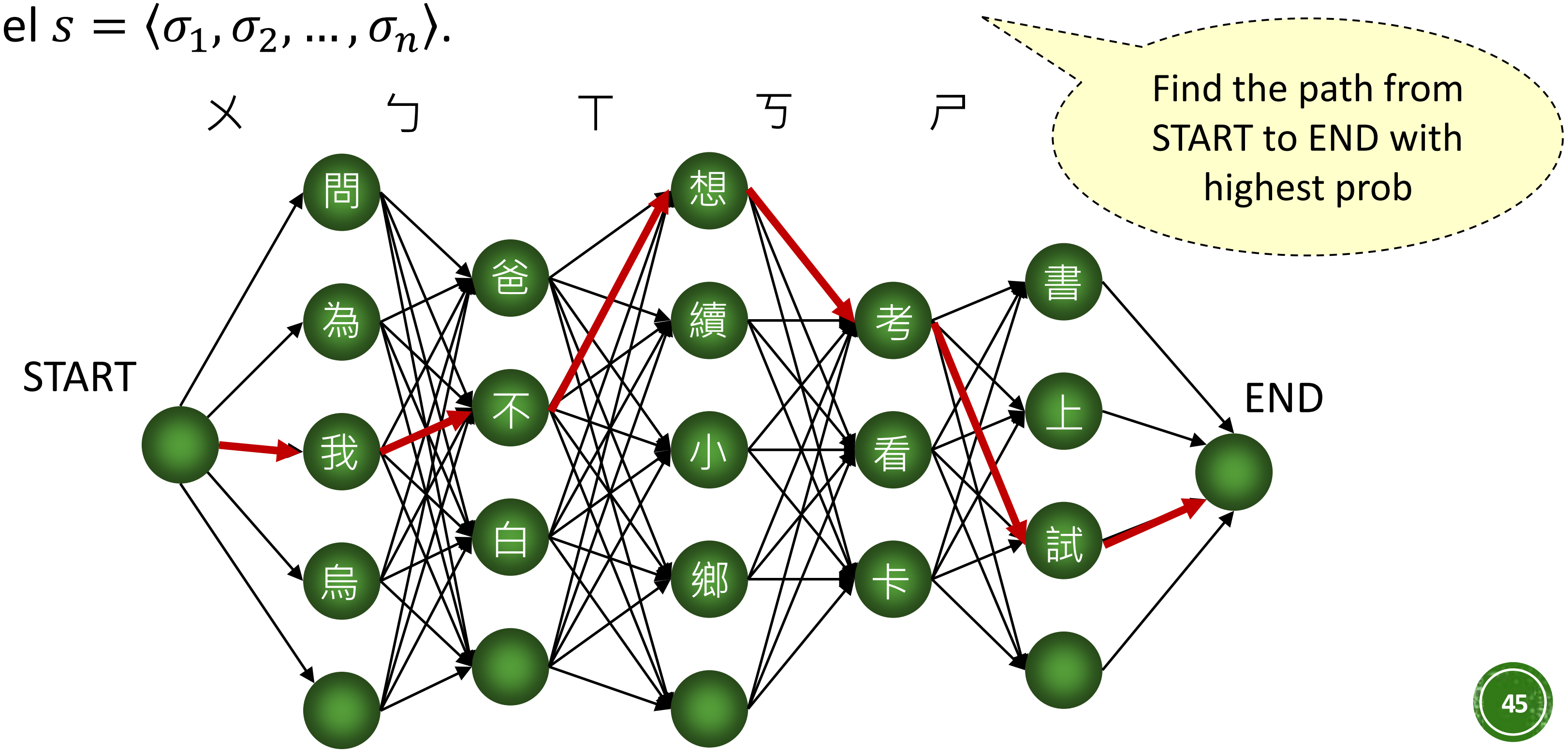
$$\leq ku\tfrac{n}{2} + k(m-u)\tfrac{n}{2} + b \cdot mn$$

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

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

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 $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$.

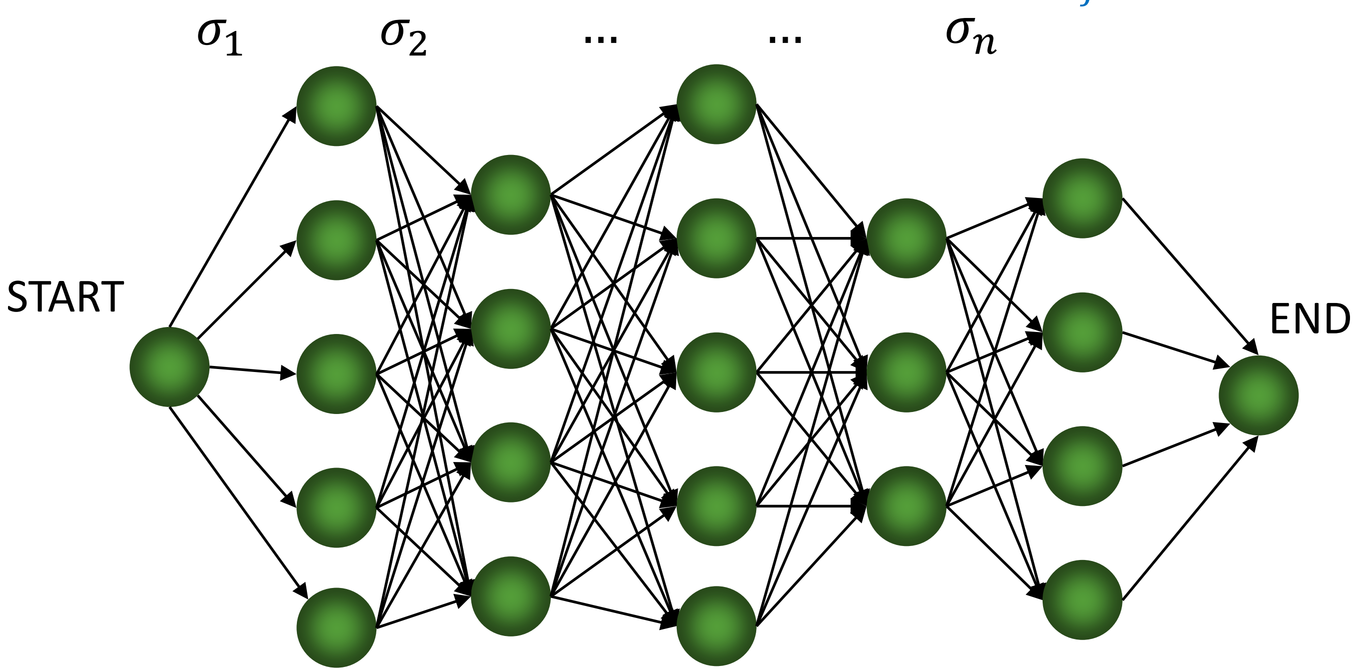


Viterbi Algorithm

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

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

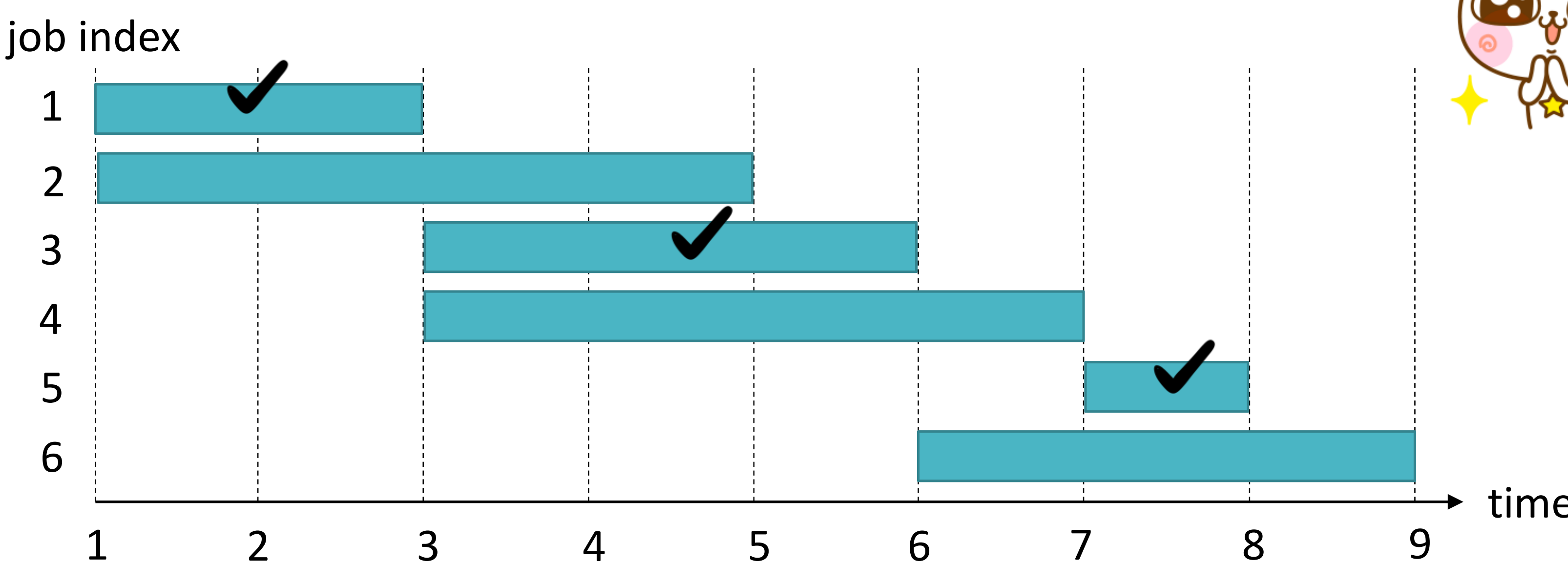
Viterbi has been applied to many AI applications, e.g. speech recognition

DP#6: Weighted Interval Scheduling

Interval Scheduling

- Input: n job requests with start times s_i, finish times f_i
- Output: the maximum number of compatible jobs
- The interval scheduling problem can be solved using an "**early-finish-time-first**" greedy algorithm in $O(n)$ time

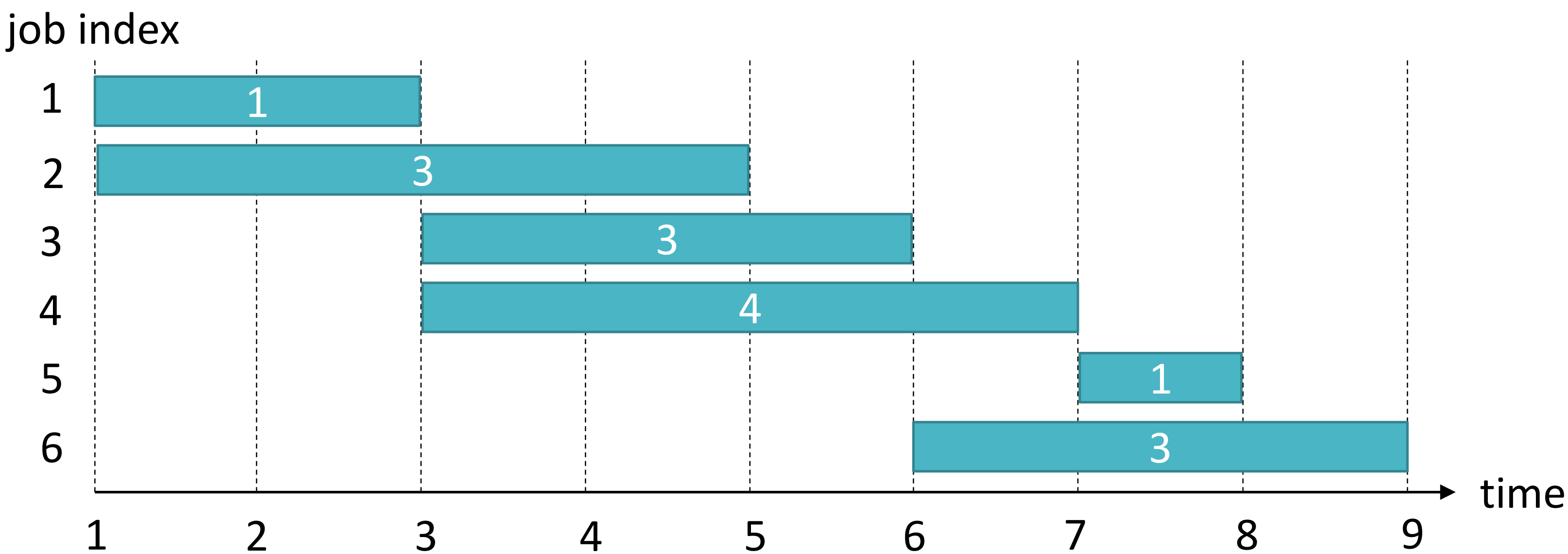
"Greedy Algorithm"
Next topic!



Weighted Interval Scheduling

- Input: n job requests with start times s_i, finish times f_i, and values v_i
- Output: the maximum total value obtainable from compatible jobs

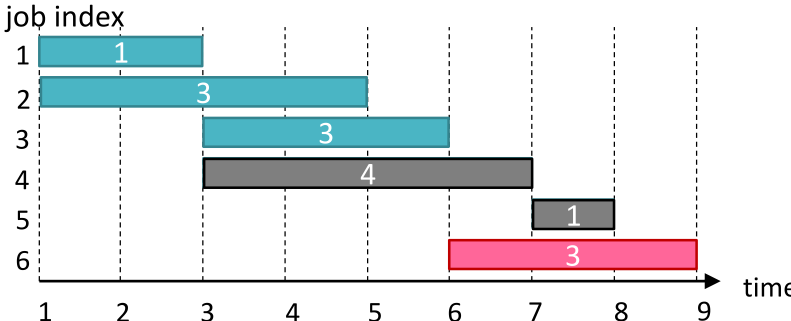
Assume that the requests are sorted in non-decreasing order ($f_i \leq f_j$ when $i < j$)
$p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
e.g. $p(1) = 0, p(2) = 0, p(3) = 1, p(4) = 1, p(5) = 4, p(6) = 3$



Step 1: Characterize an OPT Solution

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

- Subproblems
  - `WIS(i)`: weighted interval scheduling for the first i jobs
  - Goal: `WIS(n)`
- Optimal substructure: suppose OPT is an optimal solution to `WIS(i)`, there are 2 cases:
  - Case 1: job i in OPT
    - OPT\\$\{i\}$ is an optimal solution of `WIS(p(i))`
  - Case 2: job i not in OPT
    - OPT is an optimal solution of `WIS(i-1)`



Step 2: Recursively Define the Value of an OPT Solution

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

- Optimal substructure: suppose OPT is an optimal solution to `WIS(i)`, there are 2 cases:
  - Case 1: job i in OPT
    - OPT\\$\{i\}$ is an optimal solution of `WIS(p(i))`

$$M_i = v_i + M_{p(i)}$$

  - Case 2: job i not in OPT
    - OPT is an optimal solution of `WIS(i-1)`

$$M_i = M_{i-1}$$

- Recursively define the value

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

Step 3: Compute Value of an OPT Solution

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

- Bottom-up method: solve smaller subproblems first

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

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

```
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

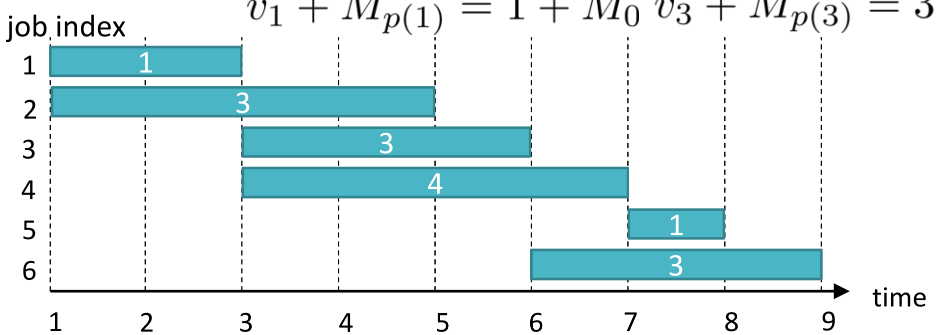
Weighted Interval Scheduling Problem
Input: n jobs with $\langle s_i, f_i, v_i \rangle$, $p(j)$ = largest index $i < j$ s.t. jobs i and j are compatible
Output: the maximum total value obtainable from compatible

- 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
```

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

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

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