

# Dynamic Programming

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

23 December, 2021

# Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”

# Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another

# Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~d itbse~~ used in designing algorithms”
- insert, ~~delete~~, substitute

# Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~itbse~~ used in designing algorithms”
- insert, ~~delete~~, substitute

## Edit distance

# Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can d itbse used in designing algorithms”
- insert, ~~delete~~, substitute

## Edit distance

- Minimum number of edit operations needed

# Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~it~~ bse used in designing algorithms”
- insert, ~~delete~~, substitute

## Edit distance

- Minimum number of edit operations needed
- In our example, 24 characters inserted, 18 ~~deleted~~, 2 substituted

# Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructuress ~~property~~ can ~~it~~ bse used in designing algorithms”
- insert, ~~delete~~, substitute

## Edit distance

- Minimum number of edit operations needed
- In our example, 24 characters inserted, 18 ~~deleted~~, 2 substituted
- Edit distance is at most 44

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

## Edit distance and LCS

- Longest common subsequence of  $u, v$

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

## Edit distance and LCS

- Longest common subsequence of  $u$ ,  $v$
- Minimum number of deletes needed to make them equal

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

## Edit distance and LCS

- Longest common subsequence of  $u$ ,  $v$ 
  - Minimum number of deletes needed to make them equal
  - Deleting a letter from  $u$  is equivalent to inserting it in  $v$

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

## Edit distance and LCS

- Longest common subsequence of  $u$ ,  $v$ 
  - Minimum number of deletes needed to make them equal
  - Deleting a letter from  $u$  is equivalent to inserting it in  $v$ 
    - `bisect`, `secret` — LCS is `sect`

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

## Edit distance and LCS

- Longest common subsequence of  $u$ ,  $v$ 
  - Minimum number of deletes needed to make them equal
- Deleting a letter from  $u$  is equivalent to inserting it in  $v$ 
  - `bisect`, `secret` — LCS is `sect`
  - Delete `b`, `i` in `bisect` and `r`, `e` in `secret`

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

## Edit distance and LCS

- Longest common subsequence of  $u$ ,  $v$ 
  - Minimum number of deletes needed to make them equal
- Deleting a letter from  $u$  is equivalent to inserting it in  $v$ 
  - $bisect$ ,  $secret$  — LCS is  $sect$
  - Delete  $b$ ,  $i$  in  $bisect$  and  $r$ ,  $e$  in  $secret$
  - Delete  $b$ ,  $i$  and then insert  $r$ ,  $e$  in  $bisect$

# Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

## Edit distance and LCS

- Longest common subsequence of  $u$ ,  $v$ 
  - Minimum number of deletes needed to make them equal
- Deleting a letter from  $u$  is equivalent to inserting it in  $v$ 
  - $bisect$ ,  $secret$  — LCS is  $sect$   

  - Delete  $b$ ,  $i$  in  $bisect$  and  $r$ ,  $e$  in  $secret$
  - Delete  $b$ ,  $i$  and then insert  $r$ ,  $e$  in  $bisect$
- LCS equivalent to edit distance without substitution

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
- 
- Recall LCS

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

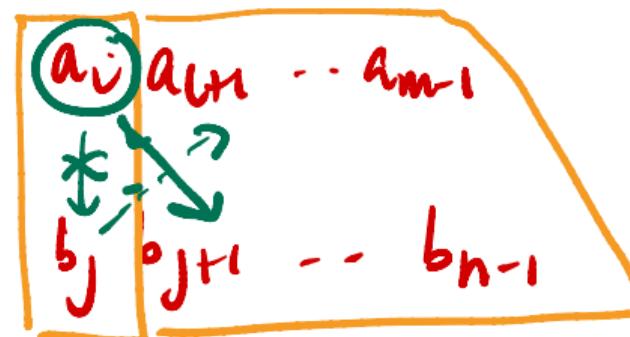
- Recall LCS

- If  $a_i = b_j$ ,

$$LCS(i, j) = 1 + LCS(i+1, j+1)$$

- If  $a_i \neq b_j$ ,

$$LCS(i, j) = \max[ LCS(i, j+1), LCS(i+1, j) ]$$



# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
- 
- Recall LCS
  - Edit distance — aim is to transform  $u$  to  $v$
- 
- If  $a_i = b_j$ ,  
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
  - If  $a_i \neq b_j$ ,  
$$LCS(i, j) = \max[ LCS(i, j+1), LCS(i+1, j) ]$$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
- 
- Recall LCS
  - Edit distance — aim is to transform  $u$  to  $v$
- 
- If  $a_i = b_j$ ,  
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
  - If  $a_i \neq b_j$ ,  
$$LCS(i, j) = \max[ LCS(i, j+1), LCS(i+1, j) ]$$
  - If  $a_i = b_j$ , nothing to be done

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
- 
- Recall LCS
  - If  $a_i = b_j$ ,  
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
  - If  $a_i \neq b_j$ ,  
$$LCS(i, j) = \max[ LCS(i, j+1), LCS(i+1, j) ]$$
- 
- Edit distance — aim is to transform  $u$  to  $v$
  - If  $a_i = b_j$ , nothing to be done
  - If  $a_i \neq b_j$ , best among

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
- 
- Recall LCS
  - If  $a_i = b_j$ ,  
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
  - If  $a_i \neq b_j$ ,  
$$LCS(i, j) = \max[ LCS(i, j+1), LCS(i+1, j) ]$$
- 
- Edit distance — aim is to transform  $u$  to  $v$
  - If  $a_i = b_j$ , nothing to be done
  - If  $a_i \neq b_j$ , best among
    - Substitute  $a_i$  by  $b_j$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
- 
- Recall LCS
  - If  $a_i = b_j$ ,  
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
  - If  $a_i \neq b_j$ ,  
$$LCS(i, j) = \max[ LCS(i, j+1), LCS(i+1, j) ]$$
  - Edit distance — aim is to transform  $u$  to  $v$
  - If  $a_i = b_j$ , nothing to be done
  - If  $a_i \neq b_j$ , best among
    - Substitute  $a_i$  by  $b_j$
    - Delete  $a_i$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
- 
- Recall LCS
  - If  $a_i = b_j$ ,  
$$LCS(i, j) = 1 + LCS(i+1, j+1)$$
  - If  $a_i \neq b_j$ ,  
$$LCS(i, j) = \max[ LCS(i, j+1), LCS(i+1, j) ]$$
  - Edit distance — aim is to transform  $u$  to  $v$
  - If  $a_i = b_j$ , nothing to be done
  - If  $a_i \neq b_j$ , best among
    - Substitute  $a_i$  by  $b_j$
    - Delete  $a_i$
    - Insert  $b_j$  before  $a_i$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform  $u$  to  $v$
- If  $a_i = b_j$ , nothing to be done
- If  $a_i \neq b_j$ , best among
  - Substitute  $a_i$  by  $b_j$
  - Delete  $a_i$
  - Insert  $b_j$  before  $a_i$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform  $u$  to  $v$
- If  $a_i = b_j$ , nothing to be done
- If  $a_i \neq b_j$ , best among
  - Substitute  $a_i$  by  $b_j$
  - Delete  $a_i$
  - Insert  $b_j$  before  $a_i$
- $ED(i, j)$  — edit distance for  
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform  $u$  to  $v$
- If  $a_i = b_j$ , nothing to be done
- If  $a_i \neq b_j$ , best among
  - Substitute  $a_i$  by  $b_j$
  - Delete  $a_i$
  - Insert  $b_j$  before  $a_i$
- $ED(i, j)$  — edit distance for  
 $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If  $a_i = b_j$ ,  
 $ED(i, j) = ED(i+1, j+1)$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform  $u$  to  $v$
- If  $a_i = b_j$ , nothing to be done
- If  $a_i \neq b_j$ , best among
  - Substitute  $a_i$  by  $b_j$
  - Delete  $a_i$
  - Insert  $b_j$  before  $a_i$
- $ED(i, j)$  — edit distance for  $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$

- If  $a_i = b_j$ ,  
 $ED(i, j) = ED(i+1, j+1)$
- If  $a_i \neq b_j$ ,  
 $ED(i, j) = 1 + \min[ ED(i+1, j+1), ED(i+1, j), ED(i, j+1) ]$

Handwritten notes illustrating the edit distance between two sequences:

Top row:  $a_{v-1} \ a_i \ a_m \ - \ -$

Bottom row:  $- \ i \ b_{j+1} \ - \ -$

A green diagonal line starts from the top-left and ends at the bottom-right. A red 'X' is drawn over the character 'i' in the bottom sequence.

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
  - Edit distance — transform  $u$  to  $v$
  - If  $a_i = b_j$ , nothing to be done
  - If  $a_i \neq b_j$ , best among
    - Substitute  $a_i$  by  $b_j$
    - Delete  $a_i$
    - Insert  $b_j$  before  $a_i$
  - $ED(i, j)$  — edit distance for  $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If  $a_i = b_j$ ,  
 $ED(i, j) = ED(i+1, j+1)$
  - If  $a_i \neq b_j$ ,  
 $ED(i, j) = 1 + \min[ ED(i+1, j+1), ED(i+1, j), ED(i, j+1) ]$
  - Base cases
    - $ED(m, n) = 0$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
  - Edit distance — transform  $u$  to  $v$
  - If  $a_i = b_j$ , nothing to be done
  - If  $a_i \neq b_j$ , best among
    - Substitute  $a_i$  by  $b_j$
    - Delete  $a_i$
    - Insert  $b_j$  before  $a_i$
  - $ED(i, j)$  — edit distance for  $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If  $a_i = b_j$ ,  
 $ED(i, j) = ED(i+1, j+1)$
  - If  $a_i \neq b_j$ ,  
 $ED(i, j) = 1 + \min[ ED(i+1, j+1), ED(i+1, j), ED(i, j+1) ]$
  - Base cases
    - $ED(m, n) = 0$
    - $ED(i, n) = m - i$  for all  $0 \leq i \leq m$   
Delete  $a_i a_{i+1} \dots a_{m-1}$  from  $u$

# Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
  - $v = b_0 b_1 \dots b_{n-1}$
  - Edit distance — transform  $u$  to  $v$
  - If  $a_i = b_j$ , nothing to be done
  - If  $a_i \neq b_j$ , best among
    - Substitute  $a_i$  by  $b_j$
    - Delete  $a_i$
    - Insert  $b_j$  before  $a_i$
  - $ED(i, j)$  — edit distance for  $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If  $a_i = b_j$ ,  
 $ED(i, j) = ED(i+1, j+1)$
  - If  $a_i \neq b_j$ ,  
 $ED(i, j) = 1 + \min[ ED(i+1, j+1), ED(i+1, j), ED(i, j+1) ]$
  - Base cases
    - $ED(m, n) = 0$
    - $ED(i, n) = m - i$  for all  $0 \leq i \leq m$   
Delete  $a_i a_{i+1} \dots a_{m-1}$  from  $u$
    - $ED(m, j) = n - j$  for all  $0 \leq j \leq n$   
Insert  $b_j b_{j+1} \dots b_{n-1}$  into  $u$

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  
 $0 \leq i \leq m, 0 \leq j \leq n$

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

The diagram shows a 7x9 grid representing a dynamic programming table for Edit Distance. The columns are labeled 0 through 6 at the top, and the rows are labeled 0 through 6 on the left. The letters 's', 'e', 'c', 'r', 'e', 't' are placed in the cells corresponding to (1,1), (1,2), (1,3), (1,4), (1,5), and (1,6) respectively. A black dot is in (1,0). Orange arrows indicate dependencies: from (1,1) to (2,1), (1,2), and (0,1); from (1,2) to (2,2) and (0,2); from (1,3) to (2,3) and (0,3); from (1,4) to (2,4) and (0,4); from (1,5) to (2,5) and (0,5); and from (1,6) to (2,6) and (0,6).

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							6
1	i							5
2	s							4
3	e							3
4	c							2
5	t							1
6	•							0

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						5	6
1	i						4	5
2	s						3	4
3	e						2	3
4	c						1	2
5	t						0	1
6	•						1	0

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					4	5	6
1	i					3	4	5
2	s					2	3	4
3	e					1	2	3
4	c					1	1	2
5	t					1	0	1
6	•					2	1	0

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b					4	4	5	6
1	i					3	3	4	5
2	s					2	2	3	4
3	e					2	1	2	3
4	c					2	1	1	2
5	t					2	1	0	1
6	•					3	2	1	0

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b			4	4	4	5	6
1	i			3	3	3	4	5
2	s			3	2	2	3	4
3	e			3	2	1	2	3
4	c			2	2	1	1	2
5	t			3	2	1	0	1
6	•			4	3	2	1	0

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	4	4	4	5	6	
1	i	4	3	3	3	4	5	
2	s	3	3	2	2	3	4	
3	e	2	3	2	1	2	3	
4	c	3	2	2	1	1	2	
5	t	4	3	2	1	0	1	
6	•	5	4	3	2	1	0	

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	4	4	4	4	5	6
1	i	3	4	3	3	3	4	5
2	s	2	3	3	2	2	3	4
3	e	3	2	3	2	1	2	3
4	c	4	3	2	2	1	1	2
5	t	5	4	3	2	1	0	1
6	•	6	5	4	3	2	1	0

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	

# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	

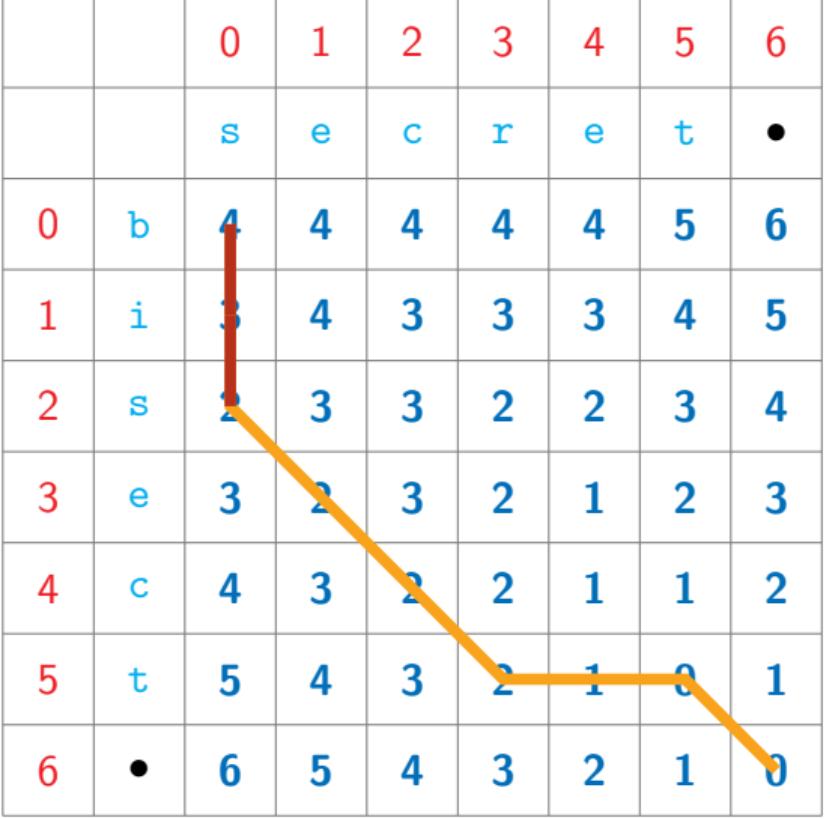
# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`, Delete `i`

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	



# Subproblem dependency

- Subproblems are  $ED(i, j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m + 1) \cdot (n + 1)$  values
- Like LCS,  $ED(i, j)$  depends on  $ED(i+1, j+1), ED(i, j+1), ED(i+1, j)$
- No dependency for  $ED(m, n)$  — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`, Delete `i`, Insert `r`

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	

# Subproblem dependency

- Subproblems are  $ED(i,j)$ , for  $0 \leq i \leq m, 0 \leq j \leq n$
- Table of  $(m+1) \cdot (n+1)$  values
- Like LCS,  $ED(i,j)$  depends on  $ED(i+1,j+1), ED(i,j+1), ED(i+1,j)$
- No dependency for  $ED(m,n)$  — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`, Delete `i`, Insert `r`, Insert `e`

		0	1	2	3	4	5	6	
		s	e	c	r	e	t	•	
0	b	4	4	4	4	4	5	6	
1	i	3	4	3	3	3	4	5	
2	s	2	3	3	2	2	3	4	
3	e	3	2	3	2	1	2	3	
4	c	4	3	2	2	1	1	2	
5	t	5	4	3	2	1	0	1	
6	•	6	5	4	3	2	1	0	

# Implementation

```
def ED(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    ed = np.zeros((m+1,n+1))  
  
    for i in range(m-1,-1,-1):  
        ed[i,n] = m-i  
    for j in range(n-1,-1,-1):  
        ed[m,j] = n-j  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                ed[i,j] = ed[i+1,j+1]  
            else:  
                ed[i,j] = 1 + min(ed[i+1,j+1],  
                                  ed[i,j+1],  
                                  ed[i+1,j])  
  
    return(ed[0,0])
```

# Implementation

```
def ED(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    ed = np.zeros((m+1,n+1))  
  
    for i in range(m-1,-1,-1):  
        ed[i,n] = m-i  
    for j in range(n-1,-1,-1):  
        ed[m,j] = n-j  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                ed[i,j] = ed[i+1,j+1]  
            else:  
                ed[i,j] = 1 + min(ed[i+1,j+1],  
                                  ed[i,j+1],  
                                  ed[i+1,j])  
  
    return(ed[0,0])
```

Complexity

# Implementation

```
def ED(u,v):  
    import numpy as np  
    (m,n) = (len(u),len(v))  
    ed = np.zeros((m+1,n+1))  
  
    for i in range(m-1,-1,-1):  
        ed[i,n] = m-i  
    for j in range(n-1,-1,-1):  
        ed[m,j] = n-j  
  
    for j in range(n-1,-1,-1):  
        for i in range(m-1,-1,-1):  
            if u[i] == v[j]:  
                ed[i,j] = ed[i+1,j+1]  
            else:  
                ed[i,j] = 1 + min(ed[i+1,j+1],  
                                  ed[i,j+1],  
                                  ed[i+1,j])  
  
    return(ed[0,0])
```

## Complexity

- Again  $O(mn)$ , using dynamic programming or memoization

LCW      Brute force  $O(mn^2)$   
DP       $O(mn)$

# Implementation

```
def ED(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    ed = np.zeros((m+1,n+1))

    for i in range(m-1,-1,-1):
        ed[i,n] = m-i
    for j in range(n-1,-1,-1):
        ed[m,j] = n-j

    for j in range(n-1,-1,-1):
        for i in range(m-1,-1,-1):
            if u[i] == v[j]:
                ed[i,j] = ed[i+1,j+1]
            else:
                ed[i,j] = 1 + min(ed[i+1,j+1],
                                  ed[i,j+1],
                                  ed[i+1,j])
    return(ed[0,0])
```

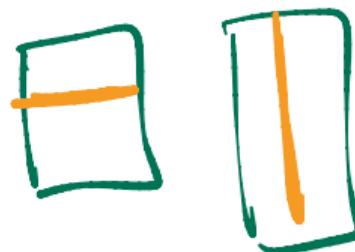
## Complexity

- Again  $O(mn)$ , using dynamic programming or memoization
  - Fill a table of size  $O(mn)$
  - Each table entry takes constant time to compute

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$



# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $$A : m \times n, B : n \times p$$
- $$AB : m \times p$$

# Multiplying matrices

- Multiply matrices  $A, B$

$$AB[i, j] = \sum_{k=0}^{n-1} A[i, k]B[k, j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in  $AB$  is  $O(n)$

map entries  
each costs  $n$  steps

# Multiplying matrices

- Multiply matrices  $A$ ,  $B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
  - $A : m \times n$ ,  $B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$

# Multiplying matrices

- Multiply matrices  $A$ ,  $B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
  - $A : m \times n$ ,  $B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
  - $A : m \times n, B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change answer

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
  - $A : m \times n, B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change answer
  - ... but can affect the complexity!

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

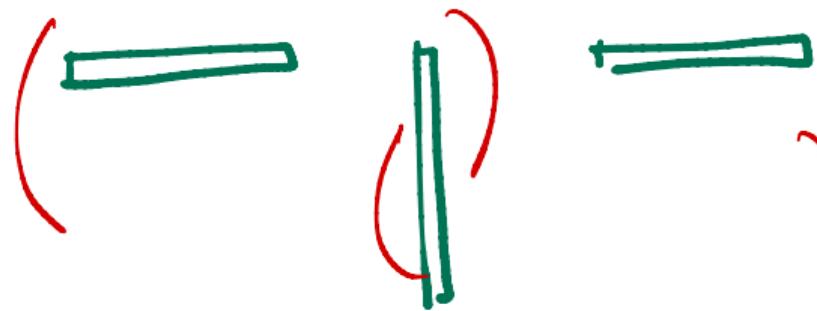
- Computing each entry in  $AB$  is  $O(n)$

- Overall, computing  $AB$  is  $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let  $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$



# Multiplying matrices

- Multiply matrices  $A, B$ 
  - $AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$
- Dimensions must be compatible
  - $A : m \times n, B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change answer
  - ... but can affect the complexity!
- Let  $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$
- Computing  $A(BC)$

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in  $AB$  is  $O(n)$

- Overall, computing  $AB$  is  $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let  $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing  $A(BC)$

- $BC : 100 \times 100$ , takes

$100 \cdot 1 \cdot 100 = 10000$  steps to compute



# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in  $AB$  is  $O(n)$

- Overall, computing  $AB$  is  $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let  $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing  $A(BC)$

- $BC : 100 \times 100$ , takes  
 $100 \cdot 1 \cdot 100 = 10000$  steps to compute
- $A(BC) : 1 \times 100$ , takes  
 $1 \cdot 100 \cdot 100 = 10000$  steps to compute

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in  $AB$  is  $O(n)$

- Overall, computing  $AB$  is  $O(mnp)$

- Matrix multiplication is associative

- $ABC = (AB)C = A(BC)$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let  $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing  $A(BC)$

- $BC : 100 \times 100$ , takes  
 $100 \cdot 1 \cdot 100 = 10000$  steps to compute
- $A(BC) : 1 \times 100$ , takes  
 $1 \cdot 100 \cdot 100 = 10000$  steps to compute

- Computing  $(AB)C$



# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in  $AB$  is  $O(n)$

- Overall, computing  $AB$  is  $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let  $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing  $A(BC)$

- $BC : 100 \times 100$ , takes  
 $100 \cdot 1 \cdot 100 = 10000$  steps to compute
- $A(BC) : 1 \times 100$ , takes  
 $1 \cdot 100 \cdot 100 = 10000$  steps to compute

- Computing  $(AB)C$

- $AB : 1 \times 1$ , takes  
 $1 \cdot 100 \cdot 1 = 100$  steps to compute

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in  $AB$  is  $O(n)$

- Overall, computing  $AB$  is  $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Let  $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing  $A(BC)$

- $BC : 100 \times 100$ , takes  
 $100 \cdot 1 \cdot 100 = 10000$  steps to compute
- $A(BC) : 1 \times 100$ , takes  
 $1 \cdot 100 \cdot 100 = 10000$  steps to compute

- Computing  $(AB)C$

- $AB : 1 \times 1$ , takes  
 $1 \cdot 100 \cdot 1 = 100$  steps to compute
- $(AB)C : 1 \times 100$ , takes  
 $1 \cdot 1 \cdot 100 = 100$  steps to compute

# Multiplying matrices

- Multiply matrices  $A, B$

$AB CD$

- $AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$

$((AB)C)D$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

$(A(B(CD)))$

- Computing each entry in  $AB$  is  $O(n)$

- Let  $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$

- Computing  $A(BC)$

- $BC : 100 \times 100$ , takes  
 $100 \cdot 1 \cdot 100 = 10000$  steps to compute

- $A(BC) : 1 \times 100$ , takes  
 $1 \cdot 100 \cdot 100 = 10000$  steps to compute

- Overall, computing  $AB$  is  $O(mnp)$

$(AB)(CD)$

- Matrix multiplication is associative

- $ABC = (AB)C = A(BC)$

$C$

- Bracketing does not change answer
- ... but can affect the complexity!

- Computing  $(AB)C$

- $AB : 1 \times 1$ , takes  
 $1 \cdot 100 \cdot 1 = 100$  steps to compute

- $(AB)C : 1 \times 100$ , takes  
 $1 \cdot 1 \cdot 100 = 100$  steps to compute

- 20000 steps vs 200 steps!

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
  - $A : m \times n, B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change answer
  - ... but can affect the complexity!

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
  - $A : m \times n, B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change answer
  - ... but can affect the complexity!

- Given  $n$  matrices  $M_0 : r_0 \times c_0, M_1 : r_1 \times c_1, \dots, M_{n-1} : r_{n-1} \times c_{n-1}$

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
  - $A : m \times n, B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change answer
  - ... but can affect the complexity!

- Given  $n$  matrices  $M_0 : r_0 \times c_0, M_1 : r_1 \times c_1, \dots, M_{n-1} : r_{n-1} \times c_{n-1}$ 
  - Dimensions match:  $r_j = c_{j-1}, 0 < j < n$

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible

- $A : m \times n, B : n \times p$
- $AB : m \times p$

- Computing each entry in  $AB$  is  $O(n)$

- Overall, computing  $AB$  is  $O(mnp)$

- Matrix multiplication is associative

- $$ABC = (AB)C = A(BC)$$
- Bracketing does not change answer
- ... but can affect the complexity!

- Given  $n$  matrices  $M_0 : r_0 \times c_0,$

- $M_1 : r_1 \times c_1, \dots, M_{n-1} : r_{n-1} \times c_{n-1}$

- Dimensions match:  $r_j = c_{j-1}, 0 < j < n$
- Product  $M_0 \cdot M_1 \cdots M_{n-1}$  can be computed

# Multiplying matrices

- Multiply matrices  $A, B$

- $$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j]$$

- Dimensions must be compatible
  - $A : m \times n, B : n \times p$
  - $AB : m \times p$
- Computing each entry in  $AB$  is  $O(n)$
- Overall, computing  $AB$  is  $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change answer
  - ... but can affect the complexity!

- Given  $n$  matrices  $M_0 : r_0 \times c_0, M_1 : r_1 \times c_1, \dots, M_{n-1} : r_{n-1} \times c_{n-1}$ 
  - Dimensions match:  $r_j = c_{j-1}, 0 < j < n$
  - Product  $M_0 \cdot M_1 \cdots M_{n-1}$  can be computed
- Find an optimal order to compute the product
  - Multiply two matrices at a time
  - Bracket the expression optimally

# Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some  $0 < k < n$

# Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some  $0 < k < n$

- First factor is  $r_0 \times c_{k-1}$ , second is  
 $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$

# Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some  $0 < k < n$

- First factor is  $r_0 \times c_{k-1}$ , second is

$$r_k \times c_{n-1}, \text{ where } r_k = c_{k-1}$$

- Let  $C(0, n-1)$  denote the overall cost

# Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some  $0 < k < n$

- First factor is  $r_0 \times c_{k-1}$ , second is  
 $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$
- Let  $C(0, n-1)$  denote the overall cost
- Final multiplication is  $O(r_0 r_k c_{n-1})$

# Inductive structure

- Final step combines two subproducts  
 $(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$   
for some  $0 < k < n$
- First factor is  $r_0 \times c_{k-1}$ , second is  
 $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$
- Let  $C(0, n-1)$  denote the overall cost
- Final multiplication is  $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are  $C(0, k-1)$   
and  $C(k, n-1)$

# Inductive structure

- Final step combines two subproducts

$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$

for some  $0 < k < n$

- First factor is  $r_0 \times c_{k-1}$ , second is

$$r_k \times c_{n-1}, \text{ where } r_k = c_{k-1}$$

- Let  $C(0, n-1)$  denote the overall cost

- Final multiplication is  $O(r_0 r_k c_{n-1})$

- Inductively, costs of factors are  $C(0, k-1)$  and  $C(k, n-1)$

- $C(0, n-1) =$

$$C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$$

$$\overbrace{(M_0 \quad M_{k-1})}^{\substack{\text{Red bracket} \\ \text{Factor 1}}} \overbrace{(M_k \quad M_{n-1})}^{\substack{\text{Red bracket} \\ \text{Factor 2}}} \quad \downarrow \quad \text{Red arrow}$$



# Inductive structure

- Final step combines two subproducts  
 $(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$   
for some  $0 < k < n$
  - First factor is  $r_0 \times c_{k-1}$ , second is  
 $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$
  - Let  $C(0, n-1)$  denote the overall cost
  - Final multiplication is  $O(r_0 r_k c_{n-1})$
  - Inductively, costs of factors are  $C(0, k-1)$   
and  $C(k, n-1)$
  - $C(0, n-1) =$   
 $C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which  $k$  should we choose?

# Inductive structure

- Final step combines two subproducts  
 $(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$   
for some  $0 < k < n$
  - First factor is  $r_0 \times c_{k-1}$ , second is  
 $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$
  - Let  $C(0, n-1)$  denote the overall cost
  - Final multiplication is  $O(r_0 r_k c_{n-1})$
  - Inductively, costs of factors are  $C(0, k-1)$   
and  $C(k, n-1)$
  - $C(0, n-1) =$   
 $C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which  $k$  should we choose?
    - Try all and choose the minimum!

# Inductive structure

- Final step combines two subproducts
$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$
for some  $0 < k < n$
  - First factor is  $r_0 \times c_{k-1}$ , second is  $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$
  - Let  $C(0, n-1)$  denote the overall cost
  - Final multiplication is  $O(r_0 r_k c_{n-1})$
  - Inductively, costs of factors are  $C(0, k-1)$  and  $C(k, n-1)$
  - $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which  $k$  should we choose?
    - Try all and choose the minimum!
  - Subproblems?

# Inductive structure

- Final step combines two subproducts
$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$
for some  $0 < k < n$
- First factor is  $r_0 \times c_{k-1}$ , second is  $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$
- Let  $C(0, n-1)$  denote the overall cost
- Final multiplication is  $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are  $C(0, k-1)$  and  $C(k, n-1)$
- $C(0, n-1) =$  
$$C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$$
- Which  $k$  should we choose?
  - Try all and choose the minimum!
- Subproblems?
  - $M_0 \cdot M_1 \cdots M_{k-1}$  would decompose as  $(M_0 \cdots M_{j-1}) \cdot (M_j \cdots M_{k-1})$
  - Generic subproblem is  $M_j \cdot M_{j+1} \cdots M_k$

# Inductive structure

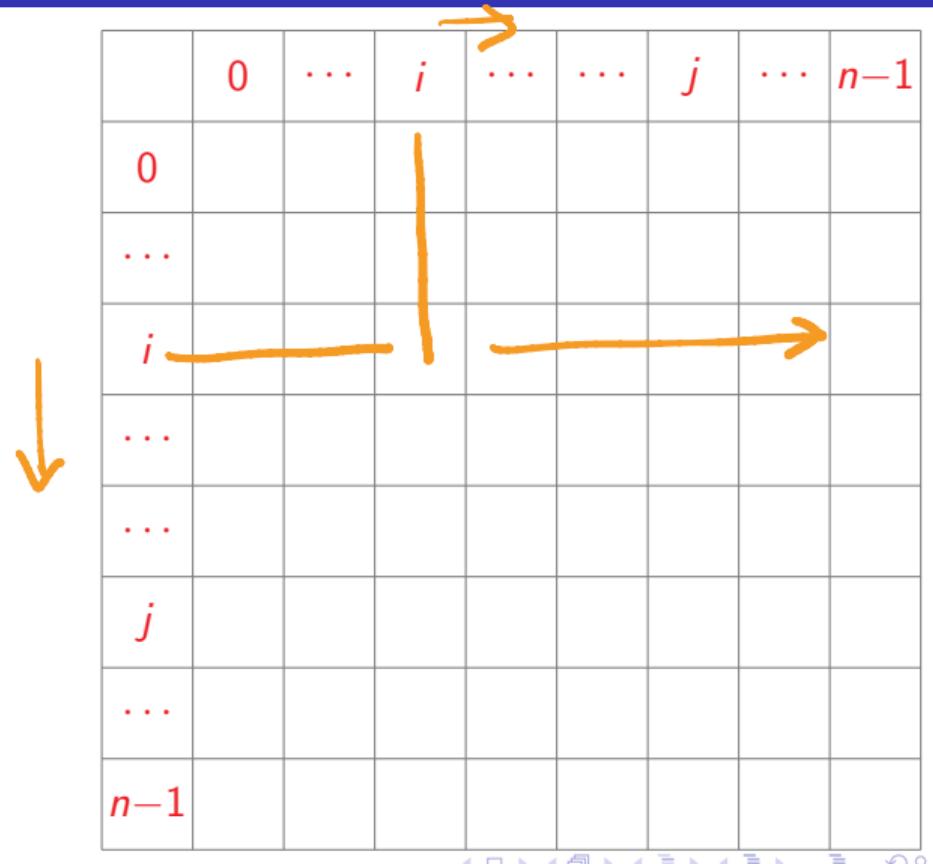
- Final step combines two subproducts
$$(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$$
for some  $0 < k < n$
- First factor is  $r_0 \times c_{k-1}$ , second is  $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$
- Let  $C(0, n-1)$  denote the overall cost
- Final multiplication is  $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are  $C(0, k-1)$  and  $C(k, n-1)$
- $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which  $k$  should we choose?
  - Try all and choose the minimum!
- Subproblems?
  - $M_0 \cdot M_1 \cdots M_{k-1}$  would decompose as  $(M_0 \cdots M_{j-1}) \cdot (M_j \cdots M_{k-1})$
  - Generic subproblem is  $M_j \cdot M_{j+1} \cdots M_k$
- $C(j, k) = \min_{j < \ell \leq k} [C(j, \ell-1) + C(\ell, k) + r_j r_\ell c_k]$

# Inductive structure

- Final step combines two subproducts  
 $(M_0 \cdot M_1 \cdots M_{k-1}) \cdot (M_k \cdot M_{k+1} \cdots M_{n-1})$   
for some  $0 < k < n$
- First factor is  $r_0 \times c_{k-1}$ , second is  
 $r_k \times c_{n-1}$ , where  $r_k = c_{k-1}$
- Let  $C(0, n-1)$  denote the overall cost
- Final multiplication is  $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are  $C(0, k-1)$  and  $C(k, n-1)$
- $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which  $k$  should we choose?
  - Try all and choose the minimum!
- Subproblems?
  - $M_0 \cdot M_1 \cdots M_{k-1}$  would decompose as  $(M_0 \cdots M_{j-1}) \cdot (M_j \cdots M_{k-1})$
  - Generic subproblem is  $M_j \cdot M_{j+1} \cdots M_k$
- $C(j, k) = \min_{j < \ell \leq k} [C(j, \ell-1) + C(\ell, k) + r_j r_\ell c_k]$
- Base case:  $C(j, j) = 0$  for  $0 \leq j < n$

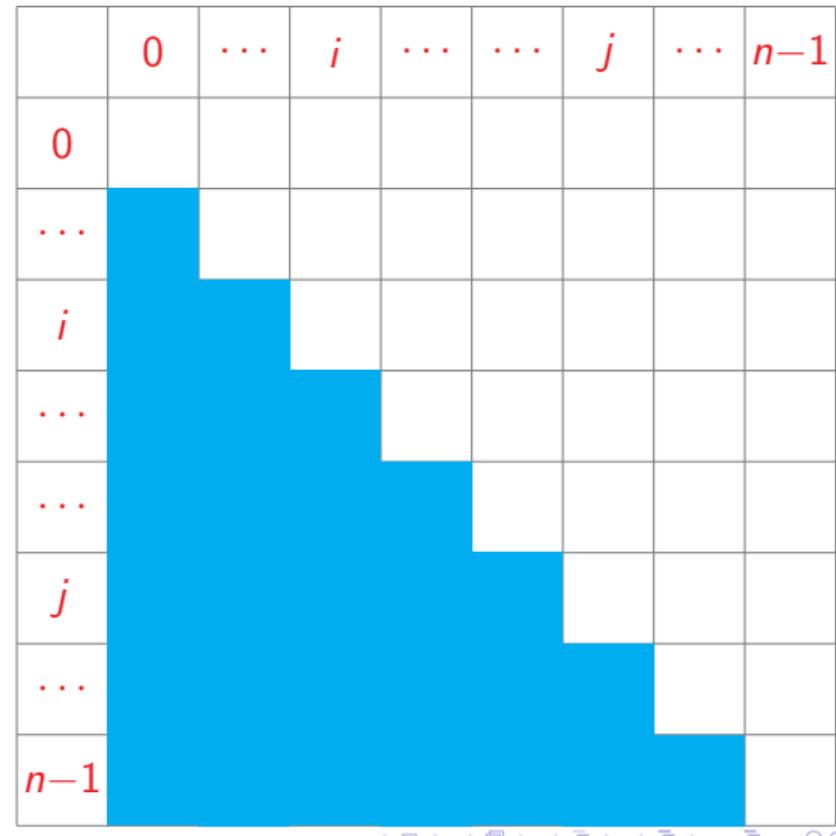
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$



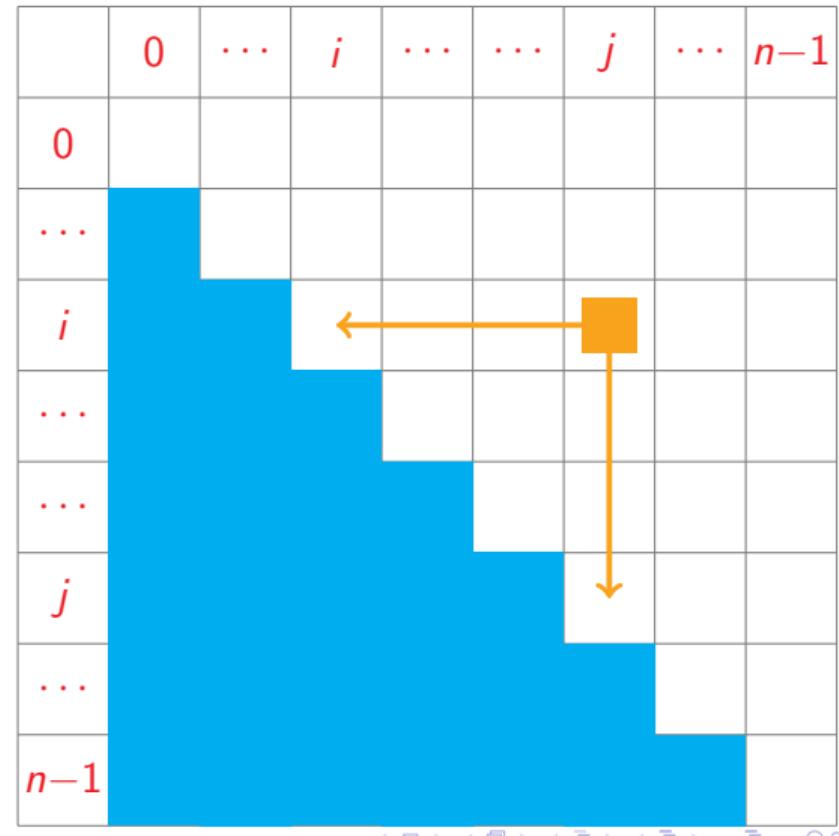
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal



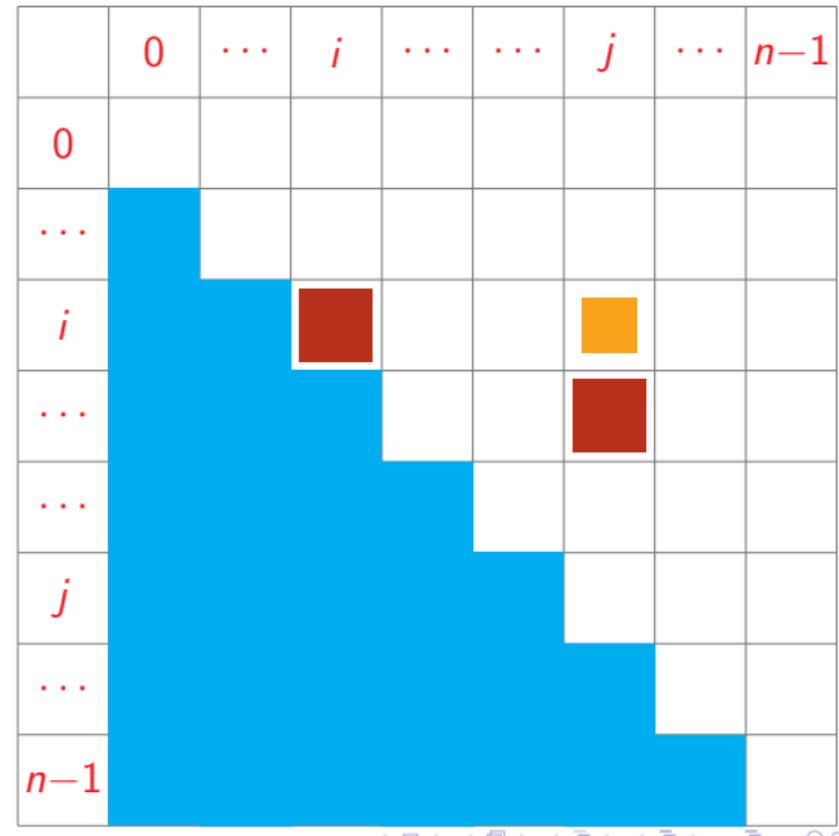
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$



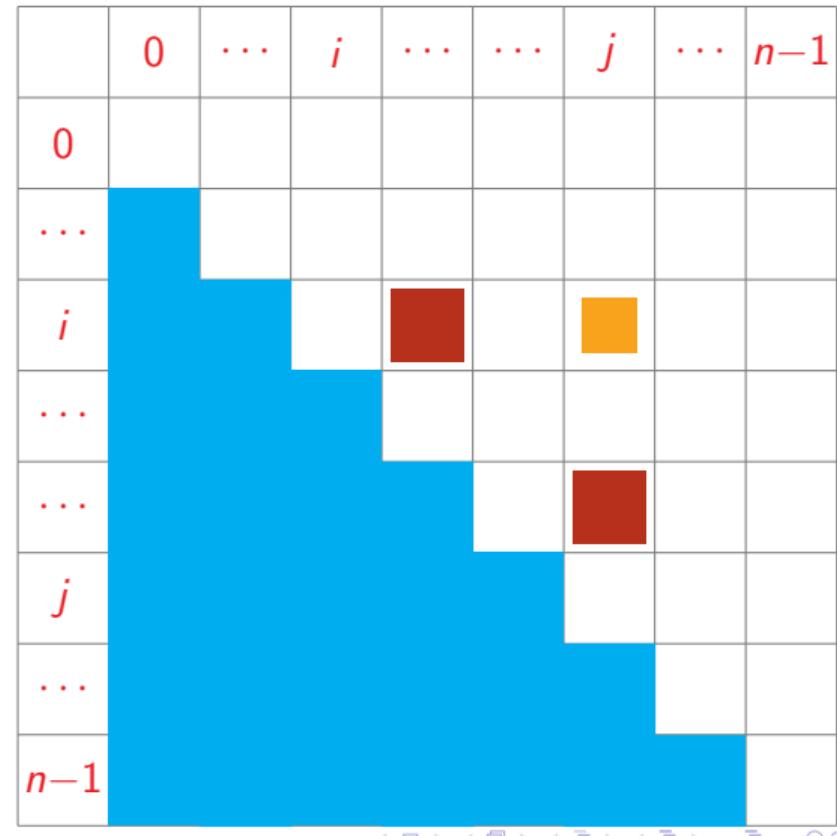
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$



# Subproblem dependency

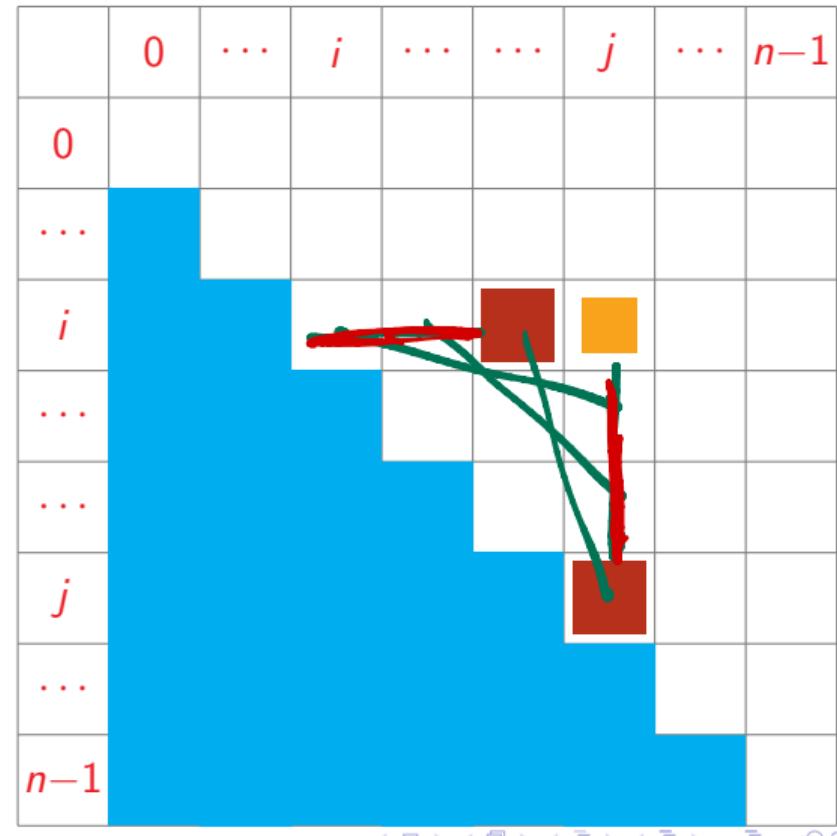
- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$



# Subproblem dependency

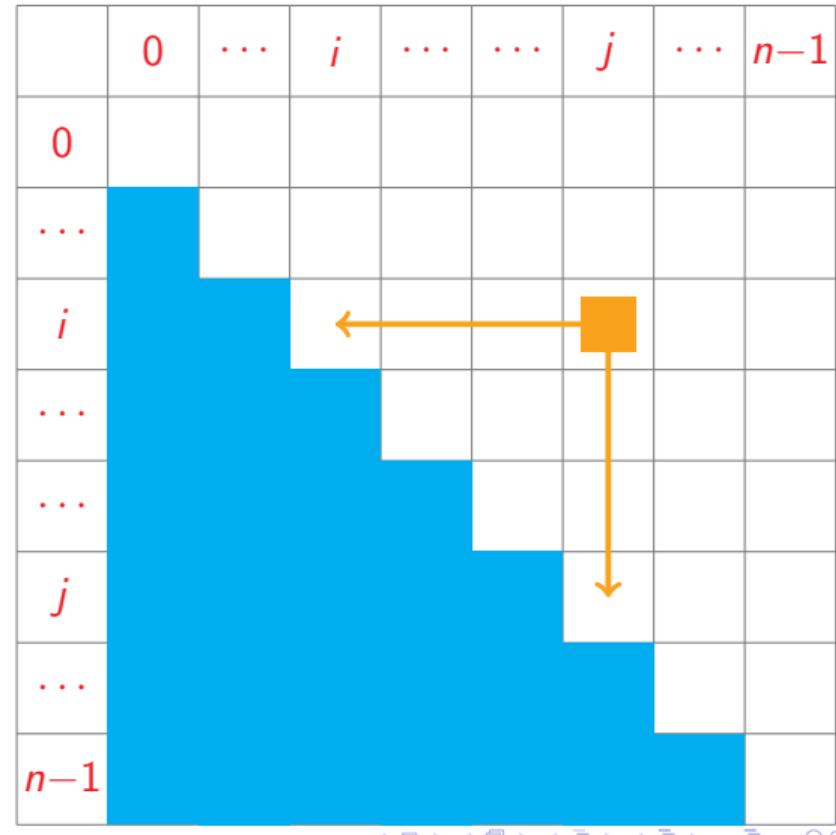
- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$

$$\min_l \quad C(u_{l_i-1}) \\ C(e_{ij})$$



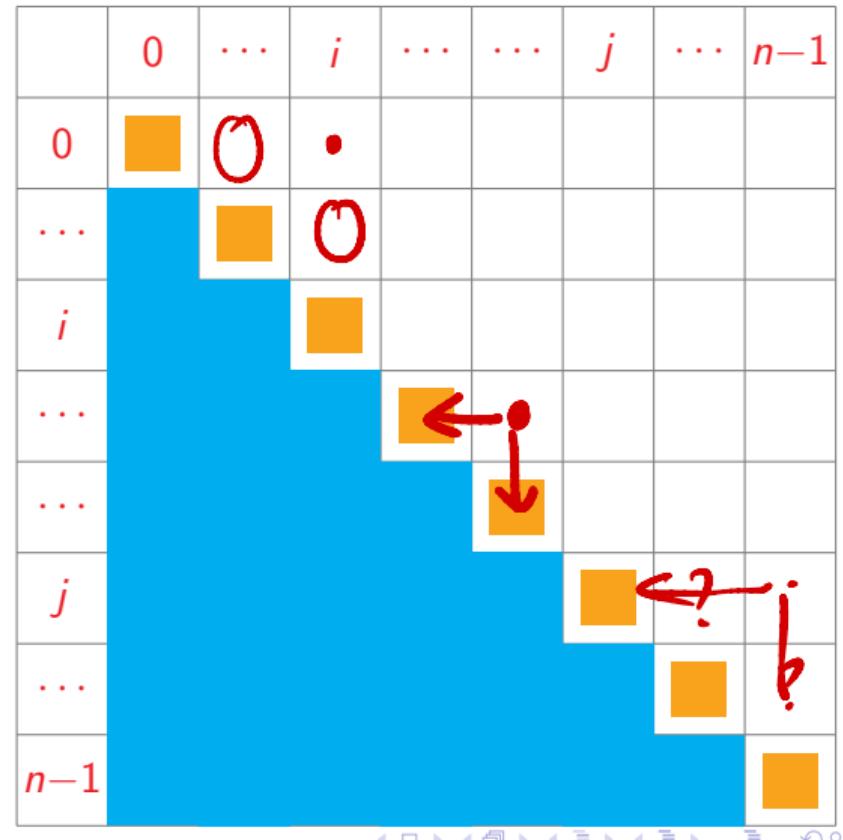
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED



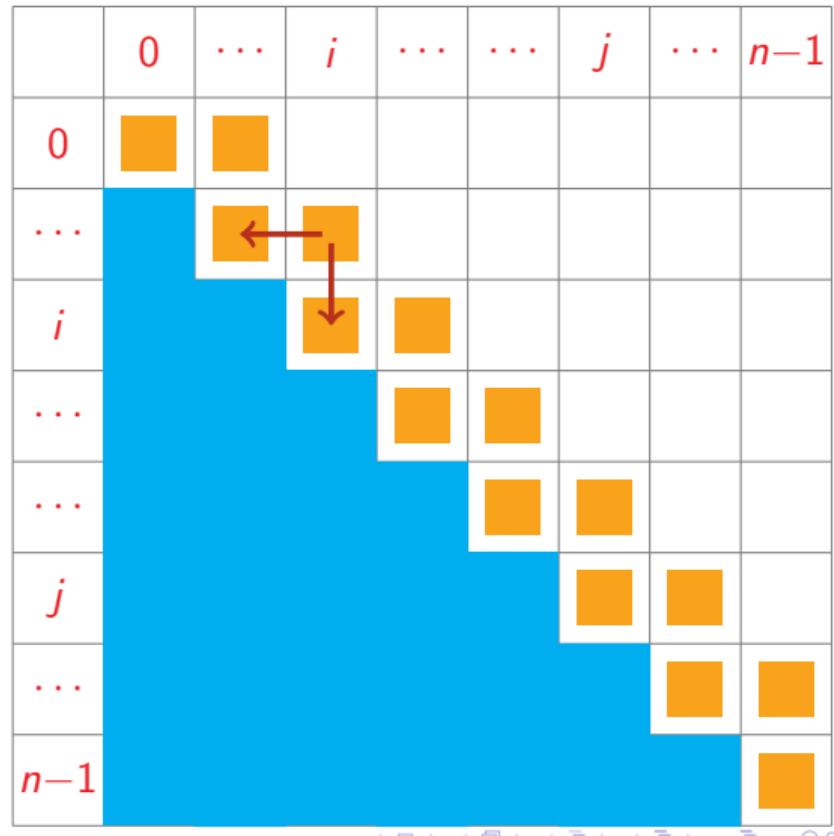
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case



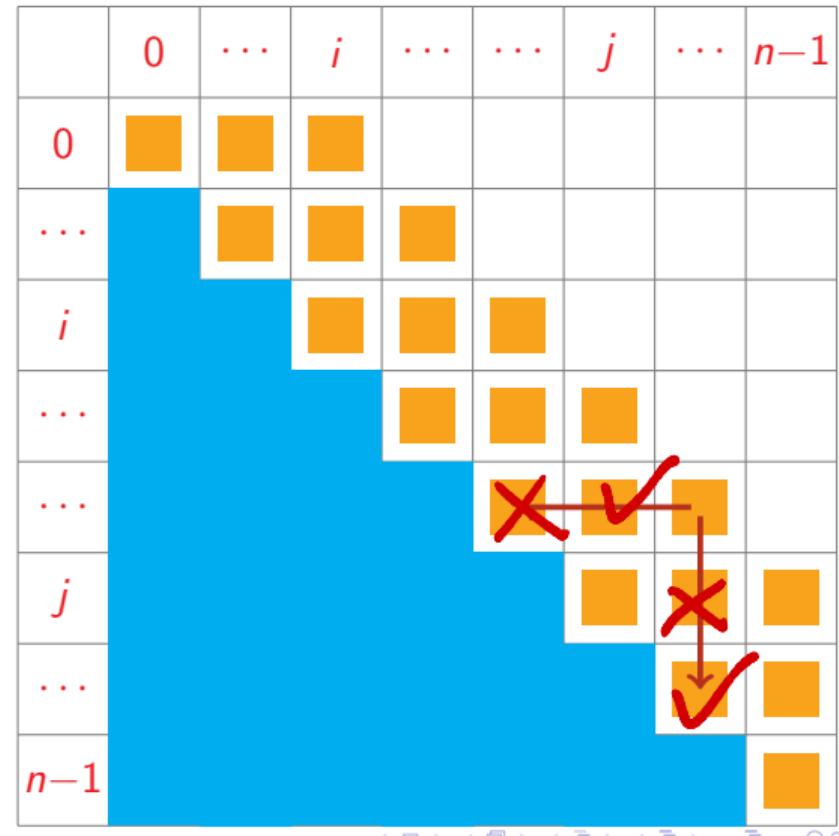
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



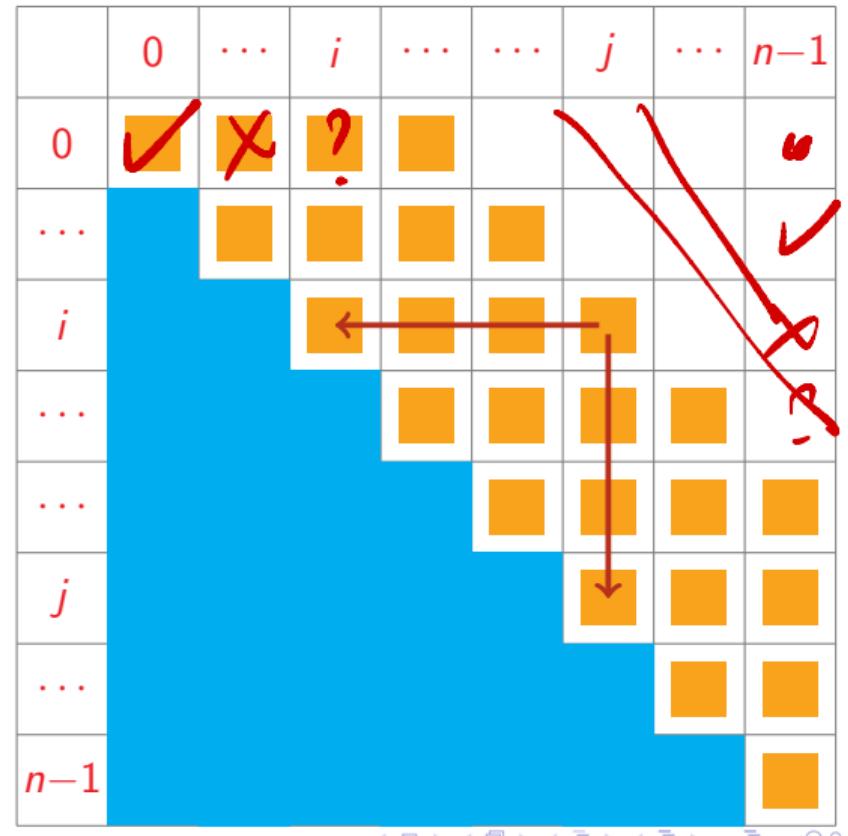
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



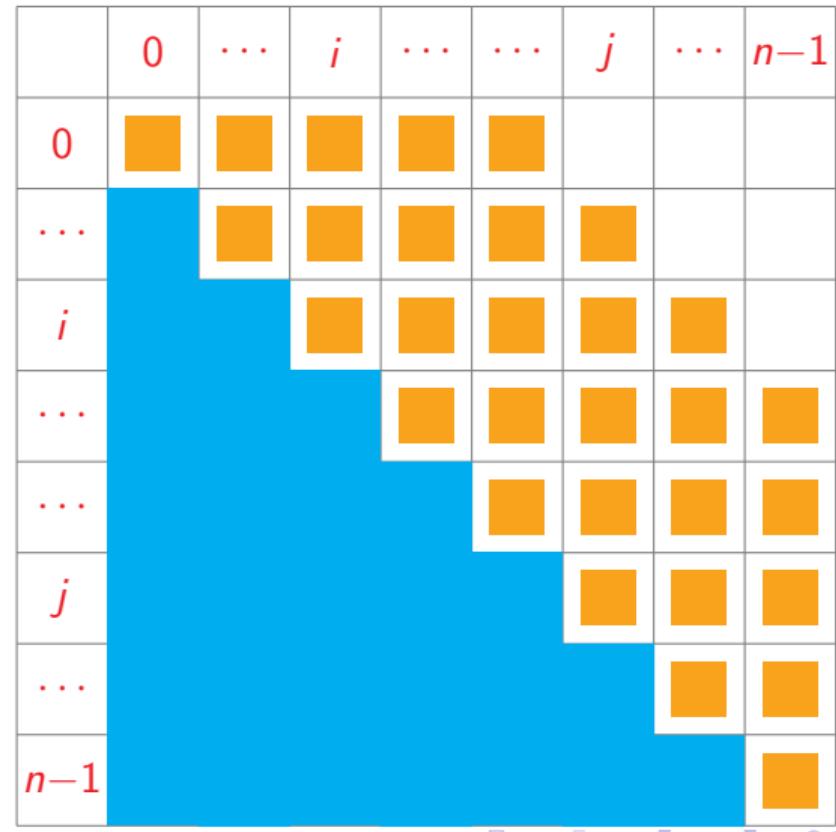
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



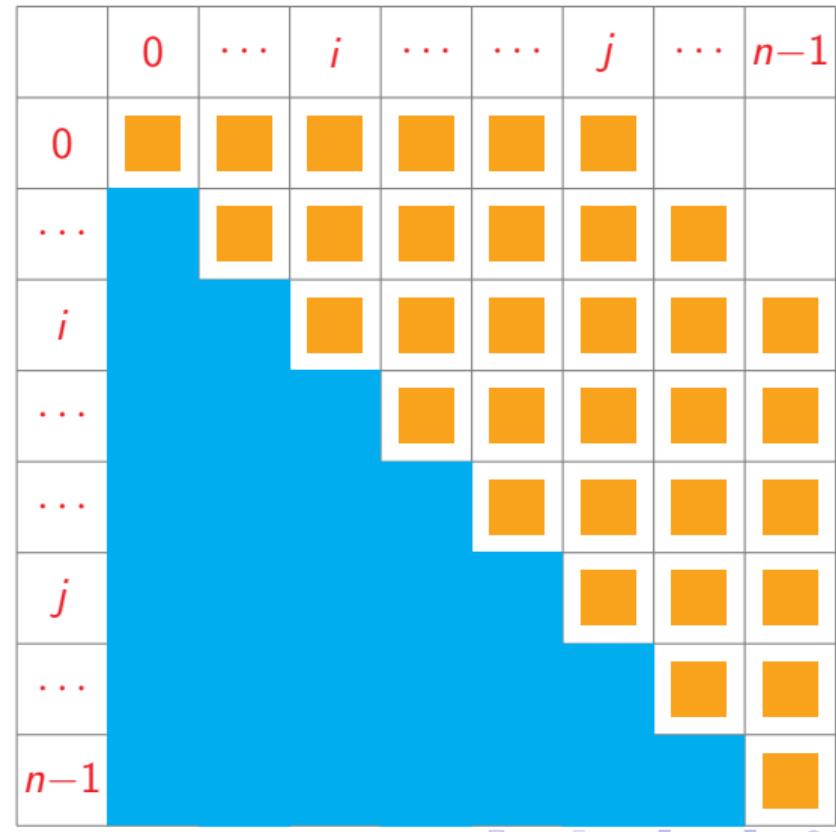
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



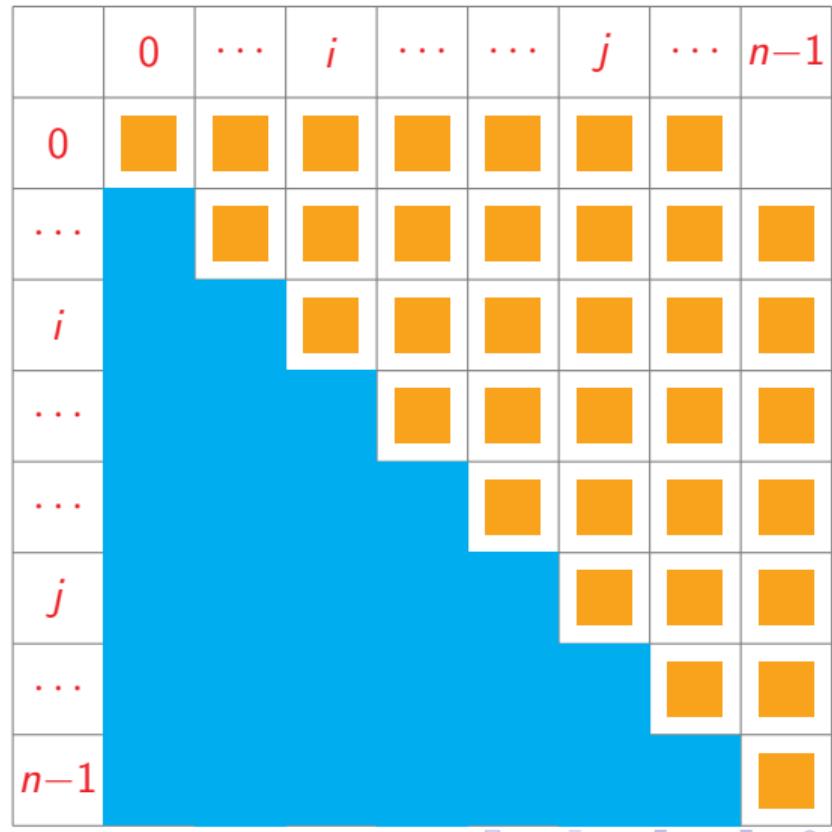
# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal



# Subproblem dependency

- Compute  $C(i,j)$ ,  $0 \leq i,j < n$ 
  - Only for  $i \leq j$
  - Entries above main diagonal
- $C(i,j)$  depends on  $C(i, k-1)$ ,  $C(k,j)$  for every  $i < k \leq j$ 
  - $O(n)$  dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal

$C(0, n-1)$

	0	...	i	...	...	j	...	$n-1$
0	X	✓						
...								X
i								✓
...								
j								
...								
$n-1$								

# Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                      C[i+1,j] +
                      dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

# Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                      C[i+1,j] +
                      dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

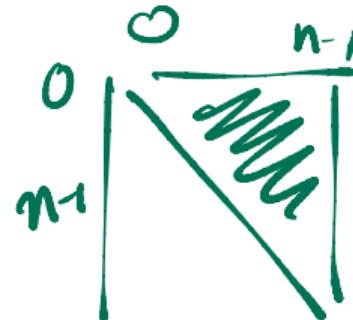
Complexity

# Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                C[i+1,j] +
                dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

## Complexity

- We have to fill a table of size  $O(n^2)$



# Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                      C[i+1,j] +
                      dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

## Complexity

- We have to fill a table of size  $O(n^2)$
- Filling each entry takes  $O(n)$

# Implementation

```
def C(dim):
    # dim: dimension matrix,
    #       entries are pairs (r_i,c_i)
    import numpy as np
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] +
                      C[i+1,j] +
                      dim[i][0]*dim[i+1][0]*dim[j][1]
            for k in range(i+1,j+1):
                C[i,j] = min(C[i,j],
                              C[i,k-1] + C[k,j] +
                              dim[i][0]*dim[k][0]*dim[j][1])
    return(C[0,n-1])
```

## Complexity

- We have to fill a table of size  $O(n^2)$
- Filling each entry takes  $O(n)$
- Overall,  $O(n^3)$