



Dynamic Programming

Kumkum Saxena

Contents

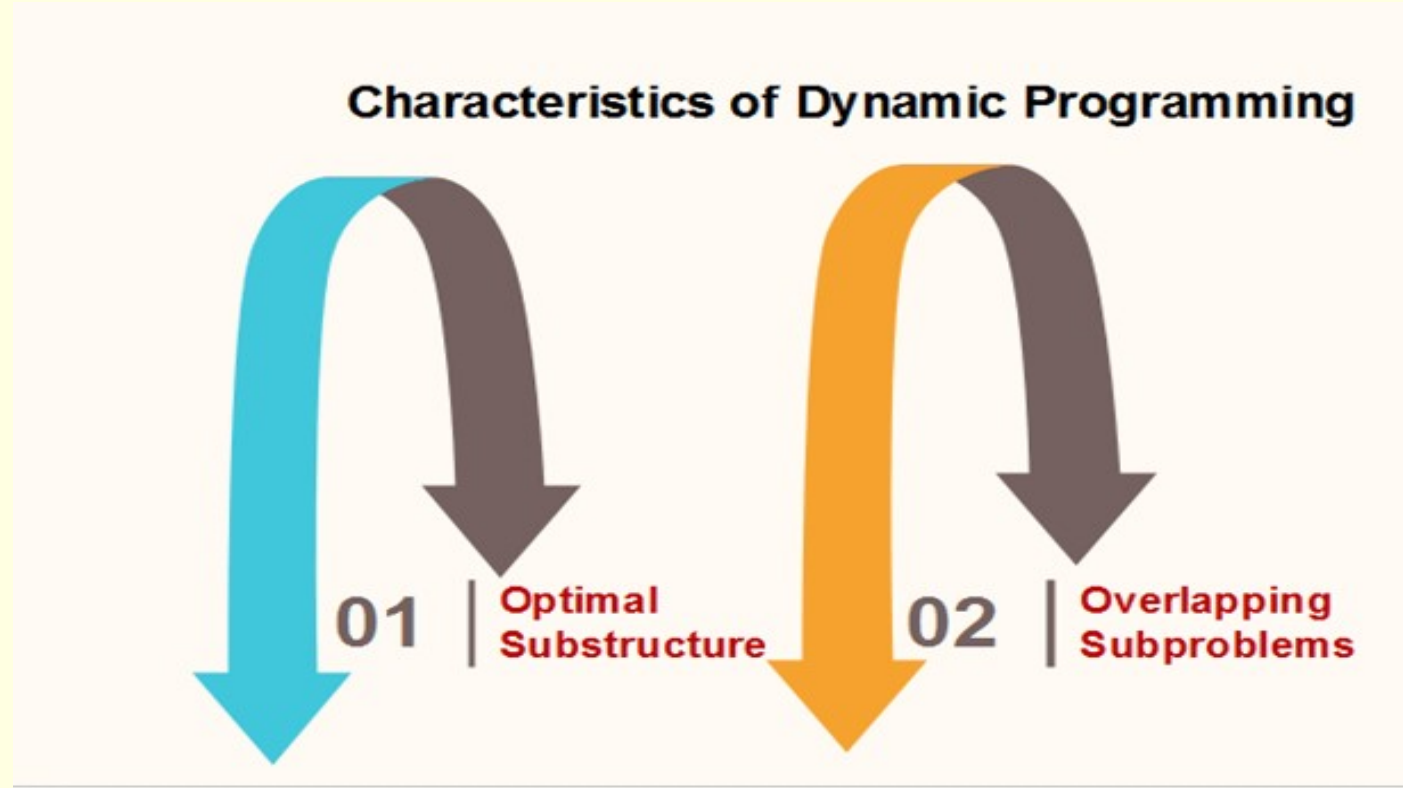
- Introduction Dynamic algorithms
- All pair shortest path
- 0/1 knapsack
- Travelling salesman problem
- Matrix Chain Multiplication
- Optimal binary search tree (OBST)
- Analysis of All problems

-
- Dynamic Programming is the most powerful design technique for solving optimization problems.
 - Divide & Conquer algorithm partition the problem into disjoint subproblems solve the subproblems recursively and then combine their solution to solve the original problems.
 - Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

-
- Dynamic Programming solves each subproblems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.
 - Dynamic Programming is a **Bottom-up approach**- we solve all possible small problems and then combine to obtain solutions for bigger problems.
 - Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appealing to the "**principle of optimality**".

Characteristics of Dynamic Programming:

- Dynamic Programming works when a problem has the following features:-

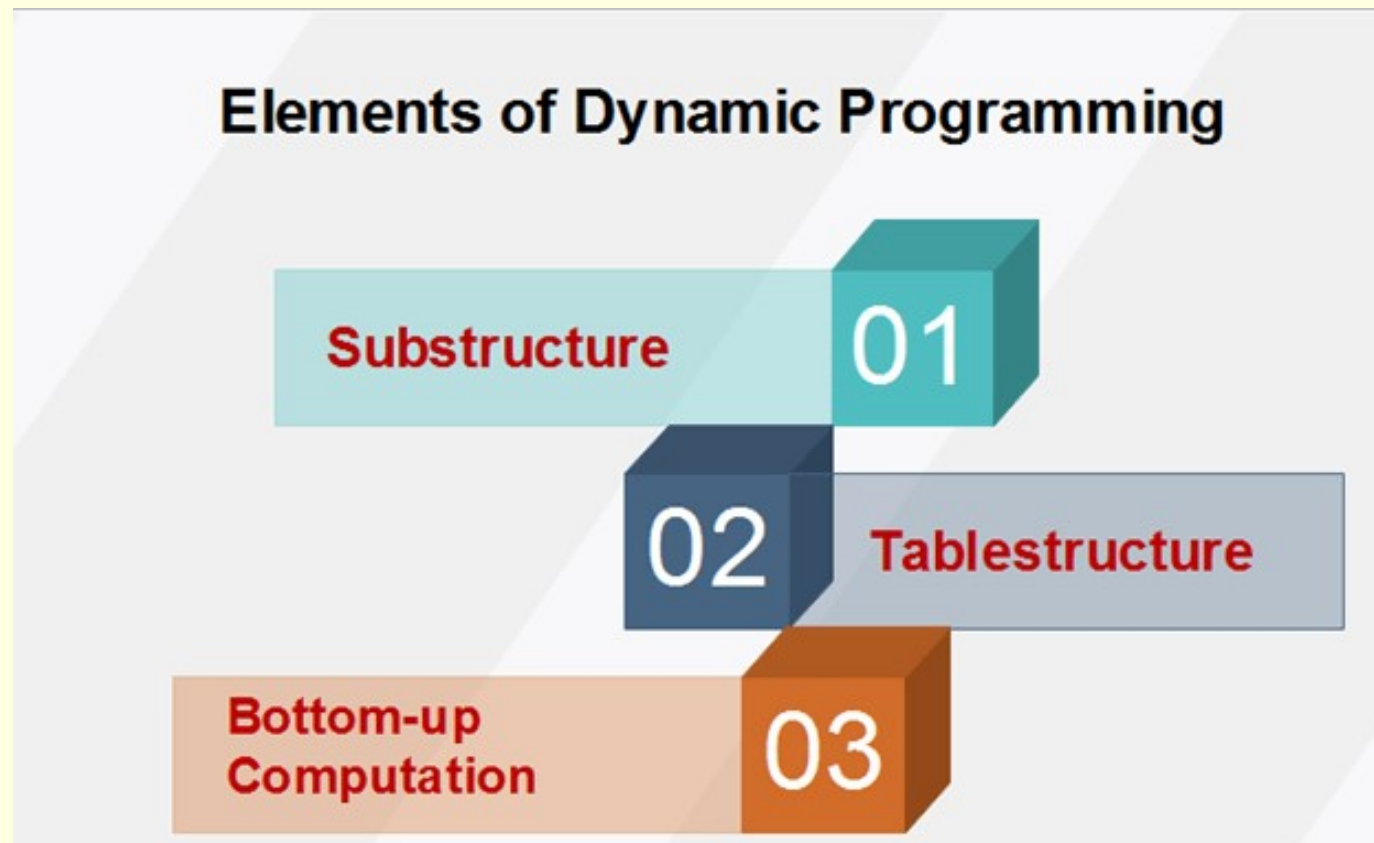


-
- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
 - **Overlapping subproblems:** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

-
- If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping subproblems, then we can improve on a recursive implementation by computing each subproblem only once.

-
- If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.
 - If the space of subproblems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

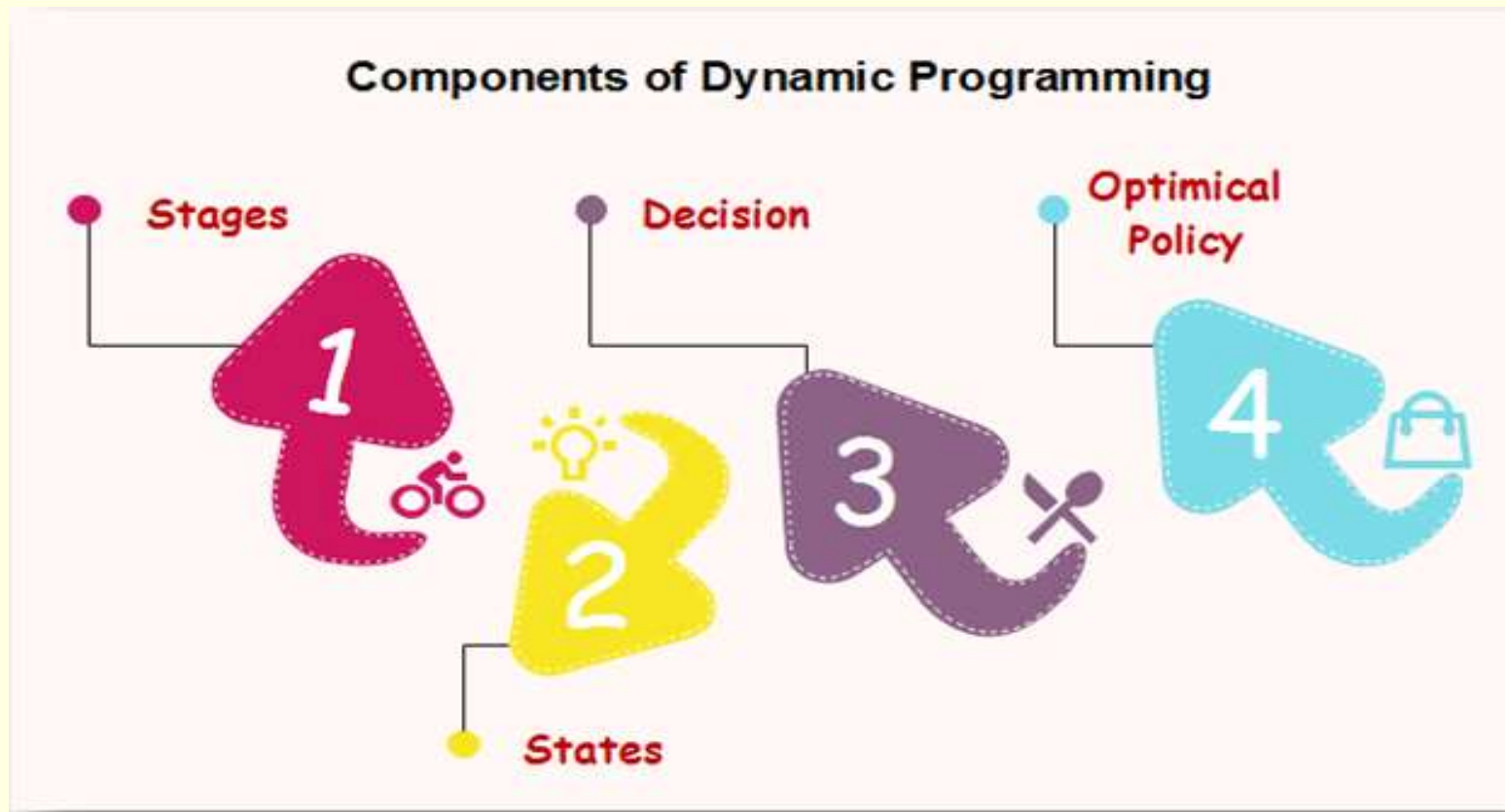
Elements of Dynamic Programming



-
- **Substructure:** Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.
 - **Table Structure:** After solving the subproblems, store the results to the subproblems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.

-
- **Bottom-up Computation:** Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.

Components of Dynamic programming



-
- **Stages:** The problem can be divided into several subproblems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
 - **States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.

-
- **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
 - **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.

Development of Dynamic Programming Algorithm

- Characterize the structure of an optimal solution.
- Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
- Compute the value of the optimal solution from the bottom up (starting with the smallest subproblems)
- Construct the optimal solution for the entire problem from the computed values of smaller subproblems.

Divide & Conquer Method vs Dynamic Programming.

Divide & Conquer Method

1. It deals (involves) three steps at each level of recursion:

Divide the problem into a number of subproblems.

Conquer the subproblems by solving them recursively.

Combine the solution to the subproblems into the solution for original subproblems.

2. It is Recursive.

3. It does more work on subproblems and hence has more time consumption.

4. It is a top-down approach.

5. In this subproblems are independent of each other.

6. **For example:** Merge Sort & Binary Search etc.

Divide & Conquer Method vs Dynamic Programming.

Dynamic Programming

1. It involves the sequence of four steps:

- Characterize the structure of optimal solutions.
- Recursively defines the values of optimal solutions.
- Compute the value of optimal solutions in a Bottom-up minimum.
- Construct an Optimal Solution from computed information.

2. It is non Recursive.

3. It solves subproblems only once and then stores in the table.

4. It is a Bottom-up approach.

5. In this subproblems are interdependent.

6. For example: Matrix Multiplication.

Dynamic Programming and Greedy Method

Dynamic Programming

1. Dynamic Programming is used to obtain the optimal solution.
2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.
3. Less efficient as compared to a greedy approach
4. Example: 0/1 Knapsack
5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.

Dynamic Programming and Greedy Method

Greedy Method

1. Greedy Method is also used to get the optimal solution.
2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
3. More efficient as compared to a greedy approach
4. Example: Fractional Knapsack
5. In Greedy Method, there is no such guarantee of getting Optimal Solution.

Matrix-Chain Multiplication

Problem: given a sequence $\langle A_1, A_2, \dots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix compatibility:

$$C = A \cdot B$$

$$\text{col}_A = \text{row}_B$$

$$\text{row}_C = \text{row}_A$$

$$\text{col}_C = \text{col}_B$$

$$C = A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$$

$$\text{col}_i = \text{row}_{i+1}$$

$$\text{row}_C = \text{row}_{A_1}$$

$$\text{col}_C = \text{col}_{A_n}$$

Matrix-chain Multiplication ...contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices

- Let us compute the product $A_1A_2A_3A_4$

- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$

2. $(A_1((A_2A_3)A_4))$

3. $((A_1A_2)(A_3A_4))$

4. $((A_1(A_2A_3))A_4)$

5. $((A_1A_2)A_3)A_4$

MATRIX-MULTIPLY(A, B)

if columns[A] \neq rows[B]

then error "incompatible dimensions"

```
else for  $i \leftarrow 1$  to rows[A]
```

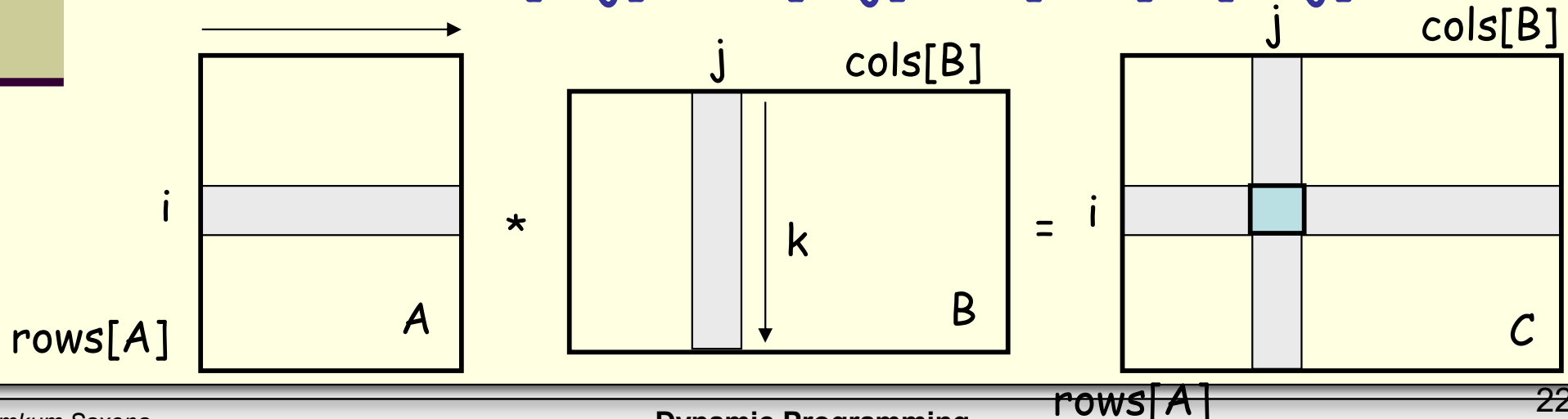
do for $j \leftarrow 1$ to $\text{columns}[B]$

do $C[i, j] = 0$

```
for k ← 1 to columns[A]
```

do $C[i, j] \leftarrow C[i, j] + A[i, k] B[k, j]$

rows[A] · cols[A] · cols[B]
multiplications



Matrix-Chain Multiplication

- In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

- Parenthesize the product to get the order in which matrices are multiplied

- *E.g.:* $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$
 $= (A_1 \cdot (A_2 \cdot A_3))$

- Which one of these orderings should we choose?
 - The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

Example

$$A_1 \cdot A_2 \cdot A_3$$

- A_1 : 10×100
- A_2 : 100×5
- A_3 : 5×50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2 = 10 \times 100 \times 5 = 5,000$ (10×5)
 $((A_1 \cdot A_2) \cdot A_3) = 10 \times 5 \times 50 = 2,500$

Total: 7,500 scalar multiplications

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3 = 100 \times 5 \times 50 = 25,000$ (100×50)
 $(A_1 \cdot (A_2 \cdot A_3)) = 10 \times 100 \times 50 = 50,000$

Total: 75,000 scalar multiplications

one order of magnitude difference!!

Matrix-chain Multiplication ...contd

Example: Consider three matrices $A_{10 \times 100}$, $B_{100 \times 5}$, and $C_{5 \times 50}$

There are 2 ways to parenthesize

■ $((AB)C) = D_{10 \times 5} \cdot C_{5 \times 50}$

■ $AB \Rightarrow 10 \cdot 100 \cdot 5 = 5,000$ scalar multiplications

■ $DC \Rightarrow 10 \cdot 5 \cdot 50 = 2,500$ scalar multiplications

} Total:
7,500

■ $(A(BC)) = A_{10 \times 100} \cdot E_{100 \times 50}$

■ $BC \Rightarrow 100 \cdot 5 \cdot 50 = 25,000$ scalar multiplications

■ $AE \Rightarrow 10 \cdot 100 \cdot 50 = 50,000$ scalar multiplications

Total:
75,000



Matrix-Chain Multiplication: Problem Statement

- Given a chain of matrices $\langle A_1, A_2, \dots, A_n \rangle$, where A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

$$\begin{array}{ccccccc} A_1 & \cdot & A_2 & \cdots & A_i & \cdot & A_{i+1} & \cdots & A_n \\ p_0 \times p_1 & & p_1 \times p_2 & & p_{i-1} \times p_i & & p_i \times p_{i+1} & & p_{n-1} \times p_n \end{array}$$

What is the number of possible parenthesizations?

- Exhaustively checking all possible parenthesizations is not efficient!
- Brute force method of exhaustive search takes time exponential in n
- It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$

Dynamic Programming Approach

The structure of an optimal solution

- Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \dots A_j$
- An optimal parenthesization of the product $A_1 A_2 \dots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \leq k < n$
- First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach

...contd

- **Key observation:** parenthesizations of the subchains $A_1A_2\dots A_k$ and $A_{k+1}A_{k+2}\dots A_n$ must also be optimal if the parenthesization of the chain $A_1A_2\dots A_n$ is optimal (why?)
- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

1. The Structure of an Optimal Parenthesization

- Notation:

$$A_{i\dots j} = A_i A_{i+1} \cdots A_j, i \leq j$$

- Suppose that an optimal parenthesization of $A_{i\dots j}$ splits the product between A_k and A_{k+1} , where $i \leq k < j$

$$\begin{aligned} A_{i\dots j} &= A_i A_{i+1} \cdots A_j \\ &= A_i A_{i+1} \cdots A_k A_{k+1} \cdots A_j \\ &= A_{i\dots k} A_{k+1\dots j} \end{aligned}$$

Optimal Substructure

$$A_{i\dots j} = A_{i\dots k} A_{k+1\dots j}$$

- The parenthesization of the “prefix” $A_{i\dots k}$ must be an optimal parenthesization
- If there were a less costly way to parenthesize $A_{i\dots k}$, we could substitute that one in the parenthesization of $A_{i\dots j}$ and produce a parenthesization with a lower cost than the optimum \Rightarrow contradiction!
- An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems

2. A Recursive Solution

- Subproblem:

determine the minimum cost of parenthesizing

$$A_{i\dots j} = A_i A_{i+1} \cdots A_j \quad \text{for } 1 \leq i \leq j \leq n$$

- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i\dots j}$

- full problem ($A_{1..n}$): $m[1, n]$

- $i = j$: $A_{i\dots i} = A_i \Rightarrow m[i, i] = 0$, for $i = 1, 2, \dots, n$

2. A Recursive Solution

- Consider the subproblem of parenthesizing

$$A_{i\dots j} = A_i A_{i+1} \dots A_j \quad \text{for } 1 \leq i \leq j \leq n$$

$$= \underbrace{A_{i\dots k}}_{m[i, k]} \underbrace{A_{k+1\dots j}}_{m[k+1, j]} \quad \text{for } i \leq k < j$$

$p_{i-1} p_k p_j$

- Assume that the optimal parenthesization splits the product $A_i A_{i+1} \dots A_j$ at k ($i \leq k < j$)

$$m[i, j] = \underbrace{m[i, k]} + \underbrace{m[k+1, j]} + \underbrace{p_{i-1} p_k p_j}$$

min # of multiplications
to compute $A_{i\dots k}$

min # of multiplications
to compute $A_{k+1\dots j}$

of multiplications
to compute $A_{i\dots k} A_{k\dots j}$

2. A Recursive Solution (cont.)

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$$

- We do not know the value of k
 - There are $j - i$ possible values for k : $k = i, i+1, \dots, j-1$
- Minimizing the cost of parenthesizing the product $A_i A_{i+1} \dots A_j$ becomes:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

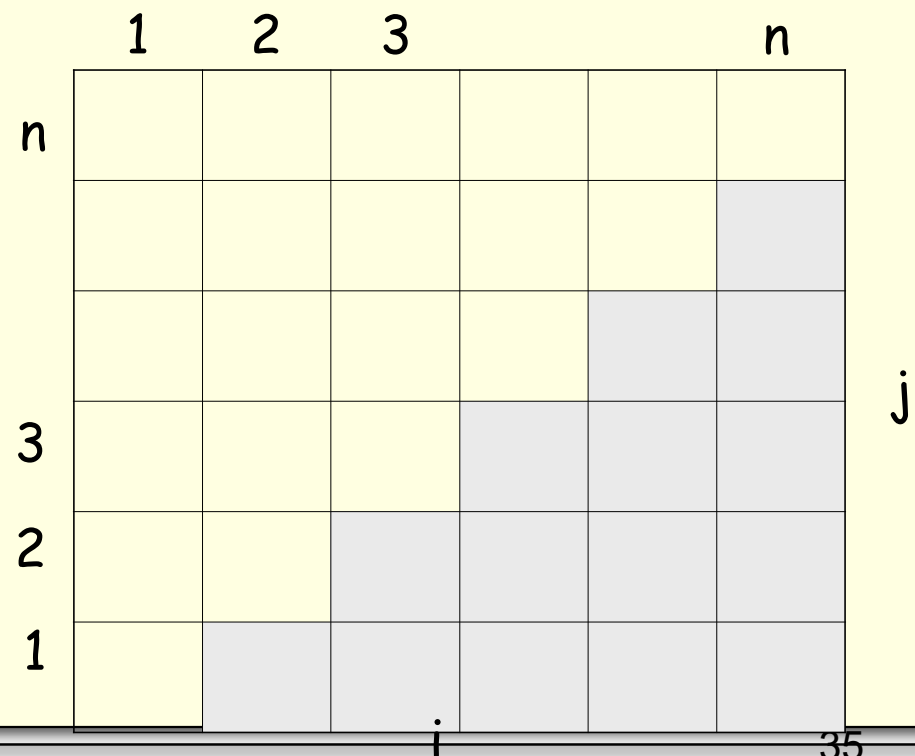
3. Computing the Optimal Costs

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- Computing the optimal solution recursively takes exponential time!
 - How many subproblems?
-
- Diagram illustrating subproblems for the Fibonacci sequence:
- | | | | | | | |
|---|---|---|---|--|--|---|
| | 1 | 2 | 3 | | | n |
| n | | | | | | |

$$\Rightarrow \Theta(n^2)$$

- Parenthesize $A_{i\dots j}$
for $1 \leq i \leq j \leq n$
- One problem for each
choice of i and j



3. Computing the Optimal Costs (cont.)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- How do we fill in the tables $m[1..n, 1..n]$?
 - Determine which entries of the table are used in computing $m[i, j]$

$$A_{i...j} = A_{i...k} A_{k+1...j}$$

- Subproblems' size is one less than the original size
- **Idea:** fill in m such that it corresponds to solving problems of increasing length

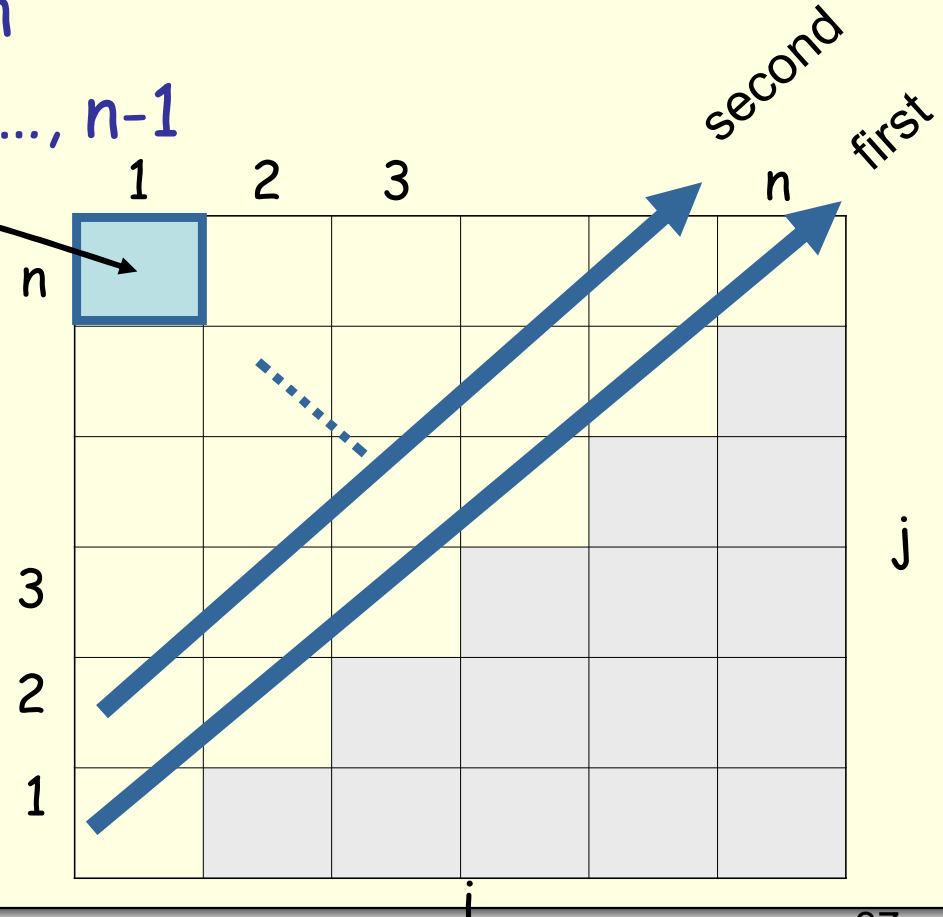
3. Computing the Optimal Costs (cont.)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- Length = 1: $i = j, i = 1, 2, \dots, n$
- Length = 2: $j = i + 1, i = 1, 2, \dots, n-1$

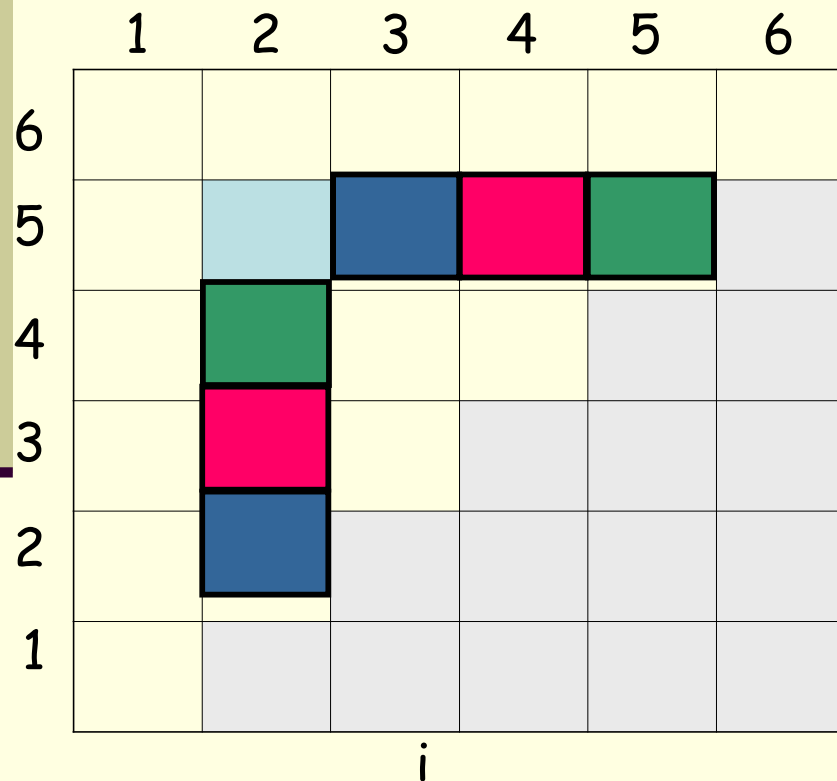
$m[1, n]$ gives the optimal solution to the problem

Compute rows from bottom to top and from left to right



Example: $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1p_2p_5 & k = 2 \\ m[2, 3] + m[4, 5] + p_1p_3p_5 & k = 3 \\ m[2, 4] + m[5, 5] + p_1p_4p_5 & k = 4 \end{cases}$$



- Values $m[i, j]$ depend only on values that have been previously computed

Example $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

Compute $A_1 \cdot A_2 \cdot A_3$

- $A_1: 10 \times 100$ ($p_0 \times p_1$)
- $A_2: 100 \times 5$ ($p_1 \times p_2$)
- $A_3: 5 \times 50$ ($p_2 \times p_3$)

$m[i, i] = 0$ for $i = 1, 2, 3$

$$\begin{aligned} m[1, 2] &= m[1, 1] + m[2, 2] + p_0p_1p_2 && (A_1A_2) \\ &= 0 + 0 + 10 * 100 * 5 = 5,000 \end{aligned}$$

$$\begin{aligned} m[2, 3] &= m[2, 2] + m[3, 3] + p_1p_2p_3 && (A_2A_3) \\ &= 0 + 0 + 100 * 5 * 50 = 25,000 \end{aligned}$$

$$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0p_1p_3 = 75,000 & (A_1(A_2A_3)) \\ m[1, 2] + m[3, 3] + p_0p_2p_3 = 7,500 & ((A_1A_2)A_3) \end{cases}$$

	1	2	3
3	2 7500	2 25000	0
2	1 5000	0	
1	0		

Matrix-Chain-Order(*p*)

```
MATRIX-CHAIN-ORDER(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3      do m[i, i] ← 0
4  for l ← 2 to n           ▷ l is the chain length.
5      do for i ← 1 to n − l + 1
6          do j ← i + l − 1
7              m[i, j] ← ∞
8              for k ← i to j − 1
9                  do q ← m[i, k] + m[k + 1, j] + pi−1pkpj
10                 if q < m[i, j]
11                     then m[i, j] ← q
12                         s[i, j] ← k
13  return m and s
```

$O(N^3)$

4. Construct the Optimal Solution

- In a similar matrix s we keep the optimal values of k
- $s[i, j] =$ a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}

	1	2	3		n
n					
			k		
3					
2					
1					

j

4. Construct the Optimal Solution

- $s[1, n]$ is associated with the entire product $A_{1..n}$
 - The final matrix multiplication will be split at $k = s[1, n]$
$$A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$$
 - For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization

	1	2	3			n
n						
3						
2						
1						

j

4. Construct the Optimal Solution

- $s[i, j]$ = value of k such that the optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1}

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

i

j

- $s[1, n] = 3 \Rightarrow A_{1..6} = A_{1..3} A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1} A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5} A_{6..6}$

4. Construct the Optimal Solution (cont.)

PRINT-OPT-PARENS(s, i, j)

if $i = j$

then print " A_i "

else print "("

PRINT-OPT-PARENS($s, i, s[i, j]$)

PRINT-OPT-PARENS($s, s[i, j] + 1, j$)

print ")"

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

i

j

Example: $A_1 \cdot \cdot \cdot A_6$ $((A_1(A_2A_3))((A_4A_5)A_6))$

PRINT-OPT-PARENS(s, i, j)

```

if i = j
  then print "A"i
else print "("
  PRINT-OPT-PARENS( $s, i, s[i, j]$ )
  PRINT-OPT-PARENS( $s, s[i, j] + 1, j$ )
  print ")"

```

P-O-P($s, 1, 6$) $s[1, 6] = 3$

$i = 1, j = 6$ "(" P-O-P($s, 1, 3$) $s[1, 3] = 1$

$i = 1, j = 3$ "(" P-O-P($s, 1, 1$) $\Rightarrow "A_1"$

P-O-P($s, 2, 3$) $s[2, 3] = 2$

$i = 2, j = 3$ "(" P-O-P($s, 2, 2$) $\Rightarrow "A_2"$

P-O-P($s, 3, 3$) $\Rightarrow "A_3"$

)

)

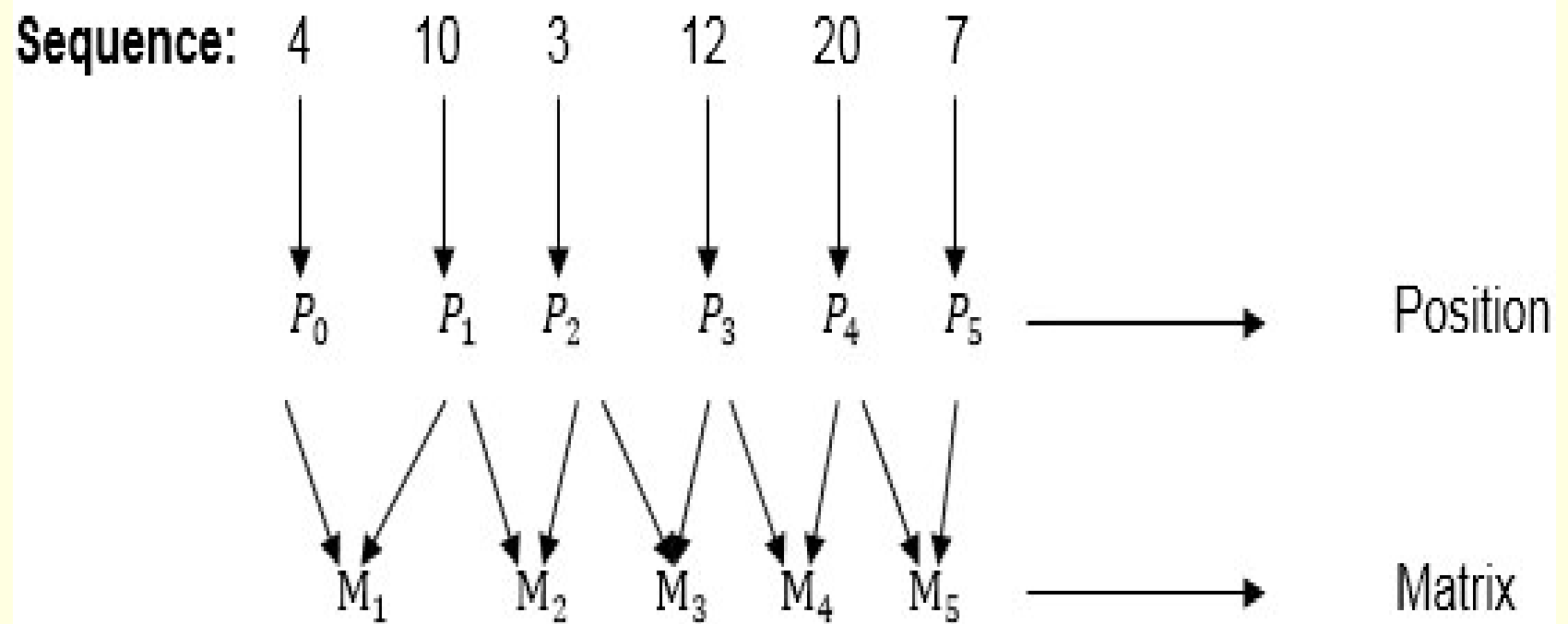
...

$s[1..6, 1..6]$	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					
	i					
						j

Example

- We are given the sequence {4, 10, 3, 12, 20, and 7}.
- The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7.
- We need to compute $M[i,j]$, $0 \leq i, j \leq 5$. We know $M[i, i] = 0$ for all i .

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5



- On the basis of sequence, we make a formula

For $M_i \longrightarrow p[i]$ as column
 $p[i-1]$ as row

Calculation of Product of 2 matrices:

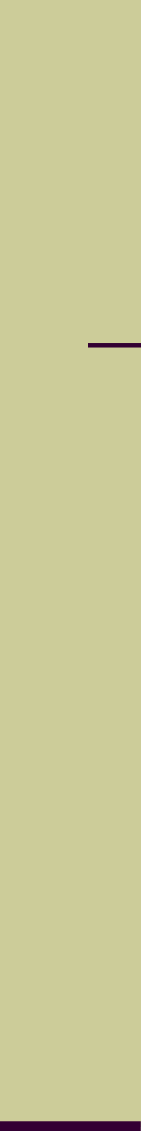
Calculation of Product of 2 matrices:

$$\begin{aligned} 1. \quad m(1, 2) &= m_1 \times m_2 \\ &= 4 \times 10 \times 10 \times 3 \\ &= 4 \times 10 \times 3 = 120 \end{aligned}$$

$$\begin{aligned} 2. \quad m(2, 3) &= m_2 \times m_3 \\ &= 10 \times 3 \times 3 \times 12 \\ &= 10 \times 3 \times 12 = 360 \end{aligned}$$

$$\begin{aligned} 3. \quad m(3, 4) &= m_3 \times m_4 \\ &= 3 \times 12 \times 12 \times 20 \\ &= 3 \times 12 \times 20 = 720 \end{aligned}$$

$$\begin{aligned} 4. \quad m(4, 5) &= m_4 \times m_5 \\ &= 12 \times 20 \times 20 \times 7 \\ &= 12 \times 20 \times 7 = 1680 \end{aligned}$$



1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Product of 3 matrices:

$$M[1, 3] = \min \left\{ \begin{array}{l} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \end{array} \right\}$$

$$\mathbf{M[1, 3] = 264}$$

$$M[2, 4] = \min \left\{ \begin{array}{l} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{array} \right\}$$

$$\mathbf{M[2, 4] = 1320}$$

$$M[3, 5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases}$$

$M[3, 5] = 1140$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Product of 4 matrices:

$$M[1, 4] = \min \left\{ \begin{array}{l} M[1,3] + M[4,4] + p_0 p_3 p_4 = 264 + 0 + 4 \cdot 12 \cdot 20 = 1224 \\ M[1,2] + M[3,4] + p_0 p_2 p_4 = 120 + 720 + 4 \cdot 3 \cdot 20 = 1080 \\ M[1,1] + M[2,4] + p_0 p_1 p_4 = 0 + 1320 + 4 \cdot 10 \cdot 20 = 2120 \end{array} \right.$$

$$M[1, 4] = 1080$$

$$M[2, 5] = \min \left\{ \begin{array}{l} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \end{array} \right\}$$

$$M[2, 5] = 1350$$

1	2	3	4	5		1	2	3	4	5	
0	120	264			1	0	120	264	1080		1
	0	360	1320		2		0	360	1320	1350	2
		0	720	1140	3			0	720	1140	3
			0	1680	4				0	1680	4
				0	5					0	5



Product of 5 matrices:

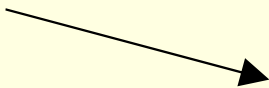
$$M[1, 5] = \min \left\{ \begin{array}{l} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \end{array} \right.$$

$$M[1, 5] = 1344$$

1	2	3	4	5			1	2	3	4	5	
0	120	264	1080		1		0	120	264	1080	1344	1
	0	360	1320	1350	2			0	360	1320	1350	2
		0	720	1140	3	→			0	720	1140	3
			0	1680	4					0	1680	4
				0	5						0	5

Example

- Show how to multiply this matrix chain optimally
- Solution on the board
 - Minimum cost 15,125
 - Optimal parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$



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

Matrix-Chain multiplication (cont.)

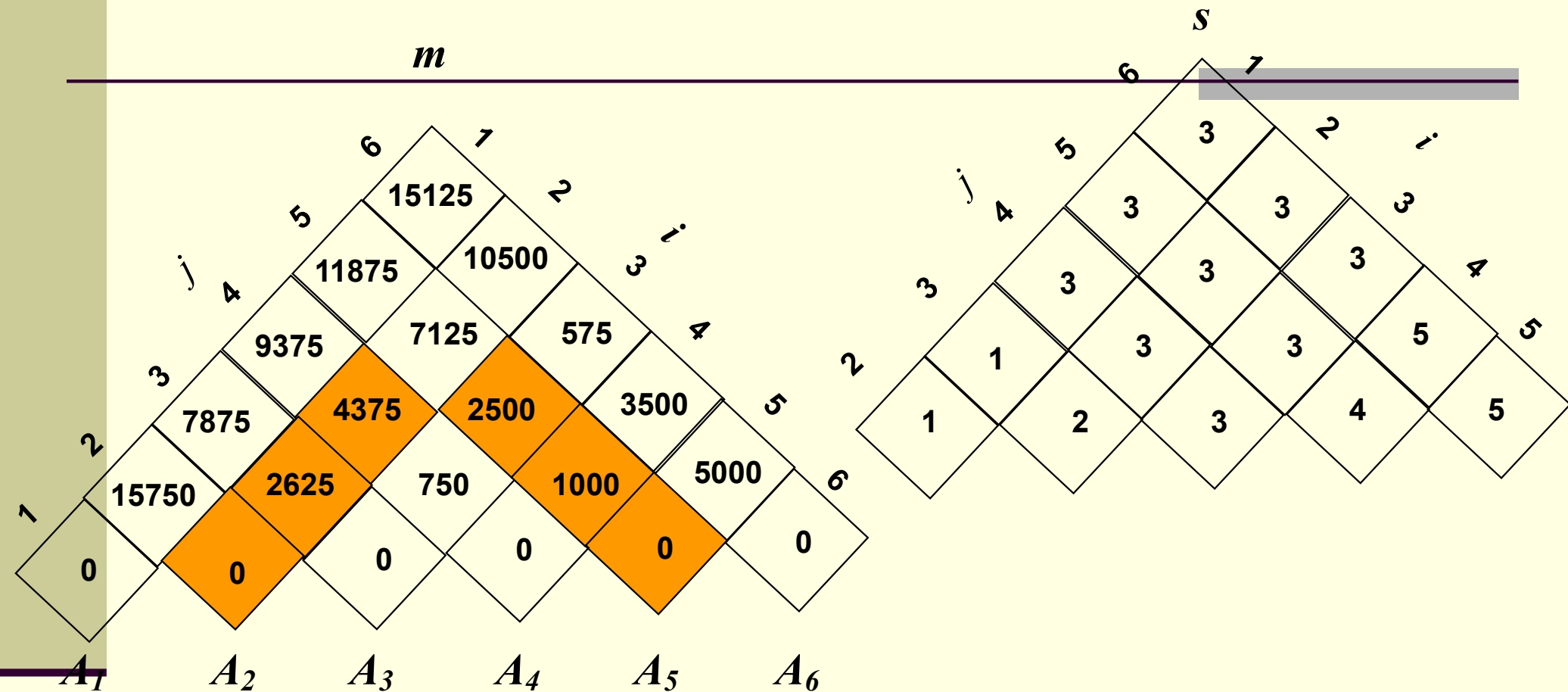
An example:

<u>matrix</u>	<u>dimension</u>
A_1	30 x 35
A_2	35 x 15
A_3	15 x 5
A_4	5 x 10
A_5	10 x 20
A_6	20 x 25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$= (7125)$

Matrix-Chain multiplication (cont.)



Matrix-Chain multiplication (Contd.)

RUNNING TIME:

Recursive solution takes exponential time.

Matrix-chain order yields a running time of $O(n^3)$