

# VE281

## Data Structures and Algorithms

### **Dynamic Programming**

#### **Learning Objectives:**

- Understand the basic idea of dynamic programming
- Know under what situation dynamic programming could be applied

# Outline

- Dynamic Programming
  - Motivation
  - Example: Matrix-Chain Multiplication
  - Summary

# Limitation of Divide and Conquer

- Recursively solving subproblems can result in the same computations being repeated when the subproblems **overlap**.

- For example: computing the **Fibonacci sequence**

$$f_0 = 0; f_1 = 1; f_n = f_{n-1} + f_{n-2}, n \geq 2$$

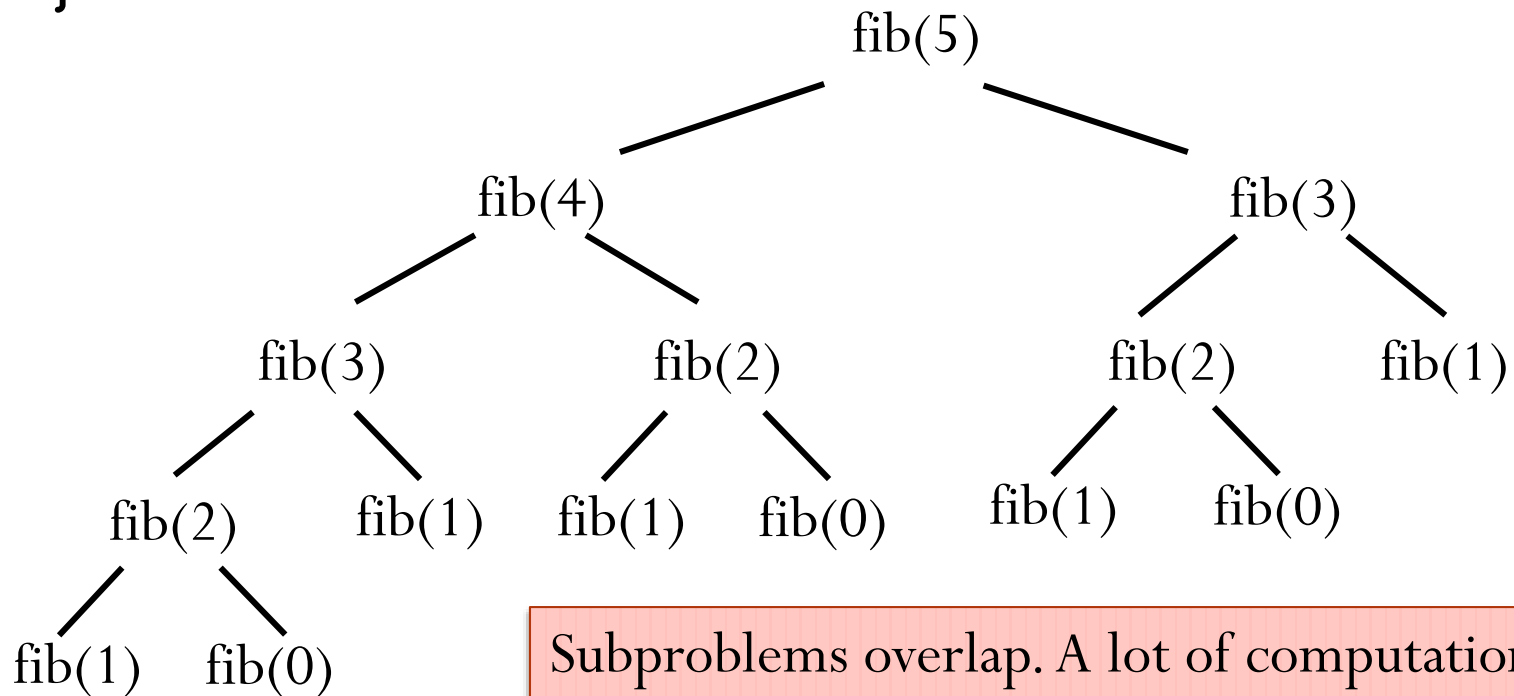
- Divide and conquer approach:

```
int fib(int n) {  
    if(n <= 1) return n;  
    return fib(n-1)+fib(n-2);  
}
```

# Fibonacci Sequence

## Divide and Conquer Solution

```
int fib(int n) {  
    if(n <= 1) return n;  
    return fib(n-1)+fib(n-2);  
}
```



Subproblems overlap. A lot of computation is wasted. Time complexity is  $\Omega(1.5^n)$ .

# Fibonacci Sequence

## Iterative Solution

- We can also compute the Fibonacci sequence in iterative way:

```
int fib(int n) {  
    f[0] = 0; f[1] = 1;  
    for(i = 2 to n)  
        f[i] = f[i-1]+f[i-2];  
    return f[n];  
}
```

- Time complexity is  $\Theta(n)$ .

# Dynamic Programming

- Used when a problem can be divided into subproblems that **overlap**.
  - Solve each subproblem **once** and store the solution in a table.
  - If a subproblem is encountered **again**, simply look up its solution in the table.
  - Reconstruct the solution to the original problem from the solutions to the subproblems.
- The more overlap the better, as this reduces the number of subproblems.
- Dynamic programming can be applied to solve **optimization problem**.

# Optimization Problem

- Many problems we encounter are **optimization problems**:
  - A problem in which some function (called the **objective function**) is to be optimized (usually minimized or maximized) subject to some **constraints**.
- The solutions that satisfy the constraints are called **feasible solutions**.
- The number of feasible solutions is typically very large.
- We obtain the optimal solution by **searching** the feasible solution space.

# Optimization Problem

## Example

- Minimum spanning tree.
  - Objective function: the sum of all edge weights.
  - Constraints: the subgraph must be a spanning tree.



# Outline

- Dynamic Programming
  - Motivation
  - Example: Matrix-Chain Multiplication
  - Summary

# Matrix-Chain Multiplication

- What is the cost of multiplying two matrices  $A$  and  $B$ ?
  - Suppose  $A$  is a  $p \times q$  matrix and  $B$  is a  $q \times r$  matrix.
  - Since the time to compute  $C = AB$  is dominated by the number of **scalar multiplications**, we use the number of scalar multiplications as the complexity measure.
- $C_{ij} = \sum_{k=1}^q A_{ik} B_{kj}$ .
  - We need  $q$  scalar multiplications to calculate  $C_{ij}$ .
  - $C$  is of size  $p \times r$ .
- The number of scalar multiplications is  $pqr$ .

# Matrix-Chain Multiplication

- Now how would you compute the multiplication of three matrices  $A \times B \times C$ ?
  - Suppose  $A$  is of size  $100 \times 1$ ,  $B$  is of size  $1 \times 100$ , and  $C$  is of size  $100 \times 1$ .
- If we multiply as  $(A \times B) \times C$ , the number of scalar multiplications is 20000.
- If we multiply as  $A \times (B \times C)$ , the number of scalar multiplications is 200.

# Matrix-Chain Multiplication

- If we want to multiply a chain of matrices  $A_1 \times A_2 \times \cdots \times A_n$ , where  $A_i$  is of size  $p_{i-1} \times p_i$ , what is the best order of multiplication to minimize the number of scalar multiplications?
- This is an optimization problem.
- It can be proved that number of different orders on  $n$  matrices is  $\Omega(4^n/n^{1.5})$ .
- Instead of enumerating all of the orders, can we do better to solve the optimization problem?

# Matrix-Chain Multiplication

- For simplicity, define the problem of finding the optimal order to multiply  $A_i \times A_{i+1} \times \cdots \times A_j$  as  $Q_{ij}$ . The minimal number of scalar multiplications is  $m_{ij}$ .
  - We ultimately want to solve  $Q_{1n}$ .

# Matrix-Chain Multiplication

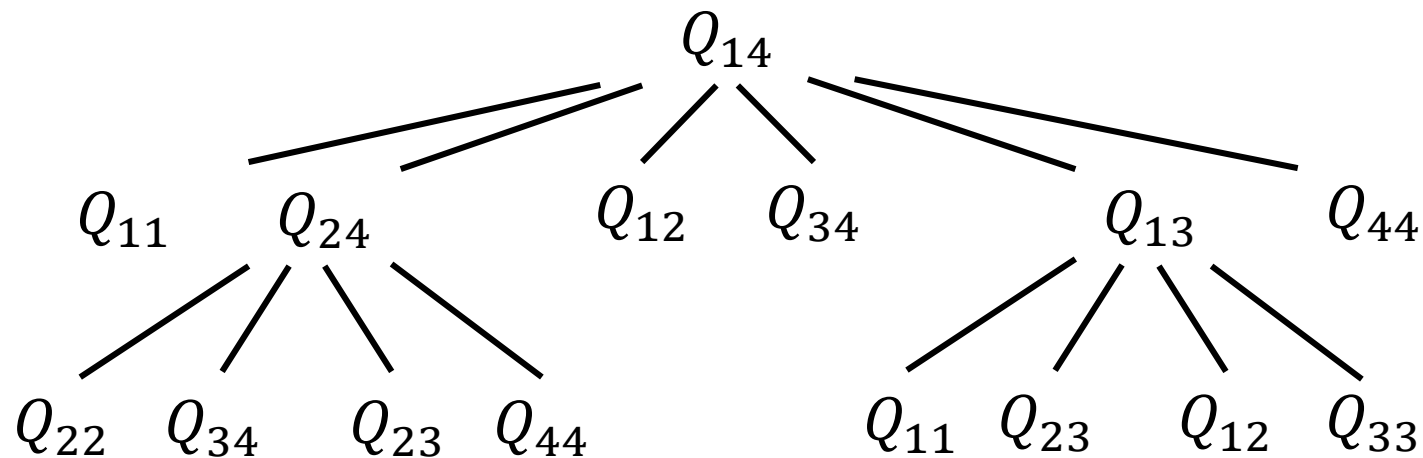
- Suppose in the optimal order for  $A_i \times \cdots \times A_j$ , the last multiplication is  $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$ .
- Then the order of computing  $A_i \times \cdots \times A_k$  in the **optimal** order of computing  $A_i \times \cdots \times A_j$  must be an **optimal** order to compute  $A_i \times \cdots \times A_k$ .
  - Why?
    - If not, then we copy and paste the better order  $\rightarrow$  we have a better order for computing  $A_i \times \cdots \times A_j$ !
    - Similar conclusion for computing  $A_{k+1} \times \cdots \times A_j$ .
- If we know  $k$ , we can divide the problem  $Q_{ij}$  into two smaller instances:  $Q_{ik}$  and  $Q_{(k+1)j}$ .

# Matrix-Chain Multiplication

- Assume we have known the minimum number of scalar multiplications for  $Q_{ik}$  and  $Q_{(k+1)j}$  as  $m_{ik}$  and  $m_{(k+1)j}$ .
  - Then  $m_{ij} = m_{ik} + m_{(k+1)j} + p_{i-1}p_kp_j$ .
- **However, we don't know  $k$ !** We need to consider all possible divisions, i.e., all  $i \leq k \leq j - 1$ .
- Thus, in order to solve  $Q_{ij}$ , we need to consider all subproblems  $Q_{ik}$  and  $Q_{(k+1)j}$ , for all  $i \leq k \leq j - 1$ .
  - $m_{ij} = \min_{i \leq k \leq j-1} (m_{ik} + m_{(k+1)j} + p_{i-1}p_kp_j)$

# Matrix-Chain Multiplication

- In summary, we can divide the problem into subproblems of the same form.



Many subproblems are overlapped.



# Matrix-Chain Multiplication

- The straightforward recursive algorithm has exponential time complexity.
  - However, it will encounter each subproblem many times in different branches of the tree.
- The total number of different subproblems is not exponential.
  - They are  $Q_{ij}$ , for  $1 \leq i \leq j \leq n$ .
  - The total number is  $n(n + 1)/2$ .
- Instead, we use a tabular, bottom-up approach.

# Matrix-Chain Multiplication

## Bottom-up Approach

- Apply the recursive relation:

$$m_{ij} = \min_{i \leq k \leq j-1} (m_{ik} + m_{(k+1)j} + p_{i-1}p_kp_j)$$

- Initial situation  $m_{11} = m_{22} = \dots = m_{nn} = 0$ .
- In the first round, we compute  $m_{12}, m_{23}, \dots, m_{(n-1)n}$ .
- In the second round, we compute  $m_{13}, m_{24}, \dots, m_{(n-2)n}$ .
- So on and so forth. In the  $l$ -th round, we compute  $m_{1(l+1)}, m_{2(l+2)}, \dots, m_{(n-l)n}$ .
- Finally, we compute  $m_{1n}$ .
- To obtain the multiplication order, we also record the partition  $k$  which gives the minimal  $m_{ij}$  as  $s_{ij}$ .

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

		$j$			
$m_{ij}$		1	2	3	4
$i$	1	0			
	2	—	0		
	3	—	—	0	
	4	—	—	—	0

		$j$			
$s_{ij}$		1	2	3	4
$i$	1	—			
	2	—	—		
	3	—	—	—	
	4	—	—	—	—

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

$m_{ij}$

	1	2	3	4
1	0			
2	—	0		
3	—	—	0	
4	—	—	—	0

$s_{ij}$

	1	2	3	4
1	—			
2	—	—		
3	—	—	—	
4	—	—	—	—

$$\begin{aligned}
 m_{i(i+1)} &= m_{ii} + m_{(i+1)(i+1)} + p_{i-1}p_i p_{i+1} \\
 &= p_{i-1}p_i p_{i+1}
 \end{aligned}$$

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

$m_{ij}$

		1	2	3	4
	$j$				
1		0	100		
2		—	0	10	
3		—	—	0	200
4		—	—	—	0
$i$					

$s_{ij}$

		1	2	3	4
	$j$				
1		—	1		
2		—	—	2	
3		—	—	—	3
4		—	—	—	—
$i$					

$$\begin{aligned}
 m_{i(i+1)} &= m_{ii} + m_{(i+1)(i+1)} + p_{i-1}p_i p_{i+1} \\
 &= p_{i-1}p_i p_{i+1}
 \end{aligned}$$

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

$m_{ij}$

		1	2	3	4
	$j$				
1		0	100		
2		—	0	10	
3		—	—	0	200
4		—	—	—	0
$i$					

$s_{ij}$

		1	2	3	4
	$j$				
1		—	1		
2		—	—	2	
3		—	—	—	3
4		—	—	—	—
$i$					

$$m_{i(i+2)} = \min\{m_{ii} + m_{(i+1)(i+2)} + p_{i-1}p_i p_{i+2},$$

$$m_{i(i+1)} + m_{(i+2)(i+2)} + p_{i-1}p_{i+1}p_{i+2}\}$$

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

$m_{ij}$

		1	2	3	4
	$j$				
1		0	100		
2		—	0	10	
3		—	—	0	200
4		—	—	—	0
$i$					

$s_{ij}$

		1	2	3	4
	$j$				
1		—	1		
2		—	—	2	
3		—	—	—	3
4		—	—	—	—
$i$					

$$m_{13} = \min\{m_{11} + m_{23} + p_0 p_1 p_3,$$

$$m_{12} + m_{33} + p_0 p_2 p_3\} = \min\{20, 200\}$$

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

		$j$			
$m_{ij}$		1	2	3	4
$i$	1	0	100	20	
	2	—	0	10	
	3	—	—	0	200
	4	—	—	—	0

		$j$			
$s_{ij}$		1	2	3	4
$i$	1	—	1	1	
	2	—	—	2	
	3	—	—	—	3
	4	—	—	—	—

$$m_{13} = \min\{m_{11} + m_{23} + p_0 p_1 p_3,$$

$$m_{12} + m_{33} + p_0 p_2 p_3\} = \min\{20, 200\}$$



# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

		$m_{ij}$			
		$j$			
		1	2	3	4
$i$	1	0	100	20	
	2	—	0	10	
	3	—	—	0	200
	4	—	—	—	0

		$s_{ij}$			
		$j$			
		1	2	3	4
$i$	1	—	1	1	
	2	—	—	2	
	3	—	—	—	3
	4	—	—	—	—

$$m_{24} = \min\{m_{22} + m_{34} + p_1 p_2 p_4,$$

$$m_{23} + m_{44} + p_1 p_3 p_4\} = \min\{400, 30\}$$

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

$m_{ij}$

		1	2	3	4
	$j$				
1	$i$	0	100	20	
2	$i$	—	0	10	30
3	$i$	—	—	0	200
4	$i$	—	—	—	0

$s_{ij}$

		1	2	3	4
	$j$				
1	$i$	—	1	1	
2	$i$	—	—	2	3
3	$i$	—	—	—	3
4	$i$	—	—	—	—

$$m_{24} = \min\{m_{22} + m_{34} + p_1 p_2 p_4,$$

$$m_{23} + m_{44} + p_1 p_3 p_4\} = \min\{400, 30\}$$

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

$m_{ij}$

		$j$			
		1	2	3	4
$i$	1	0	100	20	
	2	—	0	10	30
	3	—	—	0	200
	4	—	—	—	0

$s_{ij}$

		$j$			
		1	2	3	4
$i$	1	—	1	1	
	2	—	—	2	3
	3	—	—	—	3
	4	—	—	—	—

$$m_{i(i+3)} = \min_{i \leq k \leq i+2} (m_{ik} + m_{(k+1)(i+3)} + p_{i-1}p_kp_{(i+3)})$$

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

$m_{ij}$

		1	2	3	4
	$j$				
1		0	100	20	
2		—	0	10	30
3		—	—	0	200
4		—	—	—	0
$i$					

$s_{ij}$

		1	2	3	4
	$j$				
1		—	1	1	
2		—	—	2	3
3		—	—	—	3
4		—	—	—	—
$i$					

$$\begin{aligned}
 m_{14} &= \min_{1 \leq k \leq 3} (m_{1k} + m_{(k+1)4} + p_0 p_k p_4) \\
 &= \min\{230, 2300, 220\}
 \end{aligned}$$

# Matrix-Chain Multiplication

## Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

$m_{ij}$

	1	2	3	4
1	0	100	20	220
2	—	0	10	30
3	—	—	0	200
4	—	—	—	0

Optimal  
Value

$s_{ij}$

	1	2	3	4
1	—	1	1	3
2	—	—	2	3
3	—	—	—	3
4	—	—	—	—

$$\begin{aligned}
 m_{14} &= \min_{1 \leq k \leq 3} (m_{1k} + m_{(k+1)4} + p_0 p_k p_4) \\
 &= \min\{230, 2300, 220\}
 \end{aligned}$$

# Matrix-Chain Multiplication

## Constructing an Optimal Order

- We can construct an optimal order based on the records  $s_{ij}$ .

```
Print_Order(s, i, j) {  
    if(i == j) cout << "Ai";  
    else {  
        cout << "(";  
        Print_Order(s, i, sij);  
        cout << "*";  
        Print_Order(s, sij+1, j);  
        cout << ")";  
    }  
}
```

- Initial call is **Print\_Order(s, 1, n);**

# Matrix-Chain Multiplication

## Example

- Construct an optimal order
  - $n = 4, A_1 \times A_2 \times A_3 \times A_4$ .
  - $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$ .

		$j$			
$s_{ij}$		1	2	3	4
$i$	1	—	1	1	3
	2	—	—	2	3
	3	—	—	—	3
	4	—	—	—	—

$$s_{14} = 3 \rightarrow A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times A_2 \times A_3) \times A_4$$

$$s_{13} = 1 \rightarrow A_1 \times A_2 \times A_3 = A_1 \times (A_2 \times A_3)$$

$$s_{23} = 2 \rightarrow A_2 \times A_3 = A_2 \times A_3$$

$$A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times (A_2 \times A_3)) \times A_4$$

# Matrix-Chain Multiplication

## Time Complexity

- Get the minimum number of scalar multiplications:
  - We need to obtain all  $m_{ij}$  and  $s_{ij}$ , for  $1 \leq i \leq j \leq n$ .
    - $O(n^2)$  records
    - Each  $m_{ij}$  is the minimum of  $O(n)$  terms.
  - Total time complexity is  $O(n^3)$ .
- Obtain the optimal order:
  - $O(n)$



# Matrix-Chain Multiplication

## Summary

- Matrix-chain multiplication is an optimization problem.
- The solution is based on **dynamic programming**.
  - The original problem can be divided into same subproblems that **overlap**.
  - Each subproblem is solved once and stored in a table.
  - If a subproblem is encountered again, simply look up its solution in the table.
  - Reconstruct the solution to the original problem from the solutions to the subproblems.

# Outline

- Dynamic Programming
  - Motivation
  - Example: Matrix-Chain Multiplication
  - Summary

# Dynamic Programming for Optimization

- There are two key ingredients that an optimization problem must have in order for dynamic programming to apply:
  - Optimal substructure;
  - Overlapping subproblems.

# Optimal Substructure

- An optimal solution to the problem contains **within** it **optimal solutions to subproblems**.
  - In matrix-chain multiplication, the optimal order on calculating  $A_i \times \cdots \times A_j$  that splits the product between  $A_k$  and  $A_{k+1}$  contains within it optimal solutions to the problem of ordering  $A_i \times \cdots \times A_k$  and  $A_{k+1} \times \cdots \times A_j$ .
- You can show optimal substructure property by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction.

# Overlapping Subproblems

- A recursive algorithm for the problem solves the same subproblems **over and over**, rather than always generating new subproblems.
  - E.g., subproblems of matrix-chain multiplication overlap.
  - In contrast, a problem for which a divide-and-conquer approach is suitable usually generates **brand-new** problems at each step of the recursion.
- Dynamic-programming algorithms take advantage of overlapping subproblems by
  - solving each subproblem once ...
  - ... and then storing the solution in a table where it can be looked up when needed.

# Designing a Dynamic-Programming Algorithm

1. Characterize **the structure** of an optimal solution.
  - Usually, we need to define a **general** problem.
2. **Recursively** define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a **bottom-up** fashion.
4. Construct an optimal solution from computed information.