Dynamic Programming

Matrix Chain Multiplication

Instructor: Ashok Singh Sairam

Lecture Plan

- Introduction to DP
  - How it is different from divide and conquer
- Understand the 4 steps of DP using an example matrix chain multiplication

Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into **independent** subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

Dynamic Programming

- Applicable when subproblems are **not** independent
  - Subproblems share subsubproblems

E.g.: Fibonacci Series

```
                         fib(5)
                     /             \
               fib(4)                fib(3)
             /      \                /     \
         fib(3)      fib(2)         fib(2)    fib(1)
        /     \       /    \       /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just **once** and stores the answer in a **table**

Dynamic Programming

- Used for **optimization problems**
  - Such problems may have many solutions (eg. shortest path)
  - Each solution has a value, want to find a solution with the optimal value (minimum or maximum)
  - We call such a solution as **an optimal solution**

Dynamic Programming Algorithm

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information (not always necessary)

Matrix-Chain Multiplication

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

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix compatibility:

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

$$col_A = row_B \qquad\qquad col_i = row_{i+1}$$

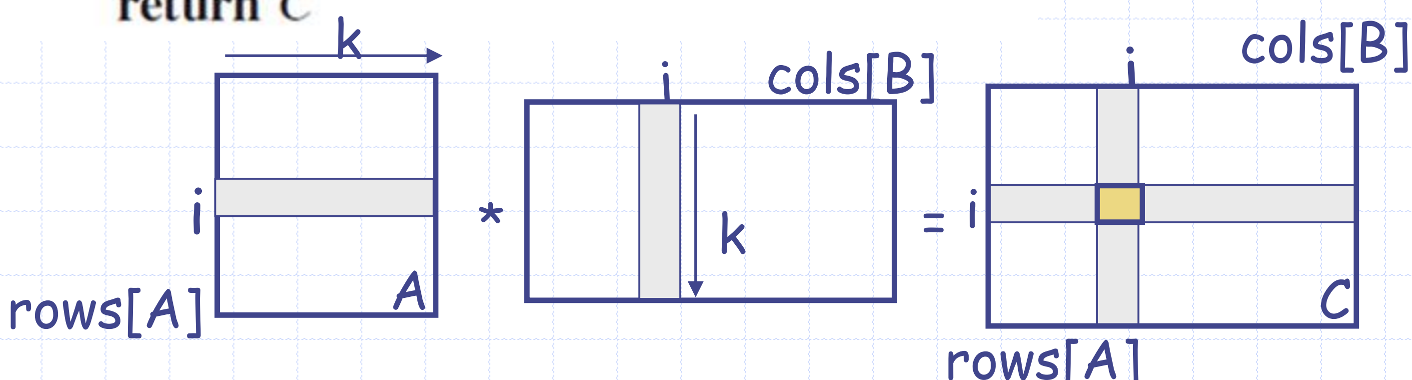
$$row_C = row_A \qquad\qquad row_C = row_{A1}$$

$$col_C = col_B \qquad\qquad col_C = col_{An}$$

MATRIX-MULTIPLY(A, B)

```
MATRIX-MULTIPLY(A, B)
    if A.columns ≠ B.rows
        error "incompatible dimensions"
    else let C be a new A.rows × B.columns matrix
        for i = 1 to A.rows
            for j = 1 to B.columns
                c_ij = 0
                for k = 1 to A.columns
                    c_ij = c_ij + a_ik · b_kj
        return C
```

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

Ex: Matrix Multiplication

$A_1 \cdot A_2 \cdot A_3$

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

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2$ = 10 x 100 x 5 = 5,000 (10 x 5)
   $((A_1 \cdot A_2) \cdot A_3)$ = 10 x 5 x 50 = 2,500
   Total: 7,500 scalar multiplications
2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3$ = 100 x 5 x 50 = 25,000 (100 x 50)
   $(A_1 \cdot (A_2 \cdot A_3))$ = 10 x 100 x 50 = 50,000
   Total: 75,000 scalar multiplications

one order of magnitude difference!!

Matrix-Chain Multiplication: Problem Statement

- Given a chain of matrices $<A_1, A_2, \ldots, A_n>$, where A_i has dimensions $\mathbf{p_{i-1} x\ p_i}$, fully parenthesize the product $A_1 . A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

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

What is the number of possible parenthesizations?

- Exhaustively checking all possible parenthesizations is not efficient!

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases}$$

- It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$

Apply dynamic programming

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information

1. The Structure of an Optimal Parenthesization

- Notation:

  $$A_{i\ldots j} = A_i \, A_{i+1} \cdots A_j, \; i \le j$$

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

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

Optimal Substructure

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

- The parenthesization of the "prefix" $A_{i\ldots k}$ must be an optimal parenthesization
- If there were a less costly way to parenthesize $A_{i\ldots k}$, we could substitute that one in the parenthesization of $A_{i\ldots j}$ 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*

Apply dynamic programming

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information

2. A Recursive Solution

- Subproblem:
  determine the minimum cost of parenthesizing $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$
- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i\ldots j}$
  - full problem $(A_{1..n})$: $m[1, n]$
  - $i = j$: $A_{i\ldots i} = A_i \Rightarrow m[i, i] = 0$, for $i = 1, 2, \ldots, n$

2. A Recursive Solution

- Consider the subproblem of parenthesizing

$$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j \qquad \text{for } 1 \le i \le j \le n$$

$$p_{i-1}p_kp_j$$

$$m[i, k] = A_{i\ldots k}\, A_{k+1\ldots j} \quad m[k+1,j] \qquad \text{for } i \le k < j$$

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

$$m[i, j] = \underbrace{m[i, k]}_{\text{min \# of multiplications to compute } A_{i\ldots k}} + \underbrace{m[k+1, j]}_{\text{min \# of multiplications to compute } A_{k+1\ldots j}} + \underbrace{p_{i-1}p_kp_j}_{\text{\# of multiplications to compute } A_{i\ldots k}A_{k\ldots j}}$$

2. A Recursive Solution – contd.

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

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

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

Apply dynamic programming

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information

3. Computing the Optimal Costs

- Requires solving the recursion, will take exponential time

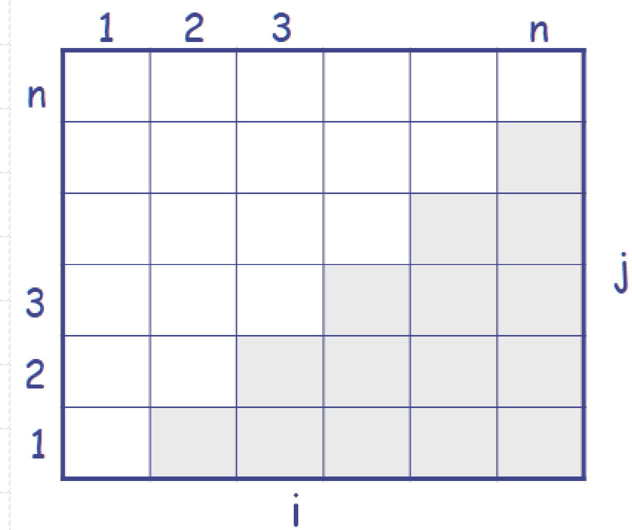
$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} & \text{if } i < j \end{cases}$$

- How many subproblems? $\Rightarrow \Theta(n^2)$
  - Parenthesize $A_{i \ldots j}$, for $1 \leq i \leq j \leq n$
  - One problem for each choice of *i* and *j*
- A recursive problem may encounter the same subproblem multiple times in the recursion tree
  - **Overlapping subproblems**, the second hallmark of dynamic programming

3. Computing the Optimal Costs

- Instead of computing the solution to the recurrence, we follow the third step of DP
  - Compute optimal cost by using a tabular **bottom-up approach**
- 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]

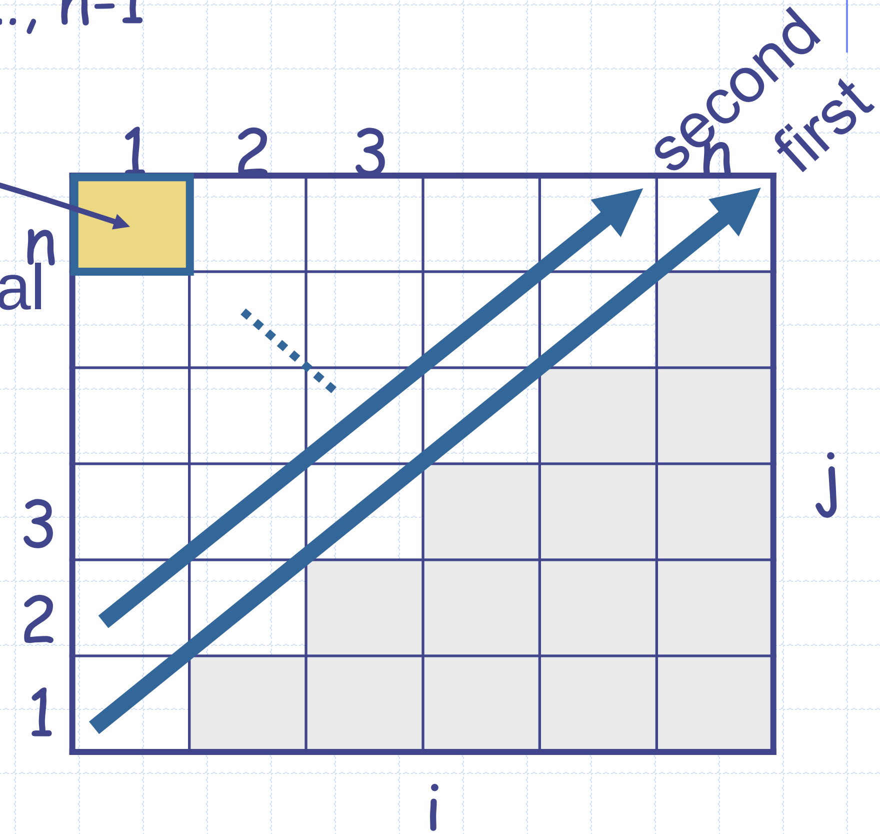
    $A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}$
  - Subproblems' size is one less than the original size
  - **<u>Idea:</u>** fill in m such that it corresponds to solving problems of increasing length



3. Computing the Optimal Costs

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

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

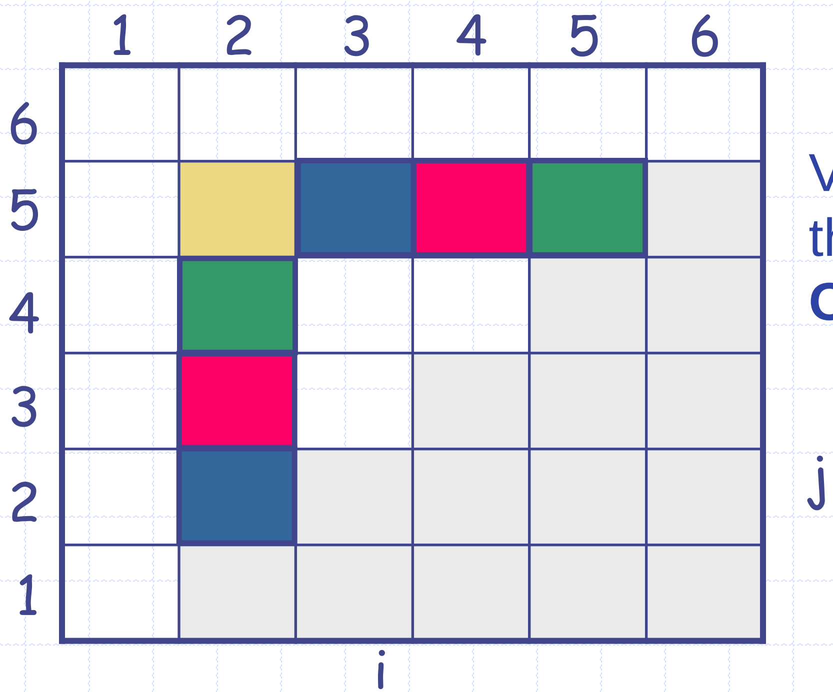


Compute rows from bottom to top and from left to right

Example: Bottom-up Approach

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

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 & k = 2 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 & k = 3 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 & k = 4 \end{cases}$$



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

Overlapping subproblems

Ex: Matrix Multiplication - contd.

- Solve $A_1 \cdot A_2 \cdot A_3$ using dynamic programming
- A_1: 10 x 100
- A_2: 100 x 5
- A_3: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2$ = 10 x 100 x 5 = 5,000 (10 x 5)
   $((A_1 \cdot A_2) \cdot A_3)$ = 10 x 5 x 50 = 2,500
   Total: 7,500 scalar multiplications
2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3$ = 100 x 5 x 50 = 25,000 (100 x 50)
   $(A_1 \cdot (A_2 \cdot A_3))$ = 10 x 100 x 50 = 50,000
   Total: 75,000 scalar multiplications

Algorithm:Matrix-Chain-Order(p)

```
MATRIX-CHAIN-ORDER(p)
n = p.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

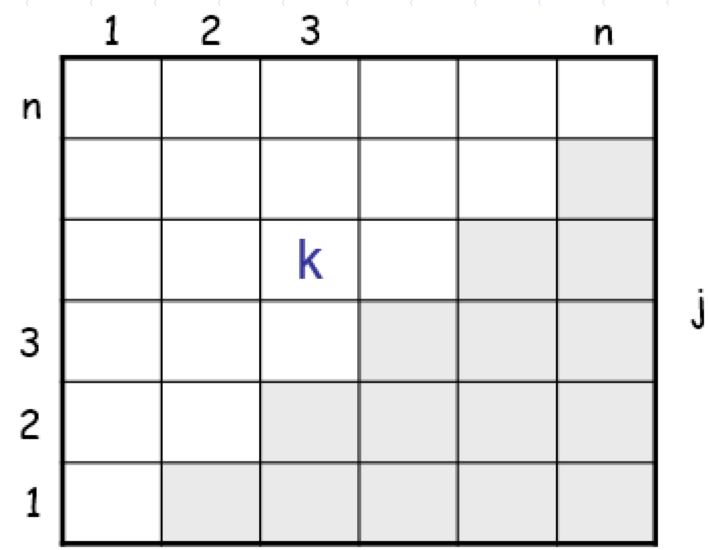
$O(N^3)$

Apply dynamic programming

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information

4. Construct the Optimal Solution

- Algorithm **Matrix-Chain-Order(p)** determines the optimal #scalar multiplications, it does NOT show how to multiple the matrices
- Not difficult to construct the optimal solution
  - Use a similar matrix s[1..n,1..n]
  - Each entry s[i,j] records the value 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}

Example: Construct Optimal Solution

- $s[i, j]$ = value of k such that the optimal parenthesization of $A_i\ A_{i+1} \cdots A_j$ 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	-					

j

i

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

Algorithm: Print-Optimal-Parens()

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Example: $A_1 \cdot \cdot \cdot A_6$

s[1..6, 1..6]

j \ i	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	-					

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"

")"

")"

Exercise

15.2-1

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.