[CS203] Design and Analysis of Algorithms

Course Instructor: Dr. Dibyendu Roy
Scribed by: Prakhar Jain (202251099)

Autumn 2023-24
Lecture (Weak 7 and 8)

- **Matrix-Chain Multiplication:**
  - Given a sequence of n matrices, aim to compute $A_1 A_2 \cdots A_n$.
  - Minimize scalar multiplications using the standard algorithm for rectangular matrices.
- **Fully Parenthesized Matrices:**
  - A product is fully parenthesized if it is a single matrix or the product of two fully parenthesized matrix products.
  - Different parenthesizations impact the cost of evaluating the product.
- **Matrix-Chain Multiplication Problem:**
  - Optimal parenthesization reduces the number of scalar multiplications.
  - Example: Chain of matrices A_1, A_2, A_3, A_4 has five distinct ways to be fully parenthesized:

$$\begin{aligned}&(A_1(A_2(A_3A_4))) \quad (1);\\ &(A_1((A_2A_3)A_4)) \quad (2);\\ &((A_1A_2)(A_3A_4)) \quad (3);\\ &(A_1(A_2A_3)A_4) \quad (4);\\ &(((A_1A_2)A_3)A_4) \quad (5).\end{aligned}$$

- **Efficiency Impact:**
  - Careful parenthesization can significantly reduce the cost of matrix multiplication.
  - Example: Chain of matrices with dimensions 10×100, 100×5, and 5×50.
  - The most efficient solution is achieved by parenthesizing as follows:

$$((A_1A_2)A_3)$$

Optimal Parenthesization:

- Optimal solution for multiplying n matrices is found by considering all possible splits and minimizing the cost.
- Optimal solution for n matrices: $A = A_1 \dots A_n$
- Optimal parenthesization:

$$A = ((A_1 \dots A_k)(A_{k+1} \dots A_n))$$

Algorithm 1: MATRIX-CHAIN-ORDER(p, n)

```
1: Let m[1...n, 1...n] and s[1...n-1, 2...n] be new tables
2: for i ← 1 to n do
       {Chain length 1}m[i, i] ← 0
3: 4: end for
5: for l ← 2 to n do
       {l is the chain length}for i ← 1 to n − l + 1 do  {Chain begins at A_i}
6:     j ← i + l − 1 {Chain ends at A_j}
8:     m[i, j] ← ∞
9:     for k ← i to j − 1 do
           {Try A_i W_k A_{k+1} W_j}q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j if q < m[i, j] then
10:        m[i, j] ← q {Update minimum cost}
13:        s[i, j − 1] ← k {Update split index}
14:    end if
15:    end for
16:  end for
17: end for
18: return m and s =0
```

- The cost is given by: $C = C_{1k} + C_{(k+1)n} + pqr$, where C_{1k} is a $p \times q$ matrix, and $C_{(k+1)n}$ is a $q \times r$ matrix, and this must be optimal.

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

Insights:

- The algorithm requires space proportional to $n^2 + n$ (two arrays are maintained).
- It performs n^3 computations in terms of time complexity.

Matrix Chain Multiplication Example

Consider the following matrices:

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

We want to find the optimal parenthesization for multiplying these matrices.

Optimal Parenthesization Table

Let's calculate the minimum number of scalar multiplications using the formula:

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

The optimal parenthesization for the given example is as follows:

$m[2,5] = \min\{m[2,2] + m[3,5] + p_1p_2p_5, m[2,3] + m[4,5] + p_1p_3p_5, m[2,4] + m[5,5] + p_1p_4p_5\}$
$= \min\{0 + m[3,5] + 30 \cdot 35 \cdot 20, 0 + m[4,5] + 30 \cdot 15 \cdot 20, 0 + 0 + 30 \cdot 10 \cdot 20\}$
$= \min\{0 + 7125, 0 + 7125, 0\}$
$= 7125.$

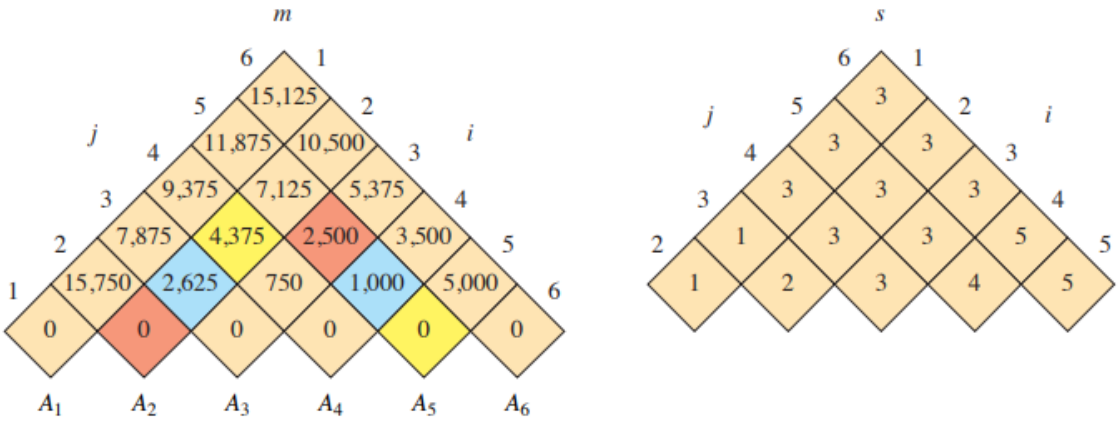

Figure 14.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the follow-

Longest Common Subsequence (LCS):

In the context of strings, the Longest Common Subsequence is the longest sequence of characters that appear in the same order within two given strings. The characters in the subsequence are not required to be consecutive, but the order of appearance must be maintained.

For example, given two strings "ABCD" and "ACDF," the common subsequences are "ACD" and "AC," but the longest common subsequence is "AC" as it has the maximum length.

The LCS problem is a fundamental problem in computer science with applications in various fields, including bioinformatics, text comparison, and version control systems.

Step 1: Characterizing a Longest Common Subsequence (LCS)

To solve the LCS problem using a brute-force approach, one could enumerate all subsequences of sequence X and check each subsequence to determine if it is also a subsequence of sequence Y, while keeping track of the longest subsequence found. However, this approach is impractical for long sequences due to its exponential time complexity.

The LCS problem exhibits an optimal-substructure property, as illustrated by the following theorem.

Theorem 14.1 (Optimal Substructure of an LCS): Let $X = \{x_1, x_2, \ldots, x_m\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ be sequences, and let $Z = \{\zeta_1, \zeta_2, \ldots, \zeta_k\}$ be any LCS of X and Y.

1. If $x_m = y_n$, then $\zeta_k = x_m = y_n$, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$ and $\zeta_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $\zeta_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Step 2: A Recursive Solution

As in the matrix-chain multiplication problem, solving the LCS problem recursively involves establishing a recurrence for the value of an optimal solution. Let $c(i, j)$ be the length of an LCS of the sequences X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c(i, j-1), c(i-1, j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

LCS-LENGTH(X, Y, m, n)

1. Let $b[1..m][1..n]$ and $c[0..m][0..n]$ be new tables.
2. For $i = 1$ to m, $c[i][0] = 0$.
3. For $j = 0$ to n, $c[0][j] = 0$.
4. For $i = 1$ to m:
   (a) For $j = 1$ to n (compute table entries in row-major order):
      i. If $x_i = y_j$:
         A. $c[i][j] = c[i-1][j-1] + 1$.
         B. $b[i][j] = \leftarrow$.
      ii. Else if $c[i-1][j] \geq c[i][j-1]$:
         A. $c[i][j] = c[i-1][j]$.
         B. $b[i][j] = \uparrow$.
      iii. Else:
         A. $c[i][j] = c[i][j-1]$.
         B. $b[i][j] = \leftarrow$.
5. Return c and b.

Concluding Remarks

The LCS-LENGTH algorithm efficiently computes the length of the Longest Common Subsequence (LCS) for two sequences X and Y. The tables c and b are crucial for reconstructing the LCS after the lengths have been calculated. This dynamic programming approach provides an optimal solution to the LCS problem, making it practical for real-world applications in various domains.