DS2030 Data Structures and Algorithms for Data Science
Week-9 Practice Set

October 22nd, 2025

1 Matrix Chain Multiplication

Problem Statement

The Matrix Chain Multiplication problem involves finding the most efficient way to multiply a chain of matrices by determining the optimal parenthesization that minimizes the number of scalar multiplications. This is a classic dynamic programming problem where subproblems overlap, and the optimal solution can be constructed bottom-up.
Given a sequence of matrices $A_0, A_1, \ldots, A_{n-1}$ with dimensions $d_0 \times d_1, d_1 \times d_2, \ldots, d_{n-1} \times d_n$, compute the minimum number of scalar multiplications required.

Tasks

Implement the Matrix Chain Multiplication algorithm with the following functionalities:

1. **Dynamic Programming Table Computation** Function: `matrix_chain_order(dims)` Where `dims` is a list of integers representing the dimensions $[d_0, d_1, \ldots, d_n]$. Compute and return a 2D table $N[i][j]$ where $N[i][j]$ is the minimum cost to multiply matrices from i to j. Print the DP table showing the costs for increasing chain lengths.
2. **Optimal Parenthesization** Function: `print_optimal_parenthesization(table, dims)` Using the DP table, reconstruct and print the optimal parenthesization for the entire chain (e.g., ((A0 * A1) * A2)).
3. **Efficiency Observation** Print the minimum cost for the entire chain and the time complexity of the algorithm.

Input

```
dims = [3, 100, 5, 5]
# Matrices: A0: 3x100, A1: 100x5, A2: 5x5
```

Expected Output

```
DP Table for chain lengths:

Length 1:
N[0][0] = 0
N[1][1] = 0
N[2][2] = 0

Length 2:
N[0][1] = 3*100*5 = 1500
N[1][2] = 100*5*5 = 2500

Length 3:
N[0][2] = min(1500 + 3*5*5, 2500 + 3*100*5) = min(1575, 4000) = 1575

Minimum cost: 1575
Optimal parenthesization: ((A0 * A1) * A2)

Time complexity: O(n^3)
```

2 Longest Common Subsequence (LCS)

Problem Statement

The Longest Common Subsequence (LCS) problem finds the length of the longest subsequence present in both given sequences. Unlike substrings, subsequences do not need to be contiguous. This problem is solved using dynamic programming by building a table that captures the lengths of LCS for prefixes of the two strings.
Given two strings X and Y, compute the length of their LCS.

Tasks

Implement the LCS algorithm with the following functionalities:

1. **Dynamic Programming Table Computation** Function: `lcs_length(X, Y)` Compute and return a 2D table $L[i][j]$ where $L[i][j]$ is the length of LCS of $X[0..i-1]$ and $Y[0..j-1]$. Print the DP table, showing how values are filled based on matches and mismatches.

2. **Reconstruct LCS** Function: `reconstruct_lcs(table, X, Y)` Using the DP table, reconstruct and print one possible LCS string along with the indices in X and Y that form this common subsequence.

3. **Efficiency Observation** Print the length of the LCS and the time/space complexity.

Input

```
X = "ABCBDAB"
Y = "BDCAB"
```

Expected Output

```
LCS DP Table:
     '' B D C A B
''   0  0  0  0  0  0
A    0  0  0  0  1  1
B    0  1  1  1  1  2
C    0  1  1  2  2  2
B    0  1  1  2  2  3
D    0  1  2  2  2  3
A    0  1  2  2  3  3
B    0  1  2  2  3  4

LCS length: 4
One possible LCS: "BCAB"
Indices in X: [1, 2, 5, 6]
Indices in Y: [0, 2, 3, 4]

Time complexity: O(nm), Space: O(nm)
```