set9.py

```python
# # Function to compute the LCS (Longest Common Subsequence) length table
def lcs_length(X, Y):
    n, m = len(X), len(Y)

    # Initialize a (n+1) x (m+1) DP table with zeros
    L = []
    for i in range(n + 1):
        L.append([0] * (m + 1))

    # Fill the DP table using the standard LCS recurrence relation
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            # If characters match, extend the LCS by 1
            if X[i - 1] == Y[j - 1]:
                L[i][j] = L[i - 1][j - 1] + 1
            # Otherwise, take the maximum from top or left cell
            else:
                L[i][j] = max(L[i - 1][j], L[i][j - 1])
    return L


# Function to reconstruct one possible LCS from the computed DP table
def reconstruct_lcs(X, Y, L):
    i, j = len(X), len(Y)
    lcs = []        # To store LCS characters
    indices_X = []  # To store corresponding indices in X
    indices_Y = []  # To store corresponding indices in Y

    # Start from the bottom-right of the DP table and trace back
    while i > 0 and j > 0:
        # If current characters match, they are part of the LCS
        if X[i - 1] == Y[j - 1]:
            lcs.append(X[i - 1])
            indices_X.append(i - 1)
            indices_Y.append(j - 1)
            i -= 1
            j -= 1
        # Move in the direction of the larger value (up or left)
        elif L[i - 1][j] >= L[i][j - 1]:
            i -= 1
        else:
            j -= 1

    # Since we built the LCS in reverse order, reverse it at the end
    lcs.reverse()
    indices_X.reverse()
    indices_Y.reverse()

```

```
    # Return the LCS string and the corresponding indices
    return ''.join(lcs), indices_X, indices_Y


# Function to neatly print the DP table with row/column labels
def print_LCS_table(X, Y, L):
    n, m = len(X), len(Y)

    print("LCS DP Table:")
    # Print the header row (Y string)
    print("   ", end="")
    print(" ".join(['""'] + list(Y)))

    # Print each row with corresponding X character
    for i in range(n + 1):
        if i == 0:
            print('""', end=" ")
        else:
            print(X[i - 1], end=" ")
        print(*L[i])  # Print the DP row

X = "ABCBDAB"
Y = "BDCAB"
L = lcs_length(X, Y)
lcs_str, indices_X, indices_Y = reconstruct_lcs(X, Y, L)
print_LCS_table(X, Y, L)
print("\nLCS length:", L[len(X)][len(Y)])
print("One possible LCS:", f"\"{lcs_str}\"")
print("Indices in X:", indices_X)
print("Indices in Y:", indices_Y)
print("Time complexity: O(nm), Space: O(nm)")




# Function to compute the optimal order of matrix multiplication
# using Dynamic Programming (Matrix Chain Multiplication problem)
def matrix_chain_order(dims):
    """
    Parameters:
        dims: list of matrix dimensions where the i-th matrix Ai has dimensions
              dims[i] x dims[i+1]
              (e.g., for matrices A1:A2:A3 with sizes 10x20, 20x30, 30x40 -> dims = [10, 20,
30, 40])

    Returns:
        N: 2D table storing minimum multiplication costs for each subchain
        P: 2D table storing the index (k) where the optimal split occurs
    """
    n = len(dims) - 1  # Number of matrices (A0, A1, ..., A_{n-1})
```

```python

    # Initialize DP tables:
    # N[i][j] → minimum cost (scalar multiplications) for matrices Ai...Aj
    # P[i][j] → split point that achieved this minimum cost
    N = [[0] * n for _ in range(n)]
    P = [[0] * n for _ in range(n)]

    print("DP Table for chain lengths:")

    # Base case: A single matrix has zero cost (no multiplication needed)
    print(" Length 1:")
    for i in range(n):
        N[i][i] = 0
        print(f" N[{i}][{i}] = 0")

    # chain_len represents the number of matrices in the current subchain
    for chain_len in range(2, n + 1):
        print(f" Length {chain_len}:")
        for i in range(n - chain_len + 1):
            j = i + chain_len - 1
            N[i][j] = float("inf")  # Initialize with infinity (will take min over all k)

            parts = []  # To store all possible cost expressions for clarity

            # Try every possible split point k between i and j
            for k in range(i, j):
                # Cost of multiplying the subchains (Ai..Ak) and (A{k+1}..Aj),
                # plus cost of multiplying the resulting two matrices
                cost = N[i][k] + N[k + 1][j] + dims[i] * dims[k + 1] * dims[j + 1]

                # Store formatted expression
                if chain_len == 2:
                    # For chain length 2, show only the multiplication cost
                    parts.append(f"{dims[i]}*{dims[k+1]}*{dims[j+1]} = {cost}")
                else:
                    # For longer chains, include subproblem cost additions
                    parts.append(f"{N[i][k]} + {dims[i]}*{dims[k+1]}*{dims[j+1]} = {cost}")

                # If this cost is smaller than the current best, update it
                if cost < N[i][j]:
                    N[i][j] = cost
                    P[i][j] = k  # store split point

            if chain_len == 2:
                print(f" N[{i}][{j}] = {parts[0]}")
            else:
                parts_str = ", ".join(parts)
                print(f" N[{i}][{j}] = min({parts_str}) = {N[i][j]}")

    return N, P
```

```python


def print_optimal_parenthesization(P, i, j):
    """
    Recursively prints the optimal parenthesization order
    based on the split table P.
    Example: ((A0 * A1) * (A2 * A3))
    """
    if i == j:
        # Single matrix (base case)
        print(f"A{i}", end="")
    else:
        # Recursively print parenthesis around subchains
        print("(", end="")
        print_optimal_parenthesization(P, i, P[i][j])
        print(" * ", end="")
        print_optimal_parenthesization(P, P[i][j] + 1, j)
        print(")", end="")


dims = [3, 100, 5, 5]  # Matrix dimensions: A0 = 3x100, A1 = 100x5, A2 = 5x5

N, P = matrix_chain_order(dims)

print(" Minimum cost:", N[0][len(dims) - 2])
print(" Optimal parenthesization: ", end="")
print_optimal_parenthesization(P, 0, len(dims) - 2)
print()
print(" Time complexity: O(n^3), Space complexity: O(n^2)")

```