Assignment -2 Code in Python

```
def matrix_chain_multiplication(dimensions):
    n = len(dimensions)
    # Create a matrix to store the minimum number of scalar multiplications
    # and a matrix to store the optimal parenthesization
    m = [[0] * n for _ in range(n)]
    parenthesization = [[0] * n for _ in range(n)]

    # Initialize the matrices for chains of length 2
    for i in range(1, n):
        m[i][i] = 0

    # Chain length
    for chain_len in range(2, n):
        for i in range(1, n - chain_len + 1):
            j = i + chain_len - 1
            m[i][j] = float('inf')
            for k in range(i, j):
                cost = m[i][k] + m[k + 1][j] + dimensions[i - 1][0] * dimensions[k][1] * dimensions[j][1]
                if cost < m[i][j]:
                    m[i][j] = cost
                    parenthesization[i][j] = k

    # Get the optimal parenthesization
    def get_optimal_parenthesization(i, j):
        if i == j:
            return f'M{str(i)}'
        else:
            k = parenthesization[i][j]
            left_chain = get_optimal_parenthesization(i, k)
            right_chain = get_optimal_parenthesization(k + 1, j)
            return f'({left_chain} x {right_chain})'

    optimal_parenthesization = get_optimal_parenthesization(1, n - 1)

    return optimal_parenthesization, m[1][n - 1]

# Example usage
dimensions = [(2, 3), (3, 4), (4, 2)]
optimal_parenthesization, min_scalar_multiplications = matrix_chain_multiplication(dimensions)
print("Optimal Parenthesization:", optimal_parenthesization)
print("Minimum Scalar Multiplications:", min_scalar_multiplications)
```

1.	Implement a dynamic programming algorithm to solve the matrix chain multiplication problem in Python.

Algorithm: Matrix Chain Multiplication

Input: List of matrix dimensions [(A1, A2), (A2, A3), ..., (An-1, An)]

Output: Optimal parenthesization and minimum scalar multiplications

```
1. Initialize n as the length of the dimensions list.
2. Create a 2D array m of size [n][n] to store minimum scalar multiplications.
3. Create a 2D array parenthesization of size [n][n] to store optimal parenthesization.
4. Set diagonal elements of m to 0 (m[i][i] = 0 for all i).
5. Loop over chain_lengths (l) from 2 to n:
  a. For each chain_length, loop over i from 1 to n - l + 1:
    i. Set j = i + l - 1.
    ii. Set m[i][j] to infinity.
    iii. Loop over k from i to j - 1:
        1. Calculate cost = m[i][k] + m[k + 1][j] + dimensions[i-1] * dimensions[k] * dimensions[j].
        2. If cost is less than m[i][j]:
            a. Update m[i][j] to cost.
            b. Set parenthesization[i][j] to k.
6. The minimum number of scalar multiplications is stored in m[1][n-1].
7. Define a function get_optimal_parenthesization(i, j):
  a. If i equals j, return "A" followed by the index i.
  b. Else, set k = parenthesization[i][j].
  c. Recursively call get_optimal_parenthesization(i, k) and get_optimal_parenthesization(k+1, j).
  d. Return the concatenation of these results enclosed in parentheses.
8. Call get_optimal_parenthesization(1, n-1) to obtain the optimal parenthesization.
9. Return the optimal parenthesization and the minimum scalar multiplications.
```

2. Apply your algorithm to the provided list of matrices to find the optimal parenthesization

We need to apply the inputs to the above algorithm

3. Calculate and provide the minimum number of scalar multiplications required for the optimal parenthesization

Optimal Parenthesization: (M1 x M2)

Minimum Scalar Multiplications: 16

4. Explain the dynamic programming approach, including the initialization, recurrence relation, and reconstruction of the optimal parenthesization.

Certainly, let's provide a more concise explanation of the dynamic programming approach for the matrix chain multiplication problem:

Dynamic Programming Approach:

Initialization:

- Initialize two matrices m and parenthesization of size n x n, where n is the number of matrices in the chain.

- Set m[i][i] to 0 for all i (base case for single matrices).

Recurrence Relation:

- Iterate over chain lengths chain_len from 2 to n, representing the number of matrices in the chain.

- For each chain length, iterate over subchains (i, j) within the chain.

- For a given subchain (i, j), find the optimal split point k that minimizes the cost of multiplication.

- Calculate m[i][j] using the formula:

  m[i][j] = min(m[i][k] + m[k+1][j] + dimensions[i-1][0] * dimensions[k][1] * dimensions[j][1])

- Update m[i][j] with the minimum cost found among all possible split points k.

Reconstruction of Optimal Parenthesization:

- Maintain the parenthesization matrix during dynamic programming to store the optimal split points.

- Use a recursive function to reconstruct the optimal parenthesization:

  - Starting from the full chain (1, n-1), find the split point k stored in parenthesization[i][j].

  - Recursively call the function on the left subchain (i, k) and the right subchain (k+1, j) until reaching the base case where i equals j.

  - Concatenate the results with parentheses to obtain the optimal parenthesized expression.

This dynamic programming approach efficiently calculates the minimum number of scalar multiplications required for matrix chain multiplication while also providing the optimal way to parenthesize the matrices for the minimum cost. It avoids redundant calculations through the use of memoization.

5. Analyze the time and space complexity of your algorithm and discuss its efficiency for large instances of the problem.

Time Complexity Analysis:

The initialization step takes O(n) time.

The recurrence relation, which fills in the m and parenthesization tables, has a time complexity of O(n^3) due to nested loops.

The reconstruction of the optimal parenthesization has a time complexity of O(n).

Overall, the time complexity of the algorithm is O(n^3).

Space Complexity Analysis:

The space complexity is primarily determined by the space required for the m and parenthesization matrices, both of which have a size of n x n, resulting in a space complexity of O(n^2).

Additional variables and the recursion call stack contribute to a space complexity of O(n).

Efficiency for Large Instances:

The algorithm is efficient for reasonably sized instances with a moderate number of matrices (where n is not very large).

However, for very large instances with a high number of matrices, the cubic time complexity may become a limiting factor, and the algorithm's efficiency can degrade.

Handling extremely large instances may require advanced optimization techniques and parallelization to manage the computational load effectively.