DAA Assignment 6

Ishaan Shaikh
231070063
Batch D

Q. **Consider meteorological data like temperature, dew point, wind direction, wind speed, cloud cover, cloud layer(s) for each city. This data is available in a two dimensional array for a week. Assuming all tables are compatible for multiplication. You have to implement the matrix chain multiplication algorithm to find the fastest way to complete the matrix multiplication to achieve timely prediction.**

Algorithm:

```
matrixChain(p):
//Input: array of matrix dimensions
//Output: Minimum number of scalar multiplications and
        optimal order of multiplication.
    n <- length(p)-1

    m[n][n] = {0}  //nxn matrix with each element 0
    s[n][n] = {0}
    for L = 2 to n+1:
        ~~for i=0 to n-L+1~~
        for i = 0 to n-L+1:
            j = i+L-1
            m[i][j] = INT_MAX
            for k = i to j
                q = m[i][k] + m[k+1][j] + p[i]*p[k+1]*p[j+1]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k
    return m[0][n-1], s

getOrder(s, i, j):
//Input: Matrix of indexes and i,j index of matrix
//Output: Substring of optimal order
    if i == j
        return "A{i+1}"
    else
        return "({getOrder(s, i, s[i][j])}x{getOrder(s, s[i][j]+1,
                j)})"
```

Test cases:

Test Cases:

1) Input: [10, 20, 30, 40, 30, 20, 10]
   Output: 38000, (A1 × (A2 × (A3 × (A4 × (A5 × A6)))))

2) Input: [30, 35, 15, 5, 10, 20, 25]
   Output: 15125, ((A1 × (A2 × A3)) × ((A4 × A5) × A6))

3) Input: [5, 10, 3, 12, 5, 50, 6]
   Output: 2010, ((A1 × A2) × ((A3 × A4) × (A5 × A6)))

4) Input: [10, "20", "30", 40, "30", 20, "10"]
   Output: All matrix dimensions must be integers.

5) Input: [10, -20, 30, 0, -40, 50, -60]
   Output: All matrix dimensions must be positive.

6) Input: []
   Output: Input list must contain at least three dimensions to form two matrices

Time Complexity:

Time Complexity:

//Input: array of matrix dimension

//Input size : n

Basic operation: Create a DP table and loop through it to find minimum cost

Let Cworst(n) be the worst case time complexity of the algorithm

$$C_{worst}(n) = \sum_{L=2}^{n+1} \sum_{i=0}^{n-L+1} \sum_{k=i}^{j} 1$$

$$= \sum_{L=2}^{n+1} \sum_{i=0}^{n-L+1} j - i$$

$$= \sum_{L=2}^{n+1} \sum_{i=0}^{n-L+1} j - \sum_{L=2}^{n+1} \sum_{i=0}^{n-L+1} i$$

$$= j \sum_{L=2}^{n+1} n - L \cancel{+1} \cancel{-1} - \sum_{L=2}^{n+1} (0 + 1 + \ldots n - L \cancel{+1} \cancel{-1})$$

$$= j \sum_{L=2}^{n+1} n - L - \sum_{L=2}^{n+1} \frac{(n-L)(n-L+1)}{2}$$

$$C_{worst}(n) \approx O(n^3)$$

$$\therefore \boxed{\text{Time complexity} = O(n^3)}$$

Program:

```
import sys

class MatrixChainMultiplication:
    """Computes minimum scalar multiplications and optimal order for
matrix chain multiplication.

    Attributes:
        dimensions (list): The list of dimensions where dimensions[i-1]
and dimensions[i] are
            dimensions for the i-th matrix in the chain.
        n (int): Number of matrices in the chain.
        m (list): DP table where m[i][j] stores the minimum cost for
```

```
multiplying matrices from i to j.
        s (list): DP table where s[i][j] stores the split point for the
optimal cost in m[i][j].
    """

    def __init__(self, dimensions):
        """Initializes MatrixChainMultiplication with dimensions and
validates input.

        Args:
            dimensions (list): List of matrix dimensions.

        Raises:
            ValueError: If dimensions contain invalid values (non-integer
or non-positive).
        """
        self.dimensions = dimensions
        self.n = len(dimensions) - 1
        self._validate_dimensions()
        self.m = [[0] * self.n for _ in range(self.n)]
        self.s = [[0] * self.n for _ in range(self.n)]

    def _validate_dimensions(self):
        """Validates dimensions to ensure all elements are positive
integers.

        Raises:
            ValueError: If any dimension is non-integer or non-positive.
        """
        if self.n < 1:
            raise ValueError("Input list must contain at least three
dimensions to form two matrices.")
        if any(type(dim) != int for dim in self.dimensions):
            raise ValueError("All matrix dimensions must be integers.")
        if any(dim <= 0 for dim in self.dimensions):
            raise ValueError("All matrix dimensions must be positive.")

    def compute_min_cost(self):
        """Calculates the minimum scalar multiplications required to
multiply the chain of matrices.
```

```
        Returns:
            int: The minimum number of scalar multiplications needed.
        """
        for chain_len in range(2, self.n + 1):
            for i in range(self.n - chain_len + 1):
                j = i + chain_len - 1
                self.m[i][j] = sys.maxsize
                for k in range(i, j):
                    cost = (self.m[i][k] + self.m[k + 1][j] +
                            self.dimensions[i] * self.dimensions[k + 1] *
self.dimensions[j + 1])
                    if cost < self.m[i][j]:
                        self.m[i][j] = cost
                        self.s[i][j] = k
        return self.m[0][self.n - 1]

    def get_optimal_order(self):
        """Generates the optimal order of matrix multiplication.

        Returns:
            str: A string representation of the optimal multiplication
order.
        """
        return self._build_order(0, self.n - 1)

    def _build_order(self, i, j):
        """Recursive helper to construct the optimal multiplication
order.


        Args:
            i (int): Start index for matrix multiplication.
            j (int): End index for matrix multiplication.

        Returns:
            str: Substring of the optimal multiplication order for
matrices from i to j.
        """
        if i == j:
            return f"A{i + 1}"
        else:
            return f"({self._build_order(i, self.s[i][j])} x
```

```
{self._build_order(self.s[i][j] + 1, j)})"


class MatrixChainTest:
    """Manages and runs test cases for MatrixChainMultiplication."""

    def __init__(self):
        """Initializes the test cases for matrix chain multiplication."""
        self.test_cases = [
            {"p": [10, 20, 30, 40, 30, 20, 10]},      # Valid case with 6
matrices
            {"p": [30, 35, 15, 5, 10, 20, 25]},       # Valid case with 6
matrices
            {"p": [5, 10, 3, 12, 5, 50, 6]},          # Valid case with 6
matrices
            {"p": [10, "20", "30", 40, "30", 20, "10"]},  # Non-integer
dimension
            {"p": [10, -20, 30, 0, -40, 50, -60]},    # Zero or negative
dimension
            {"p": []}                                 # Empty input list
        ]

    def run_tests(self):
        """Runs each test case, printing results or errors for each."""
        for i, test in enumerate(self.test_cases):
            dimensions = test["p"]
            print(f"Test Case {i + 1}: p = {dimensions}")
            try:
                matrix_chain = MatrixChainMultiplication(dimensions)
                min_cost = matrix_chain.compute_min_cost()
                order = matrix_chain.get_optimal_order()
                print(f"Minimum cost: {min_cost}")
                print(f"Optimal order: {order}")
            except ValueError as e:
                print(f"Error: {e}")
            except Exception as e:
                print(f"Unexpected error: {e}")
            print()


# Run test cases
```

```python
if __name__ == "__main__":
    MatrixChainTest().run_tests()
```

Output:

```
Test Case 1: p = [10, 20, 30, 40, 30, 20, 10]
Minimum cost: 38000
Optimal order: (A1 x (A2 x (A3 x (A4 x (A5 x A6)))))

Test Case 2: p = [30, 35, 15, 5, 10, 20, 25]
Minimum cost: 15125
Optimal order: ((A1 x (A2 x A3)) x ((A4 x A5) x A6))

Test Case 3: p = [5, 10, 3, 12, 5, 50, 6]
Minimum cost: 2010
Optimal order: ((A1 x A2) x ((A3 x A4) x (A5 x A6)))

Test Case 4: p = [10, '20', '30', 40, '30', 20, '10']
Error: All matrix dimensions must be integers.

Test Case 5: p = [10, -20, 30, 0, -40, 50, -60]
Error: All matrix dimensions must be positive.

Test Case 6: p = []
Error: Input list must contain at least three dimensions to form two matrices.
```

Conclusion: In conclusion, we have studied the 5 SOLID principles of software development and implemented it in the program of longest common subsequence. We have implemented the program of matrix chain multiplication using dynamic programming to optimize the time complexity of the algorithm.