Name: Arpita Dinesh Singh
2nd-year B.tech
Computer engineering

Roll No:231071005

DESIGN ANALYSIS OF ALGORITHM

LABORATORY 06

EXPERIMENT TASK- 1

(Longest common subsequence of 20 students)

4.CODE

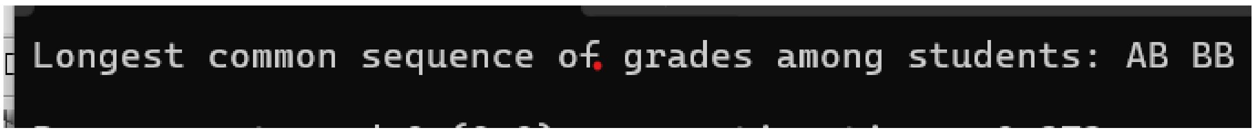
```
#include <bits/stdc++.h>
using namespace std;

// Function to find the LCS of two sequences
vector<string> lcs_two_sequences(const vector<string>& seq1, const
vector<string>& seq2) {
    int n = seq1.size();
    int m = seq2.size();

    // Create a 2D vector for dynamic programming
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    // Fill the dp array
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (seq1[i - 1] == seq2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
```

```
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
}

// Reconstruct the LCS from the dp array
vector<string> lcs;
int i = n, j = m;
while (i > 0 && j > 0) {
    if (seq1[i - 1] == seq2[j - 1]) {
        lcs.push_back(seq1[i - 1]);
        --i;
        --j;
    } else if (dp[i - 1][j] > dp[i][j - 1]) {
        --i;
    } else {
        --j;
    }
}

reverse(lcs.begin(), lcs.end());
return lcs;
}

// Function to find the longest common sequence across multiple sequences
vector<string> longest_common_sequence(const vector<vector<string>>& sequences) {
    if (sequences.empty()) {
        return {};
    }

    // Start with the LCS of the first sequence
    vector<string> lcs_result = sequences[0];

    // Compute the LCS across all sequences
    for (size_t i = 1; i < sequences.size(); ++i) {
        lcs_result = lcs_two_sequences(lcs_result, sequences[i]);
```

```
        if (lcs_result.empty()) {
            break;  // Early stop if there is no common subsequence
        }
    }

    return lcs_result;
}

int main() {
    // Sample data: grades of 20 students, each with 5 grades
    vector<vector<string>> student_grades = {
        {"AA", "AB", "BB", "CC", "FF"},
        {"AB", "BB", "CC", "FF", "GG"},
        {"AA", "AB", "BB", "GG", "CC"},
        {"AB", "BB", "CC", "FF", "GG"},
        {"AB", "BB", "CC", "FF", "AA"},
        {"AB", "BB", "GG", "FF", "CC"},
        {"AA", "AB", "BB", "FF", "GG"},
        {"AB", "BB", "CC", "FF", "CC"},
        {"AB", "BB", "FF", "GG", "AA"},
        {"AA", "AB", "BB", "GG", "FF"},
        {"AB", "BB", "CC", "FF", "GG"},
        {"AA", "AB", "BB", "CC", "GG"},
        {"AB", "BB", "GG", "FF", "CC"},
        {"AA", "AB", "BB", "FF", "GG"},
        {"AB", "BB", "CC", "FF", "GG"},
        {"AA", "AB", "BB", "GG", "CC"},
        {"AB", "BB", "CC", "FF", "GG"},
        {"AA", "AB", "BB", "FF", "GG"},
        {"AB", "BB", "CC", "FF", "GG"}
    };

    // Compute the longest common sequence
    vector<string> lcs_grades = longest_common_sequence(student_grades);

    // Output the result
    cout << "Longest common sequence of grades among students: ";
```

```
    for (const string& grade : lcs_grades) {
        cout << grade << " ";
    }
    cout << endl;

    return 0;
}
```

5.OUTPUT



6.CONCLUSION

In conclusion, the **Brute Force** approach to finding the Longest Common Subsequence (LCS) is highly inefficient for larger datasets, as it results in exponential time complexity. On the other hand, the **Dynamic Programming (DP)** approach offers a much more efficient solution by using a 2D table to store intermediate results, reducing the time complexity to $O(k \times n \times m)$ for multiple sequences.

<u>EXPERIMENT TASK- 2</u>
<u>(Matrix Chain Multiplication)</u>

4.CODE

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

// Function to perform Matrix Chain Multiplication
int matrixChainOrder(const vector<int>& p, int n, vector<vector<int>>& m,
vector<vector<int>>& s) {
    // m[i][j] will store the minimum number of scalar multiplications required to
multiply matrices from i to j
```

```
    // s[i][j] will store the index k at which the optimal split occurs
    for (int len = 2; len <= n; ++len) { // len is the chain length
        for (int i = 0; i <= n - len; ++i) { // i is the starting point of the chain
            int j = i + len - 1; // j is the endpoint of the chain
            m[i][j] = INT_MAX;
            for (int k = i; k < j; ++k) {
                // Calculate the cost for splitting at k
                int 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];
}

// Function to print the optimal parenthesization of the matrix chain multiplication
void printOptimalParenthesis(const vector<vector<int>>& s, int i, int j) {
    if (i == j) {
        cout << "M" << i + 1; // print matrix index (1-based)
    } else {
        cout << "(";
        printOptimalParenthesis(s, i, s[i][j]);
        printOptimalParenthesis(s, s[i][j] + 1, j);
        cout << ")";
    }
}

int main() {
    // Matrix dimensions (based on the 6 matrices described above)
    vector<int> p = {10, 5, 30, 15, 20, 25, 5}; // Dimension array for the matrices

    int n = p.size() - 1; // Number of matrices is p.size() - 1

    // Create tables for storing minimum cost and split index
```

```
    vector<vector<int>> m(n, vector<int>(n, 0)); // m[i][j] is the minimum number of multiplications
    vector<vector<int>> s(n, vector<int>(n, 0)); // s[i][j] is the index where the optimal split occurs

    // Call the matrix chain order function
    int minCost = matrixChainOrder(p, n, m, s);

    cout << "Minimum number of multiplications: " << minCost << endl;

    // Print the optimal parenthesization
    cout << "Optimal Parenthesization: ";
    printOptimalParenthesis(s, 0, n - 1);
    cout << endl;

    return 0;
}
```

5.OUTPUT

```
Minimum number of multiplications: 6875
Optimal Parenthesization: (M1((M2M3)(M4(M5M6))))
```

6.CONCLUSION

In this experiment, we implemented the **Matrix Chain Multiplication** problem using dynamic programming to find the optimal order for multiplying a sequence of matrices. The time complexity of the algorithm is O(n^3), making it efficient for a moderate number of matrices, like the 6 matrices in our example. The algorithm not only computes the minimal multiplication cost but also provides the optimal parenthesization for the multiplication sequence.