Name	Surti Keyur
Roll No.	25MCE026
Branch	M.Tech (CSE)

Experiment 9

Aim: Implementation of Longest Common Subsequence (LCS) using Dynamic Programming.

Objectives

- Understand the concept of subsequences and LCS.
- Learn dynamic programming optimization for overlapping subproblems.
- Implement DP matrix computation for LCS.
- Visualize the DP table with directional arrows.
- Extract the final LCS using backtracking.

Problem Statement

Given two sequences X and Y, compute their Longest Common Subsequence (LCS) using dynamic programming and visualize the DP table.

Theory

A subsequence is derived by deleting zero or more characters without changing order. LCS is the longest sequence common to both strings in the same relative order. Dynamic Programming is used due to optimal substructure and overlapping subproblems.

Dynamic Programming Recurrence

If characters match:

```
dp[i][j] = 1 + dp[i-1][j-1]
```

Else:

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

Algorithm

1. Initialize a DP table of size (m+1) × (n+1).

2. Fill the table row-wise.

3. If characters match, take diagonal + 1.

4. Otherwise, take max of top or left cell.

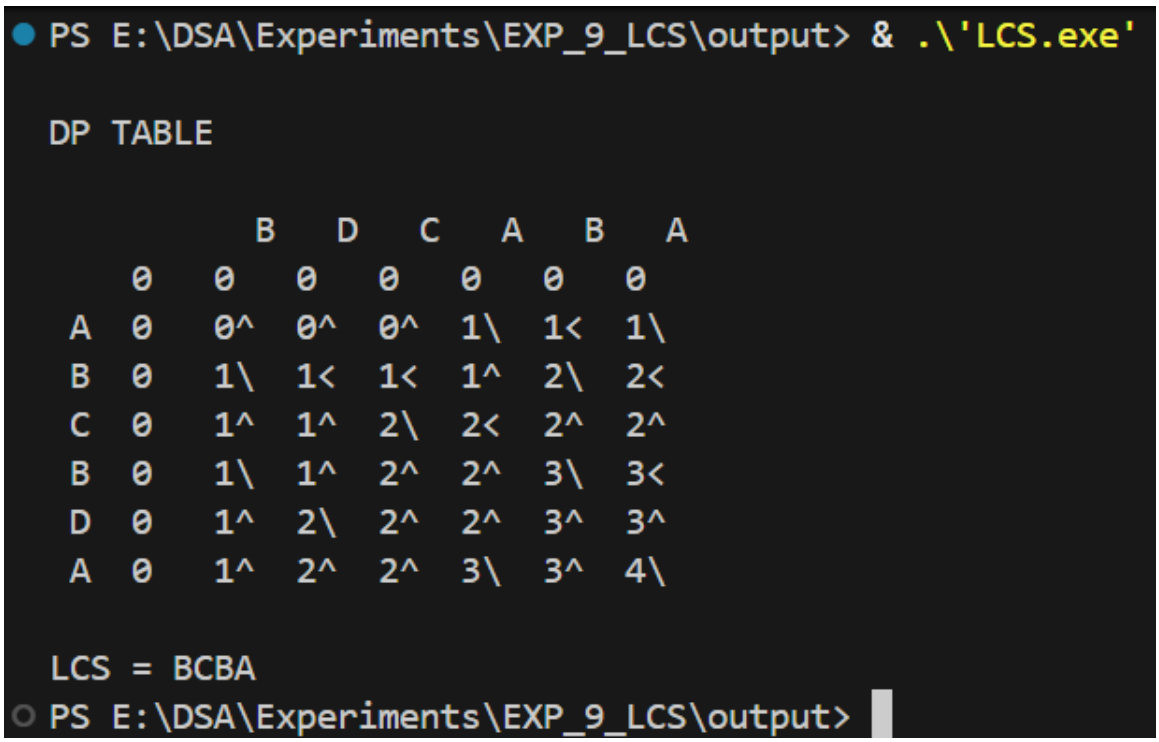
5. Reconstruct LCS through backtracking.

Source Code

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

int main() {
    string X = "ABCBDA";
    string Y = "BDCABA";

    int m = X.size();
    int n = Y.size();

    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
    vector<vector<char>> dir(m+1, vector<char>(n+1, '0'));
    // 'd' = diagonal, 'u' = up, 'l' = left, '0' = no direction

    // Build DP table
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= n; j++) {
            if(X[i-1] == Y[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
                dir[i][j] = 'd';  // diagonal arrow
            } else if(dp[i-1][j] >= dp[i][j-1]) {
                dp[i][j] = dp[i-1][j];
                dir[i][j] = 'u';  // up arrow
            } else {
                dp[i][j] = dp[i][j-1];
                dir[i][j] = 'l';  // left arrow
            }
        }
    }

    cout << "\nDP TABLE\n\n";
    cout << "       ";
    for(char c : Y) cout << setw(4) << c;
    cout << "\n";

    for(int i = 0; i <= m; i++) {
        if(i == 0) cout << "    ";
        else cout << " " << X[i-1] << " ";
```

```
        for(int j = 0; j <= n; j++) {
            char arrow = ' ';

            if(dir[i][j] == 'd') arrow = '\\';   // ↖ diagonal
            else if(dir[i][j] == 'u') arrow = '^'; // ↑ up
            else if(dir[i][j] == 'l') arrow = '<'; // ← left

            if(i == 0 || j == 0) arrow = ' ';  // First row/column

            cout << setw(2) << dp[i][j] << arrow << " ";
        }
        cout << "\n";
    }

    // Reconstruct LCS
    string lcs = "";
    int i = m, j = n;
    while(i > 0 && j > 0) {
        if(dir[i][j] == 'd') {
            lcs.push_back(X[i-1]);
            i--; j--;
        } else if(dir[i][j] == 'u') {
            i--;
        } else {
            j--;
        }
    }
    reverse(lcs.begin(), lcs.end());

    cout << "\nLCS = " << lcs << "\n";

    return 0;
}
```

Output



Time Complexity

The DP table has (m+1) × (n+1) cells.

Each cell is computed once, involving constant-time operations.

Thus, total time complexity is O(m × n).

Space Complexity:

The DP matrix also stores O(m × n) values → Space Complexity = O(m × n).

Sample Output Description

The DP table displays values and arrows:

↖ indicates a match and diagonal movement.

↑ indicates value taken from top cell.

← indicates value taken from left cell.

Final LCS for X = ABCBDA and Y = BDCABA is: BCBA.

Conclusion

LCS was successfully computed using dynamic programming. The DP table clarifies all intermediate decisions, and the experiment demonstrates efficient construction and extraction of the LCS.