2D_DP.cpp

```cpp
/**
 *    Author: devesh95
 *
 *    Topic: Basic 2D Dynamic Programming (DP) Examples
 *
 *    Description:
 *    This file presents 10 examples of problems that use 2D DP.
 *    Each problem is explained in detail along with its DP formulation,
 *    and a function is provided to demonstrate the solution.
 *
 *    Problems Covered:
 *      1. Unique Paths: Count the number of ways to traverse a grid.
 *      2. Unique Paths II: Grid traversal with obstacles.
 *      3. Minimum Path Sum: Find the minimum path sum in a grid.
 *      4. Longest Common Subsequence (LCS): Find the LCS of two strings.
 *      5. Edit Distance: Compute the minimum edit distance between two strings.
 *      6. 0/1 Knapsack: Solve the knapsack problem using a 2D DP table.
 *      7. Matrix Chain Multiplication: Find the minimum cost to multiply a chain of matrices.
 *      8. Longest Palindromic Subsequence: Determine the length of the longest palindromic subsequence.
 *      9. Count Palindromic Substrings: Count the number of palindromic substrings in a string.
 *     10. Interleaving String: Check if a string is an interleaving of two other strings.
 *
 *    Compile with:
 *         g++ -std=c++17 -O2 -Wall Basic_2D_DP.cpp -o dp2d
 *
 *    Run with:
 *         ./dp2d
 */

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

#define int long long
#define pb push_back
#define F first
#define S second

///////////////////////////////////////////////////////////////
// 1. Unique Paths
///////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an m x n grid, count the number of unique paths from the top-left corner
     to the bottom-right corner. You can only move either down or right.

   DP Approach:
     Let dp[i][j] be the number of ways to reach cell (i, j).
     - Base: dp[0][j] = 1 and dp[i][0] = 1 (only one way to reach cells in first row/column)
     - Transition: dp[i][j] = dp[i-1][j] + dp[i][j-1]
```

```cpp
*/
void solve_unique_paths() {
    cout << "\n----- Unique Paths -----\n";
    int m, n;
    cout << "Enter number of rows (m) and columns (n): ";
    cin >> m >> n;
    vector<vector<int>> dp(m, vector<int>(n, 0));

    // Initialize first row and column
    for (int i = 0; i < m; i++) dp[i][0] = 1;
    for (int j = 0; j < n; j++) dp[0][j] = 1;

    // Fill DP table
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }
    cout << "Number of unique paths: " << dp[m-1][n-1] << "\n";
}

///////////////////////////////////////////////////////////////
// 2. Unique Paths II (with obstacles)
///////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Similar to Unique Paths, but the grid contains obstacles (represented by 1).
     Cells with obstacles cannot be traversed.

   DP Approach:
     Let dp[i][j] be the number of ways to reach cell (i, j).
     - If grid[i][j] is an obstacle, dp[i][j] = 0.
     - Else, dp[i][j] = dp[i-1][j] + dp[i][j-1] (if within bounds).
*/
void solve_unique_paths_obstacle() {
    cout << "\n----- Unique Paths II (with obstacles) -----\n";
    int m, n;
    cout << "Enter number of rows (m) and columns (n): ";
    cin >> m >> n;
    vector<vector<int>> grid(m, vector<int>(n, 0));
    cout << "Enter the grid (0 for free cell, 1 for obstacle):\n";
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            cin >> grid[i][j];

    vector<vector<int>> dp(m, vector<int>(n, 0));
    // Base case: start at (0,0) if not an obstacle.
    dp[0][0] = (grid[0][0] == 0) ? 1 : 0;

    // First column
    for (int i = 1; i < m; i++) {
        dp[i][0] = (grid[i][0] == 0 && dp[i-1][0] == 1) ? 1 : 0;
    }
    // First row
```

```cpp
    for (int j = 1; j < n; j++) {
        dp[0][j] = (grid[0][j] == 0 && dp[0][j-1] == 1) ? 1 : 0;
    }
    // Fill DP table
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (grid[i][j] == 0) {
                dp[i][j] = dp[i-1][j] + dp[i][j-1];
            } else {
                dp[i][j] = 0;
            }
        }
    }
    cout << "Number of unique paths (with obstacles): " << dp[m-1][n-1] << "\n";
}

////////////////////////////////////////////////////////////////
// 3. Minimum Path Sum
////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an m x n grid filled with non-negative numbers, find a path from the top-left corner to
     the bottom-right corner which minimizes the sum of all numbers along its path.

   DP Approach:
     Let dp[i][j] be the minimum sum to reach cell (i, j).
     - Base: dp[0][0] = grid[0][0].
     - Transition:
         dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1])
         (using the available directions, considering bounds)
*/
void solve_minimum_path_sum() {
    cout << "\n----- Minimum Path Sum -----\n";
    int m, n;
    cout << "Enter number of rows (m) and columns (n): ";
    cin >> m >> n;
    vector<vector<int>> grid(m, vector<int>(n, 0));
    cout << "Enter the grid values:\n";
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            cin >> grid[i][j];

    vector<vector<int>> dp(m, vector<int>(n, 0));
    dp[0][0] = grid[0][0];

    // Fill first row
    for (int j = 1; j < n; j++) {
        dp[0][j] = dp[0][j-1] + grid[0][j];
    }
    // Fill first column
    for (int i = 1; i < m; i++) {
        dp[i][0] = dp[i-1][0] + grid[i][0];
    }
```

```
    // Fill the rest of the grid
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1]);
        }
    }
    cout << "Minimum path sum: " << dp[m-1][n-1] << "\n";
}

////////////////////////////////////////////////////////////////
// 4. Longest Common Subsequence (LCS)
////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given two strings, find the length of their longest common subsequence.
     A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

   DP Approach:
     Let dp[i][j] be the length of LCS for substrings s1[0..i-1] and s2[0..j-1].
     - If s1[i-1] == s2[j-1], then dp[i][j] = dp[i-1][j-1] + 1.
     - Else, dp[i][j] = max(dp[i-1][j], dp[i][j-1]).
*/
void solve_lcs() {
    cout << "\n----- Longest Common Subsequence (LCS) -----\n";
    string s1, s2;
    cout << "Enter first string: ";
    cin >> s1;
    cout << "Enter second string: ";
    cin >> s2;
    int n = s1.size(), m = s2.size();
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1[i-1] == s2[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    cout << "Length of LCS: " << dp[n][m] << "\n";
}

////////////////////////////////////////////////////////////////
// 5. Edit Distance
////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given two strings, compute the minimum number of operations required to convert one string into the other.
     Operations allowed are insertion, deletion, and substitution.

   DP Approach:
```

```
     Let dp[i][j] be the minimum edit distance between s1[0..i-1] and s2[0..j-1].
     - If s1[i-1] == s2[j-1], dp[i][j] = dp[i-1][j-1].
     - Otherwise, dp[i][j] = 1 + min(dp[i-1][j],   // deletion
                                      dp[i][j-1],   // insertion
                                      dp[i-1][j-1]);// substitution
*/
void solve_edit_distance() {
    cout << "\n----- Edit Distance -----\n";
    string s1, s2;
    cout << "Enter first string: ";
    cin >> s1;
    cout << "Enter second string: ";
    cin >> s2;
    int n = s1.size(), m = s2.size();
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

    // Base cases: converting to/from empty string
    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1[i-1] == s2[j-1])
                dp[i][j] = dp[i-1][j-1];
            else
                dp[i][j] = 1 + min({ dp[i-1][j], dp[i][j-1], dp[i-1][j-1] });
        }
    }
    cout << "Edit Distance: " << dp[n][m] << "\n";
}

///////////////////////////////////////////////////////////////
// 6. 0/1 Knapsack (2D DP Table)
///////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given weights and values of n items, along with a knapsack capacity W, determine the maximum value
     that can be achieved without exceeding the capacity. Each item can be included or excluded (0/1 Knapsack).

   DP Approach:
     Let dp[i][w] be the maximum value that can be achieved with the first i items and capacity w.
     - Transition:
         If weight[i-1] <= w, dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i-1]] + value[i-1]).
         Else, dp[i][w] = dp[i-1][w].
*/
void solve_knapsack() {
    cout << "\n----- 0/1 Knapsack -----\n";
    int n, W;
    cout << "Enter number of items and knapsack capacity: ";
    cin >> n >> W;
```

```
    vector<int> weight(n), value(n);
    cout << "Enter the weights of the items:\n";
    for (int i = 0; i < n; i++) cin >> weight[i];
    cout << "Enter the values of the items:\n";
    for (int i = 0; i < n; i++) cin >> value[i];

    vector<vector<int>> dp(n+1, vector<int>(W+1, 0));
    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (weight[i-1] <= w)
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - weight[i-1]] + value[i-1]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }
    cout << "Maximum value in knapsack: " << dp[n][W] << "\n";
}

///////////////////////////////////////////////////////////////
// 7. Matrix Chain Multiplication
///////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given a chain of matrices, determine the minimum number of multiplications needed
     to multiply the chain. The order in which the matrices are multiplied can affect the cost.

   DP Approach:
     Let dp[i][j] represent the minimum cost to multiply matrices from i to j.
     - Transition:
         dp[i][j] = min{ dp[i][k] + dp[k+1][j] + (dimensions cost) } for all i <= k < j.
*/
void solve_matrix_chain() {
    cout << "\n----- Matrix Chain Multiplication -----\n";
    int n;
    cout << "Enter number of matrices: ";
    cin >> n;
    // Dimensions array of size n+1 where matrix i has dimensions p[i-1] x p[i]
    vector<int> p(n+1);
    cout << "Enter the dimensions (n+1 numbers): ";
    for (int i = 0; i <= n; i++) {
        cin >> p[i];
    }
    vector<vector<int>> dp(n+1, vector<int>(n+1, 0));
    // dp[i][i] is zero cost.
    for (int len = 2; len <= n; len++) {
        for (int i = 1; i <= n - len + 1; i++) {
            int j = i + len - 1;
            dp[i][j] = LLONG_MAX;
            for (int k = i; k < j; k++) {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]);
            }
        }
    }
```

```cpp
    cout << "Minimum multiplication cost: " << dp[1][n] << "\n";
}

/////////////////////////////////////////////////////////////////
// 8. Longest Palindromic Subsequence
/////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given a string, find the length of its longest palindromic subsequence.

   DP Approach:
     Let dp[i][j] be the length of the longest palindromic subsequence in s[i...j].
     - If s[i] == s[j], then dp[i][j] = dp[i+1][j-1] + 2.
     - Otherwise, dp[i][j] = max(dp[i+1][j], dp[i][j-1]).
*/
void solve_longest_palindromic_subsequence() {
    cout << "\n----- Longest Palindromic Subsequence -----\n";
    string s;
    cout << "Enter the string: ";
    cin >> s;
    int n = s.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));
    // Base: single characters are palindromes of length 1.
    for (int i = 0; i < n; i++) dp[i][i] = 1;

    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s[i] == s[j]) {
                dp[i][j] = (len == 2 ? 2 : dp[i+1][j-1] + 2);
            } else {
                dp[i][j] = max(dp[i+1][j], dp[i][j-1]);
            }
        }
    }
    cout << "Length of Longest Palindromic Subsequence: " << dp[0][n-1] << "\n";
}

/////////////////////////////////////////////////////////////////
// 9. Count Palindromic Substrings
/////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given a string, count the number of palindromic substrings in it.

   DP Approach:
     Let dp[i][j] be true if s[i...j] is a palindrome.
     - Base: All single characters are palindromes.
     - For substrings of length 2 and more, check the outer characters and inner substring.
*/
void solve_count_palindromic_substrings() {
    cout << "\n----- Count Palindromic Substrings -----\n";
    string s;
    cout << "Enter the string: ";
```

```
    cin >> s;
    int n = s.size(), count = 0;
    vector<vector<bool>> dp(n, vector<bool>(n, false));

    // Base: single character palindromes
    for (int i = 0; i < n; i++) {
        dp[i][i] = true;
        count++;
    }

    // Check substrings of length 2
    for (int i = 0; i < n-1; i++) {
        if (s[i] == s[i+1]) {
            dp[i][i+1] = true;
            count++;
        }
    }

    // Check substrings of length >= 3
    for (int len = 3; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s[i] == s[j] && dp[i+1][j-1]) {
                dp[i][j] = true;
                count++;
            }
        }
    }
    cout << "Total palindromic substrings: " << count << "\n";
}

///////////////////////////////////////////////////////////////
// 10. Interleaving String
///////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given strings s1, s2, and s3, determine if s3 is formed by an interleaving of s1 and s2.

   DP Approach:
     Let dp[i][j] be true if s3[0...i+j-1] is an interleaving of s1[0...i-1] and s2[0...j-1].
     - Base: dp[0][0] = true.
     - Transition:
           dp[i][j] = (s1[i-1] == s3[i+j-1] and dp[i-1][j]) or
                      (s2[j-1] == s3[i+j-1] and dp[i][j-1]).
*/
void solve_interleaving_string() {
    cout << "\n----- Interleaving String -----\n";
    string s1, s2, s3;
    cout << "Enter first string: ";
    cin >> s1;
    cout << "Enter second string: ";
    cin >> s2;
```

```cpp
    cout << "Enter target interleaved string: ";
    cin >> s3;

    int n = s1.size(), m = s2.size();
    if(n + m != s3.size()){
        cout << "s3 is not an interleaving of s1 and s2 (length mismatch).\n";
        return;
    }
    vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
    dp[0][0] = true;
    // Initialize first row
    for (int i = 1; i <= n; i++) {
        dp[i][0] = dp[i-1][0] && (s1[i-1] == s3[i-1]);
    }
    // Initialize first column
    for (int j = 1; j <= m; j++) {
        dp[0][j] = dp[0][j-1] && (s2[j-1] == s3[j-1]);
    }
    // Fill DP table
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            dp[i][j] = (dp[i-1][j] && s1[i-1] == s3[i+j-1]) ||
                       (dp[i][j-1] && s2[j-1] == s3[i+j-1]);
        }
    }
    cout << "Is s3 an interleaving of s1 and s2? " << (dp[n][m] ? "Yes" : "No") << "\n";
}

///////////////////////////////////////////////////////////////
// Main function with interactive menu for 2D DP problems
///////////////////////////////////////////////////////////////
int32_t main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    cout << "===============================================\n";
    cout << "       Basic 2D Dynamic Programming (DP)       \n";
    cout << "===============================================\n";
    cout << "Select a problem to solve:\n";
    cout << "1. Unique Paths\n";
    cout << "2. Unique Paths II (with obstacles)\n";
    cout << "3. Minimum Path Sum\n";
    cout << "4. Longest Common Subsequence (LCS)\n";
    cout << "5. Edit Distance\n";
    cout << "6. 0/1 Knapsack\n";
    cout << "7. Matrix Chain Multiplication\n";
    cout << "8. Longest Palindromic Subsequence\n";
    cout << "9. Count Palindromic Substrings\n";
    cout << "10. Interleaving String\n";
    cout << "11. Run All Examples\n";
    cout << "Enter your choice: ";

    int choice;
```

```cpp
    cin >> choice;
    cout << "\n";

    switch(choice) {
        case 1:
            solve_unique_paths();
            break;
        case 2:
            solve_unique_paths_obstacle();
            break;
        case 3:
            solve_minimum_path_sum();
            break;
        case 4:
            solve_lcs();
            break;
        case 5:
            solve_edit_distance();
            break;
        case 6:
            solve_knapsack();
            break;
        case 7:
            solve_matrix_chain();
            break;
        case 8:
            solve_longest_palindromic_subsequence();
            break;
        case 9:
            solve_count_palindromic_substrings();
            break;
        case 10:
            solve_interleaving_string();
            break;
        case 11:
            solve_unique_paths();
            solve_unique_paths_obstacle();
            solve_minimum_path_sum();
            solve_lcs();
            solve_edit_distance();
            solve_knapsack();
            solve_matrix_chain();
            solve_longest_palindromic_subsequence();
            solve_count_palindromic_substrings();
            solve_interleaving_string();
            break;
        default:
            cout << "Invalid choice. Exiting...\n";
    }

    return 0;
}

```