Advanced_2D_DP.cpp

```
/**
 *    Author: devesh95
 *
 *    Topic: Advanced 2D Dynamic Programming (DP) Examples
 *
 *    Description:
 *    This file contains 15 advanced DP problems that use 2D DP techniques or
 *    interval/state DP formulations. Each problem is explained in detail, with
 *    the DP state definitions and transitions elaborated upon.
 *
 *    Problems Covered:
 *      1. Maximum Sum Rectangle in a 2D Matrix
 *      2. Longest Common Substring (Contiguous)
 *      3. Longest Increasing Path in a Matrix
 *      4. Regular Expression Matching (with '.' and '*')
 *      5. Wildcard Matching (with '?' and '*')
 *      6. Distinct Subsequences (Count ways to form t from s)
 *      7. Palindrome Partitioning II (Minimum cuts for palindrome partitioning)
 *      8. Egg Dropping Puzzle
 *      9. Count Palindromic Subsequences in a String
 *     10. Longest Common Subarray (Contiguous subsequence)
 *     11. Optimal Game Strategy (Pick coins from ends)
 *     12. Burst Balloons (Interval DP)
 *     13. Longest Arithmetic Subsequence (Using DP with difference)
 *     14. Stone Game (Optimal play for removing stones)
 *     15. Minimum Cost to Merge Stones (Interval DP)
 *
 *    Compilation:
 *         g++ -std=c++17 -O2 -Wall Advanced_2D_DP.cpp -o advanced2d_dp
 *
 *    Execution:
 *         ./advanced2d_dp
 */

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

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

////////////////////////////////////////////////////////////////////////////
// 1. Maximum Sum Rectangle in a 2D Matrix
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given a 2D matrix of integers, find the sub-rectangle (contiguous block)
     with the maximum possible sum.

   DP/Algorithm Approach:
```

```
     - Fix the left and right column boundaries.
     - For each pair, collapse the 2D problem into a 1D problem (summing rows)
       and then use Kadane's algorithm on the temporary 1D array.
*/
void solve_max_sum_rectangle() {
    cout << "\n----- Maximum Sum Rectangle in a 2D Matrix -----\n";
    int rows, cols;
    cout << "Enter number of rows and columns: ";
    cin >> rows >> cols;
    vector<vector<int>> matrix(rows, vector<int>(cols));
    cout << "Enter the matrix elements:\n";
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            cin >> matrix[i][j];

    int maxSum = LLONG_MIN;
    // Variables to store rectangle boundaries (optional)
    int finalLeft = 0, finalRight = 0, finalTop = 0, finalBottom = 0;

    // Left boundary of the rectangle
    for (int left = 0; left < cols; left++) {
        vector<int> temp(rows, 0);
        // Right boundary from left to end
        for (int right = left; right < cols; right++) {
            // Sum rows between left and right for each row
            for (int i = 0; i < rows; i++)
                temp[i] += matrix[i][right];

            // Apply Kadane's algorithm on temp[]
            int sum = 0, localMax = LLONG_MIN;
            int start = 0, localTop = 0, localBottom = 0;
            for (int i = 0; i < rows; i++) {
                sum += temp[i];
                if (sum > localMax) {
                    localMax = sum;
                    localTop = start;
                    localBottom = i;
                }
                if (sum < 0) {
                    sum = 0;
                    start = i+1;
                }
            }
            if (localMax > maxSum) {
                maxSum = localMax;
                finalLeft = left; finalRight = right;
                finalTop = localTop; finalBottom = localBottom;
            }
        }
    }
    cout << "Maximum rectangle sum is: " << maxSum << "\n";
    // Optionally print boundaries.
    // cout << "Boundaries: Top " << finalTop << ", Bottom " << finalBottom
    //      << ", Left " << finalLeft << ", Right " << finalRight << "\n";
```

```cpp
}

////////////////////////////////////////////////////////////////////////////
// 2. Longest Common Substring (Contiguous)
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given two strings, find the longest contiguous substring that appears in both.

   DP Approach:
     Let dp[i][j] be the length of the longest common suffix of 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; otherwise 0.
     Track the maximum value in dp.
*/
void solve_longest_common_substring() {
    cout << "\n----- Longest Common Substring -----\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));
    int maxLen = 0;
    string res = "";
    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;
                if (dp[i][j] > maxLen) {
                    maxLen = dp[i][j];
                    res = s1.substr(i - maxLen, maxLen);
                }
            } else {
                dp[i][j] = 0;
            }
        }
    }
    cout << "Longest common substring: \"" << res << "\" with length " << maxLen << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 3. Longest Increasing Path in a Matrix
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given a matrix, find the length of the longest strictly increasing path.
     You can move in four directions (up, down, left, right).

   DP Approach:
     Use DFS with memoization. Let memo[i][j] store the length of the longest increasing
     path starting at cell (i, j). Explore neighbors with greater value.
*/
int dx[4] = {0, 0, 1, -1};
```

```cpp
int dy[4] = {1, -1, 0, 0};

int dfs(int i, int j, vector<vector<int>>& matrix, vector<vector<int>>& memo) {
    if (memo[i][j] != 0)
        return memo[i][j];
    int maxPath = 1;
    int rows = matrix.size(), cols = matrix[0].size();
    for (int dir = 0; dir < 4; dir++) {
        int x = i + dx[dir], y = j + dy[dir];
        if (x >= 0 && x < rows && y >= 0 && y < cols && matrix[x][y] > matrix[i][j]) {
            maxPath = max(maxPath, 1 + dfs(x, y, matrix, memo));
        }
    }
    memo[i][j] = maxPath;
    return maxPath;
}

void solve_longest_increasing_path() {
    cout << "\n----- Longest Increasing Path in a Matrix -----\n";
    int rows, cols;
    cout << "Enter number of rows and columns: ";
    cin >> rows >> cols;
    vector<vector<int>> matrix(rows, vector<int>(cols));
    cout << "Enter the matrix elements:\n";
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            cin >> matrix[i][j];

    vector<vector<int>> memo(rows, vector<int>(cols, 0));
    int res = 0;
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            res = max(res, dfs(i, j, matrix, memo));

    cout << "Length of longest increasing path: " << res << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 4. Regular Expression Matching
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Implement regex matching with support for '.' and '*'.
     '.' Matches any single character.
     '*' Matches zero or more of the preceding element.

   DP Approach:
     Let dp[i][j] be true if s[0...i-1] matches p[0...j-1].
     Transition considers the '*' case carefully.
*/
void solve_regex_matching() {
    cout << "\n----- Regular Expression Matching -----\n";
    string s, p;
    cout << "Enter the input string: ";
```

```
    cin >> s;
    cout << "Enter the pattern: ";
    cin >> p;
    int n = s.size(), m = p.size();
    vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
    dp[0][0] = true;

    // Initialize patterns like a*, a*b*, etc.
    for (int j = 1; j <= m; j++) {
        if (p[j-1] == '*') {
            dp[0][j] = dp[0][j-2];
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (p[j-1] == s[i-1] || p[j-1] == '.')
                dp[i][j] = dp[i-1][j-1];
            else if (p[j-1] == '*') {
                dp[i][j] = dp[i][j-2]; // '*' represents 0 occurrence
                if (p[j-2] == s[i-1] || p[j-2] == '.')
                    dp[i][j] = dp[i][j] || dp[i-1][j];
            } else {
                dp[i][j] = false;
            }
        }
    }
    cout << "Does the string match the pattern? " << (dp[n][m] ? "Yes" : "No") << "\n";
}

////////////////////////////////////////////////////////////////////////////////
// 5. Wildcard Matching
////////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Implement wildcard matching with support for '?' and '*'.
     '?' Matches any single character.
     '*' Matches any sequence of characters (including the empty sequence).

   DP Approach:
     Let dp[i][j] be true if s[0...i-1] matches p[0...j-1].
     Update dp considering '*' can match zero or more characters.
*/
void solve_wildcard_matching() {
    cout << "\n----- Wildcard Matching -----\n";
    string s, p;
    cout << "Enter the input string: ";
    cin >> s;
    cout << "Enter the wildcard pattern: ";
    cin >> p;
    int n = s.size(), m = p.size();
    vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
    dp[0][0] = true;
    // Initialize pattern when string is empty
    for (int j = 1; j <= m; j++) {
```

```cpp
        if (p[j-1] == '*')
            dp[0][j] = dp[0][j-1];
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (p[j-1] == s[i-1] || p[j-1] == '?')
                dp[i][j] = dp[i-1][j-1];
            else if (p[j-1] == '*')
                dp[i][j] = dp[i][j-1] || dp[i-1][j];
            else
                dp[i][j] = false;
        }
    }
    cout << "Does the string match the wildcard pattern? " << (dp[n][m] ? "Yes" : "No") << "\n";
}

////////////////////////////////////////////////////////////////////////////////
// 6. Distinct Subsequences
////////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given strings s and t, count the number of distinct subsequences of s that equal t.

   DP Approach:
     Let dp[i][j] be the number of distinct subsequences of s[0...i-1] that form t[0...j-1].
     Transition:
       If s[i-1] == t[j-1], then dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
       Else, dp[i][j] = dp[i-1][j].
*/
void solve_distinct_subsequences() {
    cout << "\n----- Distinct Subsequences -----\n";
    string s, t;
    cout << "Enter source string s: ";
    cin >> s;
    cout << "Enter target string t: ";
    cin >> t;
    int n = s.size(), m = t.size();
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
    // Base: empty target is a subsequence of any string
    for (int i = 0; i <= n; i++)
        dp[i][0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s[i-1] == t[j-1])
                dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    cout << "Number of distinct subsequences: " << dp[n][m] << "\n";
}

////////////////////////////////////////////////////////////////////////////////
```

```cpp
// 7. Palindrome Partitioning II (Minimum Cuts)
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given a string, partition it such that every substring is a palindrome.
     Find the minimum number of cuts needed for a palindrome partitioning.

   DP Approach:
     Let dp[i] be the minimum cuts needed for substring s[0..i].
     Precompute a palindrome table for all substrings.
*/
void solve_palindrome_partitioning() {
    cout << "\n----- Palindrome Partitioning II (Minimum Cuts) -----\n";
    string s;
    cout << "Enter the string: ";
    cin >> s;
    int n = s.size();
    vector<vector<bool>> isPal(n, vector<bool>(n, false));
    // Precompute palindrome table
    for (int i = 0; i < n; i++) {
        isPal[i][i] = true;
    }
    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]) {
                isPal[i][j] = (len == 2) ? true : isPal[i+1][j-1];
            } else {
                isPal[i][j] = false;
            }
        }
    }
    vector<int> dp(n, 0);
    for (int i = 0; i < n; i++) {
        if (isPal[0][i]) {
            dp[i] = 0;
        } else {
            dp[i] = i; // worst-case (cut before every char)
            for (int j = 0; j < i; j++) {
                if (isPal[j+1][i])
                    dp[i] = min(dp[i], dp[j] + 1);
            }
        }
    }
    cout << "Minimum cuts required: " << dp[n-1] << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 8. Egg Dropping Puzzle
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given K eggs and N floors, determine the minimum number of trials needed in the worst
case
```

```
     to find the critical floor from which eggs start breaking.

   DP Approach:
     Let dp[k][n] be the minimum number of trials needed with k eggs and n floors.
     Use the recurrence:
       dp[k][n] = 1 + min_{1 <= x <= n}(max(dp[k-1][x-1], dp[k][n-x]))
*/
void solve_egg_dropping() {
    cout << "\n----- Egg Dropping Puzzle -----\n";
    int K, N;
    cout << "Enter number of eggs and number of floors: ";
    cin >> K >> N;
    vector<vector<int>> dp(K+1, vector<int>(N+1, 0));
    // Base cases
    for (int i = 1; i <= N; i++) dp[1][i] = i; // 1 egg: trial each floor
    for (int k = 1; k <= K; k++) dp[k][0] = 0;
    for (int k = 1; k <= K; k++) dp[k][1] = 1;

    for (int k = 2; k <= K; k++) {
        for (int n = 2; n <= N; n++) {
            dp[k][n] = INT_MAX;
            // Try dropping from each floor x
            for (int x = 1; x <= n; x++) {
                int res = 1 + max(dp[k-1][x-1], dp[k][n-x]);
                dp[k][n] = min(dp[k][n], res);
            }
        }
    }
    cout << "Minimum number of trials in worst case: " << dp[K][N] << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 9. Count Palindromic Subsequences in a String
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given a string, count the number of palindromic subsequences (not necessarily
distinct).

   DP Approach:
     Let dp[i][j] denote the count of palindromic subsequences in s[i...j].
     Use recurrence based on matching endpoints and inclusion/exclusion.
*/
void solve_count_palindromic_subsequences() {
    cout << "\n----- Count Palindromic Subsequences -----\n";
    string s;
    cout << "Enter the string: ";
    cin >> s;
    int n = s.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // Base: each single character is a palindrome
    for (int i = 0; i < n; i++)
        dp[i][i] = 1;
```

```cpp

    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] = dp[i+1][j] + dp[i][j-1] + 1;
            } else {
                dp[i][j] = dp[i+1][j] + dp[i][j-1] - dp[i+1][j-1];
            }
        }
    }
    cout << "Total palindromic subsequences: " << dp[0][n-1] << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 10. Longest Common Subarray (Contiguous)
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given two arrays, find the length of the longest subarray (contiguous) common to both.

   DP Approach:
     Let dp[i][j] be the length of the longest common suffix of A[0...i-1] and B[0...j-1].
     If A[i-1] == B[j-1], then dp[i][j] = dp[i-1][j-1] + 1.
*/
void solve_longest_common_subarray() {
    cout << "\n----- Longest Common Subarray -----\n";
    int n, m;
    cout << "Enter the size of first array and second array: ";
    cin >> n >> m;
    vector<int> A(n), B(m);
    cout << "Enter elements of first array:\n";
    for (int i = 0; i < n; i++) cin >> A[i];
    cout << "Enter elements of second array:\n";
    for (int j = 0; j < m; j++) cin >> B[j];

    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
    int maxLen = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (A[i-1] == B[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
                maxLen = max(maxLen, dp[i][j]);
            }
        }
    }
    cout << "Length of longest common subarray: " << maxLen << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 11. Optimal Game Strategy (Pick Coins from Ends)
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
```

```
     Given an array of coins, two players pick coins from either end.
     Compute the maximum amount of money the first player can collect assuming optimal play.

   DP Approach:
     Let dp[i][j] be the maximum amount the current player can collect from coins i to j.
     The recurrence:
       dp[i][j] = max( coins[i] + min(dp[i+2][j], dp[i+1][j-1]),
                       coins[j] + min(dp[i+1][j-1], dp[i][j-2]) )
*/
void solve_optimal_game_strategy() {
    cout << "\n----- Optimal Game Strategy -----\n";
    int n;
    cout << "Enter the number of coins: ";
    cin >> n;
    vector<int> coins(n);
    cout << "Enter coin values:\n";
    for (int i = 0; i < n; i++) cin >> coins[i];

    vector<vector<int>> dp(n, vector<int>(n, 0));
    // Base cases: one coin and two coins.
    for (int i = 0; i < n; i++)
        dp[i][i] = coins[i];
    for (int i = 0; i < n-1; i++)
        dp[i][i+1] = max(coins[i], coins[i+1]);

    for (int len = 3; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            int a = (i+2 <= j) ? dp[i+2][j] : 0;
            int b = (i+1 <= j-1) ? dp[i+1][j-1] : 0;
            int c = (i <= j-2) ? dp[i][j-2] : 0;
            dp[i][j] = max(coins[i] + min(a, b),
                           coins[j] + min(b, c));
        }
    }
    cout << "Maximum amount first player can collect: " << dp[0][n-1] << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 12. Burst Balloons (Interval DP)
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array of balloons, burst them in an order such that you maximize coins earned.
     When you burst balloon i, coins = nums[left] * nums[i] * nums[right], where left and
right
     are adjacent balloons.

   DP Approach:
     Let dp[i][j] be the maximum coins obtainable by bursting balloons in the open interval
(i, j).
     Use interval DP to try every possible last balloon to burst.
*/
void solve_burst_balloons() {
```

```cpp
    cout << "\n----- Burst Balloons -----\n";
    int n;
    cout << "Enter number of balloons: ";
    cin >> n;
    vector<int> nums(n);
    cout << "Enter the balloon numbers:\n";
    for (int i = 0; i < n; i++) cin >> nums[i];

    // Add boundaries 1 at the start and end
    vector<int> balloons;
    balloons.push_back(1);
    for (int x : nums) balloons.push_back(x);
    balloons.push_back(1);
    int m = balloons.size();
    vector<vector<int>> dp(m, vector<int>(m, 0));

    for (int len = 2; len < m; len++) {
        for (int i = 0; i < m - len; i++) {
            int j = i + len;
            for (int k = i+1; k < j; k++) {
                dp[i][j] = max(dp[i][j], balloons[i]*balloons[k]*balloons[j] + dp[i][k] + dp[k][j]);
            }
        }
    }
    cout << "Maximum coins obtainable: " << dp[0][m-1] << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 13. Longest Arithmetic Subsequence
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array, find the length of the longest arithmetic subsequence.

   DP Approach:
     Let dp[i][d] be the length of the longest arithmetic subsequence ending at index i with difference d.
     Use a map at each index to store differences.
*/
void solve_longest_arithmetic_subsequence() {
    cout << "\n----- Longest Arithmetic Subsequence -----\n";
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the elements:\n";
    for (int i = 0; i < n; i++) cin >> arr[i];

    int ans = 0;
    vector<unordered_map<int,int>> dp(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            int diff = arr[i] - arr[j];
```

```
            dp[i][diff] = max(dp[i][diff], dp[j].count(diff) ? dp[j][diff] + 1 : 2);
            ans = max(ans, dp[i][diff]);
        }
    }
    cout << "Length of longest arithmetic subsequence: " << ans << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 14. Stone Game (Interval DP)
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array representing piles of stones, two players remove stones optimally.
     Compute the maximum difference in score the first player can secure over the second.

   DP Approach:
     Let dp[i][j] be the maximum score difference the current player can achieve from piles i to j.
     The recurrence:
       dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]).
*/
void solve_stone_game() {
    cout << "\n----- Stone Game -----\n";
    int n;
    cout << "Enter the number of piles: ";
    cin >> n;
    vector<int> piles(n);
    cout << "Enter the number of stones in each pile:\n";
    for (int i = 0; i < n; i++) cin >> piles[i];

    vector<vector<int>> dp(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++)
        dp[i][i] = piles[i];
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]);
        }
    }
    cout << "Maximum score difference the first player can achieve: " << dp[0][n-1] << "\n";
}

////////////////////////////////////////////////////////////////////////////
// 15. Minimum Cost to Merge Stones
////////////////////////////////////////////////////////////////////////////
/*
   Problem Statement:
     Given an array of stone weights, merge adjacent stones until one stone remains.
     The cost of merging is the sum of the stones being merged.
     Find the minimum total cost to merge all stones.

   DP Approach:
     Let dp[i][j] be the minimum cost to merge stones from i to j.
     Recurrence:
```

```
       dp[i][j] = min_{i <= k < j} { dp[i][k] + dp[k+1][j] } + sum(i..j)
     Precompute prefix sums for efficient range sum calculation.
*/
void solve_minimum_cost_merge_stones() {
    cout << "\n----- Minimum Cost to Merge Stones -----\n";
    int n;
    cout << "Enter the number of stones: ";
    cin >> n;
    vector<int> stones(n);
    cout << "Enter the weights of the stones:\n";
    for (int i = 0; i < n; i++) cin >> stones[i];

    vector<int> prefix(n+1, 0);
    for (int i = 0; i < n; i++) {
        prefix[i+1] = prefix[i] + stones[i];
    }

    vector<vector<int>> dp(n, vector<int>(n, 0));
    // dp[i][i] = 0, already initialized.
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; 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] + (prefix[j+1] - prefix[i]));
            }
        }
    }
    cout << "Minimum cost to merge all stones: " << dp[0][n-1] << "\n";
}

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

    cout << "=========================================================\n";
    cout << "         Advanced 2D Dynamic Programming (DP)         \n";
    cout << "=========================================================\n";
    cout << "Select a problem to solve:\n";
    cout << " 1. Maximum Sum Rectangle in a 2D Matrix\n";
    cout << " 2. Longest Common Substring\n";
    cout << " 3. Longest Increasing Path in a Matrix\n";
    cout << " 4. Regular Expression Matching\n";
    cout << " 5. Wildcard Matching\n";
    cout << " 6. Distinct Subsequences\n";
    cout << " 7. Palindrome Partitioning II (Minimum Cuts)\n";
    cout << " 8. Egg Dropping Puzzle\n";
    cout << " 9. Count Palindromic Subsequences\n";
    cout << "10. Longest Common Subarray\n";
    cout << "11. Optimal Game Strategy (Coins from Ends)\n";
```

```
    cout << "12. Burst Balloons\n";
    cout << "13. Longest Arithmetic Subsequence\n";
    cout << "14. Stone Game (Optimal Play)\n";
    cout << "15. Minimum Cost to Merge Stones\n";
    cout << "16. Run All Examples\n";
    cout << "Enter your choice: ";

    int choice;
    cin >> choice;
    cout << "\n";

    switch(choice) {
        case 1:  solve_max_sum_rectangle(); break;
        case 2:  solve_longest_common_substring(); break;
        case 3:  solve_longest_increasing_path(); break;
        case 4:  solve_regex_matching(); break;
        case 5:  solve_wildcard_matching(); break;
        case 6:  solve_distinct_subsequences(); break;
        case 7:  solve_palindrome_partitioning(); break;
        case 8:  solve_egg_dropping(); break;
        case 9:  solve_count_palindromic_subsequences(); break;
        case 10: solve_longest_common_subarray(); break;
        case 11: solve_optimal_game_strategy(); break;
        case 12: solve_burst_balloons(); break;
        case 13: solve_longest_arithmetic_subsequence(); break;
        case 14: solve_stone_game(); break;
        case 15: solve_minimum_cost_merge_stones(); break;
        case 16:
            solve_max_sum_rectangle();
            solve_longest_common_substring();
            solve_longest_increasing_path();
            solve_regex_matching();
            solve_wildcard_matching();
            solve_distinct_subsequences();
            solve_palindrome_partitioning();
            solve_egg_dropping();
            solve_count_palindromic_subsequences();
            solve_longest_common_subarray();
            solve_optimal_game_strategy();
            solve_burst_balloons();
            solve_longest_arithmetic_subsequence();
            solve_stone_game();
            solve_minimum_cost_merge_stones();
            break;
        default:
            cout << "Invalid choice. Exiting...\n";
    }

    return 0;
}

```