2-D Dynamic Programming Solutions (Java, Recursion)

```
import java.util.*; public class UniquePathsII { static int solve(int[][] grid, int i, int j, int[][] dp) { if (i < 0 || j < 0 || grid[i][j] == 1) return 0; if (i == 0 && j == 0) return 1; if (dp[i][j] != -1) return dp[i][j]; return dp[i][j] = solve(grid, i - 1, j, dp) + solve(grid, i, j - 1, dp); } public static void main(String[] args) { Scanner sc = new Scanner(System.in); int m = sc.nextInt(), n = sc.nextInt(); int[][] grid = new int[m][n]; for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) grid[i][j] = sc.nextInt(); int[][] dp = new int[m][n]; for (int[] r : dp) Arrays.fill(r, -1); System.out.println(solve(grid, m - 1, n - 1, dp)); } }
```

```
import java.util.*; public class DungeonGame { static int solve(int[][] g, int i, int j, int[][] dp) { int m = g.length, n = g[0].length; if (i == m - 1 && j == n - 1) return Math.max(1, 1 - g[i][j]); if (i >= m || j >= n) return Integer.MAX_VALUE; if (dp[i][j] != -1) return dp[i][j]; int right = solve(g, i, j + 1, dp); int down = solve(g, i + 1, j, dp); int need = Math.min(right, down) - g[i][j]; return dp[i][j] = Math.max(1, need); } public static void main(String[] args) { Scanner sc = new Scanner(System.in); int m = sc.nextInt(), n = sc.nextInt(); int[][] g = new int[m][n]; for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) g[i][j] = sc.nextInt(); int[][] dp = new int[m][n]; for (int[] r : dp) Arrays.fill(r, -1); System.out.println(solve(g, 0, 0, dp)); } }
```

```
import java.util.*; public class TwoFingers { static int dist(int a, int b) { if (a == -1 || b == -1) return 0; return Math.abs(a / 6 - b / 6) + Math.abs(a % 6 - b % 6); } static int solve(String w, int i, int l, int r, int[][][] dp) { if (i == w.length()) return 0; if (dp[i][l + 1][r + 1] != -1) return dp[i][l + 1][r + 1]; int c = w.charAt(i) - 'A'; int left = dist(l, c) + solve(w, i + 1, c, r, dp); int right = dist(r, c) + solve(w, i + 1, l, c, dp); return dp[i][l + 1][r + 1] = Math.min(left, right); } public static void main(String[] args) { Scanner sc = new Scanner(System.in); String w = sc.next(); int[][][] dp = new int[w.length()][27][27]; for (int[][] d1 : dp) for (int[] d2 : d1) Arrays.fill(d2, -1); System.out.println(solve(w, 0, -1, -1, dp)); } }
```

```
import java.util.*; public class MinFallingPathSumII { static int solve(int[][] g, int i, int j, int[][] dp) { int m = g.length, n = g[0].length; if (i == m - 1) return g[i][j]; if (dp[i][j] != -1) return dp[i][j]; int min = Integer.MAX_VALUE; for (int k = 0; k < n; k++) if (k != j) min = Math.min(min, solve(g, i + 1, k, dp)); return dp[i][j] = g[i][j] + min; } public static void main(String[] args) { Scanner sc = new Scanner(System.in); int m = sc.nextInt(), n = sc.nextInt(); int[][] g = new int[m][n]; for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) g[i][j] = sc.nextInt(); int[][] dp = new int[m][n]; for (int[] r : dp) Arrays.fill(r, -1); int ans = Integer.MAX_VALUE; for (int j = 0; j < n; j++) ans = Math.min(ans, solve(g, 0, j, dp)); System.out.println(ans); } }
```

```
import java.util.*; public class OptimalDivision { static String solve(int[] nums, int i, int j, String[][] dp) { if (i == j) return String.valueOf(nums[i]); if (dp[i][j] != null) return dp[i][j]; if (i + 1 == j) return nums[i] + "/" + nums[j]; String res = nums[i] + "/("; for (int k = i + 1; k < j; k++) res += nums[k] + "/"; res += nums[j] + ")"; return dp[i][j] = res; } public static void main(String[] args) { Scanner sc = new Scanner(System.in); int n = sc.nextInt(); int[] nums = new int[n]; for (int i = 0; i < n; i++) nums[i] = sc.nextInt(); String[][] dp = new String[n][n]; System.out.println(solve(nums, 0, n - 1, dp)); } }
```