

2D_DP.cpp

```

1  /**
2  *   Author: devesh95
3  *
4  *   Topic: Basic 2D Dynamic Programming (DP) Examples
5  *
6  *   Description:
7  *   This file presents 10 examples of problems that use 2D DP.
8  *   Each problem is explained in detail along with its DP formulation,
9  *   and a function is provided to demonstrate the solution.
10 *
11 *   Problems Covered:
12 *       1. Unique Paths: Count the number of ways to traverse a grid.
13 *       2. Unique Paths II: Grid traversal with obstacles.
14 *       3. Minimum Path Sum: Find the minimum path sum in a grid.
15 *       4. Longest Common Subsequence (LCS): Find the LCS of two strings.
16 *       5. Edit Distance: Compute the minimum edit distance between two strings.
17 *       6. 0/1 Knapsack: Solve the knapsack problem using a 2D DP table.
18 *       7. Matrix Chain Multiplication: Find the minimum cost to multiply a chain of
    matrices.
19 *       8. Longest Palindromic Subsequence: Determine the length of the longest palindromic
    subsequence.
20 *       9. Count Palindromic Substrings: Count the number of palindromic substrings in a
    string.
21 *       10. Interleaving String: Check if a string is an interleaving of two other strings.
22 *
23 *   Compile with:
24 *       g++ -std=c++17 -O2 -Wall Basic_2D_DP.cpp -o dp2d
25 *
26 *   Run with:
27 *       ./dp2d
28 */
29
30 #include <bits/stdc++.h>
31 using namespace std;
32
33 #define int long long
34 #define pb push_back
35 #define F first
36 #define S second
37
38 //////////////////////////////////////
39 // 1. Unique Paths
40 //////////////////////////////////////
41 /*
42     Problem Statement:
43     Given an m x n grid, count the number of unique paths from the top-left corner
44     to the bottom-right corner. You can only move either down or right.
45
46     DP Approach:
47     Let dp[i][j] be the number of ways to reach cell (i, j).
48     - Base: dp[0][j] = 1 and dp[i][0] = 1 (only one way to reach cells in first row/column)
49     - Transition: dp[i][j] = dp[i-1][j] + dp[i][j-1]

```

```

50  */
51  void solve_unique_paths() {
52      cout << "\n----- Unique Paths ----- \n";
53      int m, n;
54      cout << "Enter number of rows (m) and columns (n): ";
55      cin >> m >> n;
56      vector<vector<int>>> dp(m, vector<int>(n, 0));
57
58      // Initialize first row and column
59      for (int i = 0; i < m; i++) dp[i][0] = 1;
60      for (int j = 0; j < n; j++) dp[0][j] = 1;
61
62      // Fill DP table
63      for (int i = 1; i < m; i++) {
64          for (int j = 1; j < n; j++) {
65              dp[i][j] = dp[i-1][j] + dp[i][j-1];
66          }
67      }
68      cout << "Number of unique paths: " << dp[m-1][n-1] << "\n";
69  }
70
71  ///////////////////////////////////////////////////////////////////
72  // 2. Unique Paths II (with obstacles)
73  ///////////////////////////////////////////////////////////////////
74  /*
75      Problem Statement:
76      Similar to Unique Paths, but the grid contains obstacles (represented by 1).
77      Cells with obstacles cannot be traversed.
78
79      DP Approach:
80      Let dp[i][j] be the number of ways to reach cell (i, j).
81      - If grid[i][j] is an obstacle, dp[i][j] = 0.
82      - Else, dp[i][j] = dp[i-1][j] + dp[i][j-1] (if within bounds).
83  */
84  void solve_unique_paths_obstacle() {
85      cout << "\n----- Unique Paths II (with obstacles) ----- \n";
86      int m, n;
87      cout << "Enter number of rows (m) and columns (n): ";
88      cin >> m >> n;
89      vector<vector<int>>> grid(m, vector<int>(n, 0));
90      cout << "Enter the grid (0 for free cell, 1 for obstacle): \n";
91      for (int i = 0; i < m; i++)
92          for (int j = 0; j < n; j++)
93              cin >> grid[i][j];
94
95      vector<vector<int>>> dp(m, vector<int>(n, 0));
96      // Base case: start at (0,0) if not an obstacle.
97      dp[0][0] = (grid[0][0] == 0) ? 1 : 0;
98
99      // First column
100     for (int i = 1; i < m; i++) {
101         dp[i][0] = (grid[i][0] == 0 && dp[i-1][0] == 1) ? 1 : 0;
102     }
103     // First row

```

```

104     for (int j = 1; j < n; j++) {
105         dp[0][j] = (grid[0][j] == 0 && dp[0][j-1] == 1) ? 1 : 0;
106     }
107     // Fill DP table
108     for (int i = 1; i < m; i++) {
109         for (int j = 1; j < n; j++) {
110             if (grid[i][j] == 0) {
111                 dp[i][j] = dp[i-1][j] + dp[i][j-1];
112             } else {
113                 dp[i][j] = 0;
114             }
115         }
116     }
117     cout << "Number of unique paths (with obstacles): " << dp[m-1][n-1] << "\n";
118 }
119
120 ///////////////////////////////////////////////////
121 // 3. Minimum Path Sum
122 ///////////////////////////////////////////////////
123 /*
124     Problem Statement:
125     Given an m x n grid filled with non-negative numbers, find a path from the top-left
126     corner to
127     the bottom-right corner which minimizes the sum of all numbers along its path.
128
129     DP Approach:
130     Let dp[i][j] be the minimum sum to reach cell (i, j).
131     - Base: dp[0][0] = grid[0][0].
132     - Transition:
133         dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1])
134         (using the available directions, considering bounds)
135 */
136 void solve_minimum_path_sum() {
137     cout << "\n----- Minimum Path Sum ----- \n";
138     int m, n;
139     cout << "Enter number of rows (m) and columns (n): ";
140     cin >> m >> n;
141     vector<vector<int>> grid(m, vector<int>(n, 0));
142     cout << "Enter the grid values:\n";
143     for (int i = 0; i < m; i++)
144         for (int j = 0; j < n; j++)
145             cin >> grid[i][j];
146
147     vector<vector<int>> dp(m, vector<int>(n, 0));
148     dp[0][0] = grid[0][0];
149
150     // Fill first row
151     for (int j = 1; j < n; j++) {
152         dp[0][j] = dp[0][j-1] + grid[0][j];
153     }
154     // Fill first column
155     for (int i = 1; i < m; i++) {
156         dp[i][0] = dp[i-1][0] + grid[i][0];
157     }

```

```

157 // Fill the rest of the grid
158 for (int i = 1; i < m; i++) {
159     for (int j = 1; j < n; j++) {
160         dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1]);
161     }
162 }
163 cout << "Minimum path sum: " << dp[m-1][n-1] << "\n";
164 }
165
166 ///////////////////////////////////////////////////
167 // 4. Longest Common Subsequence (LCS)
168 ///////////////////////////////////////////////////
169 /*
170     Problem Statement:
171     Given two strings, find the length of their longest common subsequence.
172     A subsequence is a sequence that appears in the same relative order, but not
173     necessarily contiguous.
174
175     DP Approach:
176     Let dp[i][j] be the length of LCS for substrings s1[0..i-1] and s2[0..j-1].
177     - If s1[i-1] == s2[j-1], then dp[i][j] = dp[i-1][j-1] + 1.
178     - Else, dp[i][j] = max(dp[i-1][j], dp[i][j-1]).
179 */
180 void solve_lcs() {
181     cout << "\n----- Longest Common Subsequence (LCS) ----- \n";
182     string s1, s2;
183     cout << "Enter first string: ";
184     cin >> s1;
185     cout << "Enter second string: ";
186     cin >> s2;
187     int n = s1.size(), m = s2.size();
188     vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
189
190     for (int i = 1; i <= n; i++) {
191         for (int j = 1; j <= m; j++) {
192             if (s1[i-1] == s2[j-1])
193                 dp[i][j] = dp[i-1][j-1] + 1;
194             else
195                 dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
196         }
197     }
198     cout << "Length of LCS: " << dp[n][m] << "\n";
199 }
200
201 ///////////////////////////////////////////////////
202 // 5. Edit Distance
203 ///////////////////////////////////////////////////
204 /*
205     Problem Statement:
206     Given two strings, compute the minimum number of operations required to convert one
207     string into the other.
208     Operations allowed are insertion, deletion, and substitution.
209
210     DP Approach:

```

```

209     Let dp[i][j] be the minimum edit distance between s1[0..i-1] and s2[0..j-1].
210     - If s1[i-1] == s2[j-1], dp[i][j] = dp[i-1][j-1].
211     - Otherwise, dp[i][j] = 1 + min(dp[i-1][j],    // deletion
212                                     dp[i][j-1],    // insertion
213                                     dp[i-1][j-1]); // substitution
214 */
215 void solve_edit_distance() {
216     cout << "\n----- Edit Distance ----- \n";
217     string s1, s2;
218     cout << "Enter first string: ";
219     cin >> s1;
220     cout << "Enter second string: ";
221     cin >> s2;
222     int n = s1.size(), m = s2.size();
223     vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
224
225     // Base cases: converting to/from empty string
226     for (int i = 0; i <= n; i++) dp[i][0] = i;
227     for (int j = 0; j <= m; j++) dp[0][j] = j;
228
229     for (int i = 1; i <= n; i++) {
230         for (int j = 1; j <= m; j++) {
231             if (s1[i-1] == s2[j-1])
232                 dp[i][j] = dp[i-1][j-1];
233             else
234                 dp[i][j] = 1 + min({ dp[i-1][j], dp[i][j-1], dp[i-1][j-1] });
235         }
236     }
237     cout << "Edit Distance: " << dp[n][m] << "\n";
238 }
239
240 ///////////////////////////////////////////////////
241 // 6. 0/1 Knapsack (2D DP Table)
242 ///////////////////////////////////////////////////
243 /*
244     Problem Statement:
245     Given weights and values of n items, along with a knapsack capacity W, determine the
246     maximum value
247     that can be achieved without exceeding the capacity. Each item can be included or
248     excluded (0/1 Knapsack).
249
250     DP Approach:
251     Let dp[i][w] be the maximum value that can be achieved with the first i items and
252     capacity w.
253     - Transition:
254         If weight[i-1] <= w, dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i-1]] + value[i-
255         1]).
256         Else, dp[i][w] = dp[i-1][w].
257 */
258 void solve_knapsack() {
259     cout << "\n----- 0/1 Knapsack ----- \n";
260     int n, W;
261     cout << "Enter number of items and knapsack capacity: ";
262     cin >> n >> W;

```

```

259     vector<int> weight(n), value(n);
260     cout << "Enter the weights of the items:\n";
261     for (int i = 0; i < n; i++) cin >> weight[i];
262     cout << "Enter the values of the items:\n";
263     for (int i = 0; i < n; i++) cin >> value[i];
264
265     vector<vector<int>>> dp(n+1, vector<int>(W+1, 0));
266     for (int i = 1; i <= n; i++) {
267         for (int w = 0; w <= W; w++) {
268             if (weight[i-1] <= w)
269                 dp[i][w] = max(dp[i-1][w], dp[i-1][w - weight[i-1]] + value[i-1]);
270             else
271                 dp[i][w] = dp[i-1][w];
272         }
273     }
274     cout << "Maximum value in knapsack: " << dp[n][W] << "\n";
275 }
276
277 ///////////////////////////////////////////////////////////////////
278 // 7. Matrix Chain Multiplication
279 ///////////////////////////////////////////////////////////////////
280 /*
281     Problem Statement:
282     Given a chain of matrices, determine the minimum number of multiplications needed
283     to multiply the chain. The order in which the matrices are multiplied can affect the
284     cost.
285
286     DP Approach:
287     Let dp[i][j] represent the minimum cost to multiply matrices from i to j.
288     - Transition:
289         dp[i][j] = min{ dp[i][k] + dp[k+1][j] + (dimensions cost) } for all i <= k < j.
290 */
291 void solve_matrix_chain() {
292     cout << "\n----- Matrix Chain Multiplication ----- \n";
293     int n;
294     cout << "Enter number of matrices: ";
295     cin >> n;
296     // Dimensions array of size n+1 where matrix i has dimensions p[i-1] x p[i]
297     vector<int> p(n+1);
298     cout << "Enter the dimensions (n+1 numbers): ";
299     for (int i = 0; i <= n; i++) {
300         cin >> p[i];
301     }
302     vector<vector<int>>> dp(n+1, vector<int>(n+1, 0));
303     // dp[i][i] is zero cost.
304     for (int len = 2; len <= n; len++) {
305         for (int i = 1; i <= n - len + 1; i++) {
306             int j = i + len - 1;
307             dp[i][j] = LLONG_MAX;
308             for (int k = i; k < j; k++) {
309                 dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]);
310             }
311         }
312     }

```

```

312     cout << "Minimum multiplication cost: " << dp[1][n] << "\n";
313 }
314
315 ///////////////////////////////////////////////////
316 // 8. Longest Palindromic Subsequence
317 ///////////////////////////////////////////////////
318 /*
319     Problem Statement:
320         Given a string, find the length of its longest palindromic subsequence.
321
322     DP Approach:
323         Let dp[i][j] be the length of the longest palindromic subsequence in s[i...j].
324         - If s[i] == s[j], then dp[i][j] = dp[i+1][j-1] + 2.
325         - Otherwise, dp[i][j] = max(dp[i+1][j], dp[i][j-1]).
326 */
327 void solve_longest_palindromic_subsequence() {
328     cout << "\n----- Longest Palindromic Subsequence ----- \n";
329     string s;
330     cout << "Enter the string: ";
331     cin >> s;
332     int n = s.size();
333     vector<vector<int>> dp(n, vector<int>(n, 0));
334     // Base: single characters are palindromes of length 1.
335     for (int i = 0; i < n; i++) dp[i][i] = 1;
336
337     for (int len = 2; len <= n; len++) {
338         for (int i = 0; i <= n - len; i++) {
339             int j = i + len - 1;
340             if (s[i] == s[j]) {
341                 dp[i][j] = (len == 2 ? 2 : dp[i+1][j-1] + 2);
342             } else {
343                 dp[i][j] = max(dp[i+1][j], dp[i][j-1]);
344             }
345         }
346     }
347     cout << "Length of Longest Palindromic Subsequence: " << dp[0][n-1] << "\n";
348 }
349
350 ///////////////////////////////////////////////////
351 // 9. Count Palindromic Substrings
352 ///////////////////////////////////////////////////
353 /*
354     Problem Statement:
355         Given a string, count the number of palindromic substrings in it.
356
357     DP Approach:
358         Let dp[i][j] be true if s[i...j] is a palindrome.
359         - Base: All single characters are palindromes.
360         - For substrings of length 2 and more, check the outer characters and inner substring.
361 */
362 void solve_count_palindromic_substrings() {
363     cout << "\n----- Count Palindromic Substrings ----- \n";
364     string s;
365     cout << "Enter the string: ";

```

```

366     cin >> s;
367     int n = s.size(), count = 0;
368     vector<vector<bool>> dp(n, vector<bool>(n, false));
369
370     // Base: single character palindromes
371     for (int i = 0; i < n; i++) {
372         dp[i][i] = true;
373         count++;
374     }
375
376     // Check substrings of length 2
377     for (int i = 0; i < n-1; i++) {
378         if (s[i] == s[i+1]) {
379             dp[i][i+1] = true;
380             count++;
381         }
382     }
383
384     // Check substrings of length >= 3
385     for (int len = 3; len <= n; len++) {
386         for (int i = 0; i <= n - len; i++) {
387             int j = i + len - 1;
388             if (s[i] == s[j] && dp[i+1][j-1]) {
389                 dp[i][j] = true;
390                 count++;
391             }
392         }
393     }
394     cout << "Total palindromic substrings: " << count << "\n";
395 }
396
397 ///////////////////////////////////////////////////
398 // 10. Interleaving String
399 ///////////////////////////////////////////////////
400 /*
401     Problem Statement:
402     Given strings s1, s2, and s3, determine if s3 is formed by an interleaving of s1 and
403     s2.
404
405     DP Approach:
406     Let dp[i][j] be true if s3[0...i+j-1] is an interleaving of s1[0...i-1] and s2[0...j-
407     1].
408     - Base: dp[0][0] = true.
409     - Transition:
410         dp[i][j] = (s1[i-1] == s3[i+j-1] and dp[i-1][j]) or
411         (s2[j-1] == s3[i+j-1] and dp[i][j-1]).
412 */
413 void solve_interleaving_string() {
414     cout << "\n----- Interleaving String ----- \n";
415     string s1, s2, s3;
416     cout << "Enter first string: ";
417     cin >> s1;
418     cout << "Enter second string: ";
419     cin >> s2;

```



```

418     cout << "Enter target interleaved string: ";
419     cin >> s3;
420
421     int n = s1.size(), m = s2.size();
422     if(n + m != s3.size()){
423         cout << "s3 is not an interleaving of s1 and s2 (length mismatch).\n";
424         return;
425     }
426     vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
427     dp[0][0] = true;
428     // Initialize first row
429     for (int i = 1; i <= n; i++) {
430         dp[i][0] = dp[i-1][0] && (s1[i-1] == s3[i-1]);
431     }
432     // Initialize first column
433     for (int j = 1; j <= m; j++) {
434         dp[0][j] = dp[0][j-1] && (s2[j-1] == s3[j-1]);
435     }
436     // Fill DP table
437     for (int i = 1; i <= n; i++) {
438         for (int j = 1; j <= m; j++) {
439             dp[i][j] = (dp[i-1][j] && s1[i-1] == s3[i+j-1]) ||
440                 (dp[i][j-1] && s2[j-1] == s3[i+j-1]);
441         }
442     }
443     cout << "Is s3 an interleaving of s1 and s2? " << (dp[n][m] ? "Yes" : "No") << "\n";
444 }
445
446 ///////////////////////////////////////////////////////////////////
447 // Main function with interactive menu for 2D DP problems
448 ///////////////////////////////////////////////////////////////////
449 int32_t main() {
450     ios_base::sync_with_stdio(false);
451     cin.tie(nullptr);
452     cout.tie(nullptr);
453
454     cout << "=====\n";
455     cout << "        Basic 2D Dynamic Programming (DP)        \n";
456     cout << "=====\n";
457     cout << "Select a problem to solve:\n";
458     cout << "1. Unique Paths\n";
459     cout << "2. Unique Paths II (with obstacles)\n";
460     cout << "3. Minimum Path Sum\n";
461     cout << "4. Longest Common Subsequence (LCS)\n";
462     cout << "5. Edit Distance\n";
463     cout << "6. 0/1 Knapsack\n";
464     cout << "7. Matrix Chain Multiplication\n";
465     cout << "8. Longest Palindromic Subsequence\n";
466     cout << "9. Count Palindromic Substrings\n";
467     cout << "10. Interleaving String\n";
468     cout << "11. Run All Examples\n";
469     cout << "Enter your choice: ";
470
471     int choice;

```

```
472     cin >> choice;
473     cout << "\n";
474
475     switch(choice) {
476         case 1:
477             solve_unique_paths();
478             break;
479         case 2:
480             solve_unique_paths_obstacle();
481             break;
482         case 3:
483             solve_minimum_path_sum();
484             break;
485         case 4:
486             solve_lcs();
487             break;
488         case 5:
489             solve_edit_distance();
490             break;
491         case 6:
492             solve_knapsack();
493             break;
494         case 7:
495             solve_matrix_chain();
496             break;
497         case 8:
498             solve_longest_palindromic_subsequence();
499             break;
500         case 9:
501             solve_count_palindromic_substrings();
502             break;
503         case 10:
504             solve_interleaving_string();
505             break;
506         case 11:
507             solve_unique_paths();
508             solve_unique_paths_obstacle();
509             solve_minimum_path_sum();
510             solve_lcs();
511             solve_edit_distance();
512             solve_knapsack();
513             solve_matrix_chain();
514             solve_longest_palindromic_subsequence();
515             solve_count_palindromic_substrings();
516             solve_interleaving_string();
517             break;
518         default:
519             cout << "Invalid choice. Exiting...\n";
520     }
521
522     return 0;
523 }
524
```