

Advanced_2D_DP.cpp

```

1  /**
2   *   Author: devesh95
3   *
4   *   Topic: Advanced 2D Dynamic Programming (DP) Examples
5   *
6   *   Description:
7   *   This file contains 15 advanced DP problems that use 2D DP techniques or
8   *   interval/state DP formulations. Each problem is explained in detail, with
9   *   the DP state definitions and transitions elaborated upon.
10  *
11  *   Problems Covered:
12  *       1. Maximum Sum Rectangle in a 2D Matrix
13  *       2. Longest Common Substring (Contiguous)
14  *       3. Longest Increasing Path in a Matrix
15  *       4. Regular Expression Matching (with '.' and '*')
16  *       5. Wildcard Matching (with '?' and '*')
17  *       6. Distinct Subsequences (Count ways to form t from s)
18  *       7. Palindrome Partitioning II (Minimum cuts for palindrome partitioning)
19  *       8. Egg Dropping Puzzle
20  *       9. Count Palindromic Subsequences in a String
21  *      10. Longest Common Subarray (Contiguous subsequence)
22  *      11. Optimal Game Strategy (Pick coins from ends)
23  *      12. Burst Balloons (Interval DP)
24  *      13. Longest Arithmetic Subsequence (Using DP with difference)
25  *      14. Stone Game (Optimal play for removing stones)
26  *      15. Minimum Cost to Merge Stones (Interval DP)
27  *
28  *   Compilation:
29  *       g++ -std=c++17 -O2 -Wall Advanced_2D_DP.cpp -o advanced2d_dp
30  *
31  *   Execution:
32  *       ./advanced2d_dp
33  */
34
35 #include <bits/stdc++.h>
36 using namespace std;
37
38 #define int long long
39 #define pb push_back
40 #define F first
41 #define S second
42
43 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
44 // 1. Maximum Sum Rectangle in a 2D Matrix
45 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
46 /*
47     Problem Statement:
48     Given a 2D matrix of integers, find the sub-rectangle (contiguous block)
49     with the maximum possible sum.
50
51     DP/Algorithm Approach:

```

```

52     - Fix the left and right column boundaries.
53     - For each pair, collapse the 2D problem into a 1D problem (summing rows)
54       and then use Kadane's algorithm on the temporary 1D array.
55 */
56 void solve_max_sum_rectangle() {
57     cout << "\n----- Maximum Sum Rectangle in a 2D Matrix ----- \n";
58     int rows, cols;
59     cout << "Enter number of rows and columns: ";
60     cin >> rows >> cols;
61     vector<vector<int>> matrix(rows, vector<int>(cols));
62     cout << "Enter the matrix elements:\n";
63     for (int i = 0; i < rows; i++)
64         for (int j = 0; j < cols; j++)
65             cin >> matrix[i][j];
66
67     int maxSum = LLONG_MIN;
68     // Variables to store rectangle boundaries (optional)
69     int finalLeft = 0, finalRight = 0, finalTop = 0, finalBottom = 0;
70
71     // Left boundary of the rectangle
72     for (int left = 0; left < cols; left++) {
73         vector<int> temp(rows, 0);
74         // Right boundary from left to end
75         for (int right = left; right < cols; right++) {
76             // Sum rows between left and right for each row
77             for (int i = 0; i < rows; i++)
78                 temp[i] += matrix[i][right];
79
80             // Apply Kadane's algorithm on temp[]
81             int sum = 0, localMax = LLONG_MIN;
82             int start = 0, localTop = 0, localBottom = 0;
83             for (int i = 0; i < rows; i++) {
84                 sum += temp[i];
85                 if (sum > localMax) {
86                     localMax = sum;
87                     localTop = start;
88                     localBottom = i;
89                 }
90                 if (sum < 0) {
91                     sum = 0;
92                     start = i+1;
93                 }
94             }
95             if (localMax > maxSum) {
96                 maxSum = localMax;
97                 finalLeft = left; finalRight = right;
98                 finalTop = localTop; finalBottom = localBottom;
99             }
100         }
101     }
102     cout << "Maximum rectangle sum is: " << maxSum << "\n";
103     // Optionally print boundaries.
104     // cout << "Boundaries: Top " << finalTop << ", Bottom " << finalBottom
105     //      << ", Left " << finalLeft << ", Right " << finalRight << "\n";

```

```

106 }
107
108 ///////////////////////////////////////////////////
109 // 2. Longest Common Substring (Contiguous)
110 ///////////////////////////////////////////////////
111 /*
112     Problem Statement:
113     Given two strings, find the longest contiguous substring that appears in both.
114
115     DP Approach:
116     Let dp[i][j] be the length of the longest common suffix of s1[0..i-1] and s2[0..j-1].
117     If s1[i-1] == s2[j-1], then dp[i][j] = dp[i-1][j-1] + 1; otherwise 0.
118     Track the maximum value in dp.
119 */
120 void solve_longest_common_substring() {
121     cout << "\n----- Longest Common Substring ----- \n";
122     string s1, s2;
123     cout << "Enter first string: ";
124     cin >> s1;
125     cout << "Enter second string: ";
126     cin >> s2;
127     int n = s1.size(), m = s2.size();
128     vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
129     int maxLen = 0;
130     string res = "";
131     for (int i = 1; i <= n; i++) {
132         for (int j = 1; j <= m; j++) {
133             if (s1[i-1] == s2[j-1]) {
134                 dp[i][j] = dp[i-1][j-1] + 1;
135                 if (dp[i][j] > maxLen) {
136                     maxLen = dp[i][j];
137                     res = s1.substr(i - maxLen, maxLen);
138                 }
139             } else {
140                 dp[i][j] = 0;
141             }
142         }
143     }
144     cout << "Longest common substring: \"" << res << "\" with length " << maxLen << "\n";
145 }
146
147 ///////////////////////////////////////////////////
148 // 3. Longest Increasing Path in a Matrix
149 ///////////////////////////////////////////////////
150 /*
151     Problem Statement:
152     Given a matrix, find the length of the longest strictly increasing path.
153     You can move in four directions (up, down, left, right).
154
155     DP Approach:
156     Use DFS with memoization. Let memo[i][j] store the length of the longest increasing
157     path starting at cell (i, j). Explore neighbors with greater value.
158 */
159 int dx[4] = {0, 0, 1, -1};

```

```

160 int dy[4] = {1, -1, 0, 0};
161
162 int dfs(int i, int j, vector<vector<int>>& matrix, vector<vector<int>>& memo) {
163     if (memo[i][j] != 0)
164         return memo[i][j];
165     int maxPath = 1;
166     int rows = matrix.size(), cols = matrix[0].size();
167     for (int dir = 0; dir < 4; dir++) {
168         int x = i + dx[dir], y = j + dy[dir];
169         if (x >= 0 && x < rows && y >= 0 && y < cols && matrix[x][y] > matrix[i][j]) {
170             maxPath = max(maxPath, 1 + dfs(x, y, matrix, memo));
171         }
172     }
173     memo[i][j] = maxPath;
174     return maxPath;
175 }
176
177 void solve_longest_increasing_path() {
178     cout << "\n----- Longest Increasing Path in a Matrix -----\n";
179     int rows, cols;
180     cout << "Enter number of rows and columns: ";
181     cin >> rows >> cols;
182     vector<vector<int>> matrix(rows, vector<int>(cols));
183     cout << "Enter the matrix elements:\n";
184     for (int i = 0; i < rows; i++)
185         for (int j = 0; j < cols; j++)
186             cin >> matrix[i][j];
187
188     vector<vector<int>> memo(rows, vector<int>(cols, 0));
189     int res = 0;
190     for (int i = 0; i < rows; i++)
191         for (int j = 0; j < cols; j++)
192             res = max(res, dfs(i, j, matrix, memo));
193
194     cout << "Length of longest increasing path: " << res << "\n";
195 }
196
197 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
198 // 4. Regular Expression Matching
199 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
200 /*
201     Problem Statement:
202     Implement regex matching with support for '.' and '*'.
203     '.' Matches any single character.
204     '*' Matches zero or more of the preceding element.
205
206     DP Approach:
207     Let dp[i][j] be true if s[0...i-1] matches p[0...j-1].
208     Transition considers the '*' case carefully.
209 */
210 void solve_regex_matching() {
211     cout << "\n----- Regular Expression Matching -----\n";
212     string s, p;
213     cout << "Enter the input string: ";

```

```

214     cin >> s;
215     cout << "Enter the pattern: ";
216     cin >> p;
217     int n = s.size(), m = p.size();
218     vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
219     dp[0][0] = true;
220
221     // Initialize patterns like a*, a*b*, etc.
222     for (int j = 1; j <= m; j++) {
223         if (p[j-1] == '*') {
224             dp[0][j] = dp[0][j-2];
225         }
226     }
227     for (int i = 1; i <= n; i++) {
228         for (int j = 1; j <= m; j++) {
229             if (p[j-1] == s[i-1] || p[j-1] == '.')
230                 dp[i][j] = dp[i-1][j-1];
231             else if (p[j-1] == '*') {
232                 dp[i][j] = dp[i][j-2]; // '*' represents 0 occurrence
233                 if (p[j-2] == s[i-1] || p[j-2] == '.')
234                     dp[i][j] = dp[i][j] || dp[i-1][j];
235             } else {
236                 dp[i][j] = false;
237             }
238         }
239     }
240     cout << "Does the string match the pattern? " << (dp[n][m] ? "Yes" : "No") << "\n";
241 }
242
243 ///////////////////////////////////////////////////
244 // 5. Wildcard Matching
245 ///////////////////////////////////////////////////
246 /*
247 Problem Statement:
248 Implement wildcard matching with support for '?' and '*'.
249 '?' Matches any single character.
250 '*' Matches any sequence of characters (including the empty sequence).
251
252 DP Approach:
253 Let dp[i][j] be true if s[0...i-1] matches p[0...j-1].
254 Update dp considering '*' can match zero or more characters.
255 */
256 void solve_wildcard_matching() {
257     cout << "\n----- Wildcard Matching ----- \n";
258     string s, p;
259     cout << "Enter the input string: ";
260     cin >> s;
261     cout << "Enter the wildcard pattern: ";
262     cin >> p;
263     int n = s.size(), m = p.size();
264     vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
265     dp[0][0] = true;
266     // Initialize pattern when string is empty
267     for (int j = 1; j <= m; j++) {

```

```

268         if (p[j-1] == '*')
269             dp[0][j] = dp[0][j-1];
270     }
271     for (int i = 1; i <= n; i++) {
272         for (int j = 1; j <= m; j++) {
273             if (p[j-1] == s[i-1] || p[j-1] == '?')
274                 dp[i][j] = dp[i-1][j-1];
275             else if (p[j-1] == '*')
276                 dp[i][j] = dp[i][j-1] || dp[i-1][j];
277             else
278                 dp[i][j] = false;
279         }
280     }
281     cout << "Does the string match the wildcard pattern? " << (dp[n][m] ? "Yes" : "No") <<
282     "\n";
283 }
284 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
285 // 6. Distinct Subsequences
286 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
287 /*
288     Problem Statement:
289     Given strings s and t, count the number of distinct subsequences of s that equal t.
290
291     DP Approach:
292     Let dp[i][j] be the number of distinct subsequences of s[0...i-1] that form t[0...j-1].
293     Transition:
294     If s[i-1] == t[j-1], then dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
295     Else, dp[i][j] = dp[i-1][j].
296 */
297 void solve_distinct_subsequences() {
298     cout << "\n----- Distinct Subsequences ----- \n";
299     string s, t;
300     cout << "Enter source string s: ";
301     cin >> s;
302     cout << "Enter target string t: ";
303     cin >> t;
304     int n = s.size(), m = t.size();
305     vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
306     // Base: empty target is a subsequence of any string
307     for (int i = 0; i <= n; i++)
308         dp[i][0] = 1;
309     for (int i = 1; i <= n; i++) {
310         for (int j = 1; j <= m; j++) {
311             if (s[i-1] == t[j-1])
312                 dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
313             else
314                 dp[i][j] = dp[i-1][j];
315         }
316     }
317     cout << "Number of distinct subsequences: " << dp[n][m] << "\n";
318 }
319
320 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

321 // 7. Palindrome Partitioning II (Minimum Cuts)
322 ///////////////////////////////////////////////////////////////////
323 /*
324     Problem Statement:
325     Given a string, partition it such that every substring is a palindrome.
326     Find the minimum number of cuts needed for a palindrome partitioning.
327
328     DP Approach:
329     Let dp[i] be the minimum cuts needed for substring s[0..i].
330     Precompute a palindrome table for all substrings.
331 */
332 void solve_palindrome_partitioning() {
333     cout << "\n----- Palindrome Partitioning II (Minimum Cuts) -----\n";
334     string s;
335     cout << "Enter the string: ";
336     cin >> s;
337     int n = s.size();
338     vector<vector<bool>> isPal(n, vector<bool>(n, false));
339     // Precompute palindrome table
340     for (int i = 0; i < n; i++) {
341         isPal[i][i] = true;
342     }
343     for (int len = 2; len <= n; len++) {
344         for (int i = 0; i <= n - len; i++) {
345             int j = i + len - 1;
346             if (s[i] == s[j]) {
347                 isPal[i][j] = (len == 2) ? true : isPal[i+1][j-1];
348             } else {
349                 isPal[i][j] = false;
350             }
351         }
352     }
353     vector<int> dp(n, 0);
354     for (int i = 0; i < n; i++) {
355         if (isPal[0][i]) {
356             dp[i] = 0;
357         } else {
358             dp[i] = i; // worst-case (cut before every char)
359             for (int j = 0; j < i; j++) {
360                 if (isPal[j+1][i])
361                     dp[i] = min(dp[i], dp[j] + 1);
362             }
363         }
364     }
365     cout << "Minimum cuts required: " << dp[n-1] << "\n";
366 }
367
368 ///////////////////////////////////////////////////////////////////
369 // 8. Egg Dropping Puzzle
370 ///////////////////////////////////////////////////////////////////
371 /*
372     Problem Statement:
373     Given K eggs and N floors, determine the minimum number of trials needed in the worst
374     case

```

```

374     to find the critical floor from which eggs start breaking.
375
376     DP Approach:
377     Let dp[k][n] be the minimum number of trials needed with k eggs and n floors.
378     Use the recurrence:
379     dp[k][n] = 1 + min_{1 <= x <= n}(max(dp[k-1][x-1], dp[k][n-x]))
380 */
381 void solve_egg_dropping() {
382     cout << "\n----- Egg Dropping Puzzle ----- \n";
383     int K, N;
384     cout << "Enter number of eggs and number of floors: ";
385     cin >> K >> N;
386     vector<vector<int>> dp(K+1, vector<int>(N+1, 0));
387     // Base cases
388     for (int i = 1; i <= N; i++) dp[1][i] = i; // 1 egg: trial each floor
389     for (int k = 1; k <= K; k++) dp[k][0] = 0;
390     for (int k = 1; k <= K; k++) dp[k][1] = 1;
391
392     for (int k = 2; k <= K; k++) {
393         for (int n = 2; n <= N; n++) {
394             dp[k][n] = INT_MAX;
395             // Try dropping from each floor x
396             for (int x = 1; x <= n; x++) {
397                 int res = 1 + max(dp[k-1][x-1], dp[k][n-x]);
398                 dp[k][n] = min(dp[k][n], res);
399             }
400         }
401     }
402     cout << "Minimum number of trials in worst case: " << dp[K][N] << "\n";
403 }
404
405 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
406 // 9. Count Palindromic Subsequences in a String
407 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
408 /*
409     Problem Statement:
410     Given a string, count the number of palindromic subsequences (not necessarily
411     distinct).
412
413     DP Approach:
414     Let dp[i][j] denote the count of palindromic subsequences in s[i...j].
415     Use recurrence based on matching endpoints and inclusion/exclusion.
416 */
417 void solve_count_palindromic_subsequences() {
418     cout << "\n----- Count Palindromic Subsequences ----- \n";
419     string s;
420     cout << "Enter the string: ";
421     cin >> s;
422     int n = s.size();
423     vector<vector<int>> dp(n, vector<int>(n, 0));
424
425     // Base: each single character is a palindrome
426     for (int i = 0; i < n; i++)
427         dp[i][i] = 1;

```



```

427
428     for (int len = 2; len <= n; len++) {
429         for (int i = 0; i <= n - len; i++) {
430             int j = i + len - 1;
431             if (s[i] == s[j]) {
432                 dp[i][j] = dp[i+1][j] + dp[i][j-1] + 1;
433             } else {
434                 dp[i][j] = dp[i+1][j] + dp[i][j-1] - dp[i+1][j-1];
435             }
436         }
437     }
438     cout << "Total palindromic subsequences: " << dp[0][n-1] << "\n";
439 }
440
441 ///////////////////////////////////////////////////
442 // 10. Longest Common Subarray (Contiguous)
443 ///////////////////////////////////////////////////
444 /*
445     Problem Statement:
446     Given two arrays, find the length of the longest subarray (contiguous) common to both.
447
448     DP Approach:
449     Let dp[i][j] be the length of the longest common suffix of A[0...i-1] and B[0...j-1].
450     If A[i-1] == B[j-1], then dp[i][j] = dp[i-1][j-1] + 1.
451 */
452 void solve_longest_common_subarray() {
453     cout << "\n----- Longest Common Subarray ----- \n";
454     int n, m;
455     cout << "Enter the size of first array and second array: ";
456     cin >> n >> m;
457     vector<int> A(n), B(m);
458     cout << "Enter elements of first array:\n";
459     for (int i = 0; i < n; i++) cin >> A[i];
460     cout << "Enter elements of second array:\n";
461     for (int j = 0; j < m; j++) cin >> B[j];
462
463     vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
464     int maxLen = 0;
465     for (int i = 1; i <= n; i++) {
466         for (int j = 1; j <= m; j++) {
467             if (A[i-1] == B[j-1]) {
468                 dp[i][j] = dp[i-1][j-1] + 1;
469                 maxLen = max(maxLen, dp[i][j]);
470             }
471         }
472     }
473     cout << "Length of longest common subarray: " << maxLen << "\n";
474 }
475
476 ///////////////////////////////////////////////////
477 // 11. Optimal Game Strategy (Pick Coins from Ends)
478 ///////////////////////////////////////////////////
479 /*
480     Problem Statement:

```

```

481     Given an array of coins, two players pick coins from either end.
482     Compute the maximum amount of money the first player can collect assuming optimal play.
483
484     DP Approach:
485     Let dp[i][j] be the maximum amount the current player can collect from coins i to j.
486     The recurrence:
487         dp[i][j] = max( coins[i] + min(dp[i+2][j], dp[i+1][j-1]),
488                        coins[j] + min(dp[i+1][j-1], dp[i][j-2]) )
489 */
490 void solve_optimal_game_strategy() {
491     cout << "\n----- Optimal Game Strategy ----- \n";
492     int n;
493     cout << "Enter the number of coins: ";
494     cin >> n;
495     vector<int> coins(n);
496     cout << "Enter coin values:\n";
497     for (int i = 0; i < n; i++) cin >> coins[i];
498
499     vector<vector<int>> dp(n, vector<int>(n, 0));
500     // Base cases: one coin and two coins.
501     for (int i = 0; i < n; i++)
502         dp[i][i] = coins[i];
503     for (int i = 0; i < n-1; i++)
504         dp[i][i+1] = max(coins[i], coins[i+1]);
505
506     for (int len = 3; len <= n; len++) {
507         for (int i = 0; i <= n - len; i++) {
508             int j = i + len - 1;
509             int a = (i+2 <= j) ? dp[i+2][j] : 0;
510             int b = (i+1 <= j-1) ? dp[i+1][j-1] : 0;
511             int c = (i <= j-2) ? dp[i][j-2] : 0;
512             dp[i][j] = max(coins[i] + min(a, b),
513                           coins[j] + min(b, c));
514         }
515     }
516     cout << "Maximum amount first player can collect: " << dp[0][n-1] << "\n";
517 }
518
519 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
520 // 12. Burst Balloons (Interval DP)
521 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
522 /*
523     Problem Statement:
524     Given an array of balloons, burst them in an order such that you maximize coins earned.
525     When you burst balloon i, coins = nums[left] * nums[i] * nums[right], where left and
526     right
527     are adjacent balloons.
528
529     DP Approach:
530     Let dp[i][j] be the maximum coins obtainable by bursting balloons in the open interval
531     (i, j).
532     Use interval DP to try every possible last balloon to burst.
533 */
534 void solve_burst_balloons() {

```

```

533     cout << "\n----- Burst Balloons ----- \n";
534     int n;
535     cout << "Enter number of balloons: ";
536     cin >> n;
537     vector<int> nums(n);
538     cout << "Enter the balloon numbers:\n";
539     for (int i = 0; i < n; i++) cin >> nums[i];
540
541     // Add boundaries 1 at the start and end
542     vector<int> balloons;
543     balloons.push_back(1);
544     for (int x : nums) balloons.push_back(x);
545     balloons.push_back(1);
546     int m = balloons.size();
547     vector<vector<int>> dp(m, vector<int>(m, 0));
548
549     for (int len = 2; len < m; len++) {
550         for (int i = 0; i < m - len; i++) {
551             int j = i + len;
552             for (int k = i+1; k < j; k++) {
553                 dp[i][j] = max(dp[i][j], balloons[i]*balloons[k]*balloons[j] + dp[i][k] +
dp[k][j]);
554             }
555         }
556     }
557     cout << "Maximum coins obtainable: " << dp[0][m-1] << "\n";
558 }
559
560 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
561 // 13. Longest Arithmetic Subsequence
562 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
563 /*
564     Problem Statement:
565     Given an array, find the length of the longest arithmetic subsequence.
566
567     DP Approach:
568     Let dp[i][d] be the length of the longest arithmetic subsequence ending at index i with
difference d.
569     Use a map at each index to store differences.
570 */
571 void solve_longest_arithmetic_subsequence() {
572     cout << "\n----- Longest Arithmetic Subsequence ----- \n";
573     int n;
574     cout << "Enter the number of elements: ";
575     cin >> n;
576     vector<int> arr(n);
577     cout << "Enter the elements:\n";
578     for (int i = 0; i < n; i++) cin >> arr[i];
579
580     int ans = 0;
581     vector<unordered_map<int, int>> dp(n);
582     for (int i = 0; i < n; i++) {
583         for (int j = 0; j < i; j++) {
584             int diff = arr[i] - arr[j];

```

```

585         dp[i][diff] = max(dp[i][diff], dp[j].count(diff) ? dp[j][diff] + 1 : 2);
586         ans = max(ans, dp[i][diff]);
587     }
588 }
589 cout << "Length of longest arithmetic subsequence: " << ans << "\n";
590 }
591
592 ///////////////////////////////////////////////////
593 // 14. Stone Game (Interval DP)
594 ///////////////////////////////////////////////////
595 /*
596     Problem Statement:
597     Given an array representing piles of stones, two players remove stones optimally.
598     Compute the maximum difference in score the first player can secure over the second.
599
600     DP Approach:
601     Let dp[i][j] be the maximum score difference the current player can achieve from piles
602     i to j.
603     The recurrence:
604     dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]).
605 */
606 void solve_stone_game() {
607     cout << "\n----- Stone Game ----- \n";
608     int n;
609     cout << "Enter the number of piles: ";
610     cin >> n;
611     vector<int> piles(n);
612     cout << "Enter the number of stones in each pile:\n";
613     for (int i = 0; i < n; i++) cin >> piles[i];
614
615     vector<vector<int>> dp(n, vector<int>(n, 0));
616     for (int i = 0; i < n; i++)
617         dp[i][i] = piles[i];
618     for (int len = 2; len <= n; len++) {
619         for (int i = 0; i <= n - len; i++) {
620             int j = i + len - 1;
621             dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]);
622         }
623     }
624     cout << "Maximum score difference the first player can achieve: " << dp[0][n-1] << "\n";
625 }
626
627 ///////////////////////////////////////////////////
628 // 15. Minimum Cost to Merge Stones
629 ///////////////////////////////////////////////////
630 /*
631     Problem Statement:
632     Given an array of stone weights, merge adjacent stones until one stone remains.
633     The cost of merging is the sum of the stones being merged.
634     Find the minimum total cost to merge all stones.
635
636     DP Approach:
637     Let dp[i][j] be the minimum cost to merge stones from i to j.
638     Recurrence:

```

```

638     dp[i][j] = min_{i <= k < j} { dp[i][k] + dp[k+1][j] } + sum(i..j)
639     Precompute prefix sums for efficient range sum calculation.
640 */
641 void solve_minimum_cost_merge_stones() {
642     cout << "\n----- Minimum Cost to Merge Stones ----- \n";
643     int n;
644     cout << "Enter the number of stones: ";
645     cin >> n;
646     vector<int> stones(n);
647     cout << "Enter the weights of the stones:\n";
648     for (int i = 0; i < n; i++) cin >> stones[i];
649
650     vector<int> prefix(n+1, 0);
651     for (int i = 0; i < n; i++) {
652         prefix[i+1] = prefix[i] + stones[i];
653     }
654
655     vector<vector<int>> dp(n, vector<int>(n, 0));
656     // dp[i][i] = 0, already initialized.
657     for (int len = 2; len <= n; len++) {
658         for (int i = 0; i <= n - len; i++) {
659             int j = i + len - 1;
660             dp[i][j] = LLONG_MAX;
661             for (int k = i; k < j; k++) {
662                 dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + (prefix[j+1] - prefix[i]));
663             }
664         }
665     }
666     cout << "Minimum cost to merge all stones: " << dp[0][n-1] << "\n";
667 }
668
669 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
670 // Main function with interactive menu for Advanced 2D DP Problems
671 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
672 int32_t main() {
673     ios_base::sync_with_stdio(false);
674     cin.tie(nullptr);
675     cout.tie(nullptr);
676
677     cout << "===== \n";
678     cout << "          Advanced 2D Dynamic Programming (DP)          \n";
679     cout << "===== \n";
680     cout << "Select a problem to solve:\n";
681     cout << " 1. Maximum Sum Rectangle in a 2D Matrix\n";
682     cout << " 2. Longest Common Substring\n";
683     cout << " 3. Longest Increasing Path in a Matrix\n";
684     cout << " 4. Regular Expression Matching\n";
685     cout << " 5. Wildcard Matching\n";
686     cout << " 6. Distinct Subsequences\n";
687     cout << " 7. Palindrome Partitioning II (Minimum Cuts)\n";
688     cout << " 8. Egg Dropping Puzzle\n";
689     cout << " 9. Count Palindromic Subsequences\n";
690     cout << "10. Longest Common Subarray\n";
691     cout << "11. Optimal Game Strategy (Coins from Ends)\n";

```

```
692     cout << "12. Burst Balloons\n";
693     cout << "13. Longest Arithmetic Subsequence\n";
694     cout << "14. Stone Game (Optimal Play)\n";
695     cout << "15. Minimum Cost to Merge Stones\n";
696     cout << "16. Run All Examples\n";
697     cout << "Enter your choice: ";
698
699     int choice;
700     cin >> choice;
701     cout << "\n";
702
703     switch(choice) {
704         case 1: solve_max_sum_rectangle(); break;
705         case 2: solve_longest_common_substring(); break;
706         case 3: solve_longest_increasing_path(); break;
707         case 4: solve_regex_matching(); break;
708         case 5: solve_wildcard_matching(); break;
709         case 6: solve_distinct_subsequences(); break;
710         case 7: solve_palindrome_partitioning(); break;
711         case 8: solve_egg_dropping(); break;
712         case 9: solve_count_palindromic_subsequences(); break;
713         case 10: solve_longest_common_subarray(); break;
714         case 11: solve_optimal_game_strategy(); break;
715         case 12: solve_burst_balloons(); break;
716         case 13: solve_longest_arithmetic_subsequence(); break;
717         case 14: solve_stone_game(); break;
718         case 15: solve_minimum_cost_merge_stones(); break;
719         case 16:
720             solve_max_sum_rectangle();
721             solve_longest_common_substring();
722             solve_longest_increasing_path();
723             solve_regex_matching();
724             solve_wildcard_matching();
725             solve_distinct_subsequences();
726             solve_palindrome_partitioning();
727             solve_egg_dropping();
728             solve_count_palindromic_subsequences();
729             solve_longest_common_subarray();
730             solve_optimal_game_strategy();
731             solve_burst_balloons();
732             solve_longest_arithmetic_subsequence();
733             solve_stone_game();
734             solve_minimum_cost_merge_stones();
735             break;
736         default:
737             cout << "Invalid choice. Exiting...\n";
738     }
739
740     return 0;
741 }
742
```