Min Cost Path - GeeksforGeeks

Source: https://www.geeksforgeeks.org/min-cost-path-dp-6/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Min Cost Path Last Updated : 24 Jan, 2026 Given a 2D matrix cost[][] , where each cell represents the cost of traversing through that position. We need to determine the minimum cost required to reach the bottom-right cell (m-1, n-1) starting from the top-left cell (0,0). The total cost of a path is the sum of all cell values along the path, including both the starting and ending positions. From any cell (i, j), you can move in the following three direction Right (i, j+1), Down (i+1, j) and Diagonal (i+1, j+1). Find the minimum cost path from (0,0) to (m-1, n-1), ensuring that movement constraints are followed. Example: Input: cost[][] = [[1, 2, 3], [4, 8, 2], [1, 5, 3]] Output: 8 Explanation: The path with minimum cost is highlighted in the following figure. The path is (0, 0) -> (0, 1) -> (1, 2) -> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3). Table of Content [Naive Approach] Using Recursion - O(3 ^ (m * n)) Time and O(1) Space [Better Approach 1] Using Dijkstra's Algorithm - O((m * n) * log(m * n)) Time and O(m * n) Space [Better Approach - 2] Using Top-Down DP (Memoization) - O(m*n) Time and O(m*n) Space [Better Approach - 3] - Using Bottom-Up DP (Tabulation) - O(m * n) Time and O(m * n) Space [Expected Approach] - Using Space Optimized DP - O(m * n) Time and O(n) Space [Naive Approach] Using Recursion - O(3 ^ (m * n)) Time and O(1) Space In this problem, we need to move from the starting cell (0,0) to the last cell (m-1, n-1). From every cell, we have three choices: we can move down (i+1, j), right (i, j+1), or diagonally (i+1, j+1). Whenever a problem gives multiple choices at every step, the first natural approach that comes to mind is recursion, because recursion allows us to break the problem into smaller independent subproblems. To solve this using recursion, we start from the starting cell (0,0) and at each step we try all three possible moves. Each recursive call represents taking one of the paths, and since we want the minimum cost path, we simply try all possibilities and take the minimum among them. The base case occurs when the recursion reaches the last cell (m-1, n-1) - at that point, we stop and return the cost of that cell because the destination is reached. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends // Function to return the cost of the minimum cost path int findMinCost (vector < vector < int >>& cost , int x , int y) { int m = cost . size (); int n = cost [0]. size (); // If indices are out of bounds, return a large value if (x >= m || y >= n) { return INT_MAX ; } // Base case: bottom cell if (x == m - 1 && y == n - 1) { return cost [x][y]; } // Recursively calculate minimum cost from // all possible paths int right = findMinCost (cost , x , y + 1); int down = findMinCost (cost , x + 1 , y); int diag = findMinCost (cost , x + 1 , y + 1); int best = min (right , min (down , diag)); return cost [x][y] + best ; } // function to find the minimum cost path // to reach (m - 1, n - 1) from (0, 0) int minCost (vector < vector < int >>& cost) { return findMinCost (cost , 0 , 0); } //Driver Code Starts int main () { vector < vector < int >> cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; cout << minCost (cost); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; public class GFG { //Driver Code Ends // Function to return the cost of the minimum cost path static int findMinCost (int [][] cost , int x , int y) { int m = cost . length ; int n = cost [0] . length ; // If indices are out of bounds, return a large value if (x >= m || y >= n) return Integer . MAX_VALUE ; // Base case: bottom cell if (x == m - 1 && y == n - 1) return cost [x][y] ; // Recursively calculate minimum cost from all possible paths int right = findMinCost (cost , x , y + 1); int down = findMinCost (cost , x + 1 , y); int diag = findMinCost (cost , x + 1 , y + 1); int best = Math . min (right , Math . min (down , diag)); return cost [x][y] + best ; } // Function to find the minimum cost path // to reach (m - 1, n - 1) from (0, 0) static int minCost (int [][] cost) { return findMinCost (cost , 0 , 0); } //Driver Code Starts public static void main (String [] args) { int [][] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } };

System . out . println (minCost (cost)); } } //Driver Code Ends Python #Driver Code Starts import sys #Driver Code Ends # Function to return the cost of the minimum cost path def findMinCost (cost , x , y): m = len (cost) n = len (cost [0]) # If indices are out of bounds, return a large value if x >= m or y >= n : return sys . maxsize # Base case: bottom cell if x == m - 1 and y == n - 1 : return cost [x][y] # Recursively calculate minimum cost from all possible paths right = findMinCost (cost , x , y + 1) down = findMinCost (cost , x + 1 , y) diag = findMinCost (cost , x + 1 , y + 1) best = min (right , down , diag) return cost [x][y] + best # Function to find the minimum cost path # to reach (m - 1, n - 1) from (0, 0) def minCost (cost): return findMinCost (cost , 0 , 0) #Driver Code Starts if __name__ == "__main__" : cost = [[1 , 2 , 3], [4 , 8 , 2], [1 , 5 , 3]] print (minCost (cost)) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends // Function to return the cost of the minimum cost path static int findMinCost (int [,] cost , int x , int y) { int m = cost . GetLength (0); int n = cost . GetLength (1); // If indices are out of bounds, return a large value if (x >= m || y >= n) return int . MaxValue ; // Base case: bottom cell if (x == m - 1 && y == n - 1) return cost [x , y]; // Recursively calculate minimum cost from all possible paths int right = findMinCost (cost , x , y + 1); int down = findMinCost (cost , x + 1 , y); int diag = findMinCost (cost , x + 1 , y + 1); int best = Math . Min (right , Math . Min (down , diag)); return cost [x , y] + best ; } // Function to find the minimum cost path // to reach (m - 1, n - 1) from (0, 0) static int minCost (int [,] cost) { return findMinCost (cost , 0 , 0); } //Driver Code Starts static void Main () { int [,] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; Console . WriteLine (minCost (cost)); } } //Driver Code Ends JavaScript // Function to return the cost of the minimum cost path function findMinCost (cost , x , y) { let m = cost . length ; let n = cost [0]. length ; // If indices are out of bounds, return a large value if (x >= m || y >= n) { return Number . MAX_SAFE_INTEGER ; } // Base case: bottom-right cell if (x === m - 1 && y === n - 1) { return cost [x][y]; } // Recursively calculate minimum cost for right, down and diagonal let right = findMinCost (cost , x , y + 1); let down = findMinCost (cost , x + 1 , y); let diag = findMinCost (cost , x + 1 , y + 1); // Find the minimum among the three paths let best = right ; if (down < best) best = down ; if (diag < best) best = diag ; return cost [x][y] + best ; } // Function to find the minimum cost path // to reach (m - 1, n - 1) from (0, 0) function minCost (cost) { return findMinCost (cost , 0 , 0); } //Driver Code Starts // Driver Code let cost = [[1 , 2 , 3], [4 , 8 , 2], [1 , 5 , 3]]; console . log (minCost (cost)); //Driver Code Ends Output 8 [Better Approach 1] Using Dijkstra's Algorithm - O((m * n) * log(m * n)) Time and O(m * n) Space The idea is to apply Dijskra's Algorithm to find the minimum cost path from the top-left to the bottom-right corner of the grid. Each cell is treated as a node and each move between adjacent cells has a cost. We use a min-heap to always expand the least costly path first. C++ //Driver Code Starts #include <iostream> using namespace std ; //Driver Code Ends int minCost (vector < vector < int >> & cost) { int m = cost . size (); int n = cost [0]. size (); vector < pair < int , int >> directions = {{ 1 , 0 }, { 0 , 1 }, { 1 , 1 }}; // Min-heap (priority queue) for Dijkstra's algorithm priority_queue < vector < int > , vector < vector < int >> , greater < vector < int >>> pq ; // Distance matrix to store the minimum // cost to reach each cell vector < vector < int >> dist (m , vector < int > (n , INT_MAX)); dist [0][0] = cost [0][0]; pq . push ({ cost [0][0], 0 , 0 }); while (! pq . empty ()) { vector < int > curr = pq . top (); pq . pop (); int x = curr [1]; int y = curr [2]; // If we reached the bottom-right // corner, return the cost if (x == m - 1 && y == n - 1) { return dist [x][y]; } // Explore the neighbors for (auto & dir : directions) { int newX = x + dir . first ; int newY = y + dir . second ; // Ensure the new cell is within bounds if (newX < m && newY < n) { // Relaxation step if (dist [newX][newY] > dist [x][y] + cost [newX][newY]) { dist [newX][newY] = dist [x][y] + cost [newX][newY]; pq . push ({ dist [newX][newY], newX , newY }); } } } } return dist [m - 1][n - 1]; } //Driver Code Starts int main () { vector < vector < int >> cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; cout << minCost (cost); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; import java.util.PriorityQueue ; public class GFG { //Driver Code Ends public static int minCost (int [][] cost) { int m = cost . length ; int n = cost [0] . length ; int [][] directions = {{ 1 , 0 }, { 0 , 1 }, { 1 , 1 }}; // Min-heap (priority queue) for Dijkstra's algorithm PriorityQueue < int []> pq = new PriorityQueue <> ((a , b) -> Integer . compare (a [0] , b [0])); // Distance matrix to store the minimum cost to reach each cell int [][] dist = new int [m][n] ; for (int [] row : dist) Arrays . fill (row , Integer . MAX_VALUE); dist [0][0] = cost [0][0] ; pq . add (new int [] { cost [0][0] , 0 , 0 }); while (! pq . isEmpty ()) { int [] curr = pq . poll (); int x = curr [1] ; int y = curr [2] ; // If we reached the bottom-right corner, return the cost if (x == m - 1 && y == n - 1) { return dist [x][y] ; } // Explore the neighbors for (int [] dir : directions) { int newX = x + dir [0] ; int newY = y + dir [1] ; // Ensure the new cell is within bounds if (newX < m && newY < n) { // Relaxation step if (dist [newX][newY] > dist [x][y] + cost [newX][newY]) { dist [newX][newY] = dist [x][y] + cost [newX][newY] ; pq . add (new int [] { dist [newX][newY] , newX , newY }); } } } } return dist [m - 1][n - 1] ; } //Driver Code Starts

public static void main (String [] args) { int [][] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; System . out . println (minCost (cost)); } } //Driver Code Ends Python #Driver Code Starts import heapq import sys #Driver Code Ends def minCost (cost): m = len (cost) n = len (cost [0]) directions = [(1 , 0), (0 , 1), (1 , 1)] # Min-heap (priority queue) for Dijkstra's algorithm pq = [] # Distance matrix to store the minimum cost to reach each cell dist = [[sys . maxsize] * n for _ in range (m)] dist [0][0] = cost [0][0] heapq . heappush (pq , (cost [0][0], 0 , 0)) while pq : currCost , x , y = heapq . heappop (pq) # If we reached the bottom-right corner, return the cost if x == m - 1 and y == n - 1 : return dist [x][y] # Explore neighbors for dx , dy in directions : newX = x + dx newY = y + dy # Ensure the new cell is within bounds if newX < m and newY < n : # Relaxation step if dist [newX][newY] > dist [x][y] + cost [newX][newY]: dist [newX][newY] = dist [x][y] + cost [newX][newY] heapq . heappush (pq , (dist [newX][newY], newX , newY)) return dist [m - 1][n - 1] #Driver Code Starts if __name__ == "__main__" : cost = [[1 , 2 , 3], [4 , 8 , 2], [1 , 5 , 3]] print (minCost (cost)) #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; class GFG { //Driver Code Ends public static int minCost (int [,] cost) { int m = cost . GetLength (0); int n = cost . GetLength (1); int [][] directions = new int [][] { new int [] { 1 , 0 }, new int [] { 0 , 1 }, new int [] { 1 , 1 } }; // Min-heap (priority queue) for Dijkstra's algorithm var pq = new PriorityQueue < (int , int , int), int > (); // Distance matrix to store the minimum cost to reach each cell int [,] dist = new int [m , n]; for (int i = 0 ; i < m ; i ++) { for (int j = 0 ; j < n ; j ++) dist [i , j] = int . MaxValue ; } dist [0 , 0] = cost [0 , 0]; pq . Enqueue ((cost [0 , 0], 0 , 0), cost [0 , 0]); while (pq . Count > 0) { var curr = pq . Dequeue (); int x = curr . Item2 ; int y = curr . Item3 ; // If we reached the bottom-right corner, return the cost if (x == m - 1 && y == n - 1) return dist [x , y]; // Explore neighbors foreach (var dir in directions) { int newX = x + dir [0]; int newY = y + dir [1]; // Ensure the new cell is within bounds if (newX < m && newY < n) { // Relaxation step if (dist [newX , newY] > dist [x , y] + cost [newX , newY]) { dist [newX , newY] = dist [x , y] + cost [newX , newY]; pq . Enqueue ((dist [newX , newY], newX , newY), dist [newX , newY]); } } } } return dist [m - 1 , n - 1]; } //Driver Code Starts static void Main () { int [,] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; Console . WriteLine (minCost (cost)); } } //Driver Code Ends JavaScript function minCost (cost) { let m = cost . length ; let n = cost [0]. length ; let directions = [[1 , 0], [0 , 1], [1 , 1]]; // Min-heap (priority queue) for Dijkstra's algorithm class MinHeap { constructor () { this . h = []; } push (val) { this . h . push (val); this . h . sort ((a , b)=> a [0] - b [0]); } pop () { return this . h . shift (); } size () { return this . h . length ; } } let pq = new MinHeap (); // Distance matrix to store the minimum cost to reach each cell let dist = Array . from ({ length : m }, () => Array (n). fill (Infinity)); dist [0][0] = cost [0][0]; pq . push ([cost [0][0], 0 , 0]); while (pq . size ()) { let [currCost , x , y] = pq . pop (); // If we reached the bottom-right corner, return the cost if (x === m - 1 && y === n - 1) return dist [x][y]; // Explore neighbors for (let [dx , dy] of directions) { let newX = x + dx ; let newY = y + dy ; // Ensure the new cell is within bounds if (newX < m && newY < n) { // Relaxation step if (dist [newX][newY] > dist [x][y] + cost [newX][newY]) { dist [newX][newY] = dist [x][y] + cost [newX][newY]; pq . push ([dist [newX][newY], newX , newY]); } } } } return dist [m - 1][n - 1]; } //Driver Code Starts //Driver Code let cost = [[1 , 2 , 3], [4 , 8 , 2], [1 , 5 , 3]]; console . log (minCost (cost)); //Driver Code Ends Output 8 [Better Approach - 2] Using Top-Down DP (Memoization) - O(m*n) Time and O(m*n) Space If we look at the recursion tree, we notice that in recursion approach, many subproblems are solved again and again, which increases the time complexity. To handle this, we use a DP (memoization) approach. The idea is to store previously computed results in a DP table. We create a 2D DP table of size (m × n) because there are two parameters that change in the recursive approach. After computing a subproblem, we store the result in the table. Later, when we encounter the same subproblem again, we check the table first. If the result is already computed, we return it instead of solving it again. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends // Function to return the cost of the minimum cost path int findMinCost (vector < vector < int >>& cost , int x , int y , vector < vector < int >> & dp) { int m = cost . size (); int n = cost [0]. size (); // If indices are out of bounds, return a large value if (x >= m || y >= n) { return INT_MAX ; } // Base case: bottom cell if (x == m - 1 && y == n - 1) { return cost [x][y]; } // Check if the result is already computed if (dp [x][y] != -1) { return dp [x][y]; } int right = findMinCost (cost , x , y + 1 , dp); int down = findMinCost (cost , x + 1 , y , dp); int diag = findMinCost (cost , x + 1 , y + 1 , dp); int best = min (right , min (down , diag)); return dp [x][y] = cost [x][y] + best ; } // function to find the minimum cost path // to reach (m - 1, n - 1) from (0, 0) int minCost (vector < vector < int >>& cost) { int m = cost . size (); int n = cost [0]. size (); // create 2d array to store the minimum cost path vector < vector < int >> dp (m , vector < int > (n , -1)); return findMinCost (cost , 0 , 0 , dp); } //Driver Code Starts int main () { vector < vector < int >> cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; cout << minCost (cost); return 0 ; } //Driver Code Ends

Java //Driver Code Starts import java.util.Arrays ; public class GFG { //Driver Code Ends // Function to return the cost of the minimum cost path static int findMinCost (int [][] cost , int x , int y , int [][] dp) { int m = cost . length ; int n = cost [0] . length ; // If indices are out of bounds, return a large value if (x >= m || y >= n) return Integer . MAX_VALUE ; // Base case: bottom cell if (x == m - 1 && y == n - 1) return cost [x][y] ; // Check if the result is already computed if (dp [x][y] != - 1) return dp [x][y] ; int right = findMinCost (cost , x , y + 1 , dp); int down = findMinCost (cost , x + 1 , y , dp); int diag = findMinCost (cost , x + 1 , y + 1 , dp); int best = Math . min (right , Math . min (down , diag)); return dp [x][y] = cost [x][y] + best ; } // function to find the minimum cost path // to reach (m - 1, n - 1) from (0, 0) static int minCost (int [][] cost) { int m = cost . length ; int n = cost [0] . length ; // create 2d array to store the minimum cost path int [][] dp = new int [m][n] ; for (int [] row : dp) Arrays . fill (row , - 1); return findMinCost (cost , 0 , 0 , dp); } //Driver Code Starts public static void main (String [] args) { int [][] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; System . out . println (minCost (cost)); } } //Driver Code Ends Python #Driver Code Starts import sys #Driver Code Ends # Function to return the cost of the minimum cost path def findMinCost (cost , x , y , dp): m = len (cost) n = len (cost [0]) # If indices are out of bounds, return a large value if x >= m or y >= n : return sys . maxsize # Base case: bottom cell if x == m - 1 and y == n - 1 : return cost [x][y] # Check if the result is already computed if dp [x][y] != - 1 : return dp [x][y] right = findMinCost (cost , x , y + 1 , dp) down = findMinCost (cost , x + 1 , y , dp) diag = findMinCost (cost , x + 1 , y + 1 , dp) best = min (right , down , diag) dp [x][y] = cost [x][y] + best return dp [x][y] # function to find the minimum cost path # to reach (m - 1, n - 1) from (0, 0) def minCost (cost): m = len (cost) n = len (cost [0]) # create 2d array to store the minimum cost path dp = [[- 1] * n for _ in range (m)] return findMinCost (cost , 0 , 0 , dp) #Driver Code Starts if __name__ == "__main__" : cost = [[1 , 2 , 3], [4 , 8 , 2], [1 , 5 , 3]] print (minCost (cost)) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends // Function to return the cost of the minimum cost path static int findMinCost (int [,] cost , int x , int y , int [,] dp) { int m = cost . GetLength (0); int n = cost . GetLength (1); // If indices are out of bounds, return a large value if (x >= m || y >= n) return int . MaxValue ; // Base case: bottom cell if (x == m - 1 && y == n - 1) return cost [x , y]; // Check if the result is already computed if (dp [x , y] != - 1) return dp [x , y]; int right = findMinCost (cost , x , y + 1 , dp); int down = findMinCost (cost , x + 1 , y , dp); int diag = findMinCost (cost , x + 1 , y + 1 , dp); int best = Math . Min (right , Math . Min (down , diag)); return dp [x , y] = cost [x , y] + best ; } // function to find the minimum cost path // to reach (m - 1, n - 1) from (0, 0) static int minCost (int [,] cost) { int m = cost . GetLength (0); int n = cost . GetLength (1); // create 2d array to store the minimum cost path int [,] dp = new int [m , n]; for (int i = 0 ; i < m ; i ++) for (int j = 0 ; j < n ; j ++) dp [i , j] = - 1 ; return findMinCost (cost , 0 , 0 , dp); } //Driver Code Starts static void Main () { int [,] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; Console . WriteLine (minCost (cost)); } } //Driver Code Ends JavaScript // Function to return the cost of the minimum cost path function findMinCost (cost , x , y , dp) { const m = cost . length ; const n = cost [0]. length ; // If indices are out of bounds, return a large value if (x >= m || y >= n) return Number . MAX_SAFE_INTEGER ; // Base case: bottom cell if (x === m - 1 && y === n - 1) return cost [x][y]; // Check if the result is already computed if (dp [x][y] !== - 1) return dp [x][y]; const right = findMinCost (cost , x , y + 1 , dp); const down = findMinCost (cost , x + 1 , y , dp); const diag = findMinCost (cost , x + 1 , y + 1 , dp); const best = Math . min (right , down , diag); dp [x][y] = cost [x][y] + best ; return dp [x][y]; } // function to find the minimum cost path // to reach (m - 1, n - 1) from (0, 0) function minCost (cost) { const m = cost . length ; const n = cost [0]. length ; // create 2d array to store the minimum cost path const dp = Array . from ({ length : m }, () => Array (n). fill (- 1)); return findMinCost (cost , 0 , 0 , dp); } //Driver Code Starts // Driver Code const cost = [[1 , 2 , 3], [4 , 8 , 2], [1 , 5 , 3]]; console . log (minCost (cost)); //Driver Code Ends Output 8 [Better Approach - 3] - Using Bottom-Up DP (Tabulation) - O(m * n) Time and O(m * n) Space In the previous approach, we used recursion along with memoization. Even though memoization helps us avoid recomputing the same subproblems, the recursive structure still makes the solution slower and harder to manage for large inputs. To make the solution more efficient, we switch to a bottom-up dynamic programming method, where we build the answer iteratively. The idea is simple: we directly fill a DP table step by step. First, we set the base case: dp[0][0] = cost[0][0], because the starting cell's minimum cost is just its own value. First row: Since movement is only from the left, dp[0][j] = dp[0][j - 1] + cost[0][j] (for j > 0) First column: Since movement is only from above, dp[i][0] = dp[i - 1][0] + cost[i][0] (for i > 0) Once these base cases are set, the remaining table can be filled using the main DP relation. For every cell (i, j), we consider the minimum cost among the three possible ways to reach it and add the current cell's cost to that minimum. By filling the table in this order, we ensure that all required subproblems are already solved when we need them. C++ //Driver Code Starts #include

<iostream> using namespace std ; //Driver Code Ends int minCost (vector < vector < int >>& cost) { int m = cost . size (); int n = cost [0]. size (); vector < vector < int >> dp (m , vector < int > (n , 0)); // Initialize the base cell dp [0][0] = cost [0][0]; // Fill the first row for (int j = 1 ; j < n ; j ++) { dp [0][j] = dp [0][j - 1] + cost [0][j]; } // Fill the first column for (int i = 1 ; i < m ; i ++) { dp [i][0] = dp [i - 1][0] + cost [i][0]; } // Fill the rest of the dp table for (int i = 1 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { dp [i][j] = cost [i][j] + min ({ dp [i - 1][j], dp [i][j - 1], dp [i - 1][j - 1]}); } } return dp [m - 1][n - 1]; } //Driver Code Starts int main () { vector < vector < int >> cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; cout << minCost (cost); return 0 ; } //Driver Code Ends Java //Driver Code Starts class GfG { //Driver Code Ends static int minCost (int [][] cost) { int m = cost . length ; int n = cost [0] . length ; int [][] dp = new int [m][n] ; // Initialize the base cell dp [0][0] = cost [0][0] ; // Fill the first row for (int j = 1 ; j < n ; j ++) { dp [0][j] = dp [0][j - 1] + cost [0][j] ; } // Fill the first column for (int i = 1 ; i < m ; i ++) { dp [i][0] = dp [i - 1][0] + cost [i][0] ; } // Fill the rest of the dp table for (int i = 1 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { dp [i][j] = cost [i][j] + Math . min (dp [i - 1][j] , Math . min (dp [i][j - 1] , dp [i - 1][j - 1])); } } return dp [m - 1][n - 1] ; } //Driver Code Starts public static void main (String [] args) { int [][] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; System . out . println (minCost (cost)); } } //Driver Code Ends Python def mincost (cost): m = len (cost) n = len (cost [0]) dp = [[0] * n for _ in range (m)] # Initialize the base cell dp [0][0] = cost [0][0] # Fill the first row for j in range (1 , n): dp [0][j] = dp [0][j - 1] + cost [0][j] # Fill the first column for i in range (1 , m): dp [i][0] = dp [i - 1][0] + cost [i][0] # Fill the rest of the dp table for i in range (1 , m): for j in range (1 , n): dp [i][j] = cost [i][j] \ + min (dp [i - 1][j], \ dp [i][j - 1], dp [i - 1][j - 1]) # Minimum cost to reach the # bottom-right cell return dp [m - 1][n - 1] // Driver Code Starts if __name__ == "__main__" : cost = [[1 , 2 , 3], [4 , 8 , 2], [1 , 5 , 3]] print (mincost (cost)) // Driver Code Ends C# //Driver Code Starts using System ; class GfG { //Driver Code Ends static int MinCost (int [,] cost) { int m = cost . GetLength (0); int n = cost . GetLength (1); int [,] dp = new int [m , n]; // Base cell dp [0 , 0] = cost [0 , 0]; // Fill first row for (int j = 1 ; j < n ; j ++) { dp [0 , j] = dp [0 , j - 1] + cost [0 , j]; } // Fill first column for (int i = 1 ; i < m ; i ++) { dp [i , 0] = dp [i - 1 , 0] + cost [i , 0]; } // Fill the rest of the table for (int i = 1 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { dp [i , j] = cost [i , j] + Math . Min (dp [i - 1 , j], Math . Min (dp [i , j - 1], dp [i - 1 , j - 1])); } } return dp [m - 1 , n - 1]; } //Driver Code Starts static void Main () { int [,] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; Console . WriteLine (MinCost (cost)); } } //Driver Code Ends JavaScript function minCost (cost) { const m = cost . length ; const n = cost [0]. length ; const dp = Array . from ({ length : m }, () => Array (n). fill (0)); // Initialize the base cell dp [0][0] = cost [0][0]; // Fill the first row for (let j = 1 ; j < n ; j ++) { dp [0][j] = dp [0][j - 1] + cost [0][j]; } // Fill the first column for (let i = 1 ; i < m ; i ++) { dp [i][0] = dp [i - 1][0] + cost [i][0]; } // Fill the rest of the dp table for (let i = 1 ; i < m ; i ++) { for (let j = 1 ; j < n ; j ++) { dp [i][j] = cost [i][j] + Math . min (dp [i - 1][j], dp [i][j - 1], dp [i - 1][j - 1]); } } // Minimum cost to reach the bottom-right cell return dp [m - 1][n - 1]; } //Driver Code Starts //Driver Code const cost = [[1 , 2 , 3], [4 , 8 , 2], [1 , 5 , 3]]; console . log (minCost (cost)); //Driver Code Ends Output 8 [Expected Approach] - Using Space Optimized DP - O(m * n) Time and O(n) Space In the previous approach, we used a 2D dp table to store the minimum cost at each cell. However, we can optimize the space complexity by observing that for calculating the current state, we only need the values from the previous row. Therefore, there is no need to store the entire dp table, and we can optimize the space to O(n) by only keeping track of the current and previous rows. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int minCost (vector < vector < int >> & cost) { int m = cost . size (); int n = cost [0]. size (); vector < int > dp (n , 0); dp [0] = cost [0][0]; // Fill the first row for (int j = 1 ; j < n ; j ++) { dp [j] = dp [j - 1] + cost [0][j]; } // Fill the rest of the rows for (int i = 1 ; i < m ; i ++) { int prev = dp [0]; // Update the first column (only depends on // the previous row) dp [0] = dp [0] + cost [i][0]; for (int j = 1 ; j < n ; j ++) { int temp = dp [j]; // Update dp[j] using the minimum of the // top, left, and diagonal cells dp [j] = cost [i][j] + min ({ dp [j], dp [j - 1], prev }); prev = temp ; } } // The last cell contains the // minimum cost path return dp [n - 1]; } //Driver Code Starts int main () { vector < vector < int >> cost = {{ 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 }}; cout << minCost (cost) << endl ; return 0 ; } //Driver Code Ends Java //Driver Code Starts class GFG { //Driver Code Ends static int minCost (int [][] cost) { int m = cost . length ; int n = cost [0] . length ; int [] dp = new int [n] ; dp [0] = cost [0][0] ; // Fill the first row for (int j = 1 ; j < n ; j ++) { dp [j] = dp [j - 1] + cost [0][j] ; } // Fill the rest of the rows for (int i = 1 ; i < m ; i ++) { int prev = dp [0] ; // Update the first column (only depends on // the previous row) dp [0] = dp [0] + cost [i][0] ; for (int j = 1 ; j < n ; j ++) { int temp = dp [j] ; // Update dp[j] using the minimum of the // top, left, and diagonal cells dp [j] = cost [i][j] + Math . min (dp [j] , Math . min (dp [j - 1] , prev)); prev = temp ; } } // The last cell contains the // minimum cost path return dp [n - 1] ; } //Driver Code Starts public static

void main (String [] args) { int [][] cost = {{ 1 , 2 , 3 },{ 4 , 8 , 2 },{ 1 , 5 , 3 }}; System . out . println (minCost (cost)); } } //Driver Code Ends Python def minCost (cost): m = len (cost) n = len (cost [0]) dp = [0] * n dp [0] = cost [0][0] # Fill the first row for j in range (1 , n): dp [j] = dp [j - 1] + cost [0][j] # Fill the rest of the rows for i in range (1 , m): prev = dp [0] # Update the first column (only depends on # the previous row) dp [0] = dp [0] + cost [i][0] for j in range (1 , n): temp = dp [j] # Update dp[j] using the minimum of the # top, left, and diagonal cells dp [j] = cost [i][j] + min (dp [j], dp [j - 1], prev) prev = temp # The last cell contains the # minimum cost path return dp [n - 1] #Driver Code Starts if __name__ == "__main__" : cost = [[1 , 2 , 3],[4 , 8 , 2],[1 , 5 , 3]] print (minCost (cost)) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int MinCost (int [,] cost) { int m = cost . GetLength (0); int n = cost . GetLength (1); int [] dp = new int [n]; dp [0] = cost [0 , 0]; // Fill the first row for (int j = 1 ; j < n ; j ++) { dp [j] = dp [j - 1] + cost [0 , j]; } // Fill the rest of the rows for (int i = 1 ; i < m ; i ++) { int prev = dp [0]; // Update the first column (only depends on // the previous row) dp [0] = dp [0] + cost [i , 0]; for (int j = 1 ; j < n ; j ++) { int temp = dp [j]; // Update dp[j] using the minimum of the // top, left, and diagonal cells dp [j] = cost [i , j] + Math . Min (dp [j], Math . Min (dp [j - 1], prev)); prev = temp ; } } // The last cell contains the // minimum cost path return dp [n - 1]; } //Driver Code Starts static void Main () { int [,] cost = { { 1 , 2 , 3 }, { 4 , 8 , 2 }, { 1 , 5 , 3 } }; Console . WriteLine (MinCost (cost)); } } //Driver Code Ends JavaScript function minCost (cost) { let m = cost . length ; let n = cost [0]. length ; let dp = new Array (n). fill (0); dp [0] = cost [0][0]; // Fill the first row for (let j = 1 ; j < n ; j ++) { dp [j] = dp [j - 1] + cost [0][j]; } // Fill the rest of the rows for (let i = 1 ; i < m ; i ++) { let prev = dp [0]; // Update the first column (only depends on // the previous row) dp [0] = dp [0] + cost [i][0]; for (let j = 1 ; j < n ; j ++) { let temp = dp [j]; // Update dp[j] using the minimum of the // top, left, and diagonal cells dp [j] = cost [i][j] + Math . min (dp [j], dp [j - 1], prev); prev = temp ; } } // The last cell contains the // minimum cost path return dp [n - 1]; } //Driver Code Starts //Driver Code let cost = [[1 , 2 , 3],[4 , 8 , 2],[1 , 5 , 3]]; console . log (minCost (cost)); //Driver Code Ends Output 8 Comment Article Tags: Article Tags: Dynamic Programming Mathematical Matrix DSA Amazon Samsung MakeMyTrip + 3 More