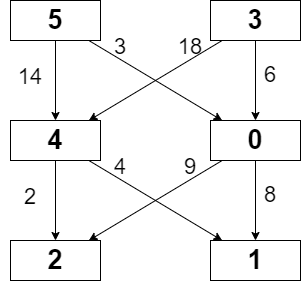
<https://leetcode.com/problems/minimum-path-cost-in-a-grid/description/>

You are given a **0-indexed** m x n integer matrix grid consisting of **distinct** integers from 0 to m \* n - 1. You can move in this matrix from a cell to any other cell in the **next** row. That is, if you are in cell (x, y) such that x < m - 1, you can move to any of the cells (x + 1, 0), (x + 1, 1), ..., (x + 1, n - 1). **Note** that it is not possible to move from cells in the last row.

Each possible move has a cost given by a **0-indexed** 2D array moveCost of size (m \* n) x n, where moveCost[i][j] is the cost of moving from a cell with value i to a cell in column j of the next row. The cost of moving from cells in the last row of grid can be ignored.

The cost of a path in grid is the **sum** of all values of cells visited plus the **sum** of costs of all the moves made. Return *the****minimum****cost of a path that starts from any cell in the****first****row and ends at any cell in the****last****row.*

**Example 1:**



**Input:** grid = [[5,3],[4,0],[2,1]], moveCost = [[9,8],[1,5],[10,12],[18,6],[2,4],[14,3]]

**Output:** 17

**Explanation:** The path with the minimum possible cost is the path 5 -> 0 -> 1.

- The sum of the values of cells visited is 5 + 0 + 1 = 6.

- The cost of moving from 5 to 0 is 3.

- The cost of moving from 0 to 1 is 8.

So the total cost of the path is 6 + 3 + 8 = 17.

**Example 2:**

**Input:** grid = [[5,1,2],[4,0,3]], moveCost = [[12,10,15],[20,23,8],[21,7,1],[8,1,13],[9,10,25],[5,3,2]]

**Output:** 6

**Explanation:** The path with the minimum possible cost is the path 2 -> 3.

- The sum of the values of cells visited is 2 + 3 = 5.

- The cost of moving from 2 to 3 is 1.

So the total cost of this path is 5 + 1 = 6.

**Constraints:**

m == grid.length

n == grid[i].length

2 <= m, n <= 50

grid consists of distinct integers from 0 to m \* n - 1.

moveCost.length == m \* n

moveCost[i].length == n

1 <= moveCost[i][j] <= 100

**Attempt 1: 2025-06-15**

**Solution 1: Native DFS (60 min, TLE 12/34)**

**Style 1: void return**

class Solution {

    int minCost = Integer.MAX\_VALUE;

    public int minPathCost(int[][] grid, int[][] moveCost) {

        int n = grid[0].length;

        for(int j = 0; j < n; j++) {

            helper(grid, moveCost, 0, j, grid[0][j]);

        }

        return minCost;

    }

    private void helper(int[][] grid, int[][] moveCost, int row, int col, int curCost) {

        // Base case: reached last row

        if(row == grid.length - 1) {

            minCost = Math.min(minCost, curCost);

            return;

        }

        // Explore all possible moves to next row

        for(int nextCol = 0; nextCol < grid[0].length; nextCol++) {

            int move = moveCost[grid[row][col]][nextCol];

            int nextCell = grid[row + 1][nextCol];

            helper(grid, moveCost, row + 1, nextCol, curCost + move + nextCell);

        }

    }

}

Time Complexity: O(n^m), where n is columns and m is maximum depth

Space Complexity: O(m), where m is maximum depth of recursion is m (number of rows)

**Style 2: int return**

class Solution {

    public int minPathCost(int[][] grid, int[][] moveCost) {

        int minCost = Integer.MAX\_VALUE;

        int n = grid[0].length;

        for(int j = 0; j < n; j++) {

            minCost = Math.min(minCost, helper(grid, moveCost, 0, j));

        }

        return minCost;

    }

    private int helper(int[][] grid, int[][] moveCost, int row, int col) {

        // Base case: reached last row

        if(row == grid.length - 1) {

            return grid[row][col];

        }

        // Explore all possible moves to next row

        int minPath = Integer.MAX\_VALUE;

        for(int nextCol = 0; nextCol < grid[0].length; nextCol++) {

            int move = moveCost[grid[row][col]][nextCol];

            int curCell = grid[row][col];

            int curPath = move + curCell + helper(grid, moveCost, row + 1, nextCol);

            minPath = Math.min(minPath, curPath);

        }

        return minPath;

    }

}

Time Complexity: O(n^m), where n is columns and m is maximum depth

Space Complexity: O(m), where m is maximum depth of recursion is m (number of rows)

**Solution 2: Memoization (10 min)**

**Style 1: void return memo (TLE 26/34)**

class Solution {

    private int minCost = Integer.MAX\_VALUE;

    private Integer[][] memo; // memo[row][col] stores min cost from (row,col) to bottom

    public int minPathCost(int[][] grid, int[][] moveCost) {

        int m = grid.length;

        int n = grid[0].length;

        memo = new Integer[m][n];

        // Try starting from each cell in the first row

        for (int j = 0; j < n; j++) {

            helper(grid, moveCost, 0, j, grid[0][j]);

        }

        return minCost;

    }

    private void helper(int[][] grid, int[][] moveCost, int row, int col, int curCost) {

        // Base case: reached last row

        if (row == grid.length - 1) {

            minCost = Math.min(minCost, curCost);

            memo[row][col] = grid[row][col]; // Store base case

            return;

        }

        // If we've already computed a path from this cell that's worse than current min,

        // we can prune this branch

        if (memo[row][col] != null && curCost >= minCost) {

            return;

        }

        // If we found a better path to this cell, update memo

        if (memo[row][col] == null || curCost < memo[row][col]) {

            memo[row][col] = curCost;

            // Explore all possible moves to next row

            for (int nextCol = 0; nextCol < grid[0].length; nextCol++) {

                int move = moveCost[grid[row][col]][nextCol];

                int nextCell = grid[row + 1][nextCol];

                helper(grid, moveCost, row + 1, nextCol, curCost + move + nextCell);

            }

        }

    }

}

**Style 2: int return memo**

class Solution {

    public int minPathCost(int[][] grid, int[][] moveCost) {

        int minCost = Integer.MAX\_VALUE;

        int m = grid.length;

        int n = grid[0].length;

        Integer[][] memo = new Integer[m][n];

        for(int j = 0; j < n; j++) {

            minCost = Math.min(minCost, helper(grid, moveCost, 0, j, memo));

        }

        return minCost;

    }

    private int helper(int[][] grid, int[][] moveCost, int row, int col, Integer[][] memo) {

        // Base case: reached last row

        if(row == grid.length - 1) {

            return grid[row][col];

        }

        if(memo[row][col] != null) {

            return memo[row][col];

        }

        // Explore all possible moves to next row

        int minPath = Integer.MAX\_VALUE;

        for(int nextCol = 0; nextCol < grid[0].length; nextCol++) {

            int move = moveCost[grid[row][col]][nextCol];

            int curCell = grid[row][col];

            int curPath = move + curCell + helper(grid, moveCost, row + 1, nextCol, memo);

            minPath = Math.min(minPath, curPath);

        }

        return memo[row][col] = minPath;

    }

}

Time Complexity: O(m \* n^2), where n is columns and m is maximum depth

Space Complexity: O(m \* n)

**Solution 3: 2D DP (30 min)**

**Based on int return Native DFS**

class Solution {

    public int minPathCost(int[][] grid, int[][] moveCost) {

        int m = grid.length;

        int n = grid[0].length;

        int[][] dp = new int[m][n];

        // Initialize last row (base case)

        for (int j = 0; j < n; j++) {

            dp[m-1][j] = grid[m-1][j];

        }

        // Build DP table from bottom up (row m-2 to 0)

        // Processes rows from bottom to top (matching DFS call stack)

        for (int i = m-2; i >= 0; i--) {

            // For each row, processes all columns (like DFS explores all moves)

            for (int j = 0; j < n; j++) {

                int minPath = Integer.MAX\_VALUE;

                // Explore all possible moves to next row

                for (int nextCol = 0; nextCol < n; nextCol++) {

                    int move = moveCost[grid[i][j]][nextCol];

                    int curCell = grid[i][j];

                    int curPath = move + curCell + dp[i+1][nextCol];

                    minPath = Math.min(minPath, curPath);

                }

                dp[i][j] = minPath;

            }

        }

        // Find minimum in first row

        int minCost = Integer.MAX\_VALUE;

        for (int j = 0; j < n; j++) {

            minCost = Math.min(minCost, dp[0][j]);

        }

        return minCost;

    }

}

Time Complexity: O(m \* n^2), where n is columns and m is maximum depth

Space Complexity: O(m \* n)

**Solution 4: 1D DP (30 min)**

**Refer to Deepseek**

**Native DFS Solution & Memoization Solution**

Here's a native Depth-First Search (DFS) solution for the Minimum Path Cost problem. While this approach is less efficient than the dynamic programming solution for larger grids, it provides a clear recursive implementation that directly models the problem's requirements.

**Approach**

The DFS solution explores all possible paths from the top row to the bottom row by:

Starting at each cell in the first row

Moving to any cell in the next row (since we can move between columns)

Keeping track of the current path cost

Returning the minimum cost found

**Solution Code**

class Solution {

private int minCost = Integer.MAX\_VALUE;

public int minPathCost(int[][] grid, int[][] moveCost) {

int n = grid[0].length;

// Try starting from each cell in the first row

for (int j = 0; j < n; j++) {

dfs(grid, moveCost, 0, j, grid[0][j]);

}

return minCost;

}

private void dfs(int[][] grid, int[][] moveCost, int row, int col, int currentCost) {

// Base case: reached last row

if (row == grid.length - 1) {

minCost = Math.min(minCost, currentCost);

return;

}

// Explore all possible moves to next row

for (int nextCol = 0; nextCol < grid[0].length; nextCol++) {

int move = moveCost[grid[row][col]][nextCol];

int nextCell = grid[row + 1][nextCol];

dfs(grid, moveCost, row + 1, nextCol, currentCost + move + nextCell);

}

}

}

**Explanation**

**Initialization**:

We initialize minCost to track the minimum path cost found

We start DFS from each cell in the first row

**Base Case**:

When we reach the last row, we update minCost if the current path cost is lower

**Recursive Exploration**:

From each cell, we recursively explore all possible moves to the next row

For each move, we calculate:

The move cost from current column to next column

The value of the next cell

Add these to the current path cost

**Result**:

After exploring all paths, minCost will contain the minimum path cost

**Optimization with Memoization**

The naive DFS solution has exponential time complexity. We can optimize it using memoization:

class Solution {

private Integer[][] memo;

public int minPathCost(int[][] grid, int[][] moveCost) {

int m = grid.length;

int n = grid[0].length;

memo = new Integer[m][n];

int minCost = Integer.MAX\_VALUE;

for (int j = 0; j < n; j++) {

minCost = Math.min(minCost, dfs(grid, moveCost, 0, j));

}

return minCost;

}

private int dfs(int[][] grid, int[][] moveCost, int row, int col) {

if (row == grid.length - 1) {

return grid[row][col];

}

if (memo[row][col] != null) {

return memo[row][col];

}

int minCost = Integer.MAX\_VALUE;

for (int nextCol = 0; nextCol < grid[0].length; nextCol++) {

int cost = grid[row][col] + moveCost[grid[row][col]][nextCol]

+ dfs(grid, moveCost, row + 1, nextCol);

minCost = Math.min(minCost, cost);

}

memo[row][col] = minCost;

return minCost;

}

}

The memoized version has O(m\*n^2) time complexity, matching the DP solution's efficiency while maintaining the recursive structure. Each state (row, col) is computed only once.

**Integer-Returning DFS Solution & Memoization Solution**

Here's a DFS solution that returns the minimum path cost as an integer instead of using a void return type with a class variable:

class Solution {

public int minPathCost(int[][] grid, int[][] moveCost) {

int minCost = Integer.MAX\_VALUE;

int n = grid[0].length;

// Try starting from each cell in the first row

for (int j = 0; j < n; j++) {

int currentCost = dfs(grid, moveCost, 0, j);

minCost = Math.min(minCost, currentCost);

}

return minCost;

}

private int dfs(int[][] grid, int[][] moveCost, int row, int col) {

// Base case: reached last row - return cell value

if (row == grid.length - 1) {

return grid[row][col];

}

int minPath = Integer.MAX\_VALUE;

// Explore all possible moves to next row

for (int nextCol = 0; nextCol < grid[0].length; nextCol++) {

// Current cell value + move cost + recursive call for next cell

int currentPath = grid[row][col]

+ moveCost[grid[row][col]][nextCol]

+ dfs(grid, moveCost, row + 1, nextCol);

minPath = Math.min(minPath, currentPath);

}

return minPath;

}

}

**Key Features:**

**Integer Return Type**:

The DFS function now returns the minimum path cost from current cell to bottom

No need for a class-level minCost variable

**Base Case**:

Returns the cell value when reaching the last row

This serves as the foundation for building up path costs

**Recursive Calculation**:

For each possible move to next row:

Adds current cell value

Adds move cost to next column

Adds recursive result from next cell

Keeps track of minimum path found

**Initialization**:

Starts DFS from each cell in first row

Takes minimum of all starting positions

**Memoization Version:**

class Solution {

private Integer[][] memo;

public int minPathCost(int[][] grid, int[][] moveCost) {

int minCost = Integer.MAX\_VALUE;

int m = grid.length;

int n = grid[0].length;

memo = new Integer[m][n];

for (int j = 0; j < n; j++) {

minCost = Math.min(minCost, dfs(grid, moveCost, 0, j));

}

return minCost;

}

private int dfs(int[][] grid, int[][] moveCost, int row, int col) {

if (row == grid.length - 1) {

return grid[row][col];

}

if (memo[row][col] != null) {

return memo[row][col];

}

int minPath = Integer.MAX\_VALUE;

for (int nextCol = 0; nextCol < grid[0].length; nextCol++) {

int currentPath = grid[row][col]

+ moveCost[grid[row][col]][nextCol]

+ dfs(grid, moveCost, row + 1, nextCol);

minPath = Math.min(minPath, currentPath);

}

memo[row][col] = minPath;

return minPath;

}

}

The memoized version avoids recomputing the same states, improving efficiency from exponential to O(m\*n^2) time complexity, where m is the number of rows and n is the number of columns.

**Time and Space Complexity Analysis for Native DFS Solution and Memozation Solution**

**Time Complexity**

**Branching Factor**:

At every cell (i, j), the DFS branches to **all n columns** in the next row.

This means **each node in the recursion tree has n children**.

**Depth of Recursion**:

The recursion goes from row 0 to row m-1, so the **maximum depth is m**.

**Total Recursive Calls**:

The recursion tree has a branching factor of n and depth m, leading to **O(n^m)** total calls in the worst case.

For a grid with m = 10 and n = 10, this could mean **10^10 calls**—extremely inefficient!

**Final Time Complexity**: O(n^m)

**Space Complexity**

**Recursion Stack**:

The maximum depth of recursion is m (number of rows).

Each call uses **O(1) space** (ignoring input storage).

**Total stack space**: O(m).

**Auxiliary Space**:

No additional data structures are used.

**Final Space Complexity**: O(m)

**Comparison with Memoized DFS**

|  |  |  |
| --- | --- | --- |
| Version | Time Complexity | Space Complexity |
| **Native DFS** | O(n^m)  (Exponential) | O(m)  (Recursion stack) |
| **Memoized DFS** | O(m \* n^2)  (DP-like) | O(m \* n)  (Memo table) |

**Why Native DFS is Inefficient?**

**Exponential Redundancy**: The same subproblems (e.g., reaching (i, j)) are recomputed repeatedly.

**No Caching**: Without memoization, the solver blindly explores all paths, even if some are provably worse than others.

**Optimization Insight**

**Memoization reduces time complexity** to O(m n^2) by storing computed results.

**DP is even better** (same time but iterative, avoiding recursion overhead).

**Final Answer**

For the **native DFS solution** (no memoization):

**Time Complexity**: O(n^m) (Exponential)

**Space Complexity**: O(m) (Recursion stack depth)

**Avoid using it for large grids!** Prefer **memoized DFS (O(m n^2) time)** or **DP (O(m n^2) time, O(n) space)**.

**2D DP Solution Derived from int return Native DFS**

Here's a 2D dynamic programming solution that directly mirrors the logic of your native DFS implementation:

class Solution {

public int minPathCost(int[][] grid, int[][] moveCost) {

int m = grid.length;

int n = grid[0].length;

int[][] dp = new int[m][n];

// Initialize last row (base case)

for (int j = 0; j < n; j++) {

dp[m-1][j] = grid[m-1][j];

}

// Build DP table from bottom up (row m-2 to 0)

for (int i = m-2; i >= 0; i--) {

for (int j = 0; j < n; j++) {

int minPath = Integer.MAX\_VALUE;

// Explore all possible moves to next row

for (int nextCol = 0; nextCol < n; nextCol++) {

int move = moveCost[grid[i][j]][nextCol];

int curCell = grid[i][j];

int curPath = move + curCell + dp[i+1][nextCol];

minPath = Math.min(minPath, curPath);

}

dp[i][j] = minPath;

}

}

// Find minimum in first row

int minCost = Integer.MAX\_VALUE;

for (int j = 0; j < n; j++) {

minCost = Math.min(minCost, dp[0][j]);

}

return minCost;

}

}

**Key Features Matching DFS Logic:**

**Same Base Case**:

Last row initialized with cell values (dp[m-1][j] = grid[m-1][j])

Matches DFS base case return

**Same Recursive Relation**:

dp[i][j] = min(moveCost + grid[i][j] + dp[i+1][nextCol]) for all nextCol

Exactly mirrors the DFS recursive calculation

**Same Processing Order**:

Processes rows from bottom to top (matching DFS call stack)

For each row, processes all columns (like DFS explores all moves)

**Same Result Calculation**:

Takes minimum of first row values (like DFS tries all starting columns)

**Time and Space Complexity:**

**Time Complexity**: O(mn²)

m rows × n columns × n moves per cell

**Space Complexity**: O(mn)

For the DP table (can be optimized to O(n) if needed)

**Optimization to 1D DP:**

class Solution {

public int minPathCost(int[][] grid, int[][] moveCost) {

int m = grid.length;

int n = grid[0].length;

int[] dp = new int[n];

// Initialize last row

System.arraycopy(grid[m-1], 0, dp, 0, n);

for (int i = m-2; i >= 0; i--) {

int[] newDp = new int[n];

for (int j = 0; j < n; j++) {

int minPath = Integer.MAX\_VALUE;

for (int nextCol = 0; nextCol < n; nextCol++) {

int cost = grid[i][j] + moveCost[grid[i][j]][nextCol] + dp[nextCol];

minPath = Math.min(minPath, cost);

}

newDp[j] = minPath;

}

dp = newDp;

}

int minCost = Integer.MAX\_VALUE;

for (int num : dp) {

minCost = Math.min(minCost, num);

}

return minCost;

}

}

This 1D version maintains the same logic with O(n) space complexity. Both solutions are direct translations of your DFS approach into iterative DP formulations.

**Refer to**

[L62.Unique Paths (Ref.L63,L64,L174,L2087,L2304,L2400)](note://E33F0B728ABE4CCF8B7E554A219CF1EA)