Matrix

Dynamic Programming

Medium Array

In this LeetCode problem, you're presented with a unique puzzle that involves navigating a matrix called grid, which is of size m x n,

where m represents the number of rows and n represents the number of columns. Each cell in the matrix contains a distinct integer value ranging from 0 to m * n - 1. Your goal is to find a path from the top row to the bottom row that minimizes the total cost of the path. The path's total cost consists of two parts:

2. The sum of additional move costs for each step you take from one row to the next.

You're allowed to move from your current cell to any cell in the directly next row. However, you cannot make moves from cells in the last row as the path ends there.

The sum of the values of all cells that you visit on the path.

To calculate the cost of moving to the next row, you're given a matrix called moveCost. This matrix has the dimensions of the number

column in the next row. The task is to compute the minimum cost of any such path starting from the first row and ending in the last row of the grid.

of distinct integers m * n by n, which means for every possible value in grid there is a corresponding row of move costs for each

Intuition The intuition behind the solution lies in dynamic programming. Since you can move to any cell in the next row, and each move has a

cost based on the current cell's value and the column you're moving to, this problem can naturally lead us to think about state and transition.

We define a state that represents the minimum cost of reaching a particular cell in the grid. The transition will be the action of

which we can arrive at the current cell and choose the one with the minimum total cost (value of the cell plus move cost). The solution approach systematically builds up the answer row by row: 1. Start with the first row, where the cost of reaching any cell is just the value of that cell.

moving from the current cell to any cell in the next row. To compute the state, we consider all possible cells in the previous row from

previous row and adding the corresponding move cost and cell value. 3. After computing the costs for the second-last row, the best path's cost will be the cell in this row with the lowest cost because there are no more moves needed to enter cells in the last row. 4. Since we're allowed to take the path starting from any cell in the first row and ending in any cell in the last row, we consider all

2. For each row after the first, compute the minimum cost of reaching each cell by checking all possible preceding cells from the

possibilities when calculating minimum costs.

Solution Approach

- In terms of implementation, the g list represents the minimum costs of reaching each cell in the current row based on the previous
- row's calculations stored in f. We iterate through each cell in the current row (j) and each cell in the previous row (k) to determine the minimum cost g[j] by considering the existing cost f[k], the additional move cost moveCost[grid[i - 1][k]][j], and the value of the current cell grid[i][j]. We continue this process until we reach the second-last row, after which we can find the minimum
- cost of the final path by taking the smallest value in f. Note: inf represents infinity and is used to initialize the minimum costs such that any actual cost computed would be lesser than inf.
- To implement the proposed solution, we use a dynamic programming approach. Let's inspect the given Python code snippet more closely to understand how the algorithm operates step by step:

Firstly, we retrieve the dimensions of the grid using m, n = len(grid), len(grid[0]) to determine the number of rows (m) and

We then initialize a list f with the values of the first row of the grid, as for the first row, the cost is simply the value of the cells

Next, an outer loop runs from 1 to m-1 (not included), iterating over the rows of the grid. We skip row 0 because we've already

themselves since no moves are yet made. 1 f = grid[0]

Within this loop, we prepare a new list g with inf (infinity) to keep track of the minimum costs for the current row (i). Since we haven't calculated any costs yet, we set them to inf as a placeholder.

1 g = [inf] * n

1 for j in range(n):

initialized f with its values.

from the previous row, respectively.

The value of the current cell (grid[i][j]).

possible cost to reach that cell from the previous row.

that run n times respectively for calculating the costs.

for k in range(n):

1 for i in range(1, m):

columns (n).

each cell k in the previous row. This cost includes: The previously calculated minimum cost to reach the cell k (f[k]).

We take the minimum of the current value of g[j] and this newly calculated potential cost. This updates the g[j] with the lowest

After we have processed all cells for the current row, we assign the values of g to f to use in the calculation of the next row.

The move cost from the value of the cell in the previous row (grid[i - 1][k]) to the current column j (moveCost[grid[i - 1]

Inside the nested loops, we update g[j] for each cell in the current row (j). We calculate the potential cost of reaching g[j] from

We then use two nested loops to consider every cell in the current row and every possible cell from which we could have reached it

```
1 g[j] = min(g[j], f[k] + moveCost[grid[i - 1][k]][j] + grid[i][j])
```

1 f = g

1 return min(f)

columns:

inf].

[k]][j]).

After exiting the outer loop, we have calculated the minimum costs of reaching each cell in the second-last row. To find the minimum cost of the path from the first row down to the last row, we return the minimum value in f.

This implementation clocks in at O(m*n^2) time complexity because for each of the m-1 rows, we are performing two nested loops

The code thus combines principles of dynamic programming - including storing intermediate results (in f) and optimizing by

considering previously made decisions - to efficiently solve the problem and avoid redundant calculations.

```
Example Walkthrough
Let's consider a small 2×3 grid example to illustrate the solution approach:
```

[1, 1, 1], // move costs from grid value 0

[2, 2, 2], // move costs from grid value 1

[3, 3, 3], // move costs from grid value 2

[4, 4, 4], // move costs from grid value 3

[5, 5, 5], // move costs from grid value 4

[6, 6, 6] // move costs from grid value 5

Assume our grid looks like this, representing the values of each cell: [1, 2, 3], [4, 5, 6]

And let's say our moveCost matrix is given as follows, where each row corresponds to move costs from a grid value to the next row's

Step 3: We calculate the minimum cost of reaching each cell in the second row (row index 1) considering all cells from the first row.

Step 1: We start with the first row. The cost of reaching any cell in the first row is simply the value of that cell. Hence, f = [1, 2, 3]. Step 2: We now consider the second row. We initiate the new cost array g to store the minimum costs for each cell with [inf, inf,

Following the solution approach:

grid considering the path and move costs.

When processing the second row: To reach the first cell (value 4) from the first row's cells, we calculate the following potentials:

• First cell: f[0] + moveCost[1][0] + grid[1][0] = 1 + 2 + 4 = 7

• First cell: f[0] + moveCost[1][1] + grid[1][1] = 1 + 2 + 5 = 8

Second cell: f[1] + moveCost[2][1] + grid[1][1] = 2 + 3 + 5 = 10

Second cell: f[1] + moveCost[2][0] + grid[1][0] = 2 + 3 + 4 = 9

To reach the second cell (value 5) from the first row's cells, similar calculations yield:

 To reach the third cell (value 6) from the first row's cells, we get: • First cell: f[0] + moveCost[1][2] + grid[1][2] = 1 + 2 + 6 = 9 Second cell: f[1] + moveCost[2][2] + grid[1][2] = 2 + 3 + 6 = 11

Since we're at the last row, we stop and return the minimum value in f, which is the minimum total cost to reach the bottom of the

Third cell: f[2] + moveCost[3][1] + grid[1][1] = 3 + 4 + 5 = 12 The minimum cost is 8, so g[1] = 8.

Third cell: f[2] + moveCost[3][2] + grid[1][2] = 3 + 4 + 6 = 13 Here, the minimum cost is 9, so g[2] = 9.

Step 4: After completing the loop for row 1, we have g = [7, 8, 9]. We copy g to f for the next iteration.

• Third cell: f[2] + moveCost[3][0] + grid[1][0] = 3 + 4 + 4 = 11 We take the minimum of these, which is 7, so g[0] = 7.

```
Python Solution
  from typing import List
   class Solution:
      def minPathCost(self, grid: List[List[int]], moveCost: List[List[int]]) -> int:
```

Initialize the minimum path cost for the first row of the grid.

Initialize a new list to store minimum path costs for the current row

Consider all possible previous steps we could take to reach this cell

Calculate the cost of the path that goes through the source_col

to destination_col, and the cost of the destination cell itself.

moveCost[grid[i - 1][source_col]][destination_col] +

in the previous row and ends in the destination_col of the current row.

This includes the path cost up to the source_col, the move cost from source_col

This is the starting point, so no move costs are added yet.

cost = (min_path_cost[source_col] +

grid[i][destination_col])

import java.util.Arrays; // Import Arrays class for operations like fill and stream

// Define infinity as a large number to simulate infinite cost

// Calculate the cost for each cell in the current row

for (int j = 0; j < cols; ++j) { // Iterating through columns of current row

// Calculate the minimum cost to move to cell j from cell k

grid[i][j]);

nextCost[j] = Math.min(nextCost[j], currentCost[k] +

for (int k = 0; k < cols; ++k) { // Iterating through columns of previous row

moveCost[grid[i - 1][k]][j] +

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

// Get the dimensions of the grid

// Initialize the cost for the first row

// Loop through rows starting from the second

// Initialize row cost with infinity

int[] nextCost = new int[cols];

Arrays.fill(nextCost, infinity);

int rows = grid.length;

int minCost = infinity;

return minCost;

for (int cost : currentCost) {

// Return the minimum path cost

minCost = Math.min(minCost, cost);

int cols = grid[0].length;

int[] currentCost = grid[0];

final int infinity = 1 << 30;

for (int i = 1; i < rows; ++i) {

The minimum of f is 7, so the minimum cost to get from top to bottom is 7.

Get the number of rows and columns in the grid

Iterate over the grid starting from the second row

new_min_path_cost = [float('inf')] * cols

For each cell in the current row

for destination_col in range(cols):

for source_col in range(cols):

rows, cols = len(grid), len(grid[0])

min_path_cost = grid[0]

for i in range(1, rows):

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

8

10

11

12

13

14

15

16

17

18

19

20

21

23

24

25

32

33

34

35

36

37

38

39 }

29 # Update the minimum path cost for the destination_col of the current row 30 new_min_path_cost[destination_col] = min(new_min_path_cost[destination_col], cost) 31 # Update the minimum path cost to the one calculated for the current row. 32 33 min path cost = new min path cost 34 35 # Return the minimum value from the last row of the updated minimum path costs, 36 # which represents the minimum cost to reach the bottom of the grid. 37 return min(min path cost) 38

```
26
27
               // Prepare for the next row calculation
28
                currentCost = nextCost;
29
30
           // Find the minimum cost among the last row's possible paths
31
```

Java Solution

class Solution {

```
40
C++ Solution
 1 class Solution {
   public:
       int minPathCost(vector<vector<int>>& grid, vector<vector<int>>& moveCost) {
           int numRows = grid.size();
                                          // Store the number of rows in the grid
           int numCols = grid[0].size(); // Store the number of columns in the grid
 6
           // Initialize 'inf' as a sufficiently large number to represent infinity
           const int inf = (1 \ll 30);
 9
10
           // Initialize the first row of costs from the grid as there is no move cost for the first row
           vector<int> currentCosts = grid[0];
11
12
13
           // Loop through each row starting from the second one
14
           for (int i = 1; i < numRows; ++i) {</pre>
15
               // Initialize the next row of costs with 'inf' to later find the minimum cost easily
               vector<int> nextCosts(numCols, inf);
16
17
               // Loop through each column of the current row
18
               for (int currentCol = 0; currentCol < numCols; ++currentCol) {</pre>
19
                   // Loop through each column of the previous row to calculate the minimum path cost
20
21
                   for (int prevCol = 0; prevCol < numCols; ++prevCol) {</pre>
22
                        // Calculate the minimum cost for the path ending in the current column
23
                        // It consists of the cost to reach the previous column, move cost from the previous column
24
                        // to the current column, and the cost of the current grid cell
25
                        nextCosts[currentCol] = min(nextCosts[currentCol], currentCosts[prevCol]
                                                    + moveCost[grid[i - 1][prevCol]][currentCol]
26
27
                                                    + grid[i][currentCol]);
28
29
30
31
               // After computing all costs for the current row, move to the next row
32
               currentCosts = move(nextCosts);
33
34
35
           // Return the minimum element from the final row of computed costs
36
           return *min_element(currentCosts.begin(), currentCosts.end());
37
38 };
39
```

14 15 16 17

8

Typescript Solution

const m = grid.length;

const n = grid[0].length;

function minPathCost(grid: number[][], moveCost: number[][]): number {

// 'previousRowCosts' holds the min path costs for the previous row

// m is the number of rows in the grid

// n is the number of columns in the grid

let previousRowCosts = grid[0].slice();

```
9
       // Iterate over each row starting with the second row
       for (let i = 1; i < m; i++) {
10
           // 'currentRowCosts' will hold the min path costs for the current row
11
           let currentRowCosts = new Array(n);
13
           // Iterate over each column in the current row
           for (let j = 0; j < n; j++) {
               const fromValue = grid[i - 1][j];
               // Check each possible move from the 'fromValue' in the previous row
18
               for (let k = 0; k < n; k++) {
19
20
                   // Cost of moving from the previous cell plus the move cost to the current cell and its value
21
                   let pathCost = previousRowCosts[j] + moveCost[fromValue][k] + grid[i][k];
22
23
                   // If this is the first calculation or the newly calculated pathCost is smaller, update it
                   if (j == 0 || currentRowCosts[k] > pathCost) {
24
25
                       currentRowCosts[k] = pathCost;
26
27
28
29
           // Update 'previousRowCosts' to be the 'currentRowCosts' before the next iteration
30
           previousRowCosts = currentRowCosts;
31
32
33
34
       // Return the minimum path cost from the last row
35
       return Math.min(...previousRowCosts);
37
Time and Space Complexity
The time complexity of the provided code can be analyzed by looking at the loops in the function minPathCost. We see that there are
three nested loops. The outermost loop runs for m - 1 iterations, where m is the number of rows in grid. The two inner loops each run
for n iterations, where n is the number of columns. The work inside the innermost loop consists of constant-time operations.
Therefore, the time complexity is 0((m-1)*n*n), which simplifies to 0(m*n^2).
```

The space complexity is impacted by the auxiliary space used. We have a list f initialized with the first row of grid, and a list g reinitialized for each of the m rows. Since each list contains n elements, and no other data structures grow with the size of the input, the space complexity is O(n).