64. Minimum Path Sum

Medium Array Dynamic Programming Matrix

Problem Description

The problem presents a $m \times n$ grid full of non-negative integers. The objective is to find a path from the top-left corner of the grid to the bottom-right corner which has the minimum sum of all numbers along the path. The rules specify that you can move only to the right or down at any given step. The aim is to compute this minimum sum.

Imagine standing at the top-left corner of the grid and needing to reach the bottom-right corner. With only two possible moves (right and down), you must choose the path with lower numbers to minimize the sum. However, the challenge lies in making choices that might not result in the minimum sum at the current step but will lead to an overall minimum path sum.

Intuition

When faced with a grid-based pathfinding problem that asks for a minimum or maximum sum, dynamic programming is a common approach. The idea is to break the problem down into smaller problems, solve these smaller problems, and use their solutions to build up an answer to the bigger problem.

For this particular problem, we use a 2-dimensional array `f` to store the minimum path sum to each cell. To fill this array, we start with the top-left corner, which is the starting point and has only one possible sum – the value of the cell itself.

Then, for the first row and column, since there's only one way to reach any of these cells (moving right for the first row and down for the first column), each cell in these positions can only have a minimum sum equal to the sum of the current cell value and the minimum sum of the previous cell.

For the rest of the grid, each cell can be reached either from the cell above it or the cell to its left. We take the minimum of the two possible cells' minimum sums and add it to the current cell's value. This ensures that for every cell, we have computed the minimum sum required to reach it.

When we reach the bottom-right corner of the grid, the `f` array will contain the minimum path sum to that cell, which will be our answer.

Solution Approach

The solution utilizes dynamic programming, which is a method for solving complex problems by breaking them down into simpler subproblems. It's particularly apt for problems where the same subproblems are solved multiple times, allowing us to store those solutions and reuse them, effectively cutting down on the computation time.

Here's how the implementation unfolds:

1. Initialize a 2D array `f` to hold the minimum sums with the same dimensions as `grid`.
2. Set `f[0][0]` to `grid[0][0]`, as this is the starting point and there's only one path to this cell, which is the cell itself.
3. Fill out the first row and first column of `f` by accumulating sums, since there's only one way to reach any cell in the first row and column.
   - For the first column, do `f[i][0] = f[i - 1][0] + grid[i][0]` for each row `i`.
   - For the first row, do `f[0][j] = f[0][j - 1] + grid[0][j]` for each column `j`.
4. Loop through the rest of the `grid`, starting from `f[1][1]`, and at each cell `f[i][j]`, calculate the minimum sum by comparing the sum through the cell above (`f[i - 1][j]`) and the cell to the left (`f[i][j - 1]`). The minimum sum up to the current cell is the minimum of the two possible sums plus the current cell's value: `f[i][j] = min(f[i - 1][j], f[i][j - 1]) + grid[i][j]`.
5. After the loops complete, the bottom-right cell `f[-1][-1]` which corresponds to `f[m - 1][n - 1]`, will contain the minimum path sum from top left to bottom right.

Let's break down why this works:

- At any cell, the decision to choose the cell above or the cell to the left is based on which of those two cells provides a lesser sum.
- Since those sums represent the minimum sum to get to those cells, we ensure the minimum sum to get to the current cell by adding the current cell's value to the lesser of those two sums.
- This builds up to the final solution by ensuring at each step, we have the optimal (minimal) path chosen.

Ultimately, by using a dynamic programming table (`f`), we efficiently calculate the required sums without re-computing sums for each cell multiple times. This is not only a powerful strategy for minimizing computation but is also fundamental in demonstrating the concept of optimal substructure, which is characteristic of a dynamic programming approach: optimal solutions to a problem incorporate optimal solutions to its subproblems.

Example Walkthrough

Let's consider a small `3 x 3` grid to illustrate the solution approach:

```
[
  [1, 3, 1],
  [1, 5, 1],
  [4, 2, 1]
]
```

Following the steps in the solution approach:

1. We initialize a 2D array `f` with the same dimensions as `grid`, which is `3 x 3` in this case.

```
f = [
  [0, 0, 0],
  [0, 0, 0],
  [0, 0, 0]
]
```

2. Set `f[0][0]` to `grid[0][0]`. So `f[0][0]` will be `1`.

```
f = [
  [1, 0, 0],
  [0, 0, 0],
  [0, 0, 0]
]
```

3. Now, fill out the first row and first column of `f`:

   For the first column:

   `f[1][0] = f[0][0] + grid[1][0]` equals `1 + 1 = 2`

   `f[2][0] = f[1][0] + grid[2][0]` equals `2 + 4 = 6`

   For the first row:

   `f[0][1] = f[0][0] + grid[0][1]` equals `1 + 3 = 4`

   `f[0][2] = f[0][1] + grid[0][2]` equals `4 + 1 = 5`

   The `f` array now looks like this:

```
f = [
  [1, 4, 5],
  [2, 0, 0],
  [6, 0, 0]
]
```

4. We then proceed to fill the rest:

   - For `f[1][1]` (middle of the grid), the minimum sum is `min(f[0][1], f[1][0]) + grid[1][1]` which is `min(4, 2) + 5 = 7`.
   - For `f[1][2]`, take `min(f[0][2], f[1][1]) + grid[1][2]` which is `min(5, 7) + 1 = 6`.
   - For `f[2][1]`, take `min(f[1][1], f[2][0]) + grid[2][1]` which is `min(7, 6) + 2 = 8`.
   - Finally, for `f[2][2]`, take `min(f[1][2], f[2][1]) + grid[2][2]` which is `min(6, 8) + 1 = 7`.

   The final `f` array is:

```
f = [
  [1, 4, 5],
  [2, 7, 6],
  [6, 8, 7]
]
```

5. The bottom-right cell `f[-1][-1]` or `f[2][2]` contains `7`, which is the minimum path sum to reach from the top-left corner to the bottom-right corner. Our path would look like `1 → 1 → 3 → 1 → 1`.

By following these steps and reasoning, we establish the power of dynamic programming to compute the minimum sum in an efficient way without repeated calculations for each cell. The approach demonstrates optimal substructure as it builds the solution for the entire grid from the solutions of its sub-grids.

Solution Implementation

Python

```python
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        # Get the number of rows (m) and columns (n) in the grid
        num_rows, num_cols = len(grid), len(grid[0])

        # Initialize a 2D array 'cost' with the same dimensions as grid to store the min path sum
        cost = [[0] * num_cols for _ in range(num_rows)]

        # The top-left corner has a path sum equal to its value as there's no other path.
        cost[0][0] = grid[0][0]

        # Populate the first column of the 'cost' array
        # Since you can only move down, the minimum path sum is just the sum of the current cell value
        # and the minimum path sum of the cell directly above it
        for i in range(1, num_rows):
            cost[i][0] = cost[i - 1][0] + grid[i][0]

        # Populate the first row of the 'cost' array
        # Since you can only move right, the minimum path sum is the sum of the current cell value
        # and the minimum path sum of the cell directly to the left of it
        for j in range(1, num_cols):
            cost[0][j] = cost[0][j - 1] + grid[0][j]

        # Fill in the rest of the 'cost' array
        # For each cell, consider the minimum path sum from the cell above and the cell to the left
        # Add the smaller of those two values to the current cell value to get the min path sum for the current cell
        for i in range(1, num_rows):
            for j in range(1, num_cols):
                cost[i][j] = min(cost[i - 1][j], cost[i][j - 1]) + grid[i][j]

        # The bottom-right corner of the 'cost' array contains the minimum path sum for the entire grid
        return cost[-1][-1]
```

The code is written in Python 3, utilizing type hints (`List` from `typing` module), and follows Pythonic naming conventions for variables and methods.

```python
from typing import List
```

Java

```java
class Solution {
    // Method to find the minimum path sum in a grid.
    public int minPathSum(int[][] grid) {
        // m and n store the dimensions of the grid.
        int m = grid.length, n = grid[0].length;

        // dp array stores the minimum path sums.
        int[][] dp = new int[m][n];

        // Initialize top-left cell with its own value as this is the starting point.
        dp[0][0] = grid[0][0];

        // Fill in the first column (vertical path) by accumulating values.
        for (int i = 1; i < m; ++i) {
            dp[i][0] = dp[i - 1][0] + grid[i][0];
        }

        // Fill in the first row (horizontal path) by accumulating values.
        for (int j = 1; j < n; ++j) {
            dp[0][j] = dp[0][j - 1] + grid[0][j];
        }

        // Fill in the rest of the grid.
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                // The cell dp[i][j] is the minimum of the cell above or to the left of it,
                // plus the value in the current cell of the grid.
                dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];
            }
        }

        // Return the bottom-right cell which contains the min path sum from top-left to bottom-right.
        return dp[m - 1][n - 1];
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // for std::min

using std::vector;
using std::min;

class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int rowCount = grid.size(); // number of rows in the grid
        int colCount = grid[0].size(); // number of columns in the grid
        vector<vector<int>> dp(rowCount, vector<int>(colCount, 0)); // create a 2D vector for dynamic programming

        // Initialize the value for the first cell
        dp[0][0] = grid[0][0];

        // Fill the first column of the dp vector
        for (int row = 1; row < rowCount; ++row) {
            dp[row][0] = dp[row - 1][0] + grid[row][0];
        }

        // Fill the first row of the dp vector
        for (int col = 1; col < colCount; ++col) {
            dp[0][col] = dp[0][col - 1] + grid[0][col];
        }

        // Fill the rest of the dp vector
        for (int row = 1; row < rowCount; ++row) {
            for (int col = 1; col < colCount; ++col) {
                // Calculate the minimum path sum for the current cell by adding the current grid value to the minimum of the top and left cell values
                dp[row][col] = min(dp[row - 1][col], dp[row][col - 1]) + grid[row][col];
            }
        }

        // Return the minimum path sum for the bottom-right corner of the grid
        return dp[rowCount - 1][colCount - 1];
    }
};
```

TypeScript

```typescript
function minPathSum(grid: number[][]): number {
    // Get the number of rows in the grid
    const numRows = grid.length;
    // Get the number of columns in the grid
    const numCols = grid[0].length;
    // Initialize a 2D array to store the minimum path sums
    const minPathSums: number[][] = Array(numRows)
        .fill(0)
        .map(() => Array(numCols).fill(0));

    // Set the first cell's minimum path sum to the first cell's value in the grid
    minPathSums[0][0] = grid[0][0];

    // Calculate minimum path sums for the first column
    for (let row = 1; row < numRows; ++row) {
        minPathSums[row][0] = minPathSums[row - 1][0] + grid[row][0];
    }

    // Calculate minimum path sums for the first row
    for (let col = 1; col < numCols; ++col) {
        minPathSums[0][col] = minPathSums[0][col - 1] + grid[0][col];
    }

    // Calculate minimum path sums for the rest of the cells
    for (let row = 1; row < numRows; ++row) {
        for (let col = 1; col < numCols; ++col) {
            // The minimum path sum is the smaller of the min paths to the cell
            // from the left or above, plus the cell's value
            minPathSums[row][col] = Math.min(minPathSums[row - 1][col], minPathSums[row][col - 1]) + grid[row][col];
        }
    }

    // Return the minimum path sum of the bottom-right corner of the grid
    return minPathSums[numRows - 1][numCols - 1];
}
```

```typescript
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        # Get the number of rows (m) and columns (n) in the grid
        num_rows, num_cols = len(grid), len(grid[0])

        # Initialize a 2D array 'cost' with the same dimensions as grid to store the min path sum
        cost = [[0] * num_cols for _ in range(num_rows)]

        # The top-left corner has a path sum equal to its value as there's no other path.
        cost[0][0] = grid[0][0]

        # Populate the first column of the 'cost' array
        # Since you can only move down, the minimum path sum is just the sum of the current cell value
        # and the minimum path sum of the cell directly above it
        for i in range(1, num_rows):
            cost[i][0] = cost[i - 1][0] + grid[i][0]

        # Populate the first row of the 'cost' array
        # Since you can only move right, the minimum path sum is the sum of the current cell value
        # and the minimum path sum of the cell directly to the left of it
        for j in range(1, num_cols):
            cost[0][j] = cost[0][j - 1] + grid[0][j]

        # Fill in the rest of the 'cost' array
        # For each cell, consider the minimum path sum from the cell above and the cell to the left
        # Add the smaller of those two values to the current cell value to get the min path sum for the current cell
        for i in range(1, num_rows):
            for j in range(1, num_cols):
                cost[i][j] = min(cost[i - 1][j], cost[i][j - 1]) + grid[i][j]

        # The bottom-right corner of the 'cost' array contains the minimum path sum for the entire grid
        return cost[-1][-1]
```

The code is written in Python 3, utilizing type hints (`List` from `typing` module), and follows Pythonic naming conventions for variables and methods.

```python
from typing import List
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the nested loops that iterate over the entire `grid` matrix. Since we iterate through all rows `m` and all columns `n` of the grid, the time complexity is $O(m * n)$.

Space Complexity

The space complexity of the code is given by the additional 2D array `f` that we use to store the minimum path sums. This array is of the same size as the input `grid`, so the space complexity is also $O(m * n)$. We do not use any other significant space that grows with the size of the input.