931. Minimum Falling Path Sum

Medium Array Dynamic Programming Matrix

Leetcode Link

Problem Description

In this problem, we have a square (`n x n`) grid of integers called `matrix`. A "falling path" can be visualized as a path starting from the first row and moving down to the last row, selecting one integer in each row as part of the sum. At each step, you can move straight down or diagonally down to the left or right. The goal is to find the falling path with the smallest sum of the numbers selected during this process.

Put simply, we're looking for the smallest total we can obtain from top to bottom by choosing numbers according to these rules:

1. Start at any number in the first row.
2. Choose a number from the next row that is either directly below, diagonally to the left, or diagonally to the right of the chosen number from the previous row.
3. Repeat until the last row is reached.
4. Add up all the numbers selected on this path.
5. The minimum sum amongst all possible paths is the solution to this problem.

Intuition

The intuition behind the provided solution is to use dynamic programming, which is a way to solve problems by breaking them down into simpler subproblems and building up the solutions to larger problems based on the solutions to the smaller problems.

For this particular problem, we utilize dynamic programming by keeping track of the minimum falling path sum that ends at each element of the current row. We leverage a temporary list `f` to hold these minimum sums for the previous row as we iterate through each row of the matrix.

As we move from one row to the next, we update our list of minimum sums `f` so that it always contains the minimum sum up to that point for each possible ending position in the row. At each step, we look at the three possible moves (down, down-left, or down-right) and choose the one that leads to the smallest sum so far.

The key dynamic programming insight here is that the minimum sum at each position in a row only depends on the sums at positions that are either directly above or diagonally above it from the previous row. Therefore, we do not need to remember the entire path to calculate the minimum sum, just the values from the previous row.

At the end of the process, `f` will contain the minimum sums for the last row, and the smallest number in `f` will be the minimum sum for the entire matrix.

Solution Approach

The solution follows a dynamic programming approach, which is a strategy for solving complex problems by breaking them down into simpler sub-problems. It efficiently solves problems by storing the sub-problem solutions in a table (usually an array) and using these solutions to construct solutions to bigger problems.

Here's how the algorithm works:

1. **Initialization**: We start with a list `f` of size `n` (where `n` is the number of rows and columns of the matrix) that will hold the minimum path sums for each column in the row that we've processed so far. At this point, we consider the first row as our base case, and our initial `f` list is just the first row of the matrix itself.
2. **Iterating Through Rows**: We iterate through each row in the matrix starting from the second row, as the first row is already in our `f` list.
3. **Updating the Minimum Sums**: For each element `x` in the current row, we:
   - Find the indices `l` and `r` which represent the range in the previous row from which we can "fall" to the current element. This range is just below, to the left below, or to the right below the current position.
   - Update the temporary list `g` to store the new minimum sums. For the current element `x`, the new minimum sum is calculated by adding `x` to the minimum of the sums in the range `(l, r)` from the list `f`.
   - `g[j] = min(f[l:r]) + x`
   - The slice `f[l:r]` gives the sums of the paths that could fall into our current position `x`.
4. **Copying the Updated Sums**: After updating all the elements in the temporary list `g` for the current row, we copy the contents of `g` into `f` so that `f` now contains the minimum sums for the current row. This process repeats for each row, essentially moving the 'window' of sums down the matrix.
5. **Finding the Result**: Once we reach the last row, `f` will contain the minimum path sums that end in each of the last row's columns. The minimum sum of any falling path through the matrix will then be the smallest value in `f`.

The dynamic programming pattern used here is known as bottom-up, where the solutions to the smallest sub-problems are computed first and then used to build up the answers to larger problems.

Here is the essence of the code that implements the above algorithm:

```
for row in matrix:
    g = row[:]
    for j, x in enumerate(row):
        l, r = max(0, j - 1), min(n, j + 2)
        g[j] = min(f[l:r]) + x
    f = g
```

This code performs the iterative step where for each number `x` in the current row, it calculates the minimum path sum ending at that position using the previously stored sums in `f`, and updates the temporary list `g`. After processing the row, it replaces `f` with `g`, so that `f` is ready to be used for the next row.

With these steps, the algorithm ensures that by the end, we have computed the minimum sum of all falling paths through the matrix, and we return the smallest value contained in `f` after processing all the rows.

```
return min(f)
```

This line returns the smallest sum, which is the answer we want.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider a `3x3` matrix:

```
matrix = [
  [2, 1, 3],
  [6, 5, 4],
  [7, 8, 9]
]
```

We are looking for the minimum falling path sum from the first row to the last.

Following the algorithm steps:

1. **Initialization**: Initialize `f` with the values from the first row.

   `f = [2, 1, 3]` (representing the path sums including each of the elements in the first row)

2. **Iterating Through Rows**: Process the second row `[4, 5, 1]`.

3. **Updating the Minimum Sums**: For each element in the second row, calculate the new path sums and temporarily store them in `g`.

   - When processing 4 (at index 0), the range is `f[0:2]`. The minimum of `f[0:2]` is `f[1]` which is 1.
     `g[0] = 1 + 4 = 5`

   - For 5 (at index 1), the range is `f[0:3]`. The minimum of `f[0:3]` is again `f[1]` which is 1.
     `g[1] = 1 + 5 = 6`

   - For 1 (at index 2), the range is `f[1:3]`. The minimum of `f[1:3]` is `f[1]` which is 1.
     `g[2] = 1 + 1 = 2`

   Now `g = [5, 6, 2]`.

4. **Copying the Updated Sums**: Overwrite `f` with the calculated sums in `g`.

   `f = g = [5, 6, 2]`

5. **Process the last row `[7, 8, 9]` following the same steps:**

   - For 7 (index 0): `g[0] = min(f[0:2]) + 7 = min(5, 6) + 7 = 5 + 7 = 12`
   - For 8 (index 1): `g[1] = min(f[0:3]) + 8 = min(5, 6, 2) + 8 = 2 + 8 = 10`
   - For 9 (index 2): `g[2] = min(f[1:3]) + 9 = min(6, 2) + 9 = 2 + 9 = 11`

   After the last row, `g = [12, 10, 11]` and we update `f` accordingly.

   `f = g = [12, 10, 11]`

6. **Finding the Result**: The smallest value in `f` will be the minimum sum of any falling path.

```
return min(f)  # Returns 10
```

The falling path corresponding to the minimum sum is `1 -> 1 -> 8`, producing a sum of `10`, which is the minimum falling path sum for this matrix.

Python Solution

```
class Solution:
    def minFallingPathSum(self, matrix: List[List[int]]) -> int:
        # Get the size of the matrix
        size = len(matrix)

        # Initialize the previous row's minimum falling path sum (top row initially)
        prev_row_min = [0] * size

        # Traverse through each row in the matrix
        for row in matrix:
            # Initialize the current row's minimum falling path sum
            current_row_min = [0] * size

            # For each element in the current row
            for index, value in enumerate(row):
                # Determine the valid range to check in previous row
                # We can pick the current position, or one step left or right
                left_bound = max(0, index - 1)
                right_bound = min(size, index + 2)

                # Calculate the minimum falling path sum for the current element by:
                # - Finding the min across the valid range from the prev row
                # - Adding the current element's value to this min
                current_row_min[index] = min(prev_row_min[left_bound:right_bound]) + value

            # Update the previous row to current row before moving to the next row
            prev_row_min = current_row_min

        # After processing all rows, return the minimum falling path sum
        return min(prev_row_min)
```

Java Solution

```
class Solution {

    // Function to calculate the minimum falling path sum in a matrix
    public int minFallingPathSum(int[][] matrix) {
        int n = matrix.length; // Get the size of the square matrix
        int[] dpCurrentRow = new int[n]; // Current row dp array

        // Initialize dp with the first row of matrix.
        for (int i = 0; i < n; ++i) {
            dpCurrentRow[i] = matrix[0][i];
        }

        // Traverse from second row to the last row
        for (int row = 1; row < n; ++row) {
            int[] dpPrevRow = dpCurrentRow; // Store the previous row for this row calculations
            dpCurrentRow = new int[n]; // Reset current row dp array

            for (int colIdx = 0; colIdx < n; ++colIdx) {
                // Initialize minimum sum as the value above the current cell
                int minSumAbove = dpPrevRow[colIdx];

                // If not in the first column, consider the upper-left neighbor
                if (colIdx > 0) {
                    minSumAbove = Math.min(minSumAbove, dpPrevRow[colIdx - 1]);
                }
                // If not in the last column, consider the upper-right neighbor
                if (colIdx < n - 1) {
                    minSumAbove = Math.min(minSumAbove, dpPrevRow[colIdx + 1]);
                }

                // Update the dp array for the next row with the new minimum
                dpNextRow[colIdx] = minSumAbove + matrix[row][colIdx];
            }
            // Move to the next row
            dpCurrentRow = dpNextRow;
        }

        // After processing all the rows, find the minimum falling path sum
        int minFallingPathSum = Integer.MAX_VALUE;
        for (int x : dpCurrentRow) {
            minFallingPathSum = Math.min(minFallingPathSum, x);
        }

        return minFallingPathSum;
    }
}
```

C++ Solution

```
class Solution {
public:
    int minFallingPathSum(vector<vector<int>>& matrix) {
        int matrixSize = matrix.size(); // Get the size of the matrix
        vector<int> prevRowCosts(matrixSize); // Initialize a vector to store cost of the previous row

        // Loop over all the rows of the matrix
        for (auto& row : matrix) {
            vector<int> currentRowCosts = prevRowCosts; // Copy the previous row costs to the current row (initially)

            // Calculate the current row costs considering the falling path sum from the previous row
            for (int j = 0; j < matrixSize; ++j) {
                // Ensure we're not on the first element to avoid going out of bounds
                if (j > 0) {
                    currentRowCosts[j] = min(currentRowCosts[j], prevRowCosts[j - 1]); // Take the smaller path from the left diagonal
                }

                // Ensure we're not on the last element to avoid going out of bounds
                if (j < matrixSize - 1) {
                    currentRowCosts[j] = min(currentRowCosts[j], prevRowCosts[j + 1]); // Take the smaller path from the right diagonal
                }

                currentRowCosts[j] += row[j]; // Add the current row's cost to the min cost found
            }

            // Update the previous row costs to be the current row costs for the next iteration
            prevRowCosts = move(currentRowCosts);
        }

        // Return the minimum element from the last row, since it contains the min falling path sum for each column
        return *min_element(prevRowCosts.begin(), prevRowCosts.end());
    }
};
```

Typescript Solution

```
/**
 * Calculates the minimum falling path sum in a square matrix.
 * A falling path starts at any element in the first row and chooses
 * an element in the next row that is either directly below or to a column
 * that is different from the previous row's column by at most one.
 *
 * @param {number[][]} matrix The input square matrix.
 * @return {number} The minimum falling path sum.
 */
function minFallingPathSum(matrix: number[][]): number {
    const size = matrix.length; // Size of the matrix
    // Initialize the array to store the minimum sum at each position.
    let currentSums: number[] = new Array(size).fill(0);

    // Iterate over each row of the matrix.
    for (const row of matrix) {
        // Copy current sums to temporary array to calculate new sums.
        let nextSums = [...currentSums];

        // Process each element in the row.
        for (let col = 0; col < size; ++col) {
            // If not the first column, take the minimum of the current sum
            // and the sum from the left neighbor.
            if (col > 0) {
                nextSums[col] = Math.min(nextSums[col], currentSums[col - 1]);
            }

            // If not the last column, take the minimum of the current sum
            // and the sum from the right neighbor.
            if (col + 1 < size) {
                nextSums[col] = Math.min(nextSums[col], currentSums[col + 1]);
            }

            // Add the current cell's value to the next sum.
            nextSums[col] += row[col];
        }

        // Update current sums with the calculated next sums.
        currentSums = nextSums;
    }

    // Return the minimum sum found in the last row.
    return Math.min(...currentSums);
}
```

Time and Space Complexity

The given Python function `minFallingPathSum` calculates the minimum falling path sum in a square matrix using dynamic programming.

- **Time Complexity**:

  For time complexity, we consider how the algorithm scales with the size of the input matrix, which for our purposes is `n x n` where `n` is the length of one side of the matrix.

  The outer loop runs once for each row in the matrix, so it iterates `n` times. The inner loop runs for each element in a row, also `n` times. Within the inner loop, we are finding the minimum of a slice of the previous row's data which is a `O(1)` operation since the slice length is at most 3, regardless of `n`. Thus, the time complexity is simply the number of times both loops run, multiplied together: `O(n) * O(n) = O(n^2)`.

- **Space Complexity**:

  The space complexity is determined by the amount of additional memory the algorithm uses as the size of the input changes.

  In this algorithm, we have two lists, `f` and `g`, each of the same size as a row of the matrix, which is `n`. The contents of `f` are replaced with the contents of `g` in each iteration of the outer loop. No other data structures are dependent on the size of the input data. Therefore, the space complexity is `O(n)` for the list `f`, plus `O(n)` for the list `g`, which still simplifies to `O(n)` overall.