1289. Minimum Falling Path Sum II

Hard | Array | Dynamic Programming | Matrix

Leetcode Link

Problem Description

Given an `n x n` integer matrix named `grid`, the objective is to calculate the minimum sum of what is termed a "falling path with non-zero shifts." A falling path with non-zero shifts is a selection of one element from each row of the matrix such that for any two elements chosen from consecutive rows, they are not situated in the same column. Essentially, we need to find a path from the top to the bottom of the matrix, selecting one element from each row, without reusing columns in consecutive rows, and adding those elements to achieve the smallest possible sum.

Intuition

To find the minimum falling path sum, we can use a dynamic programming approach that builds upon the answer from the previous row to compute the answer for the current row. Specifically, for each row, we keep track of the smallest value (`f`) and the second smallest (`g`) from the previous row. This is because we cannot pick elements from the same column in consecutive rows, so we need to have a backup option in case the smallest value from the previous row is in the same column.

The solution progressively calculates the sum of each row, taking into account the smallest sums of the previous row while avoiding the same column positions. For each row, we iterate through its columns and try to find the smallest sum (`ff`) by either choosing the smallest sum for different columns or the second smallest sum (`gg`) when the same column as the previous smallest is encountered. This way, we prevent two consecutive rows from having elements in the same column. By keeping an updated minimum (`f`) and second minimum (`g`) and their column positions (`fp`), we can continue to build up the solution row by row, until we reach the bottom of the matrix. The minimum sum of the last calculated row will be the final answer, which provides the minimum sum of a falling path with non-zero shifts as required.

Solution Approach

The solution approach makes use of dynamic programming as its core strategy. The idea is to iterate over each row and compute the minimum and the second minimum values required to reach each position without considering the previous row's column. By doing so, we ensure compliance with the problem's condition that no two elements in adjacent rows are in the same column.

A crucial aspect of the solution is tracking the minimum (`f`) and the second minimum (`g`) costs to reach the current row while recording the column index (`fp`) of the minimum cost from the previous row. This information is used to decide whether to use the minimum or second minimum cost when calculating the current row's values.

Here's a walkthrough of the algorithm using the problem's terminology:

1. Initialize `f` and `g` to 0 as the initial minimum and second minimum costs, respectively. Initialize `fp` to -1 to indicate no previous column.

2. For each row `row` in the matrix `grid`:

   - Initialize `ff` and `gg` to `inf` as placeholders for the current row's minimum and second minimum costs.
   - Initialize `ffp` to -1, which will hold the column index of the current row's minimum cost.
   - For each value `v` in `row`, with its index `j`:
     - Calculate the sum `s` by adding `v` to `g` if the current column `j` is the same as `fp` (to avoid picking two elements from the same column in adjacent rows), or to `f` otherwise.
     - If the calculated sum `s` is less than `ff`:
       - Assign the value of `ff` to `gg`, as `ff` will now hold the new minimum, `s`.
       - Update `ff` with the new minimum, `s`.
       - Update `ffp` with the current column index `j`, indicating the column of the new minimum.
     - Else if `s` is less than `gg`, update `gg` with the new second minimum, `s`.

3. After the loop, assign `ff` to `f`, `gg` to `g`, and `ffp` to `fp` for the next iteration. This step effectively moves the minimum and second minimum costs, along with the column index of the minimum, to the next row.

4. Once all rows have been processed, `f` holds the minimum sum of a falling path with non-zero shifts and is returned as the result.

Throughout this process, the algorithm avoids using the same column for consecutive rows by considering `g` (the second smallest) whenever the current column is the same as the previous row's minimum. By keeping a running tally of the smallest sums that conform to the problem's rules, the algorithm ensures optimal substructure—one of the hallmarks of dynamic programming. In the end, the minimum sum of the final row provides the answer.

This solution has a time complexity of `O(n^2)` where `n` is the number of rows (or columns) in the matrix, making it efficient enough to handle larger matrices.

The code is a practical implementation of this approach and manifests the described dynamic programming strategy effectively.

Example Walkthrough

Let's use a 3×3 matrix as a small example to illustrate the solution approach.

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

Following the dynamic programming approach to find the minimum falling path sum with non-zero shifts:

1. Initialize `f` and `g` to 0, and `fp` to -1 (no previous column):

```
f = 0, g = 0, fp = -1
```

2. For the first row, since there are no previous rows, the minimum (`f`) and second minimum (`g`) sums are the smallest and second smallest values from the row itself. We then determine `f`, `g`, and `fp`:

```
Row 0: [2, 1, 3]
f = 1, g = 2, fp = 1 (index of minimum value in row 0)
```

3. Move to the second row, update `ff`, `gg`, and `ffp` as we find the sums for this row:

```
Row 1: [6, 5, 4], Previous `fp` = 1

For column 0: s = grid[1][0] + g = 6 + 2 = 8 (since column 0 is not `fp`)
For column 1: Cannot choose since `fp` is 1 (same column as the previous minimum)
For column 2: s = grid[1][2] + f = 4 + 1 = 5 (since column 2 is not `fp`)

Update `f`, `g`, and `fp` for this row:
ff = 5, gg = 8, ffp = 2
```

4. After iterating through the second row, update `f`, `g`, and `fp` with the new values:

```
f = ff = 5, g = gg = 8, fp = ffp = 2
```

5. Repeat the process for the third row:

```
Row 2: [7, 8, 9], Previous `fp` = 2

For column 0: s = grid[2][0] + g = 7 + 8 = 15 (since column 0 is not `fp`)
For column 1: s = grid[2][1] + f = 8 + 5 = 13 (since column 1 is not `fp`)
For column 2: Cannot choose since `fp` is 2 (same column as the previous minimum)

Only two valid sums for this row:
ff = 13, gg = 15, ffp = 1
```

6. After the final row, we update `f`, `g`, and `fp` for the last time:

```
f = ff = 13, g = gg = 15, fp = ffp = 1
```

The minimum sum of a falling path with non-zero shifts is held in `f`, which is 13 in this case.

This completes the example for the 3×3 matrix, the path that leads to the minimum sum would be from the elements grid[0][1], grid[1][2], and grid[2][1], resulting in the path 1 → 4 → 8, with a sum of 13.

Python Solution

```python
class Solution:
    def minFallingPathSum(self, matrix: List[List[int]]) -> int:
        # Initialize the smallest and second smallest falls from the previous row ("f" and "g").
        # Initialize the position of the smallest fall in the previous row ("f_position").
        smallest_fall = second_smallest_fall = float('inf')
        smallest_fall_position = -1

        # Iterate over each row in the matrix.
        for row in matrix:
            # Initialize the smallest and second smallest falls of the current row
            # ("current_smallest" and "current_second_smallest"), and the position of
            # the smallest fall in the current row ("current_smallest_position").
            current_smallest = current_second_smallest = float('inf')
            current_smallest_position = -1

            # Iterate over each value in the current row with its index.
            for j, value in enumerate(row):
                # Determine whether to use the smallest or second smallest fall from
                # the previous row to add to the current element.
                total_path_sum = (second_smallest_fall if j == smallest_fall_position else smallest_fall) + value
                # Update the smallest or second smallest fall if necessary.
                if total_path_sum < current_smallest:
                    current_second_smallest = current_smallest
                    current_smallest = total_path_sum
                    current_smallest_position = j
                elif total_path_sum < current_second_smallest:
                    current_second_smallest = total_path_sum

            # Move to the next row: update the smallest and second smallest falls and the position
            smallest_fall, second_smallest_fall, smallest_fall_position = (
                current_smallest, current_second_smallest, current_smallest_position
            )

        # The matrix has been processed, "smallest_fall" contains the minimum falling path sum.
        return smallest_fall
```

Java Solution

```java
class Solution {
    public int minFallingPathSum(int[][] grid) {
        int firstMin = 0; // The smallest sum of the falling path so far.
        int secondMin = 0; // The second smallest sum of the falling path so far.
        int firstMinPos = -1; // Position of the smallest sum in the previous row.
        final int INF = Integer.MAX_VALUE; // Set a high value to represent the initial state that's effectively infinity.

        // Iterating through each row in the grid.
        for (int[] row : grid) {
            int curFirstMin = INF; // The smallest sum in the current row.
            int curSecondMin = INF; // The second smallest sum in the current row.
            int curFirstMinPos = -1; // Position of the smallest sum in the current row.

            // Iterating through each element in the current row.
            for (int j = 0; j < row.length; ++j) {
                // Calculate the sum by adding the current element to the smaller of the two sums from the previous row,
                // but not the one directly above (to avoid falling path through same column, hence j != firstMinPos check).
                int sum = (j != firstMinPos ? firstMin : secondMin) + row[j];

                // If the calculated sum is smaller than the current smallest sum, update the first and second min values and positic
                if (sum < curFirstMin) {
                    curSecondMin = curFirstMin; // The smallest becomes the second smallest.
                    curFirstMin = sum;          // The current sum becomes the new smallest.
                    curFirstMinPos = j;         // Update the current smallest sum position.
                } else if (sum < curSecondMin) { // If the current sum is between first and second min, update the second min only.
                    curSecondMin = sum;
                }
            }

            // Prepare for next row.
            firstMin = curFirstMin;
            secondMin = curSecondMin;
            firstMinPos = curFirstMinPos;
        }

        // After processing all rows, the smallest sum will represent the minimum sum of a falling path.
        return firstMin;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <climits> // For INT_MAX usage

class Solution {
public:
    int minFallingPathSum(vector<vector<int>>& grid) {
        int n = grid.size(); // Get the size of the grid.

        // Using descriptive names for variables to indicate their usage.
        int minFirstPathSum = 0; // Stores the minimum sum of the first path.
        int minSecondPathSum = 0; // Stores the minimum sum of the second best path.
        int prevMinPathCol = -1; // Index of the column resulting in the minimum sum in the previous row.

        // Infinity substitute for initialization (could use INT_MAX from <climits>).
        const int kInfinity = INT_MAX;

        // Iterate over each row in the input grid.
        for (auto& row : grid) {
            int newMinFirstPathSum = kInfinity; // The new minimum sum of the first path.
            int newMinSecondPathSum = kInfinity; // The new minimum sum of the second best path.
            int newMinPathCol = -1; // Column index for the new minimum sum.

            // Iterate over each element in the current row.
            for (int j = 0; j < n; ++j) {
                int currentSum = (prevMinPathCol != j ? minFirstPathSum : minSecondPathSum) + row[j];

                // If the current sum is less than the new minimum, update both the first and second best sums.
                if (currentSum < newMinFirstPathSum) {
                    newMinSecondPathSum = newMinFirstPathSum; // Previous min becomes second best.
                    newMinFirstPathSum = currentSum; // Current sum becomes the new min.
                    newMinPathCol = j; // Update the column index for new min.
                } else if (currentSum < newMinSecondPathSum) {
                    newMinSecondPathSum = currentSum; // Update the second best path sum if current is less than it.
                }
            }

            // Update the path sums and the column index for the next row iteration.
            minFirstPathSum = newMinFirstPathSum;
            minSecondPathSum = newMinSecondPathSum;
            prevMinPathCol = newMinPathCol;
        }
        return minFirstPathSum; // Return the minimum sum of a falling path.
    }
};
```

Typescript Solution

```typescript
// Import required modules (if needed in the context where this code will run).
// In vanilla TypeScript, importing modules like this isn't necessary.
// import { INT_MAX } from 'constants';

// Declare a constant to represent infinity (as a substitute for INT_MAX).
const kInfinity: number = Number.MAX_SAFE_INTEGER;

// Global variables acting as state (simulating class member variables)
let minFirstPathSum: number = 0; // Stores the minimum sum of the first path.
let minSecondPathSum: number = 0; // Stores the minimum sum of the second best path.
let prevMinPathCol: number = -1; // Index of the column resulting in the minimum sum in the previous row.

function minFallingPathSum(grid: number[][]): number {
    const n: number = grid.length; // Get the size of the grid.

    // Iterate over each row in the input grid.
    for (let row of grid) {
        let newMinFirstPathSum: number = kInfinity; // The new minimum sum of the first path.
        let newMinSecondPathSum: number = kInfinity; // The new minimum sum of the second best path.
        let newMinPathCol: number = -1; // Column index for the new minimum sum.

        // Iterate over each element in the current row.
        for (let j = 0; j < n; ++j) {
            let currentSum: number = (prevMinPathCol !== j ? minFirstPathSum : minSecondPathSum) + row[j];

            // If the current sum is less than the new minimum, update both the first and second best sums.
            if (currentSum < newMinFirstPathSum) {
                newMinSecondPathSum = newMinFirstPathSum; // Previous min becomes second best.
                newMinFirstPathSum = currentSum; // Current sum becomes the new min.
                newMinPathCol = j; // Update the column index for new min.
            } else if (currentSum < newMinSecondPathSum) {
                newMinSecondPathSum = currentSum; // Update the second best path sum if current is less than it.
            }
        }

        // Update the path sums and the column index for the next row iteration.
        minFirstPathSum = newMinFirstPathSum;
        minSecondPathSum = newMinSecondPathSum;
        prevMinPathCol = newMinPathCol;
    }

    return minFirstPathSum; // Return the minimum sum of a falling path.
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n * m)`, where `n` is the number of rows and `m` is the number of columns in the input grid. This is because the code iterates through each row with an outer loop and each column with an inner loop, computing the minimum falling path sum in a single pass.

Space Complexity

The space complexity of the code is `O(1)`. Although the algorithm iterates through the entire grid, it uses only a constant amount of additional space for the variables `f`, `g`, `fp`, `ff`, `gg`, and `ffp`, regardless of the size of the input grid.