Minimum Path Sum in a Grid

Created by	Chandra Prakash

Problem Statement

We are given an MxN matrix of integers. We need to find a path from the top-left corner to the bottom-right corner of the matrix, such that there is a minimum cost past that we select.

At every cell, we can move in only two directions: right and bottom. The cost of a path is given as the sum of values of cells of the given matrix.

Editorial

Greedy fails. We need to try all possible paths and find min path sum.

Recursion

Define the function statement:

`f(int i, int j)` : The min cost to go from (0, 0) to (i, j).

Direction: We chose to start from the last element.

Fix answer output:

Since we started from the last element, path ends if we reach the first element (0, 0). Return the cost of it.

Base case:

```
if(i == 0 && j == 0) return arr[0][0];
```

Corner cases:

`if(i < 0 || j < 0)` , then that path is **invalid**. You should **never pick** this path when comparing it to other valid paths.

So we return a **very large number** (e.g. `INT_MAX`) to represent that **this path is not usable**.

When you take `min()` between this and a valid path, the valid path will naturally be chosen.

Action at every step:

Add the value and either go up or left. Take the min among up and left.

Code

```
int f(int i, int j, vector<vector<int>>& matrix) {
    if(i == 0 && j == 0) return matrix[0][0];
    if(i < 0 || j < 0) return INT_MAX;

    int up = f(i-1, j, matrix);
    int left = f(i, j-1, matrix);
    return matrix[i][j] + min(up, left);
}
```

Complexity Analysis

Time Complexity: `O(2^(m + n))` - Each cell makes two recursive calls.

Space Complexity: `O(m + n)` - Maximum depth of the recursion stack can go up to `m + n` in the worst case.

Memoization

Approach

- Store the result.
- Return value if already calculated.

Code

```
int f(int i, int j, vector<vector<int>>& matrix, vector<vector<int>>& dp) {
    if(i == 0 && j == 0) return matrix[0][0];
    if(i < 0 || j < 0) return INT_MAX;
    if(dp[i][j] != -1) return dp[i][j];

    int up = f(i-1, j, matrix, dp);
    int left = f(i, j-1, matrix, dp);
    return dp[i][j] = matrix[i][j] + min(up, left);
}
```

Complexity Analysis

Time Complexity: `O(m * n)` - Each cell is computed only once and stored in the `dp` table.

Space Complexity: `O(m * n + m + n)` which is `O(m * n)` for memoization table and `O(m + n)` for recursion stack depth.

Tabulation

Approach

- Loop through all states i.e., i and j.
- If i and j are 0, return the cost.
- Else, take the min of up, left and store.

Code

```
int minPathSum(vector<vector<int>>& matrix) {
    int m = matrix.size();
    int n = matrix[0].size();
    vector<vector<int>> dp(m, vector<int>(n, 0));

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(i == 0 && j == 0) {
                dp[i][j] = matrix[i][j];
            } else {
                int up = (i > 0) ? dp[i-1][j] : INT_MAX;
```

```
            int left = (j > 0) ? dp[i][j-1] : INT_MAX;
            dp[i][j] = matrix[i][j] + min(up, left);
        }
    }
  }

  return dp[m-1][n-1];
}
```

Complexity Analysis

Time Complexity: `O(m * n)` - Every cell is visited once in nested loops over the matrix.

Space Complexity: `O(m * n)` - A 2D `dp` table of size `m x n` is used to store the minimum cost for each cell.

Space Optimisation

Approach

- Initialize 1D `prev` array.
- Loop through each row:
  - For each cell in the row, calculate the minimum cost using `up` from `prev[j]` and `left` from `curr[j-1]`.
  - Store results in `curr[j]`.
- After each row, set `prev = curr`.

Code

```
int minPathSum(vector<vector<int>>& matrix) {
  int m = matrix.size();
  int n = matrix[0].size();

  vector<int> prev(n, 0);

  for(int i = 0; i < m; i++) {
    vector<int> curr(n, 0);
    for(int j = 0; j < n; j++) {
      if(i == 0 && j == 0) {
        curr[j] = matrix[i][j];
      } else {
        int up = (i > 0) ? prev[j] : INT_MAX;
        int left = (j > 0) ? curr[j-1] : INT_MAX;
        curr[j] = matrix[i][j] + min(up, left);
      }
    }
    prev = curr;
  }

  return prev[n-1];
}
```

Complexity Analysis

Time Complexity: `O(m * n)` - All cells are visited once in nested loops.

Space Complexity: `O(n)` - Only two 1D arrays of size `n` are used for computation.

THE END