# Maximum Falling Path Sum

Created by Chandra Prakash

## Problem Statement

We are given an `m x n` matrix. We need to find the **maximum path sum** from **any cell of the first row** to **any cell of the last row**.

At every cell, we can move in **three directions**:

- to the bottom cell
- to the bottom-right cell
- to the bottom-left cell

---

## Editorial

Greedy fails. Because if we always maximise the values, there can be some other path that has total maximum value but has a minimum in the first steps.

Direction: Start from the **top row.**

## Recursion

### Approach

**Indices:** `i` (row) and `j` (column)

**Base Case:** If `i == m - 1`, we're at the last row.

Return the value at `matrix[i][j]`.

**Bounds Check:**

We must check that `j` stays inside the matrix while exploring left and right diagonals.

**Explore all directions:**

```
down = matrix[i][j] + f(i+1, j);
leftDiag = matrix[i][j] + f(i+1, j-1);
rightDiag = matrix[i][j] + f(i+1, j+1);
```

**Return max:**

`return max(down, max(leftDiag, rightDiag));`

**Final Answer:** Try all starting columns in the first row, and take the **maximum** among them.

### Code

```
int f(int i, int j, vector<vector<int>>& mat, int m, int n) {
    if(j < 0 || j >= n) return -1e9; // invalid move
    if(i == m - 1) return mat[i][j];

    int down = mat[i][j] + f(i + 1, j, mat, m, n);
    int leftDiag = mat[i][j] + f(i + 1, j - 1, mat, m, n);
    int rightDiag = mat[i][j] + f(i + 1, j + 1, mat, m, n);
```

```
    return max({down, leftDiag, rightDiag});
}

int maxPathSum(vector<vector<int>>& matrix) {
    int m = matrix.size(), n = matrix[0].size();
    int maxi = INT_MIN;
    for(int j = 0; j < n; j++) {
        maxi = max(maxi, f(0, j, matrix, m, n));
    }
    return maxi;
}
```

### Complexity Analysis

**Time Complexity:** `O(3^m)` – For each level, 3 choices (down, left, right).

**Space Complexity:** `O(m)` – Maximum recursion stack depth.

---

## Memoization

### Approach

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

### Code

```
int f(int i, int j, vector<vector<int>>& mat, int m, int n, vector<vector<int>>& dp) {
    if(j < 0 || j >= n) return -1e9;
    if(i == m - 1) return mat[i][j];
    if(dp[i][j] != -1) return dp[i][j];

    int down = mat[i][j] + f(i + 1, j, mat, m, n, dp);
    int leftDiag = mat[i][j] + f(i + 1, j - 1, mat, m, n, dp);
    int rightDiag = mat[i][j] + f(i + 1, j + 1, mat, m, n, dp);

    return dp[i][j] = max({down, leftDiag, rightDiag});
}

int maxPathSum(vector<vector<int>>& matrix) {
    int m = matrix.size(), n = matrix[0].size();
    vector<vector<int>> dp(m, vector<int>(n, -1));
    int maxi = INT_MIN;
    for(int j = 0; j < n; j++) {
        maxi = max(maxi, f(0, j, matrix, m, n, dp));
    }
    return maxi;
}
```

### Complexity Analysis

**Time Complexity:** `O(m * n)` – Each cell is computed only once.

**Space Complexity:** `O(m * n) + O(m)` – DP table + recursion stack.

## Tabulation

### Approach

- `dp[i][j]` = max path sum starting from `(i, j)` to bottom.
- Build the table from bottom row to top row.

**Transition:**

```
dp[i][j] = matrix[i][j] + max(dp[i+1][j], max(dp[i+1][j-1], dp[i+1][j+1]))
```

**Base Case:**

Last row of `dp` is same as last row of `matrix`.

### Code

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

    for(int j = 0; j < n; j++) {
        dp[m-1][j] = matrix[m-1][j];
    }

    for(int i = m - 2; i >= 0; i--) {
        for(int j = 0; j < n; j++) {
            int down = dp[i+1][j];
            int leftDiag = (j - 1 >= 0) ? dp[i+1][j-1] : -1e9;
            int rightDiag = (j + 1 < n) ? dp[i+1][j+1] : -1e9;

            dp[i][j] = matrix[i][j] + max({down, leftDiag, rightDiag});
        }
    }

    int maxi = INT_MIN;
    for(int j = 0; j < n; j++) {
        maxi = max(maxi, dp[0][j]);
    }

    return maxi;
}
```

### Complexity Analysis

**Time Complexity:** `O(m * n)` – Every cell is processed once.

**Space Complexity:** `O(m * n)` – 2D DP table.

## Space Optimization

### Approach

We only need the row below to compute current row.

So we can use two 1D arrays: `curr` and `next`.

At the end of each row computation, update `next = curr`.

### Code

```
int maxPathSum(vector<vector<int>>& matrix) {
    int m = matrix.size(), n = matrix[0].size();
    vector<int> next(matrix[m-1]);

    for(int i = m - 2; i >= 0; i--) {
        vector<int> curr(n);
        for(int j = 0; j < n; j++) {
            int down = next[j];
            int leftDiag = (j - 1 >= 0) ? next[j - 1] : -1e9;
            int rightDiag = (j + 1 < n) ? next[j + 1] : -1e9;

            curr[j] = matrix[i][j] + max({down, leftDiag, rightDiag});
        }
        next = curr;
    }

    return *max_element(next.begin(), next.end());
}
```

### Complexity Analysis

**Time Complexity:** `O(m * n)` – Each cell is processed.

**Space Complexity:** `O(n)` – Only two rows stored at a time.

---

THE END