<https://leetcode.com/problems/minimum-path-sum/description/>

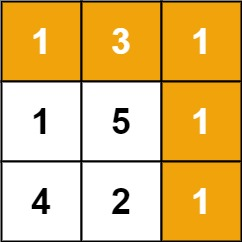
Given a

m x n

grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

**Note:** You can only move either down or right at any point in time.

**Example 1:**



Input: grid = [[1,3,1],[1,5,1],[4,2,1]]

Output: 7

Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

**Example 2:**

Input: grid = [[1,2,3],[4,5,6]]

Output: 12

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 200

0 <= grid[i][j] <= 200

**Attempt 1: 2023-08-22**

**Solution 1: Native DFS (10 min, TLE 25/61)**

class Solution {

    public int minPathSum(int[][] grid) {

        return helper(grid, 0, 0);

    }

    private int helper(int[][] grid, int i, int j) {

        if(i >= grid.length || j >= grid[0].length) {

            return Integer.MAX\_VALUE;

        }

        if(i == grid.length - 1 && j == grid[0].length - 1) {

            return grid[i][j];

        }

        int go\_down = helper(grid, i + 1, j);

        int go\_right = helper(grid, i, j + 1);

        return Math.min(go\_down, go\_right) + grid[i][j];

    }

}

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public int minPathSum(int[][] grid) {

        Integer[][] memo = new Integer[grid.length][grid[0].length];

        return helper(grid, 0, 0, memo);

    }

    private int helper(int[][] grid, int i, int j, Integer[][] memo) {

        if(i >= grid.length || j >= grid[0].length) {

            return Integer.MAX\_VALUE;

        }

        if(i == grid.length - 1 && j == grid[0].length - 1) {

            return grid[i][j];

        }

        if(memo[i][j] != null) {

            return memo[i][j];

        }

        int go\_down = helper(grid, i + 1, j, memo);

        int go\_right = helper(grid, i, j + 1, memo);

        return memo[i][j] = Math.min(go\_down, go\_right) + grid[i][j];

    }

}

**Solution 3: 2D DP (60 min)**

**Style 1: Initial 2D dp array as same size of 2D grid array**

**Note: Fully based on recursion "top" as (0, 0) and recursion "bottom" as (grid.length - 1, grid[0].length - 1) standard which exactly reflect the relation from Solution 1 Native DFS, in dp traversal, since compare to DFS solution, the dp solution will save much more time as no recursion stack push from "top" to "bottom" procedure like DFS, instead it directly process from "bottom" to "top", we should start with "bottom" and trace back to "top", the final solution will come out from dp[0][0]**

class Solution {

    public int minPathSum(int[][] grid) {

        int m = grid.length;

        int n = grid[0].length;

        int[][] dp = new int[m][n];

        dp[m - 1][n - 1] = grid[m - 1][n - 1];

        // The last column special handle as from bottom to top

        // only based on cell from downwards

        for(int i = m - 2; i >= 0; i--) {

            dp[i][n - 1] = dp[i + 1][n - 1] + grid[i][n - 1];

        }

        // The last row special handle as from right to left

        // only based on cell from rightwards

        for(int j = n - 2; j >= 0; j--) {

            dp[m - 1][j] = dp[m - 1][j + 1] + grid[m - 1][j];

        }

        for(int i = m - 2; i >= 0; i--) {

            for(int j = n - 2; j >= 0; j--) {

                dp[i][j] = Math.min(dp[i + 1][j], dp[i][j + 1]) + grid[i][j];

            }

        }

        return dp[0][0];

    }

}

**Refer to**

<https://leetcode.com/problems/minimum-path-sum/solutions/856314/sequential-thought-recursion-memo-dp-faster-easy-understanding/>

3. Dynamic Programming [ faster than 71.65% ] [ TC: (m\*n) ]

class Solution {

public:

    int minCost(vector<vector<int>> &cost,int m, int n,vector<vector<int>> dp) {

      dp[m-1][n-1]=cost[m-1][n-1];

      for(int i=n-2;i>=0;i--){

          dp[m-1][i]=dp[m-1][i+1]+cost[m-1][i];

      }

      for(int i=m-2;i>=0;i--){

          dp[i][n-1]=dp[i+1][n-1]+cost[i][n-1];

      }

      for(int i=m-2;i>=0;i--){

          for(int j=n-2;j>=0;j--){

              dp[i][j]=cost[i][j]+min(dp[i+1][j],dp[i][j+1]);

          }

      }

      return dp[0][0];

    }

**Style 2: Initial 2D dp array as one more column and row than original grid array**

**Note: Create one more row and one more column which helps uniform the formula make it even able to apply to last column and last row, even it strictly follow the conditions in Native DFS, still need to handle original last column and row specially, the difference between int[][] dp = new int[m][n] style is here the additional last column and row provide a way to do Math.min() as a uniform style as it always have a rightwards, downwards one to compare, which also mapping to base condition 1 in Native DFS**

class Solution {

    public int minPathSum(int[][] grid) {

        int m = grid.length;

        int n = grid[0].length;

        // Create one more row and one more column which helps

        // uniform the formula make it even able to apply to

        // last column and last row

        int[][] dp = new int[m + 1][n + 1];

        // But this condition still required which also mapping

        // to the DFS solution 2nd base condition

        dp[m - 1][n - 1] = grid[m - 1][n - 1];

        // Initialize additional last column

        for(int i = m; i >= 0; i--) {

            dp[i][n] = Integer.MAX\_VALUE;

        }

        // Initialize additional last row

        for(int j = n; j >= 0; j--) {

            dp[m][j] = Integer.MAX\_VALUE;

        }

        // e.g Until now for input {{1,3,1},{1,5,1},{4,2,1}}

        // 2D DP array is below:

        // [0,          0,          0,          2147483647]

        // [0,          0,          0,          2147483647]

        // [0,          0,          1,          2147483647]

        // [2147483647, 2147483647, 2147483647, 2147483647]

        // Still need to handle original last column and row specially, the

        // difference between int[][] dp = new int[m][n] style is here the

        // additional last column and row provide a way to do Math.min()

        // as a uniform style as it always have a rightwards, downwards one

        // to compare, which also mapping to base condition 1 in Native DFS

        // Speical handle for original last column, start with i = m - 2

        // because i = m - 1 plus n - 1 on 2nd dimension reserved for

        // dp[m - 1][n - 1] which setup as grid[m - 1][n - 1]

        for(int i = m - 2; i >= 0; i--) {

            dp[i][n - 1] = Math.min(dp[i + 1][n - 1], dp[i][n]) + grid[i][n - 1];

        }

        // Speical handle for original last row, start with j = n - 2

        // because j = n - 1 plus m - 1 on 1st dimension reserved for

        // dp[m - 1][n - 1] which setup as grid[m - 1][n - 1]

        for(int j = n - 2; j >= 0; j--) {

            dp[m - 1][j] = Math.min(dp[m - 1][j + 1], dp[m][j]) + grid[m - 1][j];

        }

        for(int i = m - 2; i >= 0; i--) {

            for(int j = n - 2; j >= 0; j--) {

                dp[i][j] = Math.min(dp[i + 1][j], dp[i][j + 1]) + grid[i][j];

            }

        }

        return dp[0][0];

    }

}

**Refer to**

<https://leetcode.wang/leetCode-64-Minimum-PathSum.html>

**解法二**

这里我们直接用 grid 覆盖存，不去 new 一个 n 的空间了。

public int minPathSum(int[][] grid) {

    int m = grid.length;

    int n = grid[0].length;

    //由于第一行和第一列不能用我们的递推式，所以单独更新

    //更新第一行的权值

    for (int i = 1; i < n; i++) {

        grid[0][i] = grid[0][i - 1] + grid[0][i];

    }

    //更新第一列的权值

    for (int i = 1; i < m; i++) {

        grid[i][0] = grid[i - 1][0] + grid[i][0];

    }

    //利用递推式更新其它的

    for (int i = 1; i < m; i++) {

        for (int j = 1; j < n; j++) {

            grid[i][j] = Math.min(grid[i][j - 1], grid[i - 1][j]) + grid[i][j];

        }

    }

    return grid[m - 1][n - 1];

}

时间复杂度：O（m \* n）。

空间复杂度：O（1）。

**总**

依旧是[62题](https://leetcode.windliang.cc/leetCode-62-Unique-Paths.html)的扩展，理解了 62 题的话，很快就写出来了。

**Attempt 2: 2025-06-15**

**Solution 1: Native DFS (TLE 25/66)**

**Style 1:**

class Solution {

    public int minPathSum(int[][] grid) {

        return helper(0, 0, grid);

    }

    private int helper(int i, int j, int[][] grid) {

        if (i == grid.length - 1 && j == grid[0].length - 1) {

            return grid[i][j];

        }

        int minSum = Integer.MAX\_VALUE;

        if (i + 1 < grid.length) {

            int downSum = helper(i + 1, j, grid);

            minSum = Math.min(minSum, downSum);

        }

        if (j + 1 < grid[0].length) {

            int rightSum = helper(i, j + 1, grid);

            minSum = Math.min(minSum, rightSum);

        }

        return grid[i][j] + minSum;

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Style 2:**

class Solution {

    public int minPathSum(int[][] grid) {

        return helper(grid, 0, 0);

    }

    private int helper(int[][] grid, int i, int j) {

        if(i == grid.length - 1 && j == grid[0].length - 1) {

            return grid[i][j];

        }

        if(i > grid.length - 1 || j > grid[0].length - 1) {

            return Integer.MAX\_VALUE;

        }

        int go\_down = helper(grid, i + 1, j);

        int go\_right = helper(grid, i, j + 1);

        return Math.min(go\_down, go\_right) + grid[i][j];

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Solution 2: Memoization (10 min)**

**Style 1:**

class Solution {

public int minPathSum(int[][] grid) {

// Create memoization table initialized with -1

int m = grid.length;

int n = grid[0].length;

int[][] memo = new int[m][n];

for (int i = 0; i < m; i++) {

Arrays.fill(memo[i], -1);

}

return helper(0, 0, grid, memo);

}

private int helper(int i, int j, int[][] grid, int[][] memo) {

// Base case: reached bottom-right corner

if (i == grid.length - 1 && j == grid[0].length - 1) {

return grid[i][j];

}

// Return memoized result if available

if (memo[i][j] != -1) {

return memo[i][j];

}

int minSum = Integer.MAX\_VALUE;

// Try moving down

if (i + 1 < grid.length) {

int downSum = helper(i + 1, j, grid, memo);

minSum = Math.min(minSum, downSum);

}

// Try moving right

if (j + 1 < grid[0].length) {

int rightSum = helper(i, j + 1, grid, memo);

minSum = Math.min(minSum, rightSum);

}

// Store result in memo table before returning

memo[i][j] = grid[i][j] + minSum;

return memo[i][j];

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2:**

class Solution {

public int minPathSum(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int[][] memo = new int[m][n];

// Initialize with -1 (or another value that can't occur)

for (int[] row : memo) Arrays.fill(row, -1);

return helper(grid, 0, 0, memo);

}

private int helper(int[][] grid, int i, int j, int[][] memo) {

if (i == grid.length - 1 && j == grid[0].length - 1) {

return grid[i][j];

}

if (i >= grid.length || j >= grid[0].length) {

return Integer.MAX\_VALUE;

}

if (memo[i][j] != -1) {

return memo[i][j];

}

int go\_down = helper(grid, i + 1, j, memo);

int go\_right = helper(grid, i, j + 1, memo);

memo[i][j] = Math.min(go\_down, go\_right) + grid[i][j];

return memo[i][j];

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 3: 2D DP (10 min)**

**Wrong Solution**

**Input**

**grid = [[1,3,1],[1,5,1],[4,2,1]]**

**Output = -2147483642**

**Expected = 7**

class Solution {

    public int minPathSum(int[][] grid) {

        int m = grid.length;

        int n = grid[0].length;

        int[][] dp = new int[m][n];

        dp[m - 1][n - 1] = grid[m - 1][n - 1];

        for(int i = m - 1; i >= 0; i--) {

            for(int j = n - 1; j >= 0; j--) {

                int go\_down = (i + 1 < m ? dp[i + 1][j] : Integer.MAX\_VALUE);

                int go\_right = (j + 1 < n ? dp[i][j + 1] : Integer.MAX\_VALUE);

                dp[i][j] = Math.min(go\_down, go\_right) + grid[i][j];

            }

        }

        return dp[0][0];

    }

}

**Refer to Deepseek**

The issue with your 2D DP solution is that you're **overwriting the destination cell** dp[m-1][n-1] after its initialization. Here's why it fails and how to fix it:

**Problem Analysis**

For input grid = [[1,3,1],[1,5,1],[4,2,1]]:

Your code returns -2147483642 (integer overflow)

Expected output is 7 (path: 1→1→4→2→1)

**The Bug**

**Double Counting**:

You initialize dp[2][2] = 1 (correct)

But then in the loop when i=2, j=2, you recompute it:

dp[2][2] = Math.min(MAX\_VALUE, MAX\_VALUE) + 1; // Results in MAX\_VALUE + 1

This causes integer overflow (wraps around to negative)

**Incorrect Value Propagation**:

The overflow at dp[2][2] propagates wrong values upward

**Fixed Solution**

class Solution {

public int minPathSum(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int[][] dp = new int[m][n];

// Initialize destination cell

dp[m-1][n-1] = grid[m-1][n-1];

// Fill DP table from bottom-right to top-left

for (int i = m-1; i >= 0; i--) {

for (int j = n-1; j >= 0; j--) {

// Skip destination cell (already initialized)

if (i == m-1 && j == n-1) continue;

// Get values from below and right

int down = (i + 1 < m) ? dp[i+1][j] : Integer.MAX\_VALUE;

int right = (j + 1 < n) ? dp[i][j+1] : Integer.MAX\_VALUE;

// Current cell value + minimum path

dp[i][j] = grid[i][j] + Math.min(down, right);

}

}

return dp[0][0];

}

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n)

**Key Fixes**

**Skip Destination Cell**:

Uncommented the continue statement for (m-1, n-1)

Prevents recomputation of the destination cell

**Proper Initialization**:

Only initialize dp[m-1][n-1] once

All other cells are computed based on their neighbors

**Why This Works**

Correctly computes minimum path sums without overflow

Maintains the same logic as your DFS solution:

Uses MAX\_VALUE for out-of-bounds cases

Takes minimum of down/right paths

Processes cells in the same order (bottom-up, right-to-left)

**Edge Case Handling**

The fixed solution properly handles:

Grids of any size

All positive numbers (no overflow)

The destination cell being computed only once

This will now correctly return 7 for the given input.

**Solution 4: 2 Rows DP (10 min)**

class Solution {

    public int minPathSum(int[][] grid) {

        int m = grid.length;

        int n = grid[0].length;

        int[] nextRow = new int[n];

        int[] currRow = new int[n];

        // Base case

        nextRow[n - 1] = grid[m - 1][n - 1];

        // Initialize nextRow

        for(int j = n - 2; j >= 0; j--) {

            nextRow[j] = grid[m - 1][j] + nextRow[j + 1];

        }

        // Process rows from bottom to top

        for(int i = m - 2; i >= 0; i--) {

            // Initialize currRow's last column

            currRow[n - 1] = grid[i][n - 1] + nextRow[n - 1];

            // Process columns from right to left

            for(int j = n - 2; j >= 0; j--) {

                currRow[j] = grid[i][j] + Math.min(currRow[j + 1], nextRow[j]);

            }

            // Swap rows for next iteration (next row value update

            // to current row since in next iteration, current row

            // value used as next row)

            int[] tmp = nextRow;

            nextRow = currRow;

            currRow = tmp;

        }

        return nextRow[0];

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(2 \* n)

**Refer to Deepseek**

Here's the space-optimized 2-row DP solution based on your fixed 2D DP approach:

class Solution {

public int minPathSum(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int[] currentRow = new int[n];

int[] nextRow = new int[n];

// Initialize bottom row

nextRow[n-1] = grid[m-1][n-1];

for (int j = n-2; j >= 0; j--) {

nextRow[j] = grid[m-1][j] + nextRow[j+1];

}

// Process rows from bottom to top

for (int i = m-2; i >= 0; i--) {

// Initialize current row's last column

currentRow[n-1] = grid[i][n-1] + nextRow[n-1];

// Process columns from right to left

for (int j = n-2; j >= 0; j--) {

currentRow[j] = grid[i][j] + Math.min(currentRow[j+1], nextRow[j]);

}

// Swap rows for next iteration

int[] temp = nextRow;

nextRow = currentRow;

currentRow = temp;

}

return nextRow[0];

}

}

**Key Features:**

**Space Optimization**:

Uses only two 1D arrays (currentRow and nextRow) instead of full 2D grid

Reduces space complexity from O(mn) to O(n)

**Correct Initialization**:

Properly initializes the bottom row first

Handles the destination cell correctly without recomputation

**Processing Order**:

Processes rows from bottom to top (m-2 to 0)

Processes columns from right to left (n-2 to 0)

Matches the computation order of the 2D DP solution

**Boundary Handling**:

Last column of each row is handled separately

Naturally handles grid boundaries without MAX\_VALUE checks

**Why This Works:**

**Dependency Preservation**:

nextRow stores values from the row below

currentRow[j+1] gives the right neighbor in current row

This matches the dependencies in the 2D version:

dp[i][j] depends on dp[i+1][j] (below) and dp[i][j+1] (right)

**Correct State Transition**:

currentRow[j] = grid[i][j] + Math.min(currentRow[j+1], nextRow[j])

Exactly mirrors the 2D DP logic

**Efficiency**:

Each cell is computed exactly once

Only maintains two rows at any time

**Example Walkthrough:**

For input grid = [[1,3,1],[1,5,1],[4,2,1]]:

Initialize nextRow with bottom row values: [7, 3, 1]

Process middle row:

currentRow becomes [8, 5, 2]

Process top row:

currentRow becomes [7, 6, 3]

Final result is nextRow[0] = 7

This matches the expected minimum path sum (1→1→4→2→1 = 7) while using only O(n) space.

**Solution 5: 1D DP (10 min)**

class Solution {

    public int minPathSum(int[][] grid) {

        int m = grid.length;

        int n = grid[0].length;

        int[] dp = new int[n];

        // Base case

        dp[n - 1] = grid[m - 1][n - 1];

        // Initialize last row

        for(int j = n - 2; j >= 0; j--) {

            dp[j] = grid[m - 1][j] + dp[j + 1];

        }

        // Process rows from bottom to top

        for(int i = m - 2; i >= 0; i--) {

            // Initialize currRow's last column

// The 'dp[n - 1]' on equation left side means the previous

// row's last column value (before update), the 'dp[n - 1]'

// on equation right side means the current row's last column

// value (after update), same statement, different meaning

            dp[n - 1] = grid[i][n - 1] + dp[n - 1];

            // Process columns from right to left

            for(int j = n - 2; j >= 0; j--) {

                dp[j] = grid[i][j] + Math.min(dp[j + 1], dp[j]);

            }

        }

        return dp[0];

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(n)

**Refer to Deepseek**

Here's the most space-efficient version using just a single 1D array:

class Solution {

public int minPathSum(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int[] dp = new int[n];

// Initialize last row

dp[n-1] = grid[m-1][n-1];

for (int j = n-2; j >= 0; j--) {

dp[j] = grid[m-1][j] + dp[j+1];

}

// Process rows from bottom to top

for (int i = m-2; i >= 0; i--) {

// Update last column in current row

dp[n-1] += grid[i][n-1];

// Process columns from right to left

for (int j = n-2; j >= 0; j--) {

dp[j] = grid[i][j] + Math.min(dp[j], dp[j+1]);

}

}

return dp[0];

}

}

**Key Insights:**

**Single Array Usage**:

dp[j] serves dual purpose:

Before update: stores value from below (previous row)

After update: stores value for current cell

**Right-to-Left Processing**:

Essential for correct dependencies

dp[j+1] is already updated for current row

dp[j] (before update) contains value from row below

**Efficient Updates**:

Last column handled separately (only depends on cell below)

Other columns: dp[j] = grid[i][j] + min(below, right)

**Time/Space Complexity:**

**Time**: O(mn) - processes each cell once

**Space**: O(n) - single array of size n

This maintains the same computation order as the 2D and 2-row solutions while using minimal space. The right-to-left processing is crucial for correct results.

**Refer to**

[L62.Unique Paths (Ref.L63,L64,L174,L2087,L2304,L2400)](note://E33F0B728ABE4CCF8B7E554A219CF1EA)

[L63.Unique Paths II (Ref.L62)](note://37FCA39A01734A37B2CF4CFACA8170E0)