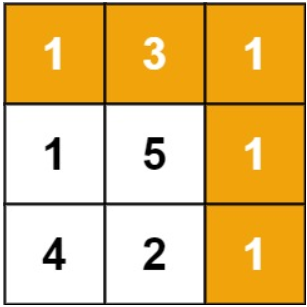
<https://leetcode.com/problems/minimum-path-sum/description/>

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。