Summary

We have to find the minimum sum of numbers over a path from the top left to the bottom right of the given matrix .

Solution

Approach 1: Brute Force

The Brute Force approach involves recursion. For each element, we consider two paths, rightwards and downwards and find the minimum sum out of those two. It specifies whether we need to take a right step or downward step to minimize the sum.

\mathrm{cost}(i, j)=\mathrm{grid}[i][j] + \min \big(\mathrm{cost}(i+1, j), \mathrm{cost}(i, j+1) \big)cost(*i*,*j*)=grid[*i*][*j*]+min(cost(*i*+1,*j*),cost(*i*,*j*+1))

**Complexity Analysis**

* Time complexity : O\big(2^{m+n}\big)*O*(2*m*+*n*). For every move, we have atmost 2 options.
* Space complexity : O(m+n)*O*(*m*+*n*). Recursion of depth m+n*m*+*n*.

public class Solution {

public int calculate(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];

return grid[i][j] + Math.min(calculate(grid, i + 1, j), calculate(grid, i, j + 1));

}

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

return calculate(grid, 0, 0);

}

}

Approach 2: Dynamic Programming 2D

**Algorithm**

We use an extra matrix dp*dp* of the same size as the original matrix. In this matrix, dp(i, j)*dp*(*i*,*j*) represents the minimum sum of the path from the index (i, j)(*i*,*j*) to the bottom rightmost element. We start by initializing the bottom rightmost element of dp*dp* as the last element of the given matrix. Then for each element starting from the bottom right, we traverse backwards and fill in the matrix with the required minimum sums. Now, we need to note that at every element, we can move either rightwards or downwards. Therefore, for filling in the minimum sum, we use the equation:

dp(i, j)= \mathrm{grid}(i,j)+\min\big(dp(i+1,j),dp(i,j+1)\big)*dp*(*i*,*j*)=grid(*i*,*j*)+min(*dp*(*i*+1,*j*),*dp*(*i*,*j*+1))

taking care of the boundary conditions.

The following figure illustrates the process:

public class Solution {

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

int[][] dp = new int[grid.length][grid[0].length];

for (int i = grid.length - 1; i >= 0; i--) {

for (int j = grid[0].length - 1; j >= 0; j--) {

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

dp[i][j] = grid[i][j] + dp[i][j + 1];

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

dp[i][j] = grid[i][j] + dp[i + 1][j];

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

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

else

dp[i][j] = grid[i][j];

}

}

return dp[0][0];

}

}

**Complexity Analysis**

* Time complexity : O(mn)*O*(*mn*). We traverse the entire matrix once.
* Space complexity : O(mn)*O*(*mn*). Another matrix of the same size is used.

Approach 3: Dynamic Programming 1D

**Algorithm**

In the previous case, instead of using a 2D matrix for dp, we can do the same work using a dp*dp* array of the row size, since for making the current entry all we need is the dp entry for the bottom and the right element. Thus, we start by initializing only the last element of the array as the last element of the given matrix. The last entry is the bottom rightmost element of the given matrix. Then, we start moving towards the left and update the entry dp(j)*dp*(*j*) as:

dp(j)=\mathrm{grid}(i,j)+\min\big(dp(j),dp(j+1)\big)*dp*(*j*)=grid(*i*,*j*)+min(*dp*(*j*),*dp*(*j*+1))

We repeat the same process for every row as we move upwards. At the end dp(0)*dp*(0) gives the required minimum sum.

**Complexity Analysis**

* Time complexity : O(mn)*O*(*mn*). We traverse the entire matrix once.
* Space complexity : O(n)*O*(*n*). Another array of row size is used.

public class Solution {

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

int[] dp = new int[grid[0].length];

for (int i = grid.length - 1; i >= 0; i--) {

for (int j = grid[0].length - 1; j >= 0; j--) {

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

dp[j] = grid[i][j] + dp[j + 1];

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

dp[j] = grid[i][j] + dp[j];

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

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

else

dp[j] = grid[i][j];

}

}

return dp[0];

}

}

Approach 4: Dynamic Programming (Without Extra Space)

**Algorithm**

This approach is same as [Approach 2](https://leetcode.com/problems/minimum-path-sum/solution/#approach-2-dynamic-programming-2d), with a slight difference. Instead of using another dp*dp* matrix. We can store the minimum sums in the original matrix itself, since we need not retain the original matrix here. Thus, the governing equation now becomes:

\mathrm{grid}(i, j)=\mathrm{grid}(i,j)+\min \big(\mathrm{grid}(i+1,j), \mathrm{grid}(i,j+1)\big)grid(*i*,*j*)=grid(*i*,*j*)+min(grid(*i*+1,*j*),grid(*i*,*j*+1))

**Complexity Analysis**

* Time complexity : O(mn)*O*(*mn*). We traverse the entire matrix once.
* Space complexity : O(1)*O*(1). No extra space is used.

public class Solution {

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

for (int i = grid.length - 1; i >= 0; i--) {

for (int j = grid[0].length - 1; j >= 0; j--) {

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

grid[i][j] = grid[i][j] + grid[i][j + 1];

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

grid[i][j] = grid[i][j] + grid[i + 1][j];

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

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

}

}

return grid[0][0];

}

}