64. Minimum Path Sum

Dec. 8, 2016 | 60.5K views

Average Rating: 4.31 (26 votes)

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:

```
Input:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
Output: 7
Explanation: Because the path 1→3→1→1→1 minimizes the sum.
```

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.

$$\mathrm{cost}(i,j) = \mathrm{grid}[i][j] + \min\left(\mathrm{cost}(i+1,j), \mathrm{cost}(i,j+1)\right)$$

Java

```java
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);
    }
}
```

Complexity Analysis

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

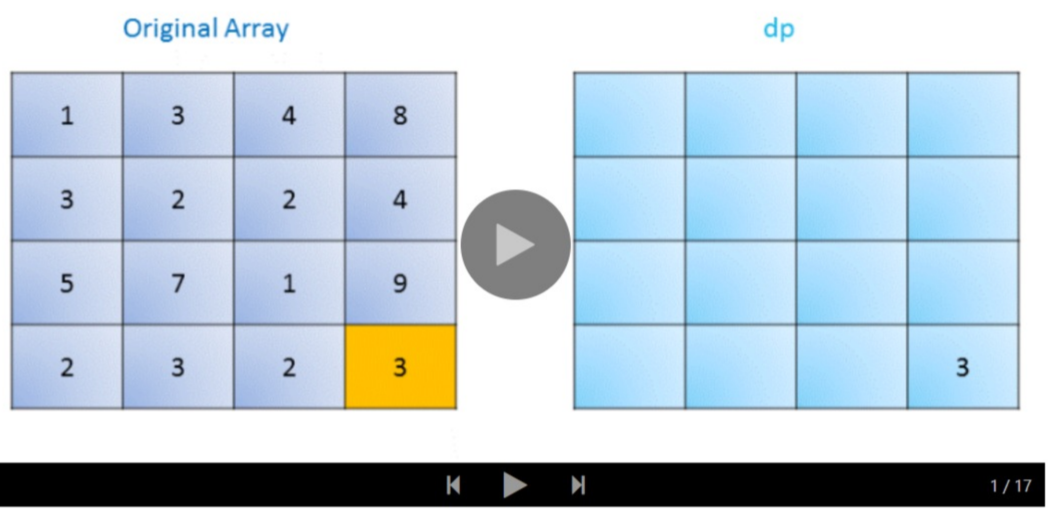
Approach 2: Dynamic Programming 2D

Algorithm

We use an extra matrix dp of the same size as the original matrix. In this matrix, $dp(i,j)$ represents the minimum sum of the path from the index (i,j) to the bottom rightmost element. We start by initializing the bottom rightmost element of 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:

$$dp(i,j) = \mathrm{grid}(i,j) + \min\left(dp(i+1,j), dp(i,j+1)\right)$$

taking care of the boundary conditions.



Java

```java
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 : $O(mn)$. We traverse the entire matrix once.
- Space complexity : $O(mn)$. 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 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 $dp(j)$ as:

$$dp(j) = \mathrm{grid}(i,j) + \min\left(dp(j), dp(j+1)\right)$$

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

Java

```java
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];
    }
}

```

Complexity Analysis

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

Approach 4: Dynamic Programming (Without Extra Space)

Algorithm

This approach is same as Approach 2, with a slight difference. Instead of using another 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:

$$\mathrm{grid}(i,j) = \mathrm{grid}(i,j) + \min\left(\mathrm{grid}(i+1,j), \mathrm{grid}(i,j+1)\right)$$

Java

```java
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];
    }
}
```

Complexity Analysis

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

Rate this article:

Previous Next

Comments: 21 Sort By

Type comment here... (Markdown is supported)

Preview Post

s961206 ★734 July 10, 2019 8:48 AM

Approach 4 can be more readable:

```
int m = grid.length, n = grid[0].length;
for(int i = 1; i < m; ++i) grid[i][0] += grid[i - 1][0];
```

Read More

66 Share Reply

SHOW 3 REPLIES

SherMM ★57 March 10, 2018 9:08 PM Report

Dijkstra might be overkill here, but it also works.

22 Share Reply

SHOW 5 REPLIES

survive ★97 January 4, 2020 2:50 PM

I don't think modify the original matrix is a good idea

13 Share Reply

ktmbdev ★203 July 4, 2019 11:01 PM

The reason why DP works here (but not in an actual shortest distance problem) is because we can only move right and down through the matrix. If we can move in all 4 directions, DP would give the wrong answer.

12 Share Reply

SHOW 5 REPLIES

ping_pong ★811 September 16, 2019 7:32 AM

Shouldn't the complexity of first approach be O(2^{M*N}). Why (M+N)?

6 Share Reply

SHOW 2 REPLIES

csgod ★7 March 20, 2020 8:41 AM

why're we starting at the bottom right corner?

3 Share Reply

SHOW 1 REPLY

madno ★302 February 10, 2020 6:07 PM

Interesting that we have the restriction on the input values of `non-negative numbers`.
The DP solution apparently works for negative numbers as well.

2 Share Reply

sys526939916 ★7 July 6, 2018 10:11 AM

are we allowed to move up or move to the left in this problem?

2 Share Reply

SHOW 5 REPLIES

alx75 ★2 January 5, 2020 8:55 PM

Here is my solution to approach 3. It think it's easier to read IMO :

```
class Solution {
    public int minPathSum(int[][] grid) {
```

Read More

1 Share Reply

donpachii ★5 April 14, 2018 2:46 AM

@adarsh4321.dsp it's O(1) space because we don't introduce any extra memory ourselves in the function. We're operating on a "constant" size multidimensional array in the scope of our function. If inputs are read only, then the only option is to make our own array and then space complexity grows to m*n or n depending on how you implement

1 Share Reply

‹ 1 2 3 ›