

Minimum Path Sum

Minimum Path Sum In a Grid (DP 10)

In this article, we will solve the most asked coding interview problem: Minimum Path Sum In a Grid.

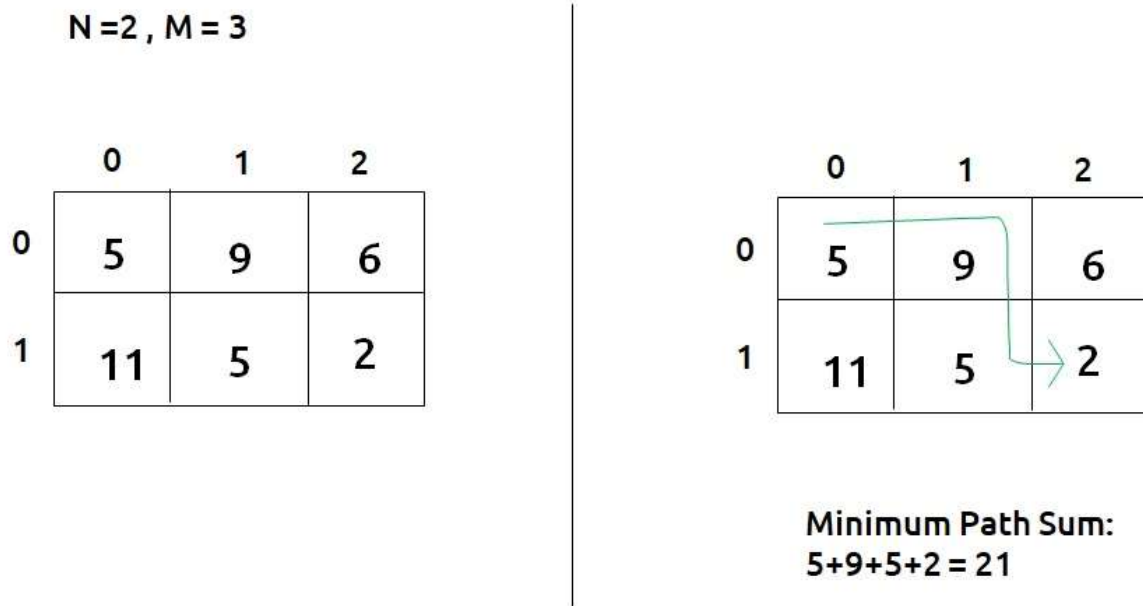
Problem Link: [Minimum Path Sum in A Grid](#)

Problem Description:

We are given an “N*M” matrix of integers. We need to find a path from the top-left corner to the bottom-right corner of the matrix, such that there is a minimum cost path that we select.

At every cell, we can move in only two directions: right and bottom. The cost of a path is given as the sum of values of cells of the given matrix.

Example:



Disclaimer. Don't jump directly to the solution, try it out yourself first.

Pre-req: [Grid Unique Path 2](#)

Solution :

This question is a slight modification of the question discussed in [Grid Unique Path 2](#). In the previous problem, there were obstacles whereas this problem has cost associated with a cell and we need to return the minimum cost path.

Why a Greedy Solution doesn't work?

As we have to return the minimum path sum, the first approach that comes to our mind is to take a greedy approach and always form a path by locally choosing the cheaper option.

This approach will not give us the correct answer. Let us look at this example to understand:

N = 3 , M = 3

	0	1	2
0	10	8	2
1	10	5	100
2	1	1	2

At every cell, we have two choices: to move right and move down. Our ultimate aim is to provide a path that provides us the least path sum. Therefore at every cell, we will make the choice to move which costs are less.

N = 3, M = 3

	0	1	2
0	10	8	2
1	10	5	100
2	1	1	2

Greedy Solution

$$10+8+2+100+2 = 112$$

N = 3, M = 3

	0	1	2
0	10	8	2
1	10	5	100
2	1	1	2

Non-Greedy Solution

$$10+10+1+1+2 = 24$$

- Figure on the left gives us a greedy solution, where we move by taking the local best choice.
- Figure on the right gives us a non-greedy solution.

We can clearly see the problem with the greedy solution. Whenever we are making a local choice, we may tend to choose a path that may cost us way more later.

Therefore, the other alternative left to us is to generate all the possible paths and see which is the path with the minimum path sum. To generate all paths we will use **recursion**.

Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in the [Dynamic Programming Introduction](#).

Step 1: Express the problem in terms of indexes.

We are given two variables N and M, representing the dimensions of the matrix.

We can define the function with two parameters i and j, where i and j represent the row and column of the matrix.

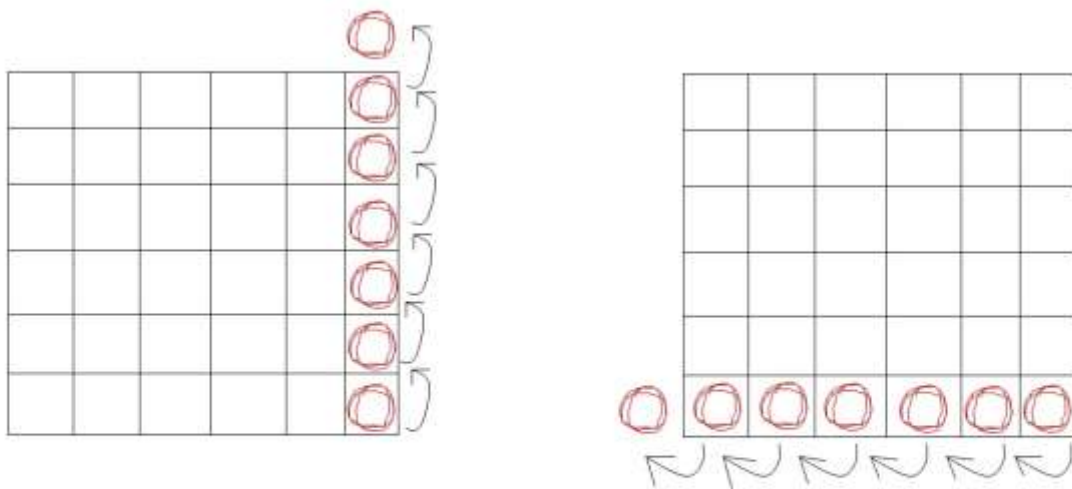
**f(i,j) -> Minimum path sum from
matrix[0,0] to matrix[i][j].**

We will be doing a top-down recursion, i.e we will move from the cell[M-1][N-1] and try to find our way to the cell[0][0]. Therefore at every index, we will try to move up and towards the left.

Base Case:

There will be three base cases:

- When $i=0$ and $j=0$, that is we have reached the destination so we can add to path the current cell value, hence we return $\text{mat}[0][0]$.
- When $i<0$ or $j<0$, it means that we have crossed the boundary of the matrix and we don't want to find a path from here, so we return a very large number(say, $1e9$) so that this path is rejected by the calling function.



As the function call crosses the boundary of the matrix, we return a large value(say 10^9) so that this path is not considered by the last calling function (as we will be returning the minimum path)

The pseudocode till this step will be:

```
f(i,j) {  
    if( i==0 && j==0) return mat[0][0]  
  
    if( i<0 || j<0) return 1e9  
  
}
```

Step 2: Try out all possible choices at a given index.

As we are writing a top-down recursion, at every index we have two choices, one to go up(↑) and the other to go left(←). To go upwards, we will reduce i by 1, and move towards left we will reduce j by 1.

Now when we get our answer for the recursive call (f(i-1,j) or f(i,j-1)), we need to also add the current cell value to it as we have to include it too for the current path sum.

```
f(i,j) {  
    if( i==0 && j==0) return mat[0][0]  
    if( i<0 || j<0) return 1e9  
    up = mat[i][j] + f(i-1,j)  
    left = mat[i][j] + f(i,j-1)  
}
```

Step 3: Take the maximum of all choices

As we have to find the **minimum path sum** of all the possible unique paths, we will return the **minimum** of the choices(up and left)

The final pseudocode after steps 1, 2, and 3:

```

f(i,j) {
    if( i==0 && j==0) return mat[0][0]

    if( i<0 || j<0) return 1e9

    up = mat[i][j] + f(i-1,j)
    left = mat[i][j] + f(i,j-1)

    return min(up,left)
}

```

Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [n][m]
2. Whenever we want to find the answer of a particular row and column (say $f(i,j)$), we first check whether the answer is already calculated using the dp array (i.e $dp[i][j] \neq -1$). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set $dp[i][j]$ to the solution we get.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>

using namespace std;

int minSumPathUtil(int i, int j, vector<vector<int>> &matrix, vector<vector<int>>
&dp)
{

```

```

    if(i==0 && j == 0)
        return matrix[0][0];
    if(i<0 || j<0)
        return 1e9;
    if(dp[i][j]!=-1) return dp[i][j];

    int up = matrix[i][j]+minSumPathUtil(i-1,j,matrix,dp);
    int left = matrix[i][j]+minSumPathUtil(i,j-1,matrix,dp);

    return dp[i][j] = min(up,left);
}

int minSumPath(int n, int m, vector<vector<int> > &matrix){
    vector<vector<int> > dp(n,vector<int>(m,-1));
    return minSumPathUtil(n-1,m-1,matrix,dp);
}

int main() {

    vector<vector<int> > matrix{{5,9,6},
                                {11,5,2}};

    int n = matrix.size();
    int m = matrix[0].size();

    cout<<minSumPath(n,m,matrix);
}

```

Output: 21

Time Complexity: $O(N*M)$

Reason: At max, there will be $N*M$ calls of recursion.

Space Complexity: $O((M-1)+(N-1)) + O(N*M)$

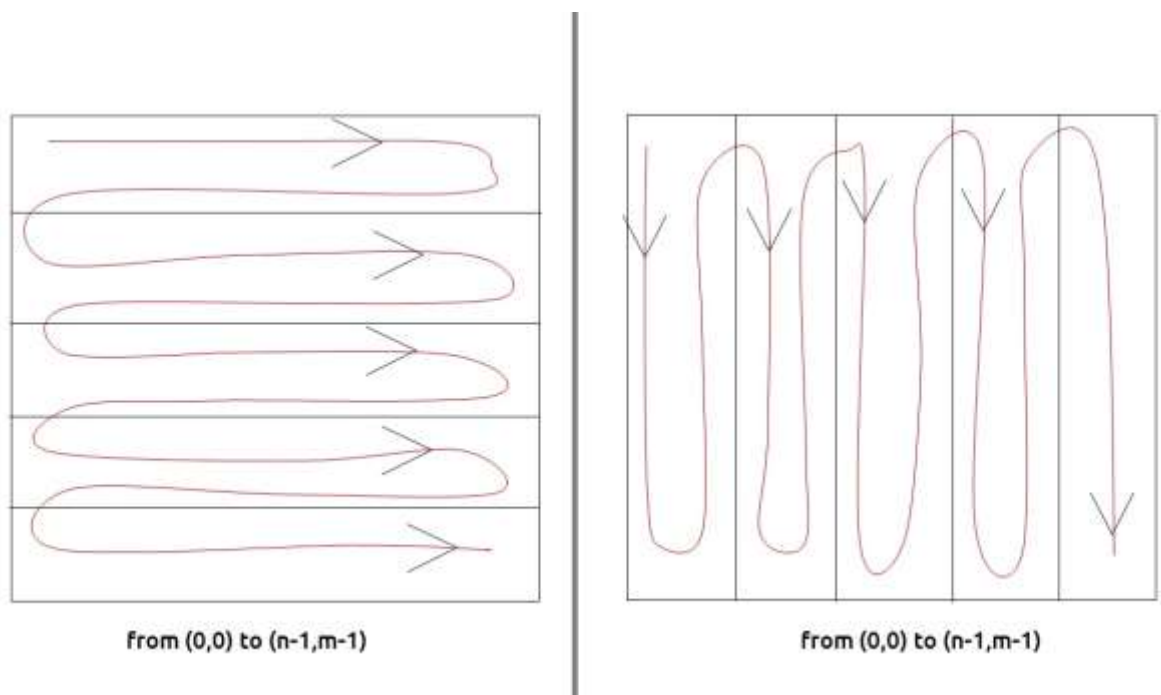
Reason: We are using a recursion stack space: $O((M-1)+(N-1))$, here $(M-1)+(N-1)$ is the path length and an external DP Array of size ' $N*M$ '.

Steps to convert Recursive Solution to Tabulation one.

Tabulation is the bottom-up approach, which means we will go from the base case to the main problem.

The steps to convert to the tabular solution are given below:

- Declare a `dp[]` array of size `[n][m]`.
- First initialize the base condition values, i.e `dp[0][0] = matrix[0][0]`
- Our answer should get stored in `dp[n-1][m-1]`. We want to move from `(0,0)` to `(n-1,m-1)`. But we can't move arbitrarily, we should move such that at a particular `i` and `j`, we have all the values required to compute `dp[i][j]`.
- If we see the memoized code, values required for `dp[i][j]` are: `dp[i-1][j]` and `dp[i][j-1]`. So we only use the previous row and column value.
- We have already filled the top-left corner (`i=0` and `j=0`), if we move in any of the two following ways(given below), at every cell we do have all the previous values required to compute its value.



- We can use two nested loops to have this traversal
- Whenever `i>0` , `j>0`, we will simply mark `dp[i][j] = matrix[i][j] + min(dp[i-1][j],dp[i][j-1])`, according to our recursive relation.
- When `i=0` or `j=0`, we add to up(or left) `1e9`, so that this path can be rejected.

Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int minSumPath(int n, int m, vector<vector<int> > &matrix){
```



```

vector<vector<int> > dp(n,vector<int>(m,0));
for(int i=0; i<n ; i++){
    for(int j=0; j<m; j++){
        if(i==0 && j==0) dp[i][j] = matrix[i][j];
        else{

            int up = matrix[i][j];
            if(i>0) up += dp[i-1][j];
            else up += 1e9;

            int left = matrix[i][j];
            if(j>0) left+=dp[i][j-1];
            else left += 1e9;

            dp[i][j] = min(up,left);
        }
    }
}

return dp[n-1][m-1];
}

int main() {

    vector<vector<int> > matrix{{5,9,6},
                                {11,5,2}};

    int n = matrix.size();
    int m = matrix[0].size();

    cout<<minSumPath(n,m,matrix);
}

```

Output:

21

Time Complexity: $O(N*M)$

Reason: There are two nested loops

Space Complexity: $O(N*M)$

Reason: We are using an external array of size ' $N*M$ '.

Part 3: Space Optimization

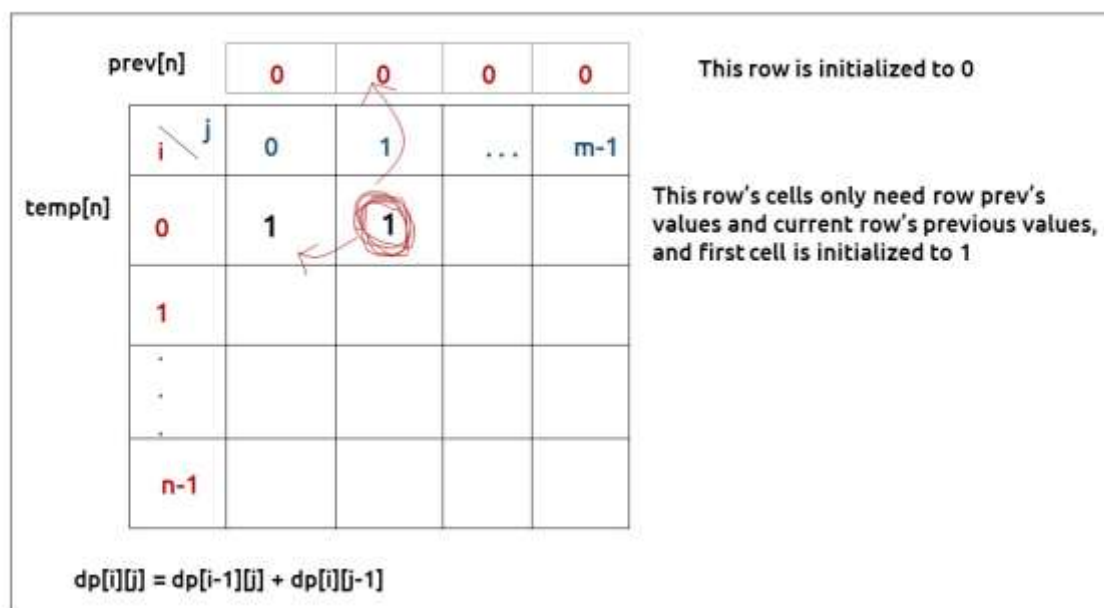
If we closely look the relation,

$$dp[i][j] = matrix[i][j] + \min(dp[i-1][j] + dp[i][j-1]))$$

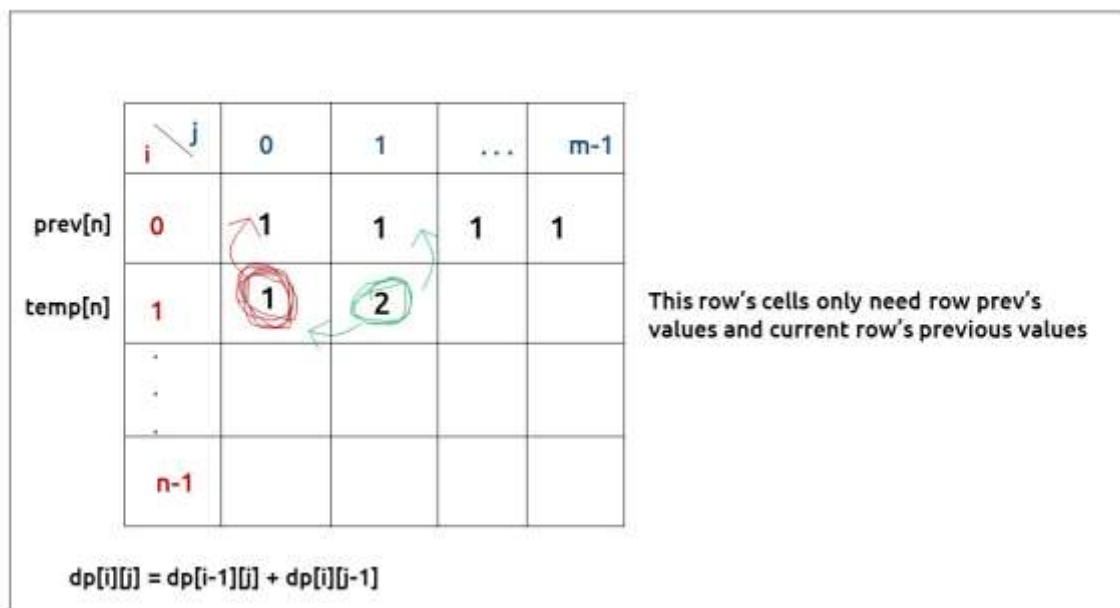
We see that we only need the previous row and column, in order to calculate $dp[i][j]$. Therefore we can space optimize it.

Initially, we can take a dummy row (say prev) and initialize it as 0.

Now the current row(say temp) **only needs the** previous row value and the current row's value in order to calculate $dp[i][j]$.



At the next step, the temp array becomes the prev of the next step and using its values we can still calculate the next row's values.



At last prev[n-1] will give us the required answer.

Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int minSumPath(int n, int m, vector<vector<int> > &matrix){
    vector<int> prev(m,0);
    for(int i=0; i<n ; i++){
        vector<int> temp(m,0);
        for(int j=0; j<m; j++){
            if(i==0 && j==0) temp[j] = matrix[i][j];
            else{
                int up = matrix[i][j];
                if(i>0) up += prev[j];
                else up += 1e9;

                int left = matrix[i][j];
                if(j>0) left+=temp[j-1];
                else left += 1e9;
            }
        }
    }
}
```

```

        temp[j] = min(up,left);
    }
}
prev=temp;
}

return prev[m-1];
}

int main() {

    vector<vector<int> > matrix{{5,9,6},
                                {11,5,2}};

    int n = matrix.size();
    int m = matrix[0].size();

    cout<<minSumPath(n,m,matrix);
}

```

Output:

21

Time Complexity: $O(M*N)$

Reason: There are two nested loops

Space Complexity: $O(N)$

Reason: We are using an external array of size 'N' to store only one row.