# Minimum Path Sum

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.

## Solution 1

This is a typical DP problem. Suppose the minimum path sum of arriving at point `(i, j)` is `S[i][j]`, then the state equation is `S[i][j] = min(S[i - 1][j], S[i][j - 1]) + grid[i][j]`.

Well, some boundary conditions need to be handled. The boundary conditions happen on the topmost row (`S[i - 1][j]` does not exist) and the leftmost column (`S[i][j - 1]` does not exist). Suppose `grid` is like `[1, 1, 1, 1]`, then the minimum sum to arrive at each point is simply an accumulation of previous points and the result is `[1, 2, 3, 4]`.

Now we can write down the following (unoptimized) code.

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<vector<int> > sum(m, vector<int>(n, grid[0][0]));
        for (int i = 1; i < m; i++)
            sum[i][0] = sum[i - 1][0] + grid[i][0];
        for (int j = 1; j < n; j++)
            sum[0][j] = sum[0][j - 1] + grid[0][j];
        for (int i = 1; i < m; i++)
            for (int j = 1; j < n; j++)
                sum[i][j]  = min(sum[i - 1][j], sum[i][j - 1]) + grid[i][j];
        return sum[m - 1][n - 1];
    }
};
```

As can be seen, each time when we update `sum[i][j]`, we only need `sum[i - 1][j]` (at the current column) and `sum[i][j - 1]` (at the left column). So we need not maintain the full `m*n` matrix. Maintaining two columns is enough and now we have the following code.

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<int> pre(m, grid[0][0]);
        vector<int> cur(m, 0);
        for (int i = 1; i < m; i++)
            pre[i] = pre[i - 1] + grid[i][0];
        for (int j = 1; j < n; j++) {
            cur[0] = pre[0] + grid[0][j];
            for (int i = 1; i < m; i++)
                cur[i] = min(cur[i - 1], pre[i]) + grid[i][j];
            swap(pre, cur);
        }
        return pre[m - 1];
    }
};
```

Further inspecting the above code, it can be seen that maintaining `pre` is for recovering `pre[i]`, which is simply `cur[i]` before its update. So it is enough to use only one vector. Now the space is further optimized and the code also gets shorter.

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<int> cur(m, grid[0][0]);
        for (int i = 1; i < m; i++)
            cur[i] = cur[i - 1] + grid[i][0];
        for (int j = 1; j < n; j++) {
            cur[0] += grid[0][j];
            for (int i = 1; i < m; i++)
                cur[i] = min(cur[i - 1], cur[i]) + grid[i][j];
        }
        return cur[m - 1];
    }
};
```

written by jianchao.li.fighter original link here

## Solution 2

```java
public int minPathSum(int[][] grid) {
    int m = grid.length;// row
    int n = grid[0].length; // column
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (i == 0 && j != 0) {
                grid[i][j] = grid[i][j] + grid[i][j - 1];
            } else if (i != 0 && j == 0) {
                grid[i][j] = grid[i][j] + grid[i - 1][j];
            } else if (i == 0 && j == 0) {
                grid[i][j] = grid[i][j];
            } else {
                grid[i][j] = Math.min(grid[i][j - 1], grid[i - 1][j])
                        + grid[i][j];
            }
        }
    }

    return grid[m - 1][n - 1];
}
```

written by wdj0xda original link here

## Solution 3

You can only reach a cell by going from its left or top neighbor.

```
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        if(!grid.size())return 0;
        const int rows=grid.size(),cols=grid[0].size();
        // r[i] == min path sum to previous row's column i.
        vector<int> r(cols,0);
        int i,j;
        r[0]=grid[0][0];
        for(j=1;j<cols;j++){
            r[j]=grid[0][j]+r[j-1];
        }
        for(i=1;i<rows;i++){
            r[0]+=grid[i][0];
            for(j=1;j<cols;j++){
                r[j]=min(r[j-1],r[j])+grid[i][j];
            }
        }
        return r[cols-1];
    }
};
```

written by lucastan original link here