62. Unique Paths solution - Dynamic Programming approach

This problem can be solved with use of Dynamic Programming:

- We make DP matrix and than for each cell in the grid we have number of unique paths from starting point to that cell stored in DP matrix.
- Base cases are cells in the top row and cells in the most left column. All that cells can be reached from starting cell in only one way (starting cell can be obviously reached only in one way, other cells in top row can be reached from the starting cell only if we move to the right in every step and cells in the most left column can be reached from the starting cell only if we move down in every step)
- We fill DP matrix from left to right and from top to bottom, so whenever we reach new cell we already have number of different ways of reaching predecessor cells calculated and stored in DP matrix.
- Solution for our problem will be stored in bottom right corner of DP matrix i.e. $DP[n-1][m-1]$ (since matrix is 0 - indexed)

Algorithm:

1. Fill DP matrix for base cases:
   $DP[0][i] = 1, \quad 0 \leq i < m$
   $DP[i][0] = 1, \quad 0 \leq i < n$
2. Fill the rest of DP matrix from left to right and from top to bottom. Every cell can be reached from the cell on its left, or from the cell on its top. That means that number of different ways to reach some cell is equal to the sum of number of different ways to reach cell to its left and number of different ways to reach cell on its top. So we calculate:
   $DP[i][j] = DP[i][j-1] + DP[i-1][j], \quad 0 < i < n, \quad 0 < j < m$
3. Return number of unique paths from top-left corner to bottom-right corner, which is value that we have stored in bottom-right corner of DP matrix i.e. $DP[n-1][m-1]$

Complexity analysis:

- **Time complexity:** We process each cell just once, and for each cell we need $O(1)$ time to calculate number of different ways to reach it. As we have $n \times m$ cells in the grid, that gives our solution overall time complexity of: $O(n \times m)$
- **Space complexity:** We used DP matrix, which has n rows and m columns, so that gives us overall memory complexity of: $O(n \times m)$

C++ code:

```
class Solution
{
public:
    int uniquePaths(int n, int m)
    {
        int DP[n][m];
        //Filling the values for the top row (base cases)
        for(int i=0;i<m;i++)
            DP[0][i]=1;
        //Filling the values for the most left column (base cases)
        for(int i=1;i<n;i++)
            DP[i][0]=1;
        //Filling the rest of the DP matrix
        for(int i=1;i<n;i++)
            for(int j=1;j<m;j++)
                DP[i][j]=DP[i][j-1]+DP[i-1][j];
        ///Return the solution from bottom-right corner of DP matrix
        return DP[n-1][m-1];
    }
};
```

Space complexity of our solution can be improved. We have 2 cases:

1. $n \leq m$: Observe that for this case, in order to calculate number of unique paths from starting point to cells in some column, we only need DP values of cells in previous column. That means that we can use DP matrix of size $n \times 2$ instead of $n \times m$. By doing this we managed to decrease space complexity for this case to $O(n)$, time complexity still remains $O(n \times m)$
2. $n > m$: Observe that for this case, in order to calculate number of unique paths from starting point to cells in some row, we only need DP values of cells in previous row. That means that we can use DP matrix of size $2 \times m$ instead of $n \times m$. By doing this we managed to decrease space complexity for this case to $O(m)$, time comlexity still remains $O(n \times m)$

Overall, space complexity of an improved solution is: $O(min(n, m))$

C++ code for solution with improved space complexity:

```cpp
class Solution
{
public:
    int uniquePaths(int n, int m)
    {
        //Base case (if we have only 1 row or 1 column than
        //there is only 1 unique path from start to finish)
        if(n==1 || m==1)
            return 1;
        if(n<=m)
        {
            int DP[n][2];
            //Filling the values for the most left column (base cases)
            for(int i=0;i<n;i++)
                DP[i][0]=1;
            //Filling DP values for the column that we
            //are currently looking, while constantly
            //keeping track of values in previous column
            //We do this until we reach last column
            for(int j=1;j<m;j++)
            {
                DP[0][1]=1; //Base case
                for(int i=1;i<n;i++)
                    DP[i][1]=DP[i][0]+DP[i-1][1];
                //Updating values in previous column
                for(int i=0;i<n;i++)
                    DP[i][0]=DP[i][1];
            }
            //Return the solution
            return DP[n-1][1];
        }
        else
        {
            int DP[2][m];
            //Filling the values for the top row (base cases)
            for(int i=0;i<m;i++)
                DP[0][i]=1;
            //Filling DP values for the row that we
            //are currently looking, while constantly
            //keeping track of values in previous row
            //We do this until we reach last row
            for(int i=1;i<n;i++)
            {
                DP[1][0]=1; //Base case
                for(int j=1;j<m;j++)
                    DP[1][j]=DP[1][j-1]+DP[0][j];
                //Updating values in previous row
                for(int j=0;j<m;j++)
                    DP[0][j]=DP[1][j];
            }
            //Return the solution
            return DP[1][m-1];
        }
    }
};
```