The problem is to count all the possible paths from top left to bottom right of a mXn matrix with the constraints that ***from each cell you can either move only to right or down***

We have discussed a solution to print all possible paths, counting all paths is easier. Let NumberOfPaths(m, n) be the count of paths to reach row number m and column number n in the matrix, NumberOfPaths(m, n) can be recursively written as following.

```
#include <iostream>
using namespace std;

// Returns count of possible paths to reach cell at row number m and column
// number n from the topmost leftmost cell (cell at 1, 1)
int  numberOfPaths(int m, int n)
{
   // If either given row number is first or given column number is first
   if (m == 1 || n == 1)
        return 1;

   // If diagonal movements are allowed then the last addition
   // is required.
   return  numberOfPaths(m-1, n) + numberOfPaths(m, n-1);
           // + numberOfPaths(m-1,n-1);
}

int main()
{
    cout << numberOfPaths(3, 3);
    return 0;
}
```

Output:

```
6
```

The time complexity of above recursive solution is exponential. There are many overlapping subproblems. We can draw a recursion tree for numberOfPaths(3, 3) and see many overlapping subproblems. The recursion tree would be similar to Recursion tree for Longest Common Subsequence problem.
So this problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array count[][] in bottom up manner using the above recursive formula.

```
#include <iostream>
using namespace std;

// Returns count of possible paths to reach cell at row number m and column
// number n from the topmost leftmost cell (cell at 1, 1)
int numberOfPaths(int m, int n)
{
    // Create a 2D table to store results of subproblems
    int count[m][n];

    // Count of paths to reach any cell in first column is 1
    for (int i = 0; i < m; i++)
        count[i][0] = 1;

    // Count of paths to reach any cell in first column is 1
    for (int j = 0; j < n; j++)
        count[0][j] = 1;

    // Calculate count of paths for other cells in bottom-up manner using
    // the recursive solution
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
```

```
            // By uncommenting the last part the code calculatest he total
            // possible paths if the diagonal Movements are allowed
            count[i][j] = count[i-1][j] + count[i][j-1]; //+ count[i-1][j-1];

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

// Driver program to test above functions
int main()
{
    cout << numberOfPaths(3, 3);
    return 0;
}
```

Output:

```
6
```

Time complexity of the above dynamic programming solution is O(mn).

Note the count can also be calculated using the formula (m-1 + n-1)!/(m-1)!(n-1)! as mentioned in the comments of this article.