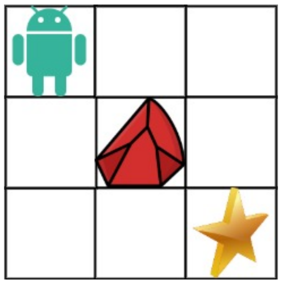
Day 49 / 100 :

Topic - Backtracking, DP

1 Problem statement: Unique Path II (medium)

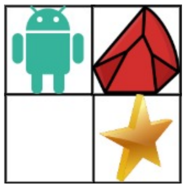
You are given an m x n integer array grid. There is a robot initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.
An obstacle and space are marked as 1 or 0 respectively in grid. A path that the robot takes cannot include any square that is an obstacle.
Return the number of possible unique paths that the robot can take to reach the bottom-right corner.
The testcases are generated so that the answer will be less than or equal to 2 * 109.

Example 1:



Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right

Example 2:



Input: obstacleGrid = [[0,1],[0,0]]
Output: 1

<u>Solutions :</u>

Approach 1 - DP

Intuition:

This problem is same as to take that path or not.(take, not take kinda)
We are storing tatal paths of the grid indexes.(by checking one by on in left direction and down).
If there's a obstacle we'll store 0 paths for that.
If we are at destination, simply return.
Otherwise take action in either direction and add both total ways.
Store it in dp table for space optimization.

Complexity:
Time complexity: O(2^n) : exponential due to recursive calls
Space complexity: O(n²)

```
class Solution {
public:
    int helper(vector<vector<int>>& grid, int i, int j, int
t[101][101]){
        if(i==grid.size() || j==grid[0].size()) return 0; // base
case
        if(grid[i][j]==1)return 0; // obstacle
        if(i==grid.size()-1 && j==grid[0].size()-1)return 1; //
destiny
        if(t[i][j]!=-1)return t[i][j]; // already calculated
        return t[i][j] = helper(grid, i+1, j, t) + helper(grid, i,
j+1, t);
    }

    int uniquePathsWithObstacles(vector<vector<int>>& grid) {
        int t[101][101];
        memset(t, -1, sizeof t);
        return helper(grid, 0, 0, t);

    }
};
```

2 Problem statement: Longest Increasing Subsequence (medium)

Given an array of integers, find the length of the longest (strictly) increasing subsequence from the given array.

Example 1:
Input:
N = 16
A = {0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15}
Output:
6
Explanation:
There are more than one LIS in this array. One such Longest increasing subsequence is {0,2,6,9,13,15}.

Example 2:
Input:
N = 6
A[] = {5,8,3,7,9,1}
Output:
3
Explanation:
There are more than one LIS in this array. One such Longest increasing subsequence is {5,7,9}.

Solutions :

Approach 1 - DP

Because of the optimal substructure and overlapping subproblem property, we can also utilize Dynamic programming to solve the problem. Instead of memorization, we can use the nested loop to implement the recursive relation.
The outer loop will run from i = 1 to N and the inner loop will run from j = 0 to i and use the recurrence relation to solve the problem.

```
int lis(int arr[], int n)
{
    int lis[n];

    lis[0] = 1;

    // Compute optimized LIS values in
    // bottom up manner
    for (int i = 1; i < n; i++) {
        lis[i] = 1;
        for (int j = 0; j < i; j++)
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
    }

    // Return maximum value in lis[]
    return *max_element(lis, lis + n);
}
```