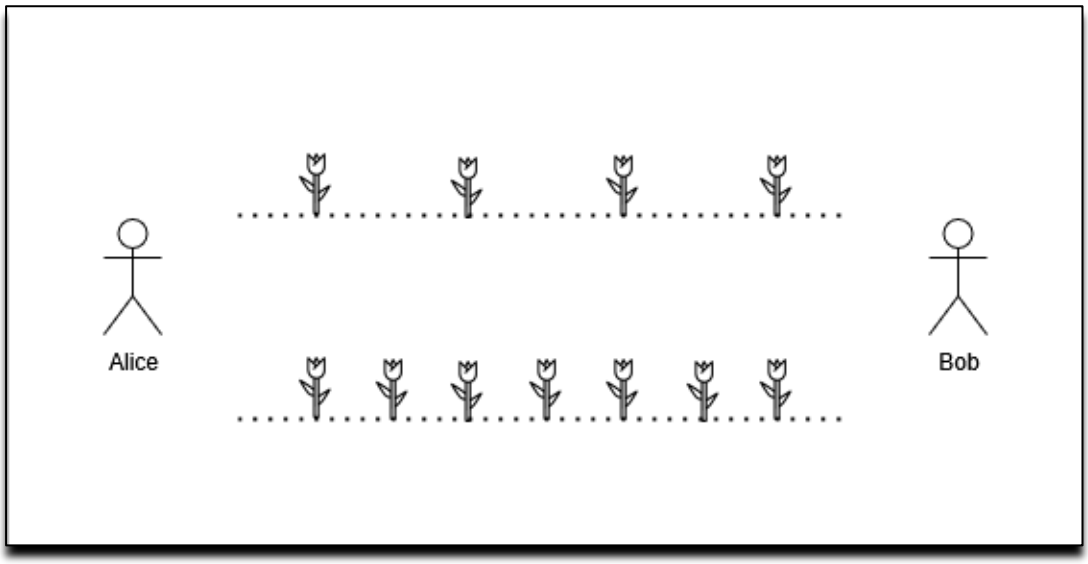
Day 65-71

FOLLOW ME 😊

3021. Alice and Bob Playing Flower Game

Alice and Bob are playing a turn-based game on a field, with two lanes of flowers between them. There are x flowers in the first lane between Alice and Bob, and y flowers in the second lane between them.



The game proceeds as follows:

1. Alice takes the first turn.
2. In each turn, a player must choose either one of the lane and pick one flower from that side.
3. At the end of the turn, if there are no flowers left at all, the **current** player captures their opponent and wins the game.

Given two integers, n and m, the task is to compute the number of possible pairs (x, y) that satisfy the conditions:

- Alice must win the game according to the described rules.
- The number of flowers x in the first lane must be in the range [1,n].
- The number of flowers y in the second lane must be in the range [1,m].

Return *the number of possible pairs* (x, y) *that satisfy the conditions mentioned in the statement*.

```
class Solution {
public:
#define ll  long long
    long long flowerGame(int n, int m) {
        ll count=0;

        //Case1: n is odd and m is even
        ll n_is_odd=(n+1)/2;
        ll m_is_even=m/2;
        count+=(n_is_odd*m_is_even);

        //Case2: n is even  and m is odd
        ll n_is_even=n/2;
        ll m_is_odd=(m+1)/2;
        count+=(n_is_even)*(m_is_odd);

        return count;

    }
};

```

1971. Find if Path Exists in Graph

There is a **bi-directional** graph with n vertices, where each vertex is labeled from 0 to n - 1 (**inclusive**). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [u_i, v_i] denotes a bi-directional edge between vertex u_i and vertex v_i. Every vertex pair is connected by **at most one** edge, and no vertex has an edge to itself.

You want to determine if there is a **valid path** that exists from vertex source to vertex destination.

Given edges and the integers n, source, and destination, return true *if there is a **valid path** from* source *to* destination*, or* false *otherwise.*

```
class Solution {
public:
bool pathHelper(int src,int dest,vector<bool>&vis,vector<vector<int>>&adj){
    //BaseCase
    if(src==dest) return true;
    vis[src]=true;

    for(int v: adj[src]){
        if(!vis[v]){
            if(pathHelper(v,dest,vis,adj)){
                return true;
            }
        }
    }
    return false;
}
    bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
        vector<vector<int>>adj(n);
        for(auto e: edges){
            adj[e[0]].push_back(e[1]);
            adj[e[1]].push_back(e[0]);

        }
        vector<bool>vis(n,false);
        return pathHelper(source,destination,vis,adj);

    }
```

```
};
```

3446. Sort Matrix by Diagonals

You are given an n x n square matrix of integers grid. Return the matrix such that:

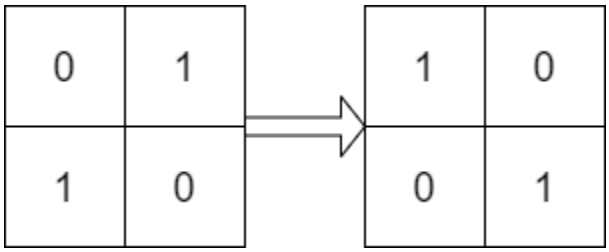
- The diagonals in the **bottom-left triangle** (including the middle diagonal) are sorted in **non-increasing order**.
- The diagonals in the **top-right triangle** are sorted in **non-decreasing order**.

```
class Solution {
public:
void sortDiagonal(vector<vector<int>>&grid,int startX,int startY,bool increasing){
    vector<int>diagonalElements;
    int x=startX;
    int y=startY;
    int n=grid.size();

    //1. Extract all the diagonal elements;
    while(x<n && y<n){
        diagonalElements.push_back(grid[x][y]);
        x++,y++;
    }

    //step2. Sort the elements
    if(increasing){
        sort( diagonalElements.begin(), diagonalElements.end());
    }else{
        sort( diagonalElements.begin(), diagonalElements.end(),greater<int>());
    }

    //step3: put the sorted elements back into the frid
    x=startX,y=startY;
    for(int val:  diagonalElements){
        grid[x][y]=val;
        x++,y++;
    }
}
    vector<vector<int>> sortMatrix(vector<vector<int>>& grid) {
        int n=grid.size();

//Process elements top most
        for(int i=0;i<n;i++){
            sortDiagonal(grid,i,0,false);
        }

        for(int j=1;j<n;j++){
            sortDiagonal(grid,0,j,true);
        }

        return grid;

    }
};
```

1886. Determine Whether Matrix Can Be Obtained By Rotation

Given two n x n binary matrices mat and target, return true *if it is possible to make* mat *equal to* target *by* ***rotating*** mat *in* ***90-degree increments****, or* false *otherwise.*

Example 1:



Input: mat = [[0,1],[1,0]], target = [[1,0],[0,1]]

```
class Solution {
public:
    bool findRotation(vector<vector<int>>& mat, vector<vector<int>>& target) {
        int n=mat.size();
        for(int r=0;r<4;r++){
            if(mat==target) return true;
            for(int i=0;i<n;i++){
            for(int j=i+1;j<n;j++){
                swap(mat[i][j],mat[j][i]);
            }
        }

        //reverse
        for(int i=0;i<n;i++){
            reverse(mat[i].begin(),mat[i].end());
        }


        }
        return false;

    }
};

```

36. Valid Sudoku

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits 1-9 without repetition.
2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

```
class Solution {
public:
bool validBox(vector<vector<char>>& board,int sr,int er,int sc,int ec){
    unordered_set<char>s;
    for(int i=sr;i<er;i++){
        for(int j=sc;j<ec;j++){
            if(board[i][j]=='.') continue;
            if(s.find(board[i][j])!=s.end()) return false;
            s.insert(board[i][j]);
```

```
        }
    }
    return true;
}
    bool isValidSudoku(vector<vector<char>>& board) {
        //1. check validity of rows
        for(int row=0;row<9;row++){
        unordered_set<char>s;
          for(int col=0;col<9;col++){
            if(board[row][col]=='.') continue;
            if(s.find(board[row][col])!=s.end()) return false;
            s.insert(board[row][col]);
          }
        }


        //2. check validity of col
        for(int col=0;col<9;col++){
            unordered_set<char>s;
            for(int row=0;row<9;row++){
                if(board[row][col]=='.') continue;
                if(s.find(board[row][col])!=s.end()) return false;
                s.insert(board[row][col]);
            }
        }

        //3. check the validity of grid
        for(int sr=0;sr<9;sr+=3){
            int er=sr+3;
            unordered_set<char>s;
            for(int sc=0;sc<9;sc+=3){
               int ec=sc+3;

               if(!validBox(board,sr,er,sc,ec)) return false;

            }
        }
        return true;

    }
};

```

Wave Array

Given an **sorted** array **arr[]** of integers. Sort the array into a **wave-like** array(In Place). In other words, **arrange the elements** into a sequence such that arr[1] >= arr[2] <= arr[3] >= arr[4] <= arr[5] ..... and so on. If there are multiple solutions, find the **lexicographically smallest** one.

Note: The given array is sorted in ascending order, and modify the given array in-place without returning a new array.

```
class Solution {
  public:
    void sortInWave(vector<int>& arr) {
        int n=arr.size();
        for(int i=0;i<n-1;i+=2){
            swap(arr[i],arr[i+1]);
        }

    }
};

```

Rotate by 90 degree

Given a square matrix **mat**[][] of size **n x n**. The task is to rotate it by **90 degrees** in an **anti-clockwise** direction without using any extra space.

Examples:

Input: mat[][] = [[0, 1, 2],
[3, 4, 5],
[6, 7, 8]]

Output: [[2, 5, 8],

[1, 4, 7],

[0, 3, 6]]

```
class Solution {
  public:
    void rotateMatrix(vector<vector<int>>& mat) {
        int n=mat.size();

           //reverse
        for(int i=0;i<n;i++){
            reverse(mat[i].begin(),mat[i].end());
        }

        //adjoint
        for(int i=0;i<n;i++){
            for(int j=i+1;j<n;j++){
                swap(mat[i][j],mat[j][i]);
            }
        }
    }
};

```