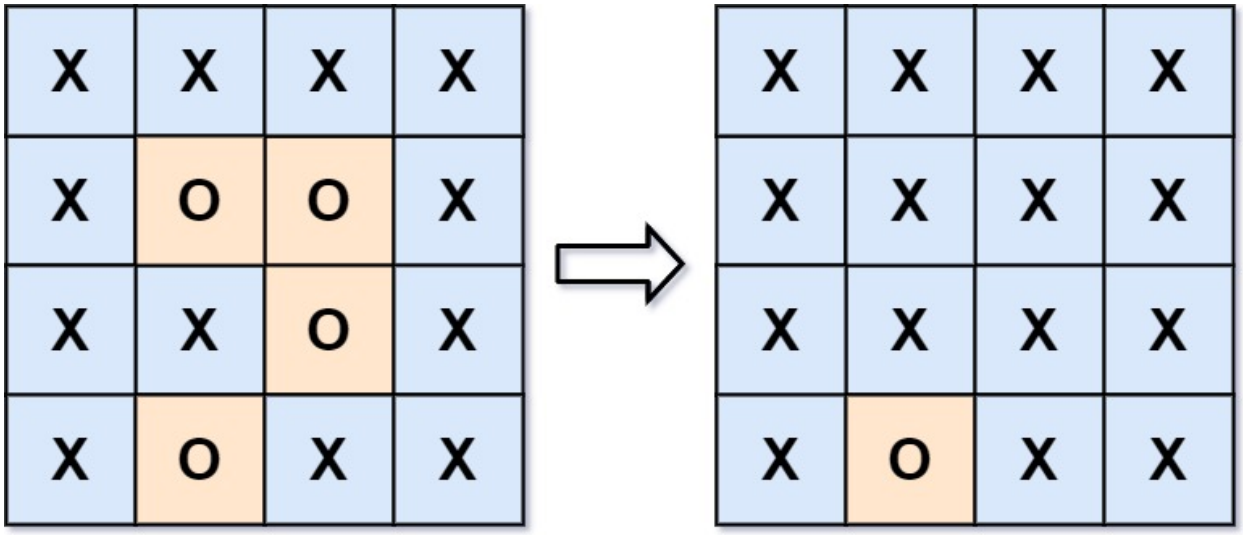
Problem Statement :

Given an m x n matrix board containing 'X' and 'O', capture all regions that are 4-directionally surrounded by 'X'. A region is captured by flipping all 'O's into 'X's in that surrounded region.



Input: board =
[["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]

Output:
[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]

Explanation: Notice that an 'O' should not be flipped if:
- It is on the border, or
- It is adjacent to an 'O' that should not be flipped.
The bottom 'O' is on the border, so it is not flipped.
The other three 'O' form a surrounded region, so they are flipped.

Observation: We notice that all the O's form islands or groups and X's do so too, So now we need to make all the O's present to X's that are surrounded in four directions by X islands. Now we notice that the O's that are to the side of the square can not obviously be surrounded by X's so all the O islands that have one or more Os that border the side of the square can not be surrounded by Xs however the other islands will be turned to X.

Step one: find all the Os in the borders
Step two: Do a Depth first search from the above Os and turn them to Y
Step Three: Then Make the other O's Xs and the Y's Os

My Code:

```
n=int(input("Enter the number of rows:"))
m=int(input("Enter the number of columns:"))
board=[]
for i in range(0,n):
    temp=[]
    for j in range (0,m):
        temp.append(input())
    board.append(temp)
n=len(board)
m=len(board[0])
print(board)
def dfs(board,i,j,n,m):
    if(i>=n or i<0 or j>=m or j<0 or board[i][j]!='O' ):
        return
    board[i][j]='Y'
    dfs(board,i-1,j,n,m)
    dfs(board,i+1,j,n,m)
    dfs(board,i,j-1,n,m)
    dfs(board,i,j+1,n,m)
for i in range (0,n):
  for j in range (0,m):
    if (board[i][j]=='O' and (i in [0,n-1] or j in [0,m-1])):
        dfs(board,i,j,n,m)
for i in range (0,n):
    for j in range (0,m):
        if(board[i][j]=='Y'):
            board[i][j]='O'
```

```
        else:
            board[i][j]='X'
print(board)
```

Rohith's Code:

```
def solve(board):
    m=len(board) #dimensions
    n=len(board[0])
    # DFS
    def dfs(row,col):
        if row>=m or row<0 or col>=n or col<0 or board[row][col] != 'O':
            return
        board[row][col]='$'
        dfs(row+1,col) # Moving Down
        dfs(row,col+1) # Moving Right
        dfs(row-1,col) # Moving Up
        dfs(row,col-1) # Moving Left
# Traversing every element in board and If we come across 'O' we move it
to dfs and change the connected 'O'
    for i in range(m):
        if board[i][0] == 'O':
            dfs(i,0)    # Traversing [0][0] to [m-1][0] and [0][n-1] to
[m-1][n-1]
        if board[i][n-1] == 'O':
            dfs(i, n-1)
    for j in range(n):
#time complexity - O(m*n)
        if board[0][j] == 'O':
#space complexity - O(m*n)(stack space)
          dfs(0, j)         # Traversing [m-1][0] to [m-1][n-1] and [0][0]
#to [0][n-1]
        if board[m-1][j] == 'O':
          dfs(m-1, j)
    # Traversing every element in board and If we come across $ we make it
'O' else we make it X.
    for i in range(m):
        for j in range(n):
            if board[i][j]=='O':
                board[i][j]='X'
            if board[i][j]=='$':
                board[i][j]='O'
board =
[["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]
print("Board before captures - ")
for i in board:
   for j in i:
      print(j,end = " ")
```

```
   print()
solve(board)
print("Board after captures - ")
for i in board:
    for j in i:
        print(j,end=" ")
    print()
```

Mohan's Code:

```
# Solving the problem in DFS Depth first search

def solve(board):
    # finding length of the column of the board
    m=len(board)
    # Finding len of the row of the board
    n=len(board[0])

    #Creating dfs function
    def dfs(m,n):
        if n <0 or m<0 or n == len(board[0]) or m == len(board) or
board[m][n]!='O':
            return

        board[m][n]='Z'
        dfs(m+1,n) # Moving Down
        dfs(m,n+1) # Moving Right
        dfs(m-1,n) # Moving Up
        dfs(m,n-1) # Moving Left
    # Traversing every element in board and If we come across 'O' we move
it to dfs and explore its neighbours.
    for i in range(m):
        for j in range(n):
            #If element is 'O; and is in borders of board we move it to
dfs
            if (board[i][j]=='O' and (i in [0,m-1] or j in [0,n-1])):
                dfs(i,j)
    # Traversing every element in board and If we come across Z we make it
'O' else we make it. X.
    for i in range(m):
        for j in range(n):
            if board[i][j]=='O':
                board[i][j]='X'
            if board[i][j]=='Z':
                board[i][j]='O'


# Getting the input from the user
```

```
no_columns = int(input("Enter no of columns"))
no_rows =int(input("Enter no of rows"))
board = [[""]*no_columns]*no_rows
print(board)
for i in range(no_columns):
    for j in range(no_rows):
        board[j][i] = input(f"enter the value in the {i}, {j} th
positon\n")
print(board)


# Default board input
board =
[["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]
print("Solution using Breadth First Search")
print("\nBoard before captures")
for r in board:
   for c in r:
      print(c,end = " ")
   print()

solve(board)
print("\nBoard after captures - ")
for r in board:
   for c in r:
      print(c,end = " ")
   print()
```

Sonali's Code:

```
def solve(board):
    # Dimensions for the board
    m=len(board)
    n=len(board[0])
    # Defining a function for dfs
    def dfs(m,n):
        # Base condition, we do nothing if we find 'X' or if limit exceeds
        if n <0 or m<0 or n == len(board[0]) or m == len(board) or
board[m][n]!='O':
            return
        # Else we change it to N
        board[m][n]='N'
        # We explore its directions and repeat the manpulation
        dfs(m+1,n) # Lower cell
        dfs(m,n+1) # Right cell
        dfs(m-1,n) # Upper cell
        dfs(m,n-1) # Left cell
```

```
    # We traverse every element present on the board, if we find 'O' we
move it to dfs
    for i in range(m):
        for j in range(n):
            #If element is 'O' and is on the borders of the board we move
it to dfs
            if (board[i][j]=='O' and (i in [0,m-1] or j in [0,n-1])):
                dfs(i,j)
    # Now traverse the entire board, if we find N, change it to O else we
change it to X.
    for i in range(m):
        for j in range(n):
            if board[i][j]=='O':
                board[i][j]='X'
            if board[i][j]=='Y':
                board[i][j]='O'

m = int(input("Enter no of columns = "))
n = int(input("Enter no of rows = "))

board = [[""]*m]*n # creating board
print(board)

for i in range(m):
    for j in range(n):
        board[j][i] = input(f"Enter value in the {i}, {j} th positon\n")

# Default board input
board =
[["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]
print("Solution using Breadth First Search")
print("\nBoard before captures")
for x in board:
   for y in x:
      print(y,end = " ")
   print()

solve(board)
print("\nBoard after captures - ")
for x in board:
   for y in x:
      print(y,end = " ")
   print()
```

Two of my peers and I have the same approach although after going through Rohith's program I have come to the conclusion that my program runs slowly when the input size becomes larger, as my code iterates through the whole 2d list instead of just iterating through the boundaries and then calling for the dfs function which causes unnecessary wastage of time.

My code:

```
for i in range (0,n):
  for j in range (0,m):
    if (board[i][j]=='O' and (i in [0,n-1] or j in [0,m-1])):
        dfs(board,i,j,n,m)
#time complexity - O(m*n) although has to traverse through the whole
matrix
```

Rohith's Code:

```
  for i in range(m):
        if board[i][0] == 'O':
            dfs(i,0)
        if board[i][n-1] == 'O':
            dfs(i, n-1)
    for j in range(n):
        if board[0][j] == 'O':
          dfs(0, j)
        if board[m-1][j] == 'O':
          dfs(m-1, j)
#time complexity - O(m*n)
```