**Name : Ayush Patil**

**Roll no : TYCOC210**

**Batch : C-3**

**Prn no : 122B1B210**

**Assignment – 5**

**Title :**

A classic problem that can be solved by backtracking is called the Knight's tour Problem. It is a problem in which we are provided with a NxN chessboard and a knight. For a person who is not familiar with chess, the knight moves two squares horizontally and one square vertically, or two squares vertically and one square horizontally. In this problem, there is an empty chess board, and a knight starting from any location in the board, our task is to check whether the knight can visit all of the squares in the board or not. When It can visit all of the squares, then place the number of jumps needed to reach that location from the starting point.

**Theory :**

The Knight's Tour problem is a classic challenge in the realm of combinatorial optimization and algorithm design. It involves a knight piece on a chessboard, with the goal of determining a sequence of moves that allows the knight to visit every square exactly once. The knight moves in an "L" shape, which consists of two squares in one direction and one square perpendicular to that, or vice versa. This unique movement capability provides the knight with up to eight potential positions to which it can leap from any given square.

To tackle the Knight's Tour problem, one common approach is to utilize a backtracking algorithm. This technique incrementally builds the knight's tour by exploring possible moves and employing a recursive strategy to find a solution. Initially, all squares are marked as unvisited, often denoted by a value like -1.

During the backtracking process, the algorithm checks for valid moves at each step. A valid move is one that stays within the bounds of the chessboard and lands on an unvisited square. The recursion continues until either all squares have been visited, indicating a successful tour or it reaches a point where no further valid moves are available, prompting the algorithm to backtrack.

**Time Complexity :** The worst-case time complexity for the Knight's Tour problem is O(8^{N^2}) This arises because the knight has up to 8 possible moves from any given position on the board. As the knight moves and explores each square, the number of potential paths grows exponentially, especially since the knight can visit each square only once.

**Space Complexity :** The space complexity is O(N^2). The space required for the board itself, which needs to store the visited status of each square (N×N matrix).The maximum depth of the recursive call stack can also reach O(N^2) in the worst case, as the knight may potentially visit all squares on the board before backtracking.

**Code :**

def is\_safe(x, y, board):

N = len(board)

return 0 <= x < N and 0 <= y < N and board[x][y] == -1

def knight\_tour\_util(x, y, move\_count, board, x\_move, y\_move):

if move\_count == len(board) \*\* 2:

return True

for i in range(8):

next\_x = x + x\_move[i]

next\_y = y + y\_move[i]

if is\_safe(next\_x, next\_y, board):

board[next\_x][next\_y] = move\_count

if knight\_tour\_util(next\_x, next\_y, move\_count + 1, board, x\_move, y\_move):

return True

board[next\_x][next\_y] = -1

return False

def knight\_tour(N, start\_x, start\_y):

board = [[-1 for \_ in range(N)] for \_ in range(N)]

x\_move = [2, 1, -1, -2, -2, -1, 1, 2]

y\_move = [1, 2, 2, 1, -1, -2, -2, -1]

board[start\_x][start\_y] = 0

if not knight\_tour\_util(start\_x, start\_y, 1, board, x\_move, y\_move):

print("Solution does not exist")

else:

print("Knight's tour solution:")

for row in board:

print(row)

N = int(input("Enter the size of the chessboard (N x N): "))

start\_x = int(input("Enter the starting x position (0 to N-1): "))

start\_y = int(input("Enter the starting y position (0 to N-1): "))

if 0 <= start\_x < N and 0 <= start\_y < N:

knight\_tour(N, start\_x, start\_y)

else:

print("Invalid starting position. Please enter values between 0 and", N-1)

**Output :**

