**PROGRAM TITLE -2**

**8-QUEEN PROBLEM**

**AIM:**

To write and execute the python program for solving 8-Queen problem.

**ALGORITHM**

1. **Initialize the Chessboard**: Create an 8x8 chessboard represented by a 2D array. Initially, all cells are empty.
2. **Place Queens**: Start placing queens on the chessboard row by row, starting from the first row.
3. **Recursive Backtracking Function**:
   * Base Case: If all 8 queens are placed successfully (i.e., all rows are filled), return true.
   * Recursive Case:
     + For the current row, try placing a queen in each column.
     + Check if placing the queen at the current position is safe:
       - Check if no two queens threaten each other horizontally, vertically, or diagonally.
       - If it's safe, mark the current cell as occupied by the queen and move to the next row.
       - Recursively call the backtracking function for the next row.
       - If placing the queen leads to a solution (i.e., all queens are placed), return true.
       - If placing the queen does not lead to a solution, backtrack by removing the queen from the current position and try the next column.
   * If all columns are tried and no queen can be placed, return false to trigger backtracking to the previous row.
4. **Main Function**:
   * Call the backtracking function with the initial state (empty chessboard) starting from the first row.
   * If the backtracking function returns true, the solution is found. Otherwise, no solution exists.

Here's a Python implementation of the algorithm:

.

**CODING:**

def is\_goal(state):

return state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

def find\_empty(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return (i, j)

def get\_valid\_moves(state, empty\_pos):

moves = []

i, j = empty\_pos

if i > 0:

moves.append("up")

if i < 2:

moves.append("down")

if j > 0:

moves.append("left")

if j < 2:

moves.append("right")

return moves

def solve(state, visited):

if is\_goal(state):

return state

visited.add(tuple(map(tuple, state)))

empty\_pos = find\_empty(state)

for move in get\_valid\_moves(state, empty\_pos):

new\_state = [row.copy() for row in state]

i, j = empty\_pos

if move == "up":

new\_state[i][j], new\_state[i - 1][j] = new\_state[i - 1][j], new\_state[i][j]

elif move == "down":

new\_state[i][j], new\_state[i + 1][j] = new\_state[i + 1][j], new\_state[i][j]

elif move == "left":

new\_state[i][j], new\_state[i][j - 1] = new\_state[i][j - 1], new\_state[i][j]

else:

new\_state[i][j], new\_state[i][j + 1] = new\_state[i][j + 1], new\_state[i][j]

if tuple(map(tuple, new\_state)) not in visited:

solution = solve(new\_state, visited.copy())

if solution:

return [move] + solution

return None

initial\_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]

visited = set()

solution = solve(initial\_state, visited)

if solution:

print("Solution found!")

for move in solution:

print(move)

else:

print("No solution found.")

N = 8

def solveNQueens(board, col):

if col == N:

print(board)

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQueens(board, col + 1):

return True

board[i][col] = 0

return False

def isSafe(board, row, col):

for x in range(col):

if board[row][x] == 1:

return False

for x, y in zip(range(row, -1, -1), range(col, -1, -1)):

if board[x][y] == 1:

return False

for x, y in zip(range(row, N, 1), range(col, -1, -1)):

if board[x][y] == 1:

return False

return True

board = [[0 for x in range(N)] for y in range(N)]

if not solveNQueens(board, 0):

print("No solution found")

**OUTPUT:**

****

**RESULT:**

Thus the program has been successfully executed and verified.