**LAB 4:**

**Program:** Implement Hill Climbing to Solve N-Queen’s Problem

**Code:**

import random

class NQueens:

def \_\_init\_\_(self, n, initial\_board=None):

self.n = n

self.board = initial\_board if initial\_board is not None else self.initialize\_board()

def initialize\_board(self):

# Place one queen in each row randomly

return [random.randint(0, self.n - 1) for \_ in range(self.n)]

def calculate\_conflicts(self, board):

conflicts = 0

for i in range(self.n):

for j in range(i + 1, self.n):

# Check for conflicts between queens

if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):

conflicts += 1

return conflicts

def get\_neighbors(self, board):

neighbors = []

for row in range(self.n):

for col in range(self.n):

if col != board[row]:

# Create a new board configuration by moving the queen

new\_board = board[:]

new\_board[row] = col

neighbors.append(new\_board)

return neighbors

def hill\_climbing(self):

current\_board = self.board

current\_conflicts = self.calculate\_conflicts(current\_board)

print(f"Initial State: {current\_board} | Conflicts: {current\_conflicts}")

while current\_conflicts > 0:

neighbors = self.get\_neighbors(current\_board)

next\_board = None

next\_conflicts = float('inf')

for neighbor in neighbors:

conflicts = self.calculate\_conflicts(neighbor)

print(f"Neighbor: {neighbor} | Conflicts: {conflicts}")

# Choose the neighbor with the smallest number of conflicts

# If there's a tie, choose the one with the smallest column number

if conflicts < next\_conflicts or (conflicts == next\_conflicts and self.choose\_smaller(neighbor, next\_board)):

next\_board = neighbor

next\_conflicts = conflicts

if next\_conflicts >= current\_conflicts:

# No better neighbors found

return None

print(f"Moving to: {next\_board} | Conflicts: {next\_conflicts}")

current\_board = next\_board

current\_conflicts = next\_conflicts

return current\_board

def choose\_smaller(self, new\_board, current\_board):

if current\_board is None:

return True

for i in range(self.n):

if new\_board[i] != current\_board[i]:

return new\_board[i] < current\_board[i]

return False

def print\_board(self, board):

for row in range(self.n):

line = ['Q' if board[i] == row else '.' for i in range(self.n)]

print(' '.join(line))

print()

if \_\_name\_\_ == "\_\_main\_\_":

n = int(input("Enter the size of the board (N): "))

initial\_board = []

print("Enter the initial positions of the queens (0-indexed, one number per row):")

for i in range(n):

while True:

try:

pos = int(input(f"Row {i} (0 to {n-1}): "))

if 0 <= pos < n:

initial\_board.append(pos)

break

else:

print(f"Invalid position! Please enter a value between 0 and {n-1}.")

except ValueError:

print("Invalid input! Please enter an integer.")

solver = NQueens(n, initial\_board)

solution = solver.hill\_climbing()

if solution:

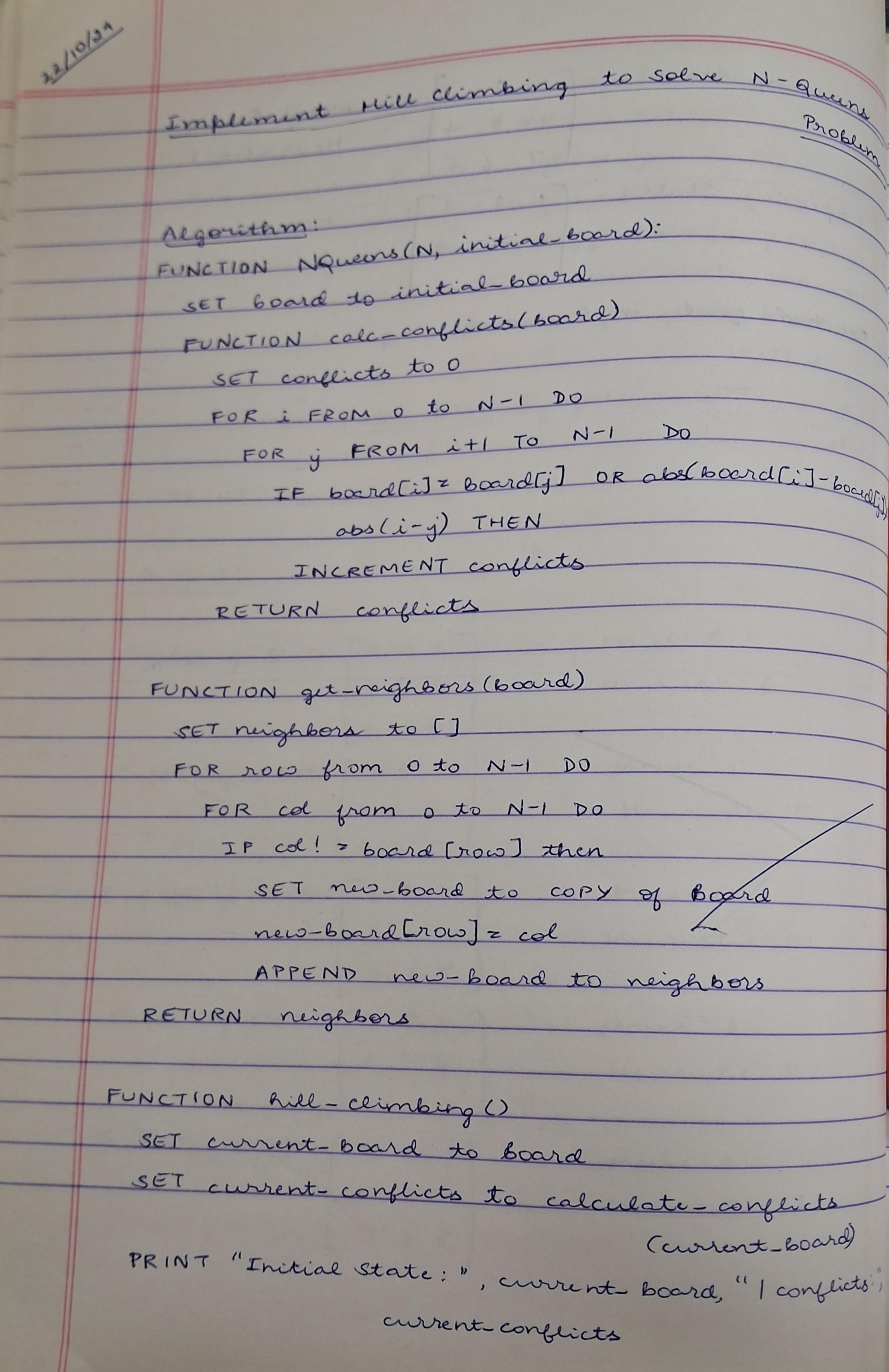
print("\nSolution found:")

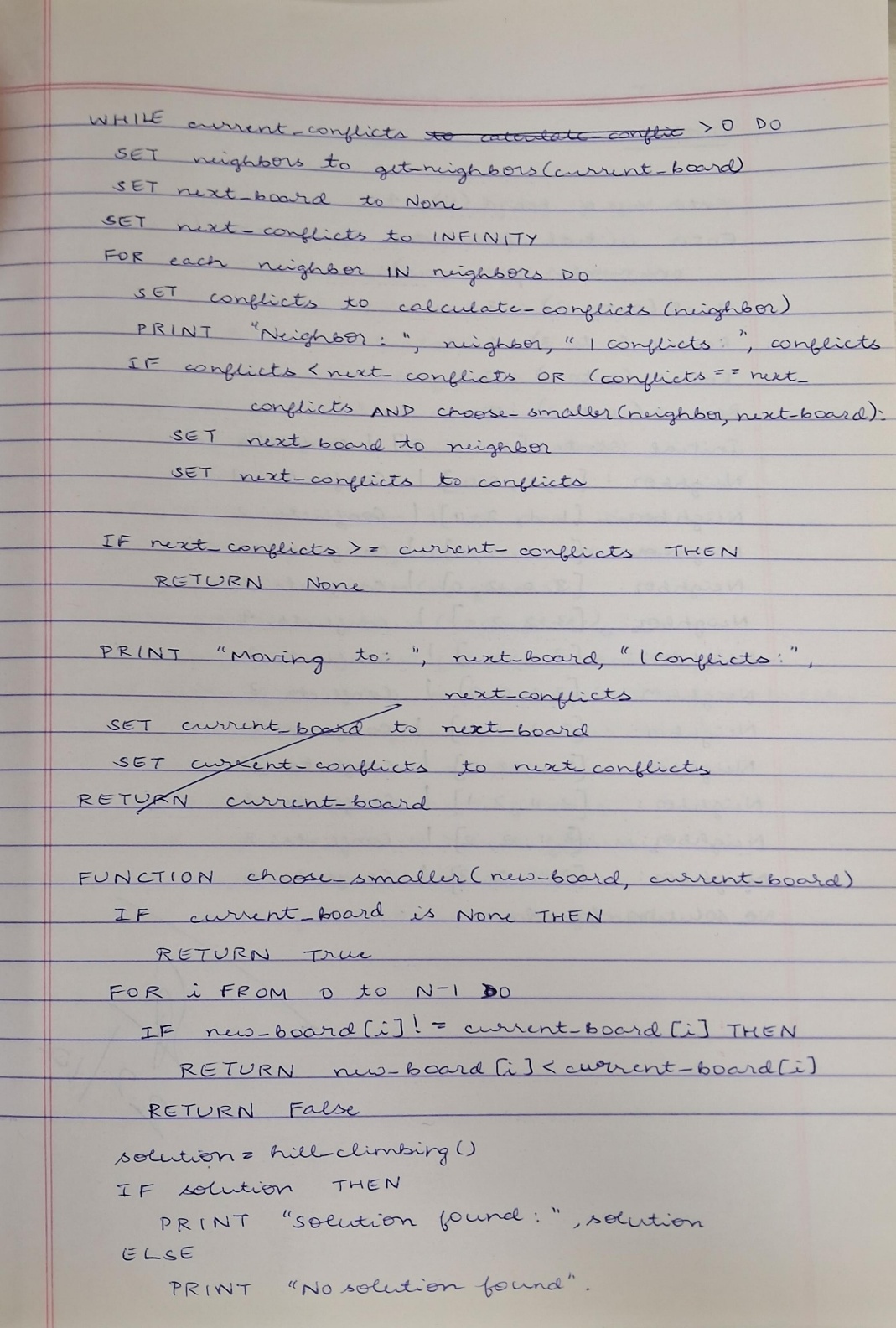
solver.print\_board(solution)

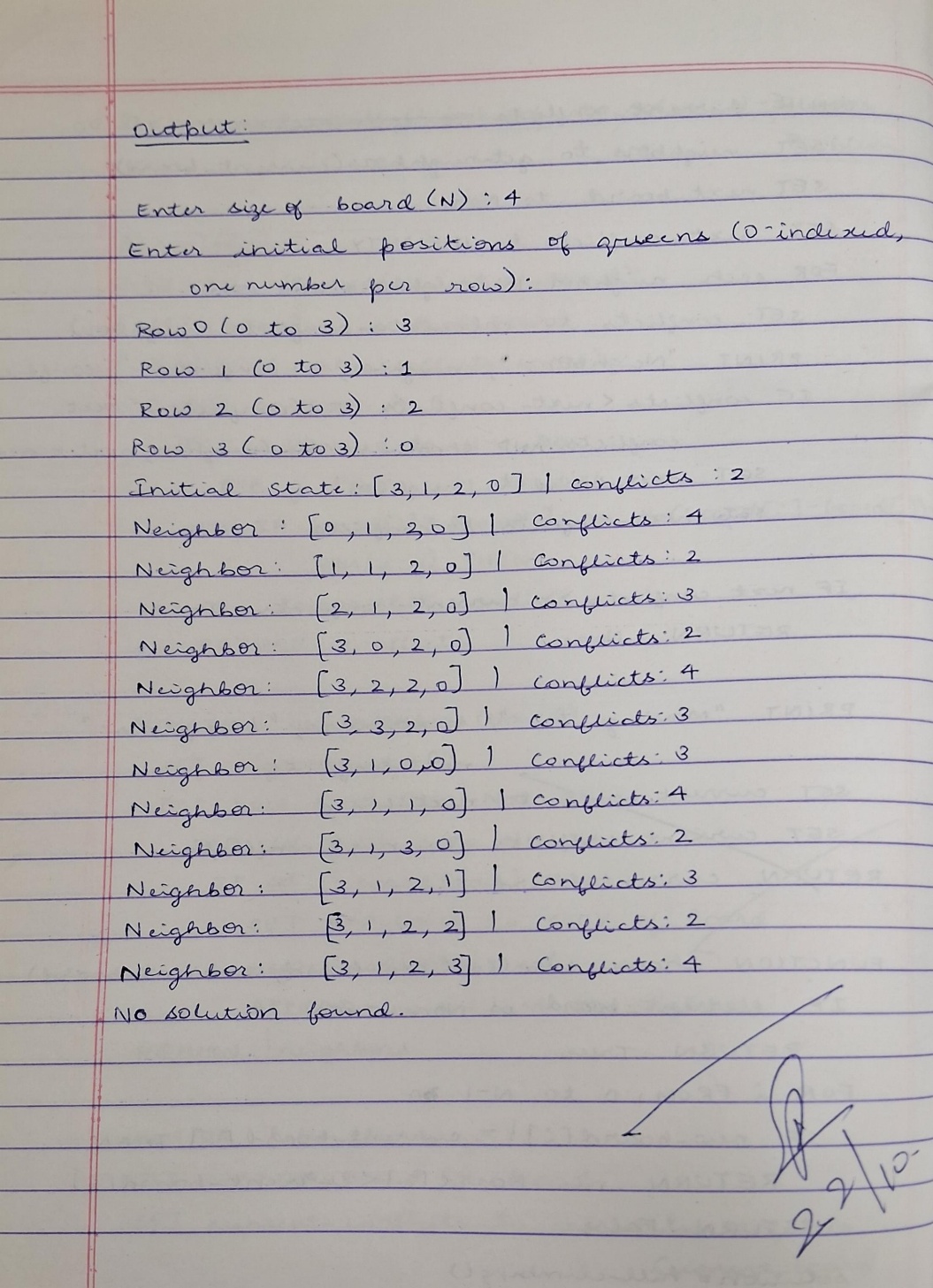
else:

print("No solution found.")

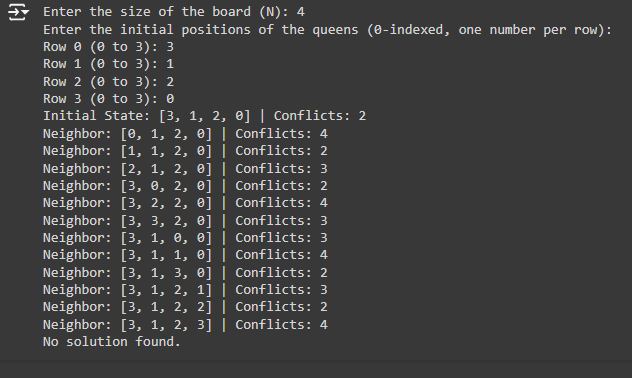
**Algorithm:**

****

****

****

**Output:**

****