```
N = 4 # Size of the board

# A utility function that prints the board representation based on the state
def printBoard(state):
    # Create a 2D array initialized to zeros to represent the board
    board = [[0] * N for _ in range(N)]

    # Place a queen on the board based on the current state
    for col in range(N):
        board[state[col]][col] = 1  # Place a queen in the specified row for each column

    # Print the board row by row
    for row in board:
        print(*row)  # Unpack the row list for clean output
    print()  # Print a blank line for better readability

# A utility function that calculates the number of attacking queens
def calculateObjective(state):
    attacking = 0  # Initialize the count of attacking queens to zero

    # Check each pair of queens to see if they attack each other
    for i in range(N):
        for j in range(i + 1, N):  # Only check queens to the right to avoid double counting
            if state[i] == state[j]:  # Check if they are in the same row
                attacking += 1  # Increment the count if they are in the same row
            # Check if they are on the same diagonal
            if abs(state[i] - state[j]) == abs(i - j):
                attacking += 1  # Increment the count for diagonal attack

    return attacking  # Return the total number of attacking pairs

# This function finds the neighbor of the current state with the least objective value
def getNeighbour(state):
    best_state = state[:]  # Start with the current state as the best state
    best_objective = calculateObjective(state)  # Calculate the current objective value (attacks)

    # Loop through each queen to find a better configuration
    for i in range(N):
        original_row = state[i]  # Store the original row position of the queen

        # Try moving the queen to each possible row in the current column
        for j in range(N):
            if j != original_row:  # Skip the current row
                state[i] = j  # Move the queen to the new row
                current_objective = calculateObjective(state)  # Calculate attacks for the new state

                # Check if this new state has fewer attacks
                if current_objective < best_objective:
                    best_objective = current_objective  # Update best objective
                    best_state = state[:]  # Update best state to the new configuration

        state[i] = original_row  # Restore the original position of the queen

    return best_state  # Return the best state found

# The main function that implements the hill climbing algorithm
def hillClimbing(state):
    while True:  # Loop indefinitely until a solution is found or no better state exists
        current_objective = calculateObjective(state)  # Calculate attacks for the current state

        # Check if a solution has been found (no attacks)
        if current_objective == 0:
            print("Final board configuration:")  # Indicate that a solution has been found
            printBoard(state)  # Print the final configuration
            break  # Exit the loop

        next_state = getNeighbour(state)  # Find the best neighboring state

        # Check if no better state was found (stuck in a local minimum)
        if next_state == state:
            print("Stuck in local minimum.")  # Indicate that no improvement is possible
            printBoard(state)  # Print the current state
            break  # Exit the loop
        else:
            state = next_state  # Update the current state to the better neighbor

# Driver code
state = [1, 3, 2, 4]  # Initial position of queens, specifying their row positions

# Run the hill climbing algorithm on the initial state
hillClimbing(state)
```

```
Final board configuration:
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

```
# Python3 implementation of the
# above approach
from random import randint

N = 8

# A utility function that configures
# the 2D array "board" and
# array "state" randomly to provide
# a starting point for the algorithm.
def configureRandomly(board, state):

    # Iterating through the
    # column indices
    for i in range(N):

        # Getting a random row index
        state[i] = randint(0, 100000) % N;

        # Placing a queen on the
        # obtained place in
        # chessboard.
        board[state[i]][i] = 1;

# A utility function that prints
# the 2D array "board".
def printBoard(board):

    for i in range(N):
        print(*board[i])

# A utility function that prints
# the array "state".
def printState( state):
    print(*state)

# A utility function that compares
# two arrays, state1 and state2 and
# returns True if equal
# and False otherwise.
def compareStates(state1, state2):


    for i in range(N):
        if (state1[i] != state2[i]):
            return False;

    return True;

# A utility function that fills
# the 2D array "board" with
# values "value"
def fill(board, value):

    for i in range(N):
        for j in range(N):
            board[i][j] = value;

# This function calculates the
# objective value of the
# state(queens attacking each other)
# using the board by the
# following logic.
def calculateObjective( board, state):

    # For each queen in a column, we check
    # for other queens falling in the line
    # of our current queen and if found,
    # any, then we increment the variable
    # attacking count.

    # Number of queens attacking each other,
    # initially zero.
```

```
    attacking = 0;

    # Variables to index a particular
    # row and column on board.
    for i in range(N):

        # At each column 'i', the queen is
        # placed at row 'state[i]', by the
        # definition of our state.

        # To the left of same row
        # (row remains constant
        # and col decreases)
        row = state[i]
        col = i - 1;
        while (col >= 0 and board[row][col] != 1) :
            col -= 1

        if (col >= 0 and board[row][col] == 1) :
            attacking += 1;

        # To the right of same row
        # (row remains constant
        # and col increases)
        row = state[i]
        col = i + 1;
        while (col < N and board[row][col] != 1):
            col += 1;

        if (col < N and board[row][col] == 1) :
            attacking += 1;

        # Diagonally to the left up
        # (row and col simultaneously
        # decrease)
        row = state[i] - 1
        col = i - 1;
        while (col >= 0 and row >= 0 and board[row][col] != 1) :
            col-= 1;
            row-= 1;

        if (col >= 0 and row >= 0 and board[row][col] == 1) :
            attacking+= 1;

        # Diagonally to the right down
        # (row and col simultaneously
        # increase)
        row = state[i] + 1
        col = i + 1;
        while (col < N and row < N and board[row][col] != 1) :
            col+= 1;
            row+= 1;

        if (col < N and row < N and board[row][col] == 1) :
            attacking += 1;

        # Diagonally to the left down
        # (col decreases and row
        # increases)
        row = state[i] + 1
        col = i - 1;
        while (col >= 0 and row < N and board[row][col] != 1) :
            col -= 1;
            row += 1;

        if (col >= 0 and row < N and board[row][col] == 1) :
            attacking += 1;

        # Diagonally to the right up
        # (col increases and row
        # decreases)
        row = state[i] - 1
        col = i + 1;
        while (col < N and row >= 0 and board[row][col] != 1) :
            col += 1;
            row -= 1;

        if (col < N and row >= 0 and board[row][col] == 1) :
            attacking += 1;

    # Return pairs.
    return int(attacking / 2);
```

```
# A utility function that
# generates a board configuration
# given the state.
def generateBoard( board, state):
    fill(board, 0);
    for i in range(N):
        board[state[i]][i] = 1;

# A utility function that copies
# contents of state2 to state1.
def copyState( state1, state2):

    for i in range(N):
        state1[i] = state2[i];

# This function gets the neighbour
# of the current state having
# the least objective value
# amongst all neighbours as
# well as the current state.
def getNeighbour(board, state):

    # Declaring and initializing the
    # optimal board and state with
    # the current board and the state
    # as the starting point.
    opBoard = [[0 for _ in range(N)] for _ in range(N)]
    opState = [0 for _ in range(N)]

    copyState(opState, state);
    generateBoard(opBoard, opState);

    # Initializing the optimal
    # objective value
    opObjective = calculateObjective(opBoard, opState);

    # Declaring and initializing
    # the temporary board and
    # state for the purpose
    # of computation.
    NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]

    NeighbourState = [0 for _ in range(N)]
    copyState(NeighbourState, state);
    generateBoard(NeighbourBoard, NeighbourState);

    # Iterating through all
    # possible neighbours
    # of the board.
    for i in range(N):
        for j in range(N):

            # Condition for skipping the
            # current state
            if (j != state[i]) :

                # Initializing temporary
                # neighbour with the
                # current neighbour.
                NeighbourState[i] = j;
                NeighbourBoard[NeighbourState[i]][i] = 1;
                NeighbourBoard[state[i]][i] = 0;

                # Calculating the objective
                # value of the neighbour.
                temp = calculateObjective( NeighbourBoard, NeighbourState);

                # Comparing temporary and optimal
                # neighbour objectives and if
                # temporary is less than optimal
                # then updating accordingly.

                if (temp <= opObjective) :
                    opObjective = temp;
                    copyState(opState, NeighbourState);
                    generateBoard(opBoard, opState);

                # Going back to the original
                # configuration for the next
                # iteration.
                NeighbourBoard[NeighbourState[i]][i] = 0;
```

```
                NeighbourState[i] = state[i];
                NeighbourBoard[state[i]][i] = 1;

    # Copying the optimal board and
    # state thus found to the current
    # board and, state since c+= 1 doesn't
    # allow returning multiple values.
    copyState(state, opState);
    fill(board, 0);
    generateBoard(board, state);

def hillClimbing(board, state):

    # Declaring and initializing the
    # neighbour board and state with
    # the current board and the state
    # as the starting point.

    neighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
    neighbourState = [0 for _ in range(N)]

    copyState(neighbourState, state);
    generateBoard(neighbourBoard, neighbourState);

    while True:

        # Copying the neighbour board and
        # state to the current board and
        # state, since a neighbour
        # becomes current after the jump.

        copyState(state, neighbourState);
        generateBoard(board, state);

        # Getting the optimal neighbour

        getNeighbour(neighbourBoard, neighbourState);

        if (compareStates(state, neighbourState)) :

            # If neighbour and current are
            # equal then no optimal neighbour
            # exists and therefore output the
            # result and break the loop.

            printBoard(board);
            break;

        elif (calculateObjective(board, state) == calculateObjective( neighbourBoard,neighbourState)):

            # If neighbour and current are
            # not equal but their objectives
            # are equal then we are either
            # approaching a shoulder or a
            # local optimum, in any case,
            # jump to a random neighbour
            # to escape it.

            # Random neighbour
            neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N;
            generateBoard(neighbourBoard, neighbourState);

# Driver code
state = [0] * N
board = [[0 for _ in range(N)] for _ in range(N)]

# Getting a starting point by
# randomly configuring the board
configureRandomly(board, state);

# Do hill climbing on the
# board obtained
hillClimbing(board, state);

# This code is contributed by phasing17.
```

```
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0
```

```
    0 0 0 0 0 0 1 0
    0 0 0 0 1 0 0 0
```

```python
N = 4 # Size of the board

# A utility function that prints the board representation based on the state
def printBoard(state):
    # Create a 2D array initialized to zeros to represent the board
    board = [[0] * N for _ in range(N)]

    # Place a queen on the board based on the current state
    for col in range(N):
        board[state[col]][col] = 1  # Place a queen in the specified row for each column

    # Print the board row by row
    for row in board:
        print(*row)  # Unpack the row list for clean output
    print()  # Print a blank line for better readability

# A utility function that calculates the number of attacking queens
def calculateObjective(state):
    attacking = 0  # Initialize the count of attacking queens to zero

    # Check each pair of queens to see if they attack each other
    for i in range(N):
        for j in range(i + 1, N):  # Only check queens to the right to avoid double counting
            if state[i] == state[j]:  # Check if they are in the same row
                attacking += 1  # Increment the count if they are in the same row
            # Check if they are on the same diagonal
            if abs(state[i] - state[j]) == abs(i - j):
                attacking += 1  # Increment the count for diagonal attack

    return attacking  # Return the total number of attacking pairs

# This function finds the neighbor of the current state with the least objective value
def getNeighbour(state):
    best_state = state[:]  # Start with the current state as the best state
    best_objective = calculateObjective(state)  # Calculate the current objective value (attacks)

    # Loop through each queen to find a better configuration
    for i in range(N):
        original_row = state[i]  # Store the original row position of the queen

        # Try moving the queen to each possible row in the current column
        for j in range(N):
            if j != original_row:  # Skip the current row
                state[i] = j  # Move the queen to the new row
                current_objective = calculateObjective(state)  # Calculate attacks for the new state

                # Check if this new state has fewer attacks
                if current_objective < best_objective:
                    best_objective = current_objective  # Update best objective
                    best_state = state[:]  # Update best state to the new configuration

        state[i] = original_row  # Restore the original position of the queen

    return best_state  # Return the best state found

# The main function that implements the hill climbing algorithm
def hillClimbing(state):
    while True:  # Loop indefinitely until a solution is found or no better state exists
        current_objective = calculateObjective(state)  # Calculate attacks for the current state

        # Check if a solution has been found (no attacks)
        if current_objective == 0:
            print("Final board configuration:")  # Indicate that a solution has been found
            printBoard(state)  # Print the final configuration
            break  # Exit the loop

        next_state = getNeighbour(state)  # Find the best neighboring state

        # Check if no better state was found (stuck in a local minimum)
        if next_state == state:
            print("Stuck in local minimum.")  # Indicate that no improvement is possible
            printBoard(state)  # Print the current state
            break  # Exit the loop
        else:
            state = next_state  # Update the current state to the better neighbor

# Driver code
state = [1,2,3, 4]  # Initial position of queens, specifying their row positions
```

```
# Run the hill climbing algorithm on the initial state
hillClimbing(state)
```

```
Stuck in local minimum.
1 0 0 0
0 0 0 1
0 1 0 0
0 0 1 0
```

```
from random import randint

N = 8

# A utility function that configures
# the 2D array "board" and
# array "state" randomly to provide
# a starting point for the algorithm.
def configureRandomly(board, state):
    for i in range(N):
        state[i] = randint(0, 100000) % N
        board[state[i]][i] = 1

# A utility function that prints the 2D array "board".
def printBoard(board):
    for i in range(N):
        print(*board[i])

# A utility function that prints the array "state".
def printState(state):
    print(*state)

# A utility function that compares two arrays, state1 and state2 and returns True if equal.
def compareStates(state1, state2):
    return all(state1[i] == state2[i] for i in range(N))

# A utility function that fills the 2D array "board" with values "value"
def fill(board, value):
    for i in range(N):
        for j in range(N):
            board[i][j] = value

# This function calculates the objective value of the state (queens attacking each other)
def calculateObjective(board, state):
    attacking = 0
    for i in range(N):
        row = state[i]

        # Check row, diagonal left and right
        for j in range(i + 1, N):
            if state[j] == row or abs(state[j] - row) == abs(j - i):
                attacking += 1
    return attacking

# A utility function that generates a board configuration given the state.
def generateBoard(board, state):
    fill(board, 0)
    for i in range(N):
        board[state[i]][i] = 1

# A utility function that copies contents of state2 to state1.
def copyState(state1, state2):
    for i in range(N):
        state1[i] = state2[i]

# This function gets the neighbour of the current state having the least objective value
def getNeighbour(board, state):
    opBoard = [[0 for _ in range(N)] for _ in range(N)]
    opState = [0 for _ in range(N)]
    copyState(opState, state)
    generateBoard(opBoard, opState)

    opObjective = calculateObjective(opBoard, opState)

    NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
    NeighbourState = [0 for _ in range(N)]
    copyState(NeighbourState, state)
    generateBoard(NeighbourBoard, NeighbourState)

    for i in range(N):
        for j in range(N):
            if i != state[i]:
```

```
            if j != state[i]:
                NeighbourState[i] = j
                NeighbourBoard[NeighbourState[i]][i] = 1
                NeighbourBoard[state[i]][i] = 0

                temp = calculateObjective(NeighbourBoard, NeighbourState)

                if temp < opObjective:
                    opObjective = temp
                    copyState(opState, NeighbourState)
                    generateBoard(opBoard, opState)

                NeighbourBoard[NeighbourState[i]][i] = 0
                NeighbourState[i] = state[i]
                NeighbourBoard[state[i]][i] = 1

    copyState(state, opState)
    fill(board, 0)
    generateBoard(board, state)

def hillClimbing(board, state):
    neighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
    neighbourState = [0 for _ in range(N)]

    copyState(neighbourState, state)
    generateBoard(neighbourBoard, neighbourState)

    while True:
        copyState(state, neighbourState)
        generateBoard(board, state)

        getNeighbour(neighbourBoard, neighbourState)

        if compareStates(state, neighbourState):
            print("Stuck in local minimum.")
            printBoard(board)
            break
        elif calculateObjective(board, state) == calculateObjective(neighbourBoard, neighbourState):
            neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N
            generateBoard(neighbourBoard, neighbourState)

# Driver code
state = [0] * N
board = [[0 for _ in range(N)] for _ in range(N)]

# Manually configure a local minimum to start
# Example of a configuration that may lead to local minimum
state = [0, 2, 4, 1, 3, 5, 7, 6]  # This configuration has multiple pairs attacking each other
generateBoard(board, state)

# Perform hill climbing on the configured board
hillClimbing(board, state)
```

```
Stuck in local minimum.
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
```