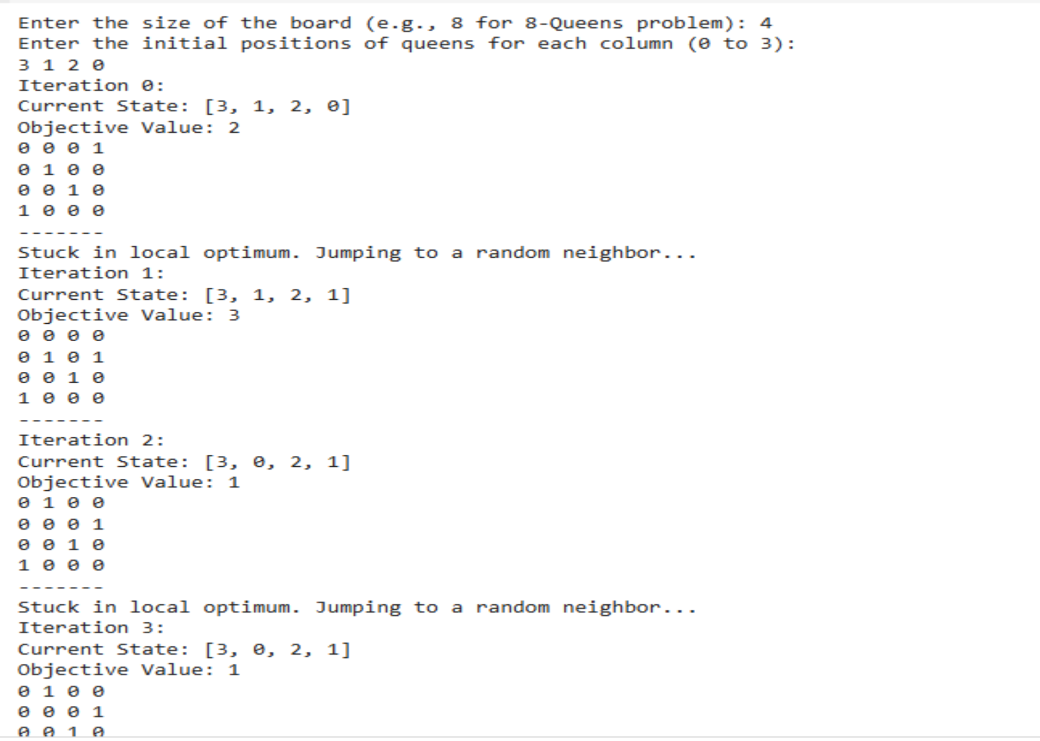
5. N queens using Hill climbing algorithm

Code:

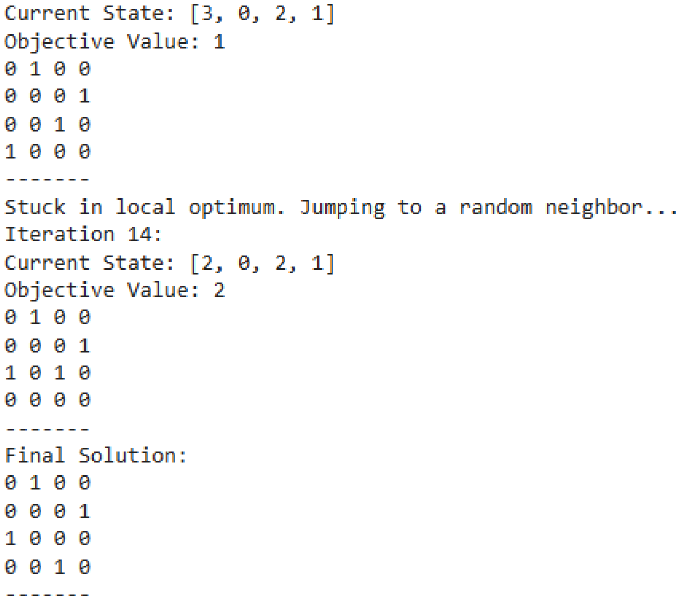
```
from random import randint

# Function to print the board
def printBoard(board, N):
    for i in range(N):
        print(" ".join(map(str, board[i])))
    print("-" * (2 * N - 1))

# Function to calculate the objective value
def calculateObjective(board, state, N):
    attacking = 0
    for i in range(N):
        row = state[i]

        # Check row conflicts
        col = i - 1
        while col >= 0 and board[row][col] != 1:
            col -= 1
        if col >= 0 and board[row][col] == 1:
            attacking += 1

        col = i + 1
        while col < N and board[row][col] != 1:
            col += 1
        if col < N and board[row][col] == 1:
            attacking += 1

        # Check diagonal conflicts
        for d_row, d_col in [(-1, -1), (1, 1), (1, -1), (-1, 1)]:
            r, c = row + d_row, i + d_col
            while 0 <= r < N and 0 <= c < N and board[r][c] != 1:
                r += d_row
                c += d_col
            if 0 <= r < N and 0 <= c < N and board[r][c] == 1:
                attacking += 1

    return attacking // 2

# Function to generate the board from the state
def generateBoard(board, state, N):
    for i in range(N):
        for j in range(N):
            board[i][j] = 0
    for i in range(N):
        board[state[i]][i] = 1

# Function to get the best neighbor
def getNeighbour(board, state, N):
    opState = state[:]
    opBoard = [[0] * N for _ in range(N)]
    generateBoard(opBoard, opState, N)
    opObjective = calculateObjective(opBoard, opState, N)

    for i in range(N):
        original_row = state[i]
        for new_row in range(N):
            if new_row != original_row:
                state[i] = new_row
                generateBoard(board, state, N)
                tempObjective = calculateObjective(board, state, N)
                if tempObjective < opObjective:
                    opObjective = tempObjective
                    opState = state[:]
        state[i] = original_row
    generateBoard(board, opState, N)
    return opState, opObjective

# Hill climbing algorithm
def hillClimbing(N, initial_state):
    board = [[0] * N for _ in range(N)]
    state = initial_state[:]
    generateBoard(board, state, N)

    iteration = 0
    while True:
        print(f"Iteration {iteration}:")
        print(f"Current State: {state}")
        currentObjective = calculateObjective(board, state, N)
        print(f"Objective Value: {currentObjective}")
        printBoard(board, N)

        nextState, nextObjective = getNeighbour(board, state, N)

        # Break if we reach an optimal solution with objective 0
        if nextObjective == 0:
            print("Final Solution:")
            printBoard(board, N)
            break
```

```python
        # If stuck in a local optimum, pick a random neighboring state
        if nextObjective >= currentObjective:
            print("Stuck in local optimum. Jumping to a random neighbor...")
            state[randint(0, N - 1)] = randint(0, N - 1)
            generateBoard(board, state, N)
        else:
            state = nextState  # Move to the next better state

        iteration += 1

# Main code to accept user input
if __name__ == "__main__":
    N = int(input("Enter the size of the board (e.g., 8 for 8-Queens problem): "))
    print(f"Enter the initial positions of queens for each column (0 to {N-1}):")
    initial_state = list(map(int, input().split()))

    if len(initial_state) != N or any(pos < 0 or pos >= N for pos in initial_state):
        print("Invalid input. Please ensure each queen position is within the board size.")
    else:
        hillClimbing(N, initial_state)
```

Output:

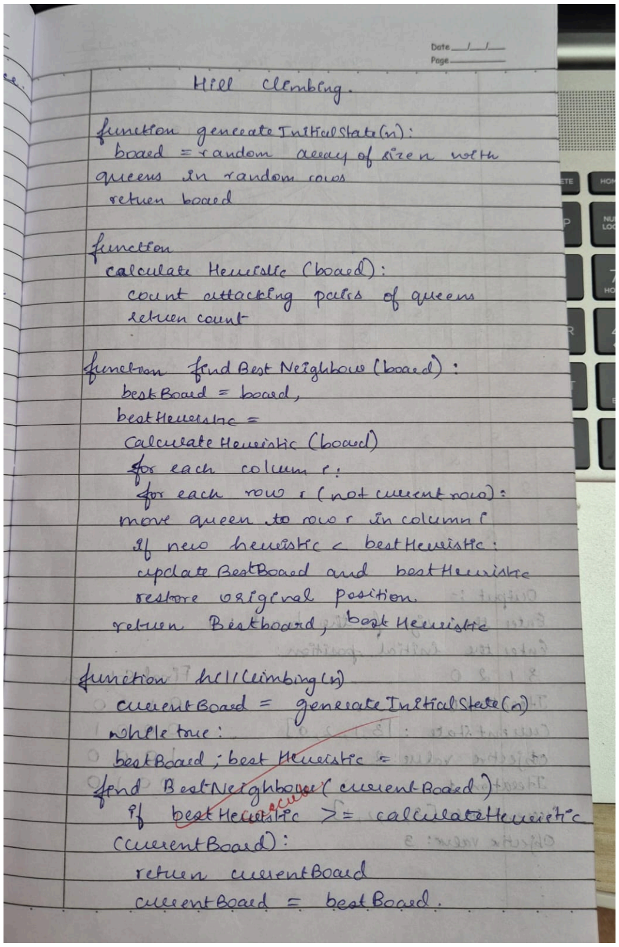


```
Enter the size of the board (e.g., 8 for 8-Queens problem): 4
Enter the initial positions of queens for each column (0 to 3):
3 1 2 0
Iteration 0:
Current State: [3, 1, 2, 0]
Objective Value: 2
0 0 0 1
0 1 0 0
0 0 1 0
1 0 0 0
-------
Stuck in local optimum. Jumping to a random neighbor...
Iteration 1:
Current State: [3, 1, 2, 1]
Objective Value: 3
0 0 0 0
0 1 0 1
0 0 1 0
1 0 0 0
-------
Iteration 2:
Current State: [3, 0, 2, 1]
Objective Value: 1
0 1 0 0
0 0 0 1
0 0 1 0
1 0 0 0
-------
Stuck in local optimum. Jumping to a random neighbor...
Iteration 3:
Current State: [3, 0, 2, 1]
Objective Value: 1
0 1 0 0
0 0 0 1
0 0 1 0
```



```
Current State: [3, 0, 2, 1]
Objective Value: 1
0 1 0 0
0 0 0 1
0 0 1 0
1 0 0 0
-------
Stuck in local optimum. Jumping to a random neighbor...
Iteration 14:
Current State: [2, 0, 2, 1]
Objective Value: 2
0 1 0 0
0 0 0 1
1 0 1 0
0 0 0 0
-------
Final Solution:
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
-------
```

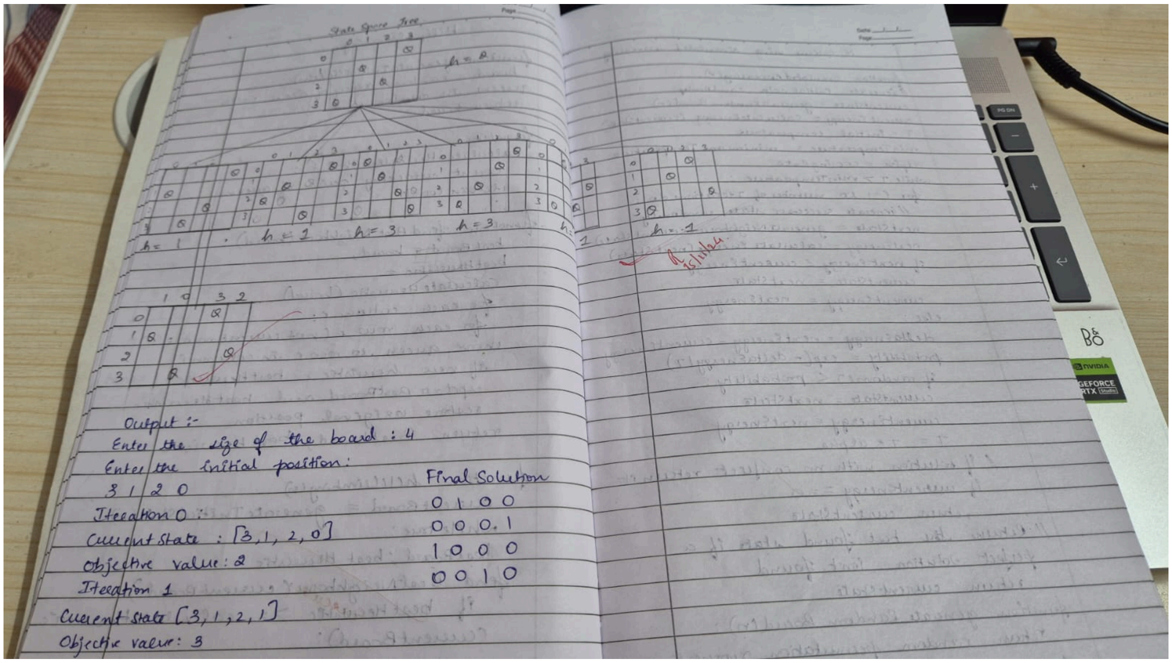
Observation book screenshots:



Date __/__/__
Page ____

Hill climbing.

```
function generateInitialState(n):
  board = random array of size n with
queens in random rows
  return board

function
  calculate Heuristic (board):
    count attacking pairs of queens
    return count

function find Best Neighbour (board):
    bestBoard = board,
    bestHeuristic =
    Calculate Heuristic (board)
     for each column c:
      for each row r (not current row):
    move queen to row r in column c
     if new heuristic < bestHeuristic:
     update BestBoard and bestHeuristic
     restore original position.
   return Bestboard, best Heuristic

function hillclimbing(n)
   currentBoard = generateInitialState(n)
  while true:
   bestBoard ; best Heuristic =
  find BestNeighbour(current Board)
     if bestHeuristic >= calculateHeuristic
   (currentBoard):
     return currentBoard
     currentBoard = bestBoard.
```



State Space Tree
h = 2
h = 1
h = 1
h = 3
h = 3
h = 1

```
Output :-
Enter the size of the board : 4
Enter the initial position:
3 1 2 0
Iteration 0 :
Current State : [3, 1, 2, 0]
Objective value: 2
Iteration 1
Current state [3, 1, 2, 1]
Objective value: 3
```

```
Final Solution
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```