**DSA MINI PROJECT**

**NAME:** ARYAN GAWDE

CLASS: SY Bsc IT

**ROLL NO:** A08

**SAP ID:** 45207220012

**Topic: N-Queens problem**

* **Introduction:**

The N-Queens Problem is a classic combinatorial puzzle that has been studied extensively in the field of computer science and mathematics. It is a challenging problem that involves placing N chess queens on an N×N chessboard in such a way that no two queens threaten each other, meaning that no two queens can share the same row, column, or diagonal. This problem has various applications in computer science, including artificial intelligence, constraint satisfaction problems, and optimization algorithms. This report provides an overview of the N-Queens Problem, its significance, and a pseudocode representation of a typical solution algorithm**.**The N-Queens problem is a classic combinatorial problem that has been studied for centuries. It is a seemingly simple problem to state, but it is actually quite difficult to solve for large values of N.The problem is to place N chess queens on an N×N chessboard so that no two queens attack each other. Two queens attack each other if they are on the same row, column, or diagonal.

* **Problem Statement:**

The N-Queens Problem can be formally stated as follows:

Input: An integer N, which represents the size of the chessboard.

Output: A placement of N queens on the N×N chessboard such that no two queens threaten each other.

* **Significance:**

The N-Queens Problem has several significant aspects:

1. Combinatorial Challenge: It is a classic example of a combinatorial problem that requires finding a valid arrangement of pieces on a board while satisfying specific constraints. Solving this problem for larger N values becomes increasingly difficult.
2. Algorithmic Research: The N-Queens Problem has been a subject of extensive research in computer science, leading to the development of various algorithms and heuristics for solving it efficiently.
3. Applications: It serves as a foundational problem in constraint satisfaction, artificial intelligence, and optimization, with applications in tasks such as scheduling, layout design, and error-correcting codes.

* **Here is a more detailed explanation of the backtracking algorithm for the N-Queens problem:**

1. Start with an empty chessboard.
2. Place a queen in the first column, in the first row.
3. Recursively place queens in the remaining columns, starting from the second column. For each column:

* Try to place a queen in each row of the column.
* If a row is safe, place the queen in that row and move on to the next column.
* If no row is safe, backtrack and remove the queen from the previous column.

1. If the algorithm reaches the last column and there is a queen in each row, then the algorithm has found a solution.
2. If the algorithm backtracks all the way to the beginning and finds no solution, then there is no solution to the N-Queens problem.

* **Pseudocode for Solving the N-Queens Problem:**

The N-Queens problem is a classic problem in which you need to place N chess queens on an N×N chessboard in such a way that no two queens threaten each other. This means that no two queens can be in the same row, column, or diagonal. Here's a Python implementation of the N-Queens problem using backtracking:

**CODE:**

def is\_safe(board, row, col, N):

# Check column

for i in range(row):

if board[i][col] == 1:

return False

# Check upper-left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check upper-right diagonal

for i, j in zip(range(row, -1, -1), range(col, N)):

if board[i][j] == 1:

return False

return True

def solve\_nqueens\_util(board, row, N):

# If all queens are placed, return True

if row >= N:

return True

for col in range(N):

if is\_safe(board, row, col, N):

board[row][col] = 1 # Place the queen

# Recur to place the rest of the queens

if solve\_nqueens\_util(board, row + 1, N):

return True

# Backtrack if placing queen here doesn't lead to a solution

board[row][col] = 0

return False

def solve\_nqueens(N):

board = [[0 for \_ in range(N)] for \_ in range(N)]

if not solve\_nqueens\_util(board, 0, N):

print("Solution does not exist")

return False

print("Solution:")

for row in board:

print(" ".join(["Q" if cell == 1 else "." for cell in row]))

return True

# Example usage:

n = 8 # Number of queens

solve\_nqueens(n)

This code will print a solution to the N-Queens problem for the given **n** value. For **n = 8**, the output would be one of the possible solutions:

* **OUTPUT:**

Q . . . . . . .

. . . . Q . . .

. . . . . . . Q

. . . . . Q . .

. . . . . . Q .

. Q . . . . . .

. . . Q . . . .

. . Q . . . . .

* **Conclusion:**

The N-Queens Problem is a fascinating and important problem in computer science and mathematics. It challenges us to find a valid placement of N queens on an N×N chessboard without any of them threatening each other. While the problem becomes increasingly complex as N grows, it has been the subject of extensive research, resulting in various algorithms and techniques for finding solutions. The provided pseudocode outlines a backtracking algorithm that can be used as a starting point to solve the N-Queens Problem, demonstrating the core logic of how to approach this problem computationally.