The Superior University, Lahore

LAB-TASK (4) (Spring-2026)

Course Title:	Programming for AI (Lab)				Course Code:	-	Credit Hours: 1
Instructor:	Sir Rasikh				Program:	BSDS	
Semester:	4th	Batch:	Morning	Section:	4A	Date:	-
Time Allowed:	-				Maximum Marks:		N/A
Student's Name:	Shumaila Maryam				Roll No.	SU92-BSDSM-S25-062	

Work Assigned:

Implement the N-Queens Problem (Dynamic)

DEPARTMENT OF SOFTWARE ENGINEERING

Introduction

The N-Queens Problem is one of the most famous puzzles in computer science and artificial intelligence. It challenges us to place N queens on an N×N chessboard such that no two queens can attack each other. While the concept is simple, solving it requires logical thinking and systematic exploration of possibilities. The difficulty comes from the fact that queens can attack in all directions: horizontally, vertically, and diagonally.

This problem isn't just a puzzle it teaches us how computers can solve complex constraint satisfaction problems by exploring possibilities systematically and intelligently. When we try to solve the 4-Queens or 8Queens problem by hand, we quickly realize how many possibilities need to be checked. A computer can check thousands of possibilities in seconds, making this problem a perfect learning tool for understanding algorithms and problem-solving techniques.

Why I Made This

This project was created with several learning goals in mind:

- Understanding State Representation: Learning how to represent a problem in a way that a computer can understand. For N-Queens, we represent the solution as a list where each index is a row and the value is the column where a queen is placed.
- Learning Backtracking Algorithms: Backtracking is a powerful technique where we try a solution, and if it doesn't work, we undo our choices and try something different. This is exactly what's needed for N-Queens.
- Practicing Constraint Checking: Understanding how to check multiple constraints simultaneously. In N-Queens, we need to ensure no two queens share the same column or diagonal.
- Problem-Solving Skills: Developing the ability to think about problems differently. Instead of trying every possible arrangement (which would take forever), we use intelligent constraints to reduce the number of possibilities we need to check.

- Code Implementation: Writing clean, working code that actually solves the problem, not just theory.

Through this project, I understood that many real-world problems—like scheduling, routing, and resource allocation—use similar principles to the N-Queens Problem. Once you can solve N-Queens, you have the foundation to solve many other challenging problems.

How It Works

The Problem Definition

The N-Queens Problem has a simple definition:

- We have an N×N chessboard
- We need to place N queens on the board
- No two queens should attack each other
- A queen can attack any piece on the same row, column, or diagonal

For example, in the 4-Queens problem, we need to place 4 queens on a 4×4 board. This problem has exactly 2 solutions.

Representing the Solution

The program represents each solution as a list where:
• The index represents the row number
• The value represents the column number where the queen is placed

For example: [1, 3, 0, 2] means:
• Row 0: Queen at column 1
• Row 1: Queen at column 3
• Row 2: Queen at column 0
• Row 3: Queen at column 2

This representation automatically ensures no two queens are on the same row (since each row has exactly one queen).

Checking If a Position Is Safe

Before placing a queen at a position, we need to check if it's safe:

- Column Check: Is there already a queen in this column?
- Diagonal Check: Are there any queens on the diagonals that pass through this position?

The diagonals can be checked mathematically:
• For a queen at position (i, col) and another at (j, prev_col):
  • If prev_col - col == j - i: They're on the same diagonal
  • If prev_col - col == i - j: They're on the same diagonal

The Backtracking Algorithm
The solution uses backtracking, which works like this:

1. Try to place: Try to place a queen in the current row
2. Check constraints: For each column, check if placing a queen there is safe
3. Move forward: If safe, place the queen and move to the next row
4. Recurse: Try to solve for the remaining rows
5. Backtrack: If we can't find a solution with this placement, remove the queen and try the next column
6. Found it: When all N queens are placed successfully, we've found a solution

Algorithm in Detail

The is_safe() Function
This function checks if placing a queen at position (row, col) is safe:

```
def is_safe(self, board, row, col):
    for i in range(row):
        if board[i] == col:
            return False
    for i in range(row):
        if board[i] - col == i - row:
            return False
        if board[i] - col == row - i:
            return False

    return True
```

We only check above the current row because we haven't placed any queens below yet.

The solve() Function

This is where backtracking happens:

```
def solve(self, board, row):
    if row == self.n:
        self.solutions.append(board[:])
        return

    for col in range(self.n):
        if self.is_safe(board, row, col):
            board[row] = col
            self.solve(board, row + 1)
            board[row] = -1
```

Complexity Analysis

Time Complexity: O(N!) in the worst case, but with pruning it's much faster

Space Complexity: O(N)

Why backtracking is efficient:

- We don't check all N! arrangements
- We prune invalid branches early
- For 8-Queens, we find solutions in milliseconds instead of years

Learning Points

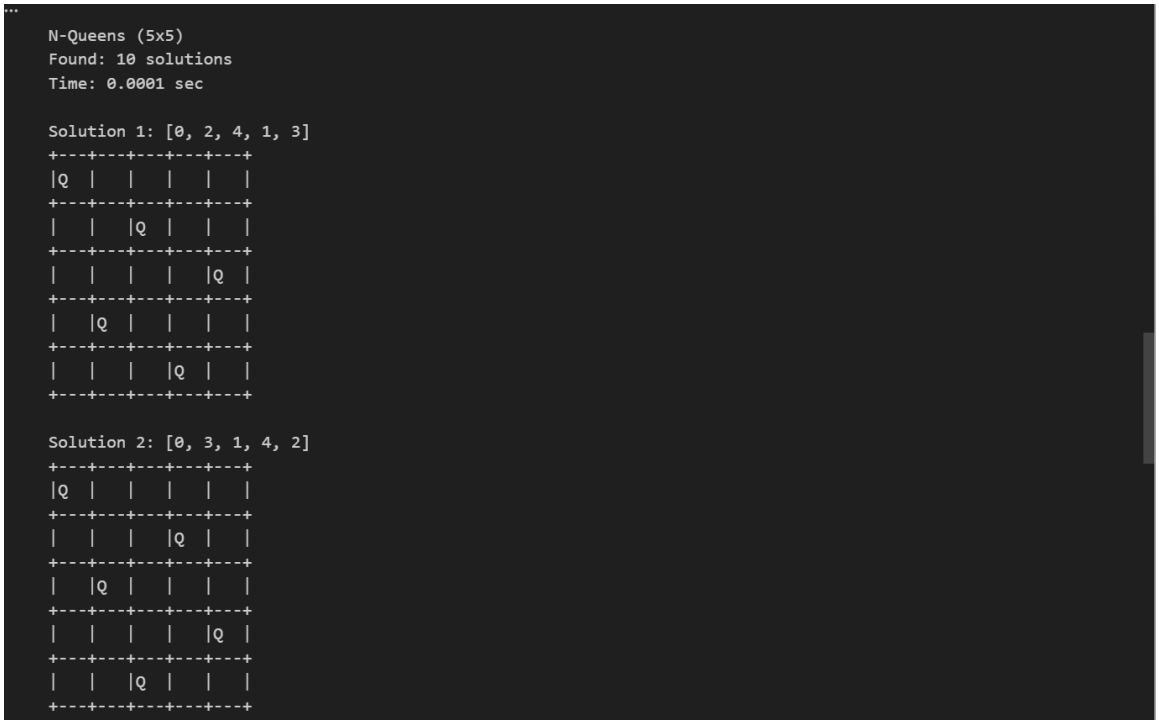
- Representation is Everything: How we represent a problem affects how easy it is to solve.
- Constraint Checking: Many problems are just about checking constraints efficiently.
- Backtracking is Powerful: It allows us to avoid checking invalid paths by failing fast.
- Optimization Matters: A good algorithm can solve something in seconds that would take hours.

- Systematic Problem-Solving: Breaking down a complex problem into smaller checks makes it manageable.

Facts

- 4-Queens: 2 solutions
- 8-Queens: 92 solutions (the classic version)
- 12-Queens: 14,200+ solutions
- The first person to solve the 8-Queens puzzle was Carl Friedrich Gauss in 1850
- The N-Queens problem is NP-hard, meaning there's no known algorithm to solve it quickly for all cases
- People use advanced versions of N-Queens to solve real-world problems like placing radio antennas

Output:



Conclusion

The N-Queens Problem is a beautiful example of how computers solve complex problems through systematic exploration and intelligent constraint checking. While it seems difficult to solve by hand, the backtracking algorithm makes it solvable for reasonable board sizes.

• Complex problems can often be solved by breaking them into smaller constraint checks
• The way we represent a problem matters greatly
• Backtracking is a powerful technique for exploring possibilities systematically
• What seems impossible becomes manageable with the right approach