N–QUEENS

Concepts of Programming

Contents:

1. Introduction and Overview
   i. Project Idea
   ii. Problem Description
2. Applied Paradigms
   i. Pure Functional
   ii. Imperative
3. Results and Analysis
   i. Results and Comparison
   ii. Analysis and Discussion

Introduction and Overview

1. Project Idea:

The N-Queens project aims to design and implement a program capable of solving the N-Queen problem for different board sizes.

The value of N is selected by user, allowing the program to work with small and large boards alike.

To deal with this challenge, the project explores 2 different programming paradigms:

- Pure Functional
- Imperative

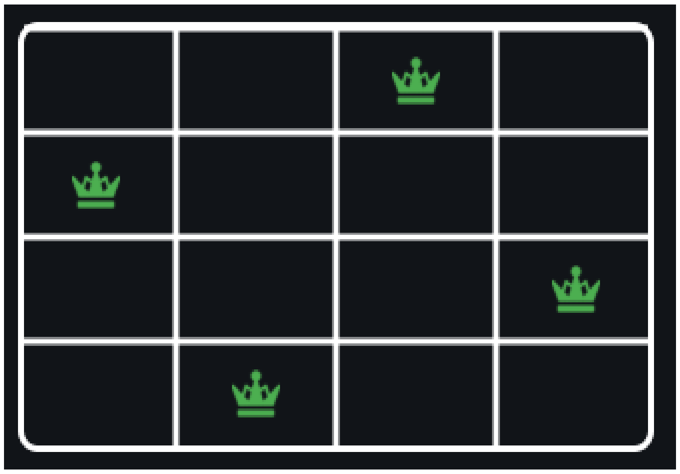
Using multiple approaches allow comparison of performance, effectiveness, and search behavior.

2. Problem Description:

In the N-Queen problem, the task is to arrange N queens on a NxN board so that no queen is attacking any other queen.

So to satisfy that no 2 queens must share the same row, column, or diagonal

As N increases, the number of possible board states grows exponentially



Valid solution for 4-Queens

Applied Paradigm

1. Pure Functional:

Pure Functional programming emphasizes immutability, recursion, and no side effects

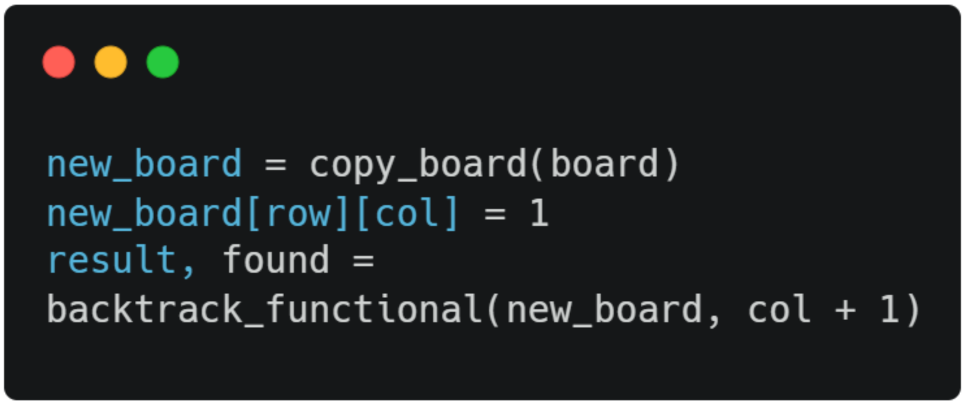
Functions don’t modify external state, instead, they return new values

Loops are avoided, and recursion or higher-order functions are used to process data

In the N-Queens code:

- Functions like *is_safe_row*, *is_safe_upper* are recursive and don’t modify the board, they only check conditions
- *copy_board* creates a new board each time a queen is placed instead of modifying the original board

- *try_rows* and *backtrack_functional* recursively explore rows and columns without altering the original board
- Each function returns a new board state or Boolean avoiding the side effects



```
new_board = copy_board(board)
new_board[row][col] = 1
result, found =
backtrack_functional(new_board, col + 1)
```

Here the *new_board* is a new copy of the board with the queen placed, the original board remains unchanged

Advantages:

- Predictable behavior
- More suitable for concurrency
- Easier testing

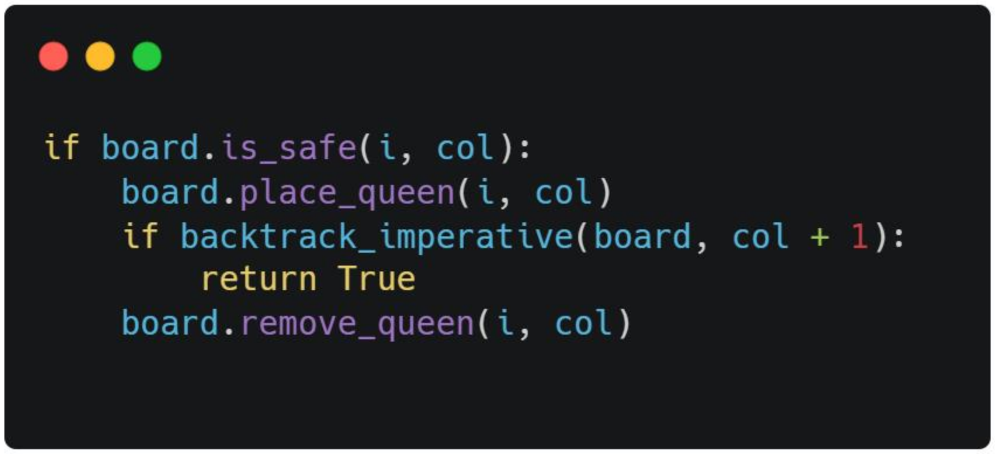
2. Imperative:

Imperative programming focuses on changing program state through statements, loops, and explicit instructions

The program describes how to achieve a goal by updating variables step by step

In the N-Queens code:

- *backtrack_imperative* uses loops to iterate through rows *for i in range(board.N)*
- It directly *modifies the board via board.place_queen(i,col) and board.remove_queen(i,col) which has side effects*
- The board is a mutable object, recursion is still used but the state is constantly updated in place rather than creating copies



```python
if board.is_safe(i, col):
    board.place_queen(i, col)
    if backtrack_imperative(board, col + 1):
        return True
    board.remove_queen(i, col)
```

Here the *same board object is modified and restored at each step, instead of creating new copies*

Advantages:

- More efficient in memory and time
- Easier to implement step by step

Paradigm	Functional	Imperative
Mutability	Immutable	Mutable
Iteration	Recursion	Loop

Results and Analysis

1. Results and Comparison

- To keep comparison fair, we used the same start state and same board size
- We ran each paradigm several times to get average
- For N=5

Paradigm	Avg Time
Functional	0.000100
Imperative	0.000041

- For N=15

Paradigm	Avg Time
Functional	0.158944
Imperative	0.040047

2. Analysis and Discussion

1)Functional:

- Uses less memory
- Executes faster

2)Imperative:

- Introduces significant overhead which slows down algorithm

Paradigm	Advantage	Disadvantage
Functional	Fast, memory-efficient and straightforward	Side effects can make it harder to understand
Imperative	Predictable, avoid side effects	Slower and higher memory usage

So in the N-Queens problem, using imperative paradigm is more efficient