N-Queens Puzzle Documentation

The N-Queens puzzle is a classic problem in computer science and mathematics. The task is to place N queens on an N×N chessboard in a way that ensures no two queens can attack each other.

A queen can attack another queen if they are placed on the same row, the same column, or the same diagonal.
The goal of the problem is to find all valid arrangements where all queens are positioned safely.

This file provides an implementation of the N-Queens solver using the chosen programming paradigm (functional or imperative) to demonstrate key differences such as:

- mutable vs. immutable data
- recursion vs. loops
- pure functions vs. side effects
- declarative vs. state-based logic

Evaluation Criteria	**Imperative Paradigm (imperative.py)**	**Functional Paradigm (functional.py)**
Implementation	Relies on **Explicit Control Flow**. The solution is built by modifying a reserved memory space (List) step-by-step, manually managing the "undo" operation (backtracking) when a dead end is reached.	Relies on **Data Flow** and **Expression Evaluation**. The solution is built by passing data through functions, relying on Pure Functions that do not depend on or modify external state.
Mutability Vs Immutability	**Mutable:** Uses a `List` structure. The algorithm modifies the list contents in-place using indices (`cols[row] = c`) and manually reverts changes (`cols[row] = -1`).	**Immutable:** Uses a `Tuple` structure. Tuples cannot be modified; instead, a new tuple instance is created for every recursive step (`cols + (c,)`), eliminating the need for a manual undo step.
Recursion Vs Loop	**Loops + Recursion:** Uses explicit `for` loops to iterate over columns. While it uses recursion to advance rows, the primary control logic involves mutating variables inside a loop.	**Pure Recursion (Declarative):** Relies entirely on recursion. Traditional loops are replaced by **List Comprehensions** (a declarative approach) to define the set of valid solutions.
Higher order programming	**Low / None:** The code is largely procedural. Functions execute direct commands and statements without passing other functions as arguments or returning them.	**High:** Utilizes Higher-Order Functions like `all()` (to check multiple conditions at once) and List Comprehensions, which act as syntactic sugar for `map` and `filter` operations.

1. Understand the Core Differences Between Paradigms

The project serves as a direct comparative study of **State Mutation** versus **Immutability**.

- In the **Imperative** approach (imperative.py), the focus is on *how* to calculate the result by explicitly changing the computer's state (using a mutable list and backtracking steps). This highlights the concept of **Control Flow**—telling the computer exactly what to do and when to undo it.
- In the **Functional** approach (functional.py), the focus is on *what* the result is. By using immutable Tuples and Recursion, the project demonstrates **Data Flow**—passing information through a chain of functions without side effects, where the "state" is preserved in the function stack rather than a variable.

2. Explore Strengths and Weaknesses

Implementing both versions exposes the trade-offs inherent in each paradigm:

- **Imperative Strengths:** It is often more intuitive for defining step-by-step algorithms and can be more memory-efficient by reusing a single data structure (the board list).
- **Imperative Weaknesses:** Managing state changes manually (like the "undo" step in backtracking) increases the risk of bugs and makes the code harder to debug in parallel environments.
- **Functional Strengths:** The code is more declarative and easier to reason about because functions are "pure" (output depends only on input). This reduces bugs related to hidden state changes.
- **Functional Weaknesses:** It can be less intuitive for those accustomed to loops, and creating new data structures (new Tuples) for every step can incur a higher memory overhead compared to in-place modification.

3. Develop a Deeper Appreciation for Problem-Solving

- The **Imperative** mindset encourages thinking in terms of **Time**: "First place a queen, then check, then maybe remove it."
- The **Functional** mindset encourages thinking in terms of **Definitions**: "A solution is a set of columns where no queens attack each other."

Detailed Code Analysis

This section analyzes how each file satisfies the paradigm constraints.

A. Imperative Paradigm (imperative.py)

This implementation demonstrates the traditional imperative style through:

1. **State Mutation:**
   - The state is initialized as a mutable list: cols = [-1] * N.
   - The algorithm modifies this single object in memory and explicitly handles the "Backtracking" logic by resetting the value:

```
cols[row] = c      # Do: Place Queen (Mutate state)
backtrack(row + 1)
cols[row] = -1     # Undo: Remove Queen (Revert mutation)
```

2. **Explicit Control Flow:**
   - It uses a standard for c in range(N) loop to drive the logic, focusing on *how* to iterate rather than *what* the result should be.

B. Functional Paradigm (functional.py)

This implementation demonstrates functional principles through:

1. **Immutability:**
   - The state is passed as a Tuple.
   - Instead of modifying the existing state, a new version of the data is created and passed to the next function call:

```
# Pass a NEW tuple (cols + (c,)) to the recursive call
extend_solutions(cols + (c,))
```

2. **Declarative Style & Higher-Order Functions:**
   - Instead of a loop with if statements to check safety, it uses the all() function with a generator expression. This reads declaratively: "Return True if all checks pass":

```
return all(cc != col ... for rr, cc in enumerate(cols))
```

   - The main solver uses a **List Comprehension** to strictly define the solution set, combining mapping and filtering in a single expression:

```
return [s for c in range(N) if safe(...) for s in extend_solutions(...)]
```

3. Conclusion

- The **Imperative** approach focuses on **"How to do it"**: by describing the exact steps to change the program state (memory) and recover from invalid states.
- The **Functional** approach focuses on **"What is the solution"**: by defining the relationships between data (safe columns, valid extensions) without managing the underlying state changes or side effects.