REPORT: Sudoku Solver

→By: Anirudh Pratap Singh Yadav (24070722004)

Problem Statement: Write a program in C to solve a given Sudoku puzzle. The puzzle will be provided as a 9x9 grid where empty cells are represented by zeros. Your program must fill in the missing numbers so that every row, every column, and every 3x3 sub-grid contains the digits 1 to 9 exactly once.

Introduction: This report outlines my individual contribution to the Sudoku Solver project developed in the C programming language. The primary objective of the assignment was to implement a Sudoku-solving application using the backtracking algorithm.

Initial Idea: The initial idea for solving the Sudoku puzzle was to use a brute-force approach. This method would involve trying all possible combinations of numbers in the empty cells until the puzzle was solved.

→Why Didn't I choose the initial idea?

→The Brute Force method would involve trying all possible combinations of number in the empty cells so that the sudoku remains valid and solved. But as it was 9x9 grid, there was a probability of large number of configurations which would make it inefficient and consume more memory and time.

→Reason of Rejection?

- Inefficient for large puzzles
- High time complexity
- Difficult to optimise or find errors
- More memory
- Less Effective and no proper exception handling

Method Used: Backtracking

Instead of brute-force, a backtracking algorithm was used. This technique is more efficient and suitable for solving constraint satisfaction problems like N-Queen, Sudoku. Backtracking systematically searches for a solution by trying out different values in a cell and reverting (backtracking) when a conflict is encountered.

Challenges Faced:

Challenge 1: Input Validation

- **Users could enter numbers outside the valid range (0-9).**

  Solution: Implemented input checks and re-prompted the user if the value was invalid.

Challenge 2: Validity Check Efficiency

- **Needed to efficiently check if a number can be placed.**

  Solution: Created a helper function valid_ puzzle () to check for row, column, and 3x3 grid validity.

Challenge 3: Unsatisfiable Puzzles

- **Some user inputs could not result in valid solutions.**

  Solution: Added checks to return a message indicating the puzzle is unsolvable.

Challenge 4: Lack Of Knowledge about Backtracking, Brute Force Algorithm

- **I didn't know much about the algorithm like Brute Force, N-Queen and Backtracking.**
- **Solution: I took help from YouTube, documentation, etc and got more crisper in these algorithms.**

Learning Skills Gained:

1. **This project helped me to learn about new data structure algorithms like Brute Force, Backtracking, etc.**
2. **Learned about Exception Handling**
3. **I learned about the recursive functions.**
4. **I developed skills in writing clean and modular C code**
5. **I understood how to debug complex functions.**

What is the approach followed?

1. **Read the puzzle input with 0s representing empty cells.**
2. **Start from the top-left cell and move row by row.**
3. **If the cell is empty, try inserting numbers 1 through 9.**
4. **For each number, check if it's valid using the valid_ puzzle() function.**
5. **If valid, place the number and proceed recursively.**
6. **If none of the numbers work, reset the cell and backtrack.**
7. **Repeat until the puzzle is either solved or deemed unsolvable.**
8. **If unsolvable then simply print that it is unsolvable**