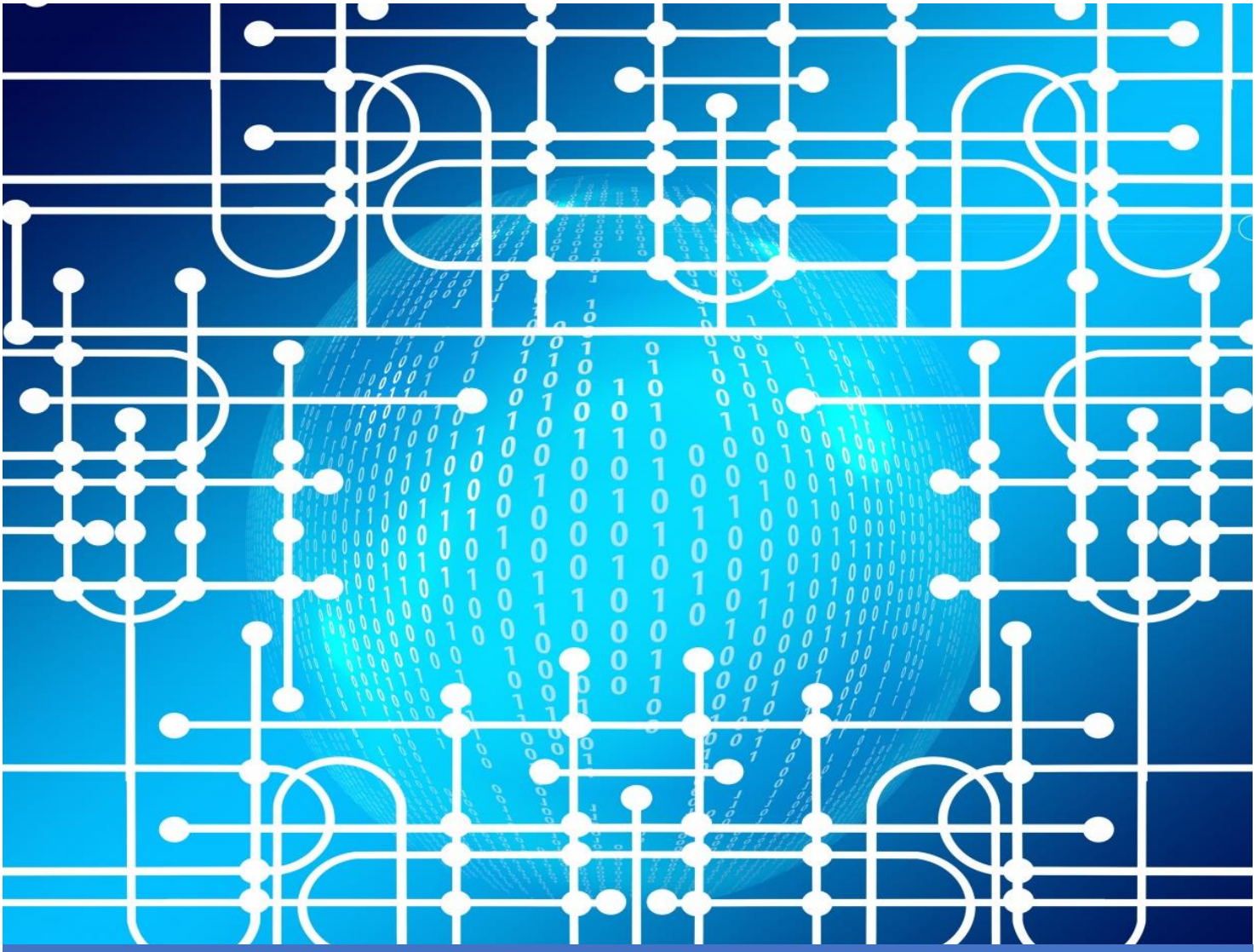

# Sudoku Solver

DSA Project


| | |
|---|---|
| **Muhammad Mazhar Rehan** | **L1F21BSSE001** |
| **Muhammad Uman** | **L1F21BSSE002** |
| **Muhammad Zohaib** | **L1F21BSSE003** |
| **Asad Ali** | **L1F21BSSE004** |

## 1. Introduction:

This code is a C++ implementation of a Sudoku solver using a backtracking algorithm. The program prompts the user to input a 9x9 Sudoku grid, where known values are represented by numbers from 1 to 9, and empty cells are represented by 0. After receiving the input, the program attempts to solve the Sudoku puzzle using a backtracking approach. If a solution is found, it displays the solved Sudoku grid; otherwise, it prints "No solution exists."

## 2. Functions:

### Arrays

The time complexity analysis for each function used in the Sudoku solver program:

**1**. **SudokuGrid()** - Constructor:

- Time Complexity: O(1)
- Explanation: The constructor initializes the Sudoku grid with empty cells. Since the grid size is fixed, the time complexity is constant.

**2. insertValue(int row, int col, int value)**

- Time Complexity: O(1)
- Explanation: The function inserts a value into the specified row and column of the Sudoku grid. Since the grid size is fixed, the time complexity is constant.

**3. getValue(int row, int col)**

- Time Complexity: O(1)
- Explanation: The function retrieves the value at the specified row and column of the Sudoku grid. Since the grid size is fixed, the time complexity is constant.

**4. displayGrid()**

- Time Complexity: O(1)
- Explanation: The function displays the Sudoku grid. Since the grid size is fixed, the time complexity is constant.

5. **isSafe(int row, int col, int value):**

- Time Complexity: O(1)
- Explanation: The function checks if it is safe to place a value in the specified row and column of the Sudoku grid. It checks the current row, current column, and the corresponding 3x3 box. Since the grid size is fixed and the box size is also fixed (3x3), the time complexity is constant.

6. **solveSudoku():**

- Time Complexity: Exponential
- Explanation: The function uses a backtracking algorithm to solve the Sudoku grid. It tries assigning numbers from 1 to 9 to empty cells and recursively solves for the next cell until a solution is found or all possibilities are exhausted. The time complexity of the backtracking algorithm for a typical Sudoku grid is exponential.

7. **findEmptyCell(int& row, int& col):**

- Time Complexity: O(1)
- Explanation: The function finds the next empty cell in the Sudoku grid. It iterates through the grid row by row until an empty cell is found. Since the grid size is fixed, the time complexity is constant.

8. **validateInput(int row, int col, int value):**

- Time Complexity: O(1)
- Explanation: The function validates the user input for row, column, and value. It checks if they are within the valid range. Since the range is fixed, the time complexity is constant.

9. **stringToInt(const string& str):**

- Time Complexity: O(n)
- Explanation: The function converts a string to an integer. It iterates through each character in the string to perform the conversion. The time complexity is proportional to the length of the string.

**10. isStringDigits(const string& str):**

- Time Complexity: O(n)
- Explanation: The function checks if a string contains only digits. It iterates through each character in the string to perform the check. The time complexity is proportional to the length of the string.

**11. inputGrid(SudokuGrid& sudoku):**

- Time Complexity: O(1)
- Explanation: The function takes input for the Sudoku grid from the user. It prompts the user for values for each cell in the grid. Since the grid size is fixed, the time complexity is constant per cell.

## Linked List:

The time complexity of the functions used in Linked List program is as follows:

1. **SudokuGrid::SudokuGrid()**
    - O(1)
    - This constructor initializes the grid with empty cells. Since the size of the grid is fixed, the time complexity is constant.

2. **SudokuGrid::createNode()**
    - O(1)
    - This function creates a new linked list node. Again, the time complexity is constant.

3. **SudokuGrid::insertValue()**
    - O(1)
    - This function inserts a value into the grid at the specified row and column. Since the grid is a 2D array and the access is direct, the time complexity is constant.

4. **SudokuGrid::getValue()**
    - O(1)

- This function retrieves the value at the specified row and column. Similar to `insertValue()`, the time complexity is constant.

5. **SudokuGrid::displayGrid()**
   - O(1)
   - This function displays the Sudoku grid. The number of cells in the grid is fixed, so the time complexity is constant.

6. **SudokuGrid::isSafe()**
   - O(1)
   - This function checks if a value can be placed in a particular cell. It performs constant time operations to check the constraints.

7. **SudokuGrid::solveSudoku()**
   - O(9^m)
   - This function solves the Sudoku grid using backtracking. The time complexity depends on the number of empty cells in the grid, denoted by 'm'. In the worst case, where all cells are empty, the time complexity becomes O(9^m), as there are 9 possible values to try at each empty cell.

8. **SudokuGrid::findEmptyCell()**
   - O(1)
   - This function finds the next empty cell in the Sudoku grid. It iterates over the grid once, so the time complexity is constant.

9. **validateInput()**
   - O(1)
   - This function validates the user input. It performs simple comparisons, so the time complexity is constant.

10. **stringToInt()**
    - O(n)
    - This function converts a string to an integer. The time complexity depends on the length of the string, denoted by 'n'.

**11. isStringDigits()**

- O(n)
- This function checks if a string contains only digits. It iterates over each character of the string, so the time complexity depends on the length of the string, denoted by 'n'.

**12. inputGrid()**

- O(81)
- This function inputs the Sudoku grid from the user. It iterates over each cell in the grid, which has a fixed size of 9x9.

---