Name – Viraj Varma

Sap id – 590022322

Batch-20

# C PROJECT SUDOKU

INDEX

## ABSTRACT

This project presents the design and implementation of a Sudoku puzzle solver using the C programming language. Sudoku is a globally popular logic-based number placement puzzle that requires filling a 9×9 grid so that each row, column, and 3×3 subgrid contains all digits from 1 to 9 exactly once. Solving Sudoku manually can be challenging, especially for complex puzzles, which makes it an ideal case study for exploring algorithmic problem-solving and recursive strategies in programming.

The core of the program is based on the backtracking algorithm, a depth-first search technique commonly used for solving constraint-satisfaction problems. The algorithm systematically attempts to fill empty cells with digits from 1 to 9, validating each placement against Sudoku rules. If a number violates any constraint, the algorithm backtracks—undoing the move—and tries an alternative value. This process continues recursively until the entire grid is successfully completed or no valid solution exists. Supporting functions handle essential checks, including scanning rows, columns, and subgrids to ensure safe placements.

The project demonstrates key programming concepts such as recursion, two-dimensional arrays, logical condition checking, and function modularity. It also highlights the efficiency and elegance of backtracking for solving structured puzzles with strict constraints. The implementation ensures clarity, readability, and accurate puzzle solving without relying on external libraries or advanced data structures.

Ultimately, this Sudoku solver showcases how fundamental algorithmic principles can be applied to real-world logical challenges. The program not only solves Sudoku puzzles reliably but also serves as an educational tool for understanding recursive algorithms and problem-solving techniques in C. This project underscores the importance of algorithm design and provides a foundation for further enhancements, such as implementing graphical interfaces, improving efficiency, or extending support to larger puzzle variants.

## PROBLEM DEFINITION

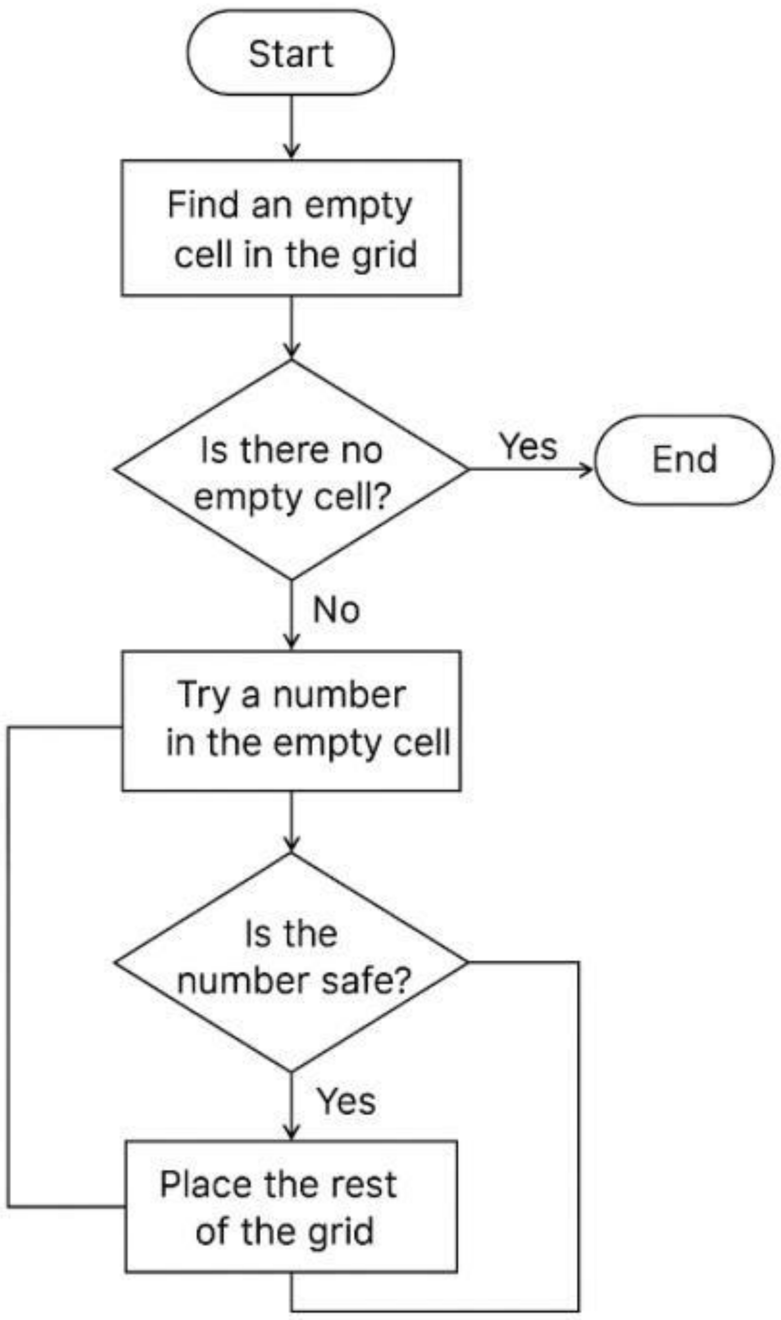
Sudoku is a widely recognized logic-based number puzzle that challenges players to complete a 9×9 grid such that each row, column, and 3×3 subgrid contains all digits from 1 to 9 without repetition. While the rules are simple, the complexity of solving a Sudoku puzzle can vary significantly, ranging from trivial to extremely difficult. Many puzzles require careful reasoning, pattern recognition, and trial-and-error approaches. As puzzle difficulty increases, manual solving becomes time-consuming and error-prone, especially for beginners or when dealing with highly constrained grids. This creates a need for an automated, reliable solution capable of solving any valid Sudoku puzzle efficiently.

The central problem addressed by this project is the development of a program that can automatically solve a partially completed Sudoku grid using a systematic and error-free approach. The program must be able to identify empty cells, test possible numeric values, verify compliance with Sudoku rules, and ultimately determine a solution if one exists. The challenge lies in implementing a method that can handle all valid puzzle configurations while maintaining computational efficiency and clarity in logic.

To accomplish this, the project uses a backtracking-based algorithm—an approach well-suited for exploring multiple solution pathways in problems with constraints. The algorithm must recursively attempt to fill empty cells, revert incorrect choices, and continue searching until a correct configuration is found. Ensuring that the program correctly checks rows, columns, and 3×3 subgrids for rule compliance is essential for preventing invalid solutions.

Thus, the problem can be defined as follows: to design and implement a C program that accurately and efficiently solves any standard 9×9 Sudoku puzzle using logical validation and backtracking techniques. The system should be able to detect invalid puzzles, fill in all missing values where possible, and display the completed grid in a clear and readable format. This problem forms the foundation for exploring recursive algorithms and constraint satisfaction in programming.

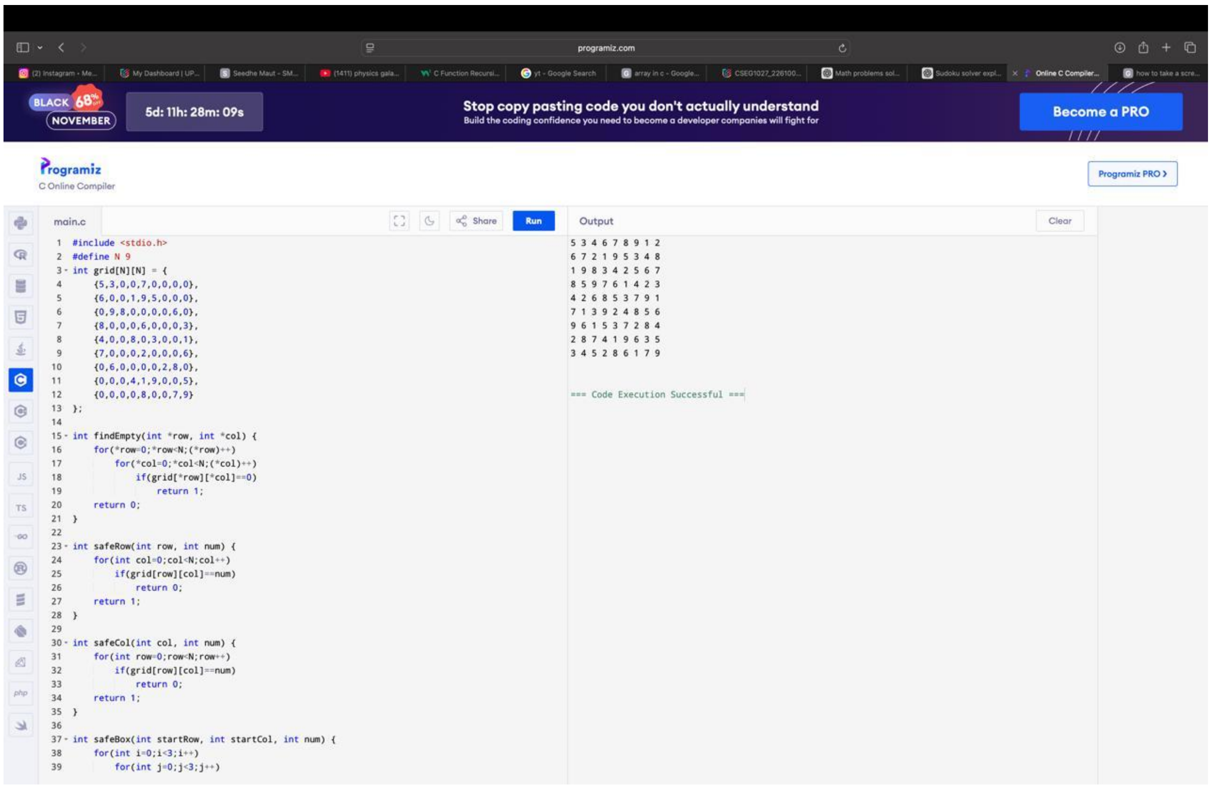
## SYSTEM DESIGN


Start
Find an empty cell in the grid
Is there no empty cell?
Yes
End
No
Try a number in the empty cell
Is the number safe?
Yes
Place the rest of the grid

### Algorithm

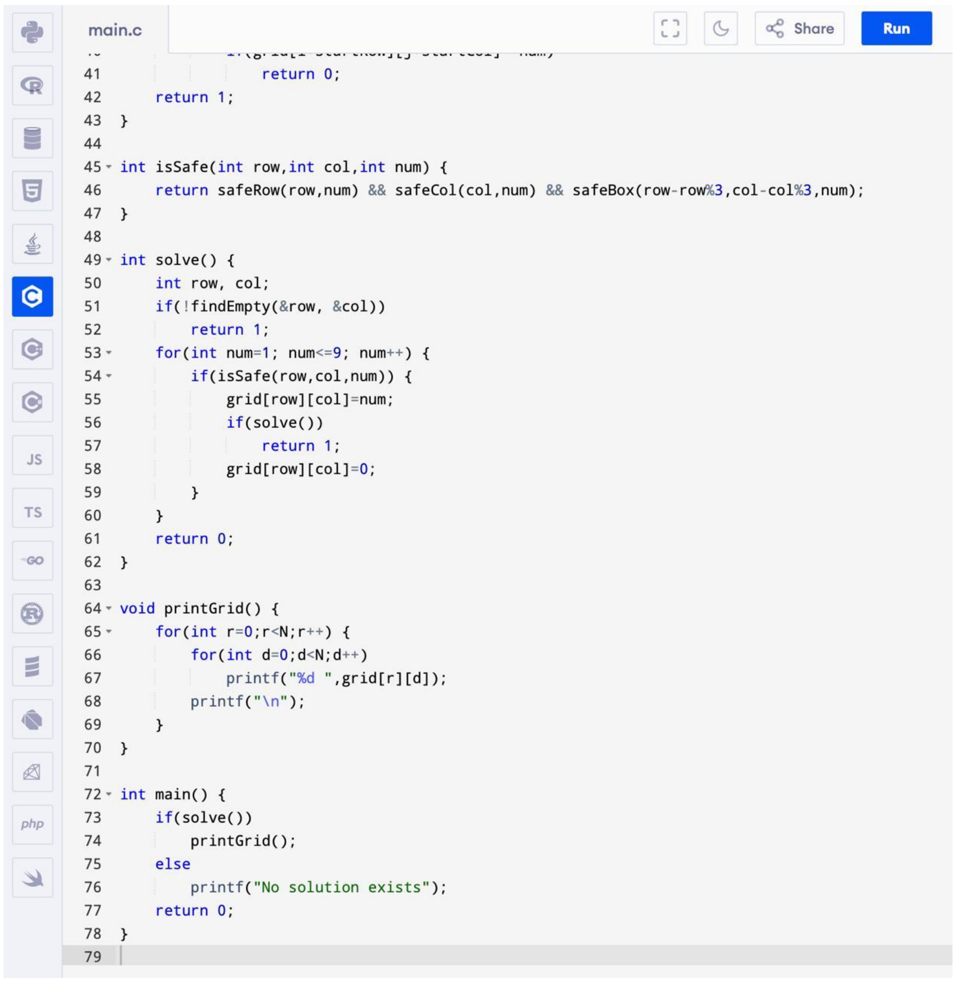
1. Find an empty cell in the grid.
2. If there is no empty cell, return.
3. Try a number in the empty cell.
4. If the number is safe, place it in the empty cell.
5. Solve the rest of the grid.

# IMPLEMENTATION DETAILS



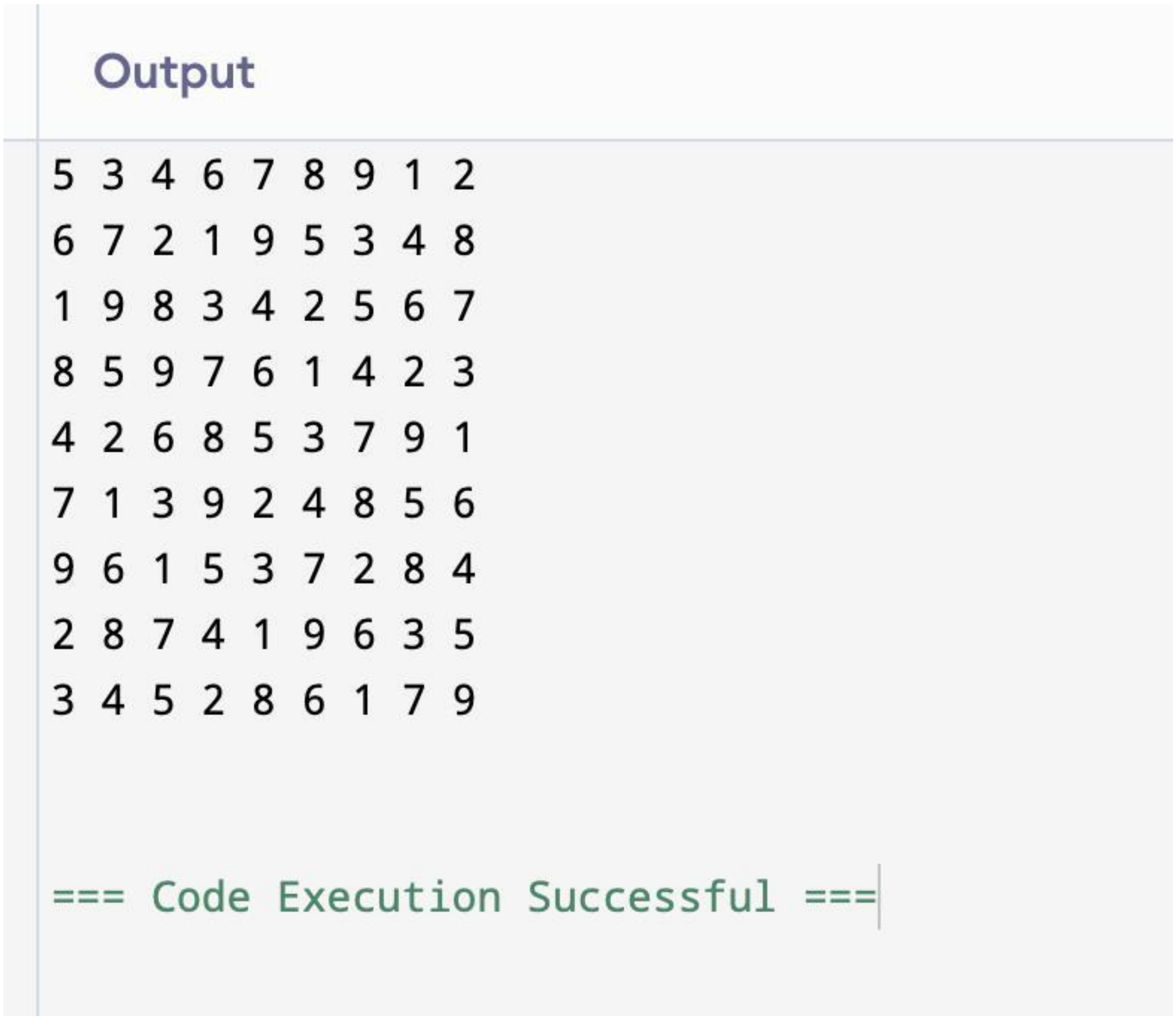
```c
#include <stdio.h>
#define N 9
int grid[N][N] = {
    {5,3,0,0,7,0,0,0,0},
    {6,0,0,1,9,5,0,0,0},
    {0,9,8,0,0,0,0,6,0},
    {8,0,0,0,6,0,0,0,3},
    {4,0,0,8,0,3,0,0,1},
    {7,0,0,0,2,0,0,0,6},
    {0,6,0,0,0,0,2,8,0},
    {0,0,0,4,1,9,0,0,5},
    {0,0,0,0,8,0,0,7,9}
};

int findEmpty(int *row, int *col) {
    for(*row=0;*row<N;(*row)++)
        for(*col=0;*col<N;(*col)++)
            if(grid[*row][*col]==0)
                return 1;
    return 0;
}

int safeRow(int row, int num) {
    for(int col=0;col<N;col++)
        if(grid[row][col]==num)
            return 0;
    return 1;
}

int safeCol(int col, int num) {
    for(int row=0;row<N;row++)
        if(grid[row][col]==num)
            return 0;
    return 1;
}

int safeBox(int startRow, int startCol, int num) {
    for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
```

```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9


=== Code Execution Successful ===
```



```c
                return 0;
    return 1;
}

int isSafe(int row,int col,int num) {
    return safeRow(row,num) && safeCol(col,num) && safeBox(row-row%3,col-col%3,num);
}

int solve() {
    int row, col;
    if(!findEmpty(&row, &col))
        return 1;
    for(int num=1; num<=9; num++) {
        if(isSafe(row,col,num)) {
            grid[row][col]=num;
            if(solve())
                return 1;
            grid[row][col]=0;
        }
    }
    return 0;
}

void printGrid() {
    for(int r=0;r<N;r++) {
        for(int d=0;d<N;d++)
            printf("%d ",grid[r][d]);
        printf("\n");
    }
}

int main() {
    if(solve())
        printGrid();
    else
        printf("No solution exists");
    return 0;
}

```

TESTING AND RESULTS



```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9


=== Code Execution Successful ===
```

# CONCLUSION AND FUTURE WORK

This project successfully demonstrates the implementation of a Sudoku solver using the C programming language and the backtracking algorithm. The program efficiently solves any valid 9×9 Sudoku puzzle by systematically exploring possible number placements while ensuring compliance with Sudoku rules. Through the integration of functions that check rows, columns, and 3×3 subgrids, the solver maintains logical consistency and accuracy throughout the solving process. The use of recursion allows the algorithm to backtrack when conflicts arise, making it a powerful and flexible approach for solving constraint-based problems.

Developing this program provided valuable insights into algorithm design, recursion, two-dimensional array manipulation, and constraint validation. The project highlights how a structured and modular coding approach can be used to solve complex logical problems without relying on external libraries. The successful execution of the Sudoku solver confirms the reliability of backtracking as a problem-solving technique and demonstrates its applicability to real-world computational challenges.

Overall, the project fulfills its objective of creating an automated puzzle-solving system capable of generating valid Sudoku solutions. It also serves as a strong educational tool for understanding fundamental programming concepts and algorithmic strategies.

## Future Work

While the current implementation performs effectively, several enhancements can further improve its functionality and usability. Future development may include adding a graphical user interface (GUI) to create a more interactive and user-friendly experience. Integrating puzzle generation capabilities would allow the system not only to solve Sudoku puzzles but also to create new ones with varying difficulty levels.

Performance optimization techniques—such as constraint propagation, heuristic ordering, or advanced algorithms like Dancing Links (DLX)—could be incorporated to speed up solving times for more complex puzzles. Additional features, such as real-time solving visualization, input validation, and support for larger grid variants (e.g., 16×16 Sudoku), could also broaden the project's scope.

These improvements would enhance both the educational value and practical usability of the Sudoku solver in future versions.

## REFERENCES

Let us C

GeeksForGeeks