Solving Sudoku with Recursion

Sudoku: The Classic Puzzle

What is Sudoku?
Sudoku is a logic-based number placement puzzle where
you fill a 9x9 grid with numbers from 1 to 9.

The Rules
The goal is to fill the grid so that each row, column, and
3x3 subgrid contains all the numbers from 1 to 9 without repetition.

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

How Sudoku is Played

1

Start with a Partial Grid

Sudoku puzzles begin with a partially filled grid, leaving empty cells to be completed.

2

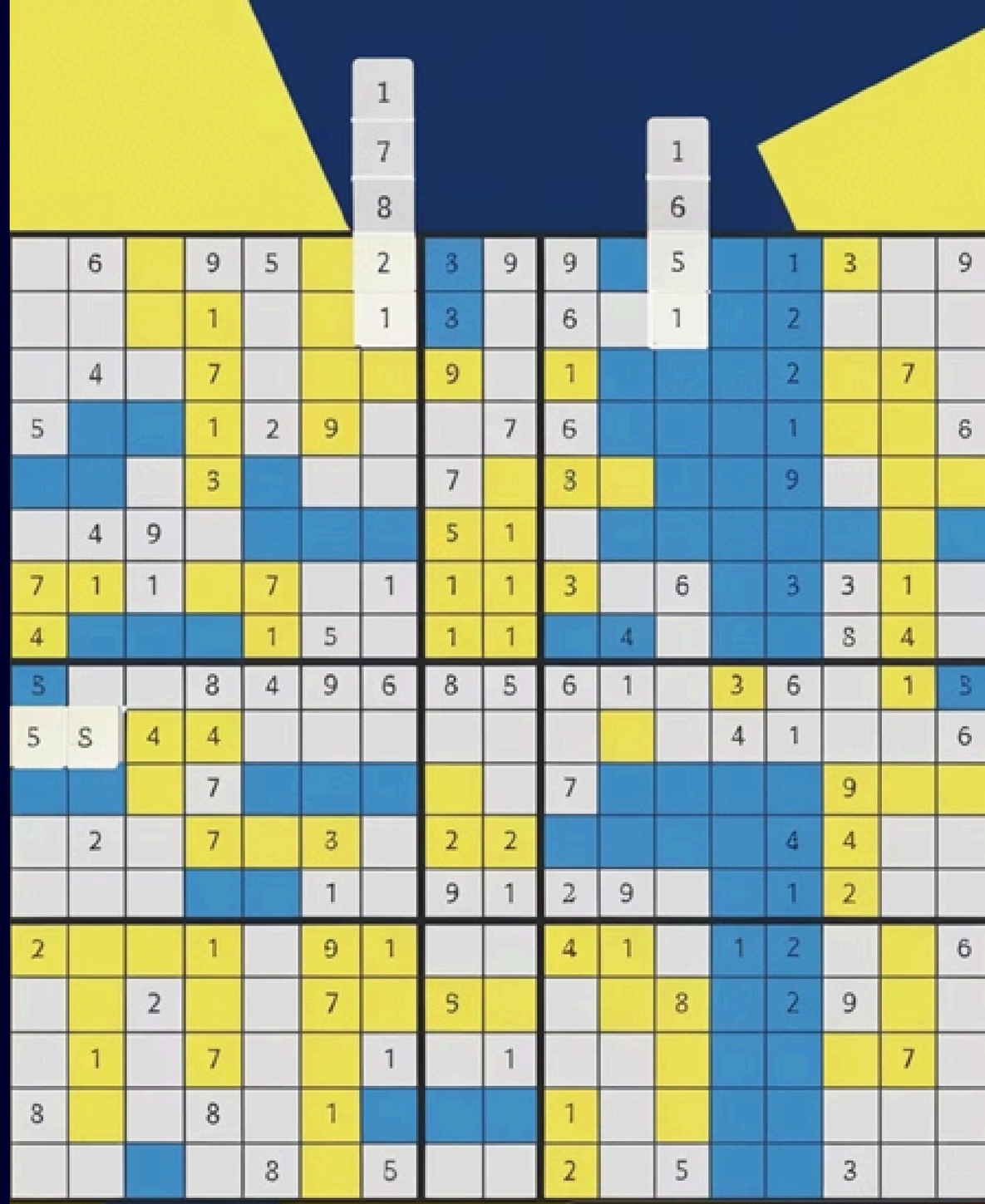
The Challenge

The challenge is to fill in the empty cells while adhering to the rules, using logic and deduction.

3

Solve and Complete

The goal is to complete the grid by placing numbers strategically to satisfy all the rules.



Introducing Recursion

- A Function Calling Itself

Recursion is a programming technique where a function calls itself

- Breaking Down Complexity

Recursion effectively breaks down complex problems into manageable pieces that can be solved iteratively.

- Repetitive Patterns

It's ideal for problems with repetitive patterns, like Sudoku, where the rules are consistent throughout the grid

Backtracking Algorithm

The key for using recursion for solving sudoku is the backtracking algorithm

Place a number

We Place a number in an empty cell and recursively check if its valid

Move To The Next Cell

If Valid, we move to the next cell and repeat the process

Backtrack if conflict

If a conflict arise, we backtrack to the previous cell and try a different number

VALID FUNCTION

Check if the number already exists in the current 3x3 grid

```
// Determine the starting indices of the 3x3 s
start_row = (row / 3) * 3;
end_row = start_row + 3;

start_col = (col / 3) * 3;
end_col = start_col + 3;

for (int i = start_row; i < end_row; i++)  //
{
    for (int j = start_col; j < end_col; j++)
    {
        if (puzzle[i][j] == val)
        {
            return false;
        }
    }
}
```

VALID FUNCTION

Check if the number already exists in the current row or column

```
int start_row, end_row, start_col, end_col;

for (int i = 0; i < 9; i++)
{
    if (puzzle[row][i] == val)
    {
        return false;
    }
}

for (int i = 0; i < 9; i++)
{
    if (puzzle[i][col] == val)
    {
        return false;
    }
```

ALGORITHM EXPLANATION

If all rows are processed, the puzzle is solved

```
bool solver(int puzzle[9][9], int row, int col)

    if (row == 9)
    {
        return true;
    }
    else if (col == 9)
    {
        return solver(puzzle, row + 1, 0);
    }

    else if (puzzle[row][col] != 0)
    {
        return solver(puzzle, row, col + 1);
    }
    for (int i = 1; i <= 9; i++)
    {
        if (is_valid(puzzle, i, row, col))
        {
            puzzle[row][col] = i;
```

ALGORITHM EXPLANATION

If the current column exceeds 8, move to the next row

```
bool solver(int puzzle[9][9], int row, int col)

    if (row == 9)
    {
        return true;
    }
    else if (col == 9)
    {
        return solver(puzzle, row + 1, 0);
    }

    else if (puzzle[row][col] != 0)
    {
        return solver(puzzle, row, col + 1);
    }
    for (int i = 1; i <= 9; i++)
    {
        if (is_valid(puzzle, i, row, col))
        {
            puzzle[row][col] = i;
```

ALGORITHM EXPLANATION

If the current cell is pre-filled, move to the next cell in the row

```
bool solver(int puzzle[9][9], int row, int col)

    if (row == 9)
    {
        return true;
    }
    else if (col == 9)
    {
        return solver(puzzle, row + 1, 0);
    }

    else if (puzzle[row][col] != 0)
    {
        return solver(puzzle, row, col + 1);
    }
    for (int i = 1; i <= 9; i++)
    {
        if (is_valid(puzzle, i, row, col))
        {
            puzzle[row][col] = i;
```

ALGORITHM EXPLANATION

Hit and trial by placing numbers 1 to 9 in the current empty cell

```
        return solver(puzzle, row + 1, 0);
    }

    else if (puzzle[row][col] != 0)
    {
        return solver(puzzle, row, col + 1);
    }
    for (int i = 1; i <= 9; i++)
    {
        if (is_valid(puzzle, i, row, col))
        {
            puzzle[row][col] = i;

            if (solver(puzzle, row, col + 1))
            {
                return true;
            }
            puzzle[row][col] = 0;
        }
    }
    return false;
```

ALGORITHM EXPLANATION

Recursively solve for the next cells

```
        return solver(puzzle, row + 1, 0);
    }

    else if (puzzle[row][col] != 0)
    {
        return solver(puzzle, row, col + 1);
    }
    for (int i = 1; i <= 9; i++)
    {
        if (is_valid(puzzle, i, row, col))
        {
            puzzle[row][col] = i;

            if (solver(puzzle, row, col + 1))
            {
                return true;
            }
            puzzle[row][col] = 0;
        }
    }
    return false;
```

ALGORITHM EXPLANATION

If placing this number doesn't lead to a solution, backtrack

```
        return solver(puzzle, row + 1, 0);
    }

    else if (puzzle[row][col] != 0)
    {
        return solver(puzzle, row, col + 1);
    }
    for (int i = 1; i <= 9; i++)
    {
        if (is_valid(puzzle, i, row, col))
        {
            puzzle[row][col] = i;

            if (solver(puzzle, row, col + 1))
            {
                return true;
            }
            puzzle[row][col] = 0;
        }
    }
    return false;
```

ALGORITHM EXPLANATION

Return false if no solution to the puzzle exists

```
        return solver(puzzle, row + 1, 0);
    }

    else if (puzzle[row][col] != 0)
    {
        return solver(puzzle, row, col + 1);
    }
    for (int i = 1; i <= 9; i++)
    {
        if (is_valid(puzzle, i, row, col))
        {
            puzzle[row][col] = i;

            if (solver(puzzle, row, col + 1))
            {
                return true;
            }
            puzzle[row][col] = 0;
        }
    }
    return false;
```

ANY QUESTIONS?