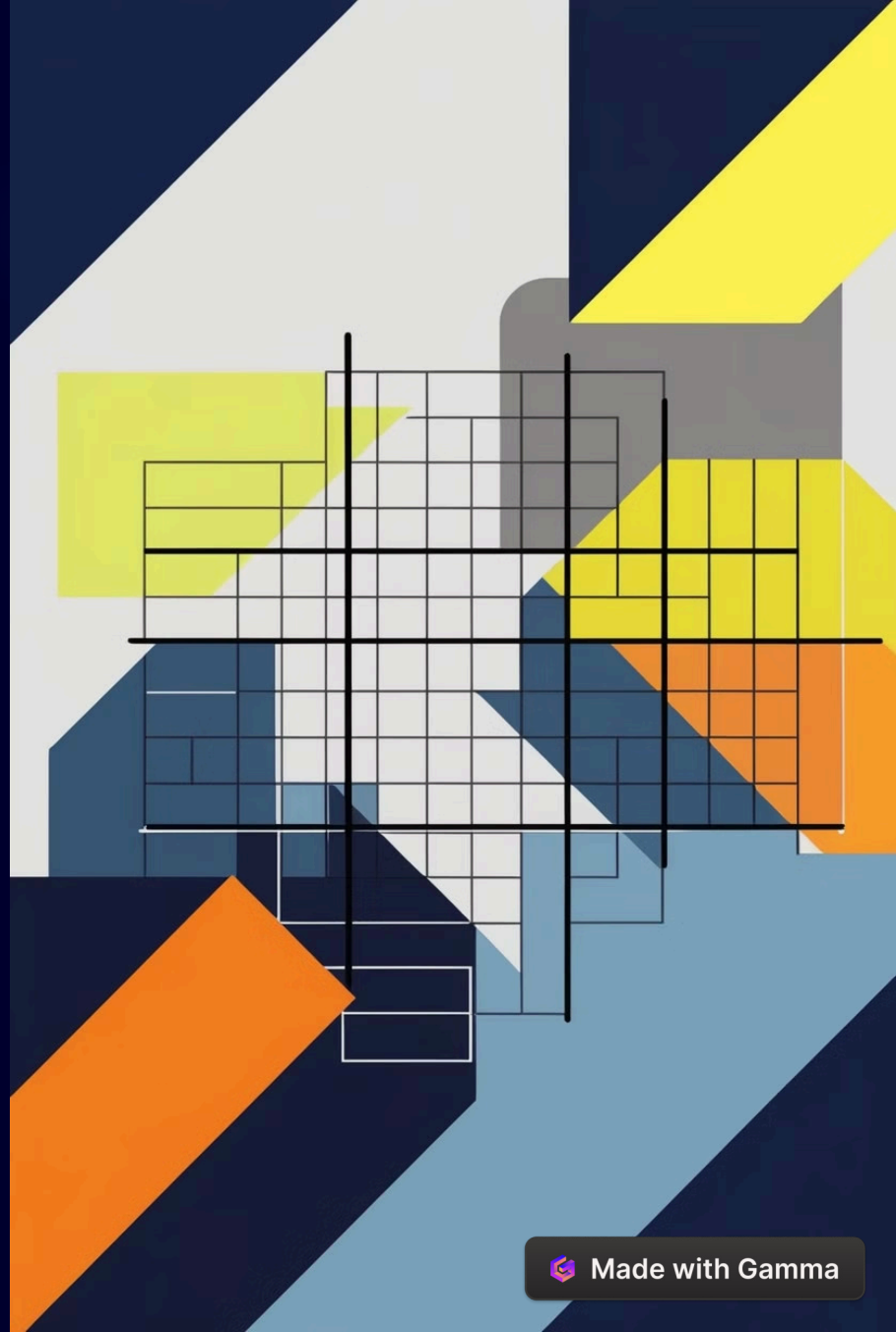
Solving Sudoku with Recursion

Welcome! Today, we'll delve into the fascinating world of Sudoku and discover how recursion provides a powerful and elegant solution to this popular logic puzzle.

T **by Talha Ismail**



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.

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

1 A Function Calling Itself

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.

2 Breaking Down Complexity

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

3 Repetitive Patterns

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

Recursion Solves Sudoku

Backtracking Algorithm

The key to using recursion for Sudoku is the backtracking algorithm.

Place a Number

We place a number in an empty cell and recursively check if it's valid.

Move to the Next Cell

If valid, we move to the next empty cell and repeat the process.

Backtrack if a Conflict

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

Code Explanation

High-Level Workflow

- Start from the first empty cell
- Try all valid numbers
- Recursively solve the rest of the grid
- If no number works, backtrack

Code Snippets

```
def is_valid(grid, row, col, num):
  # Check row, col, and 3x3 subgrid
  # ...

def solver(grid):
  for row in range(9):
    for col in range(9):
      if grid[row][col] == 0:
        for num in range(1, 10):
          if is_valid(grid, row, col, num):
            grid[row][col] = num
            if solver(grid):
              return True
            grid[row][col] = 0
        return False
  return True
```

Sudoku Solution in Action

1

Initial State

Start with a partially filled Sudoku grid.

2

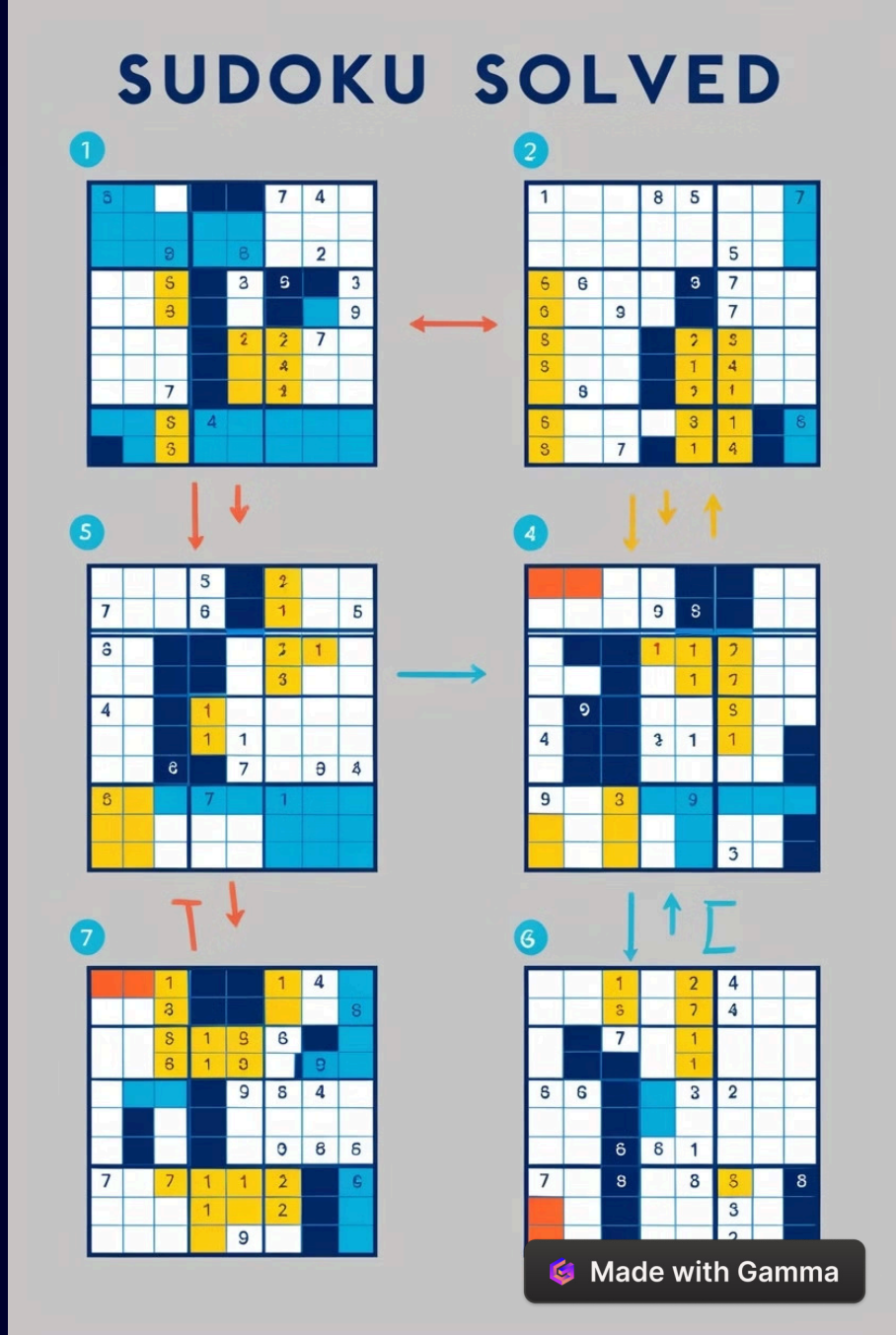
Recursive Steps

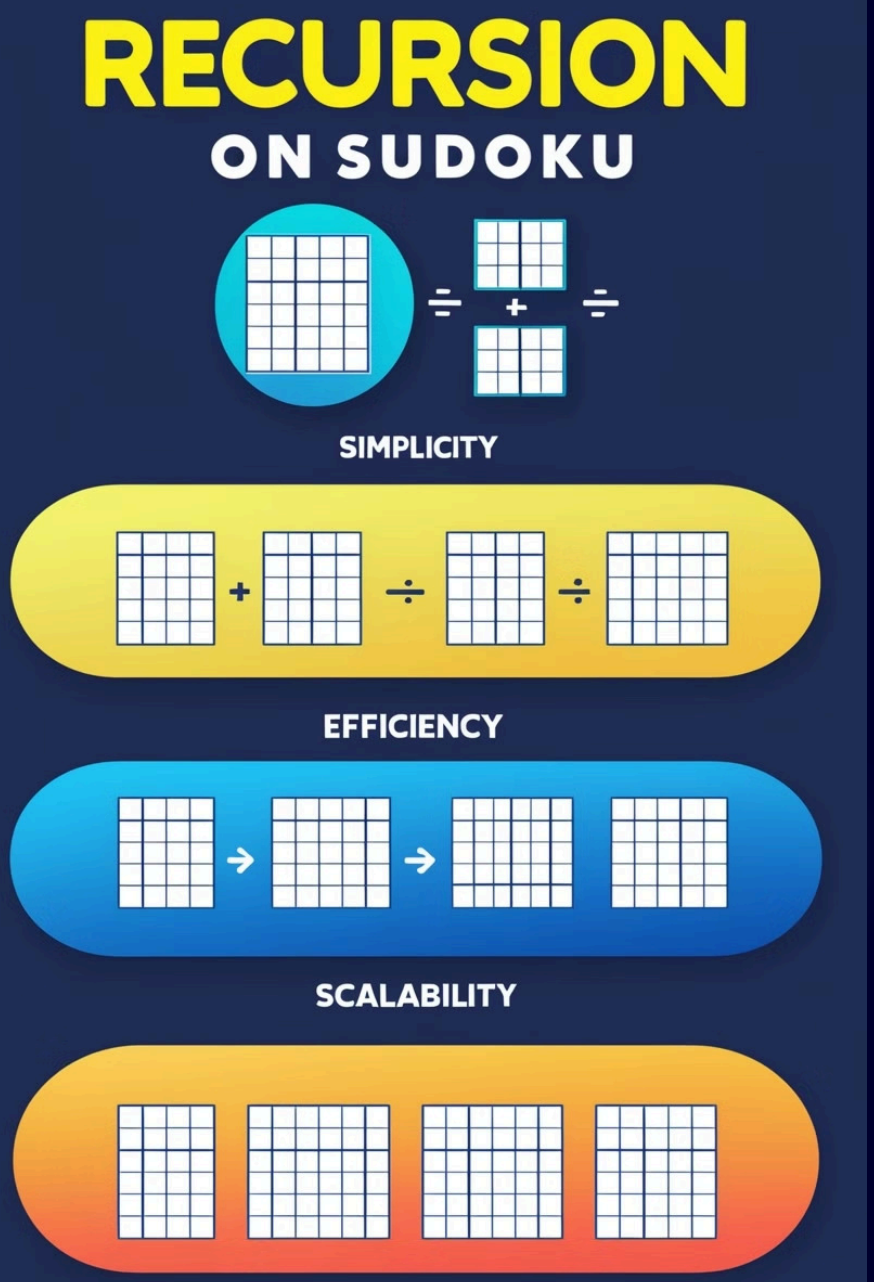
The algorithm iteratively tries numbers, backtracks when necessary.

3

Solved Puzzle

The final result is a completed Sudoku grid, satisfying all rules.





Benefits of Recursion

Simplicity

The recursive code structure naturally mirrors the problem-solving approach, making it easier to understand.

Efficiency

Recursion systematically explores all possible solutions, ensuring a valid solution is found.

Scalability

The recursive approach can be adapted to handle Sudoku grids of varying sizes and complexities.

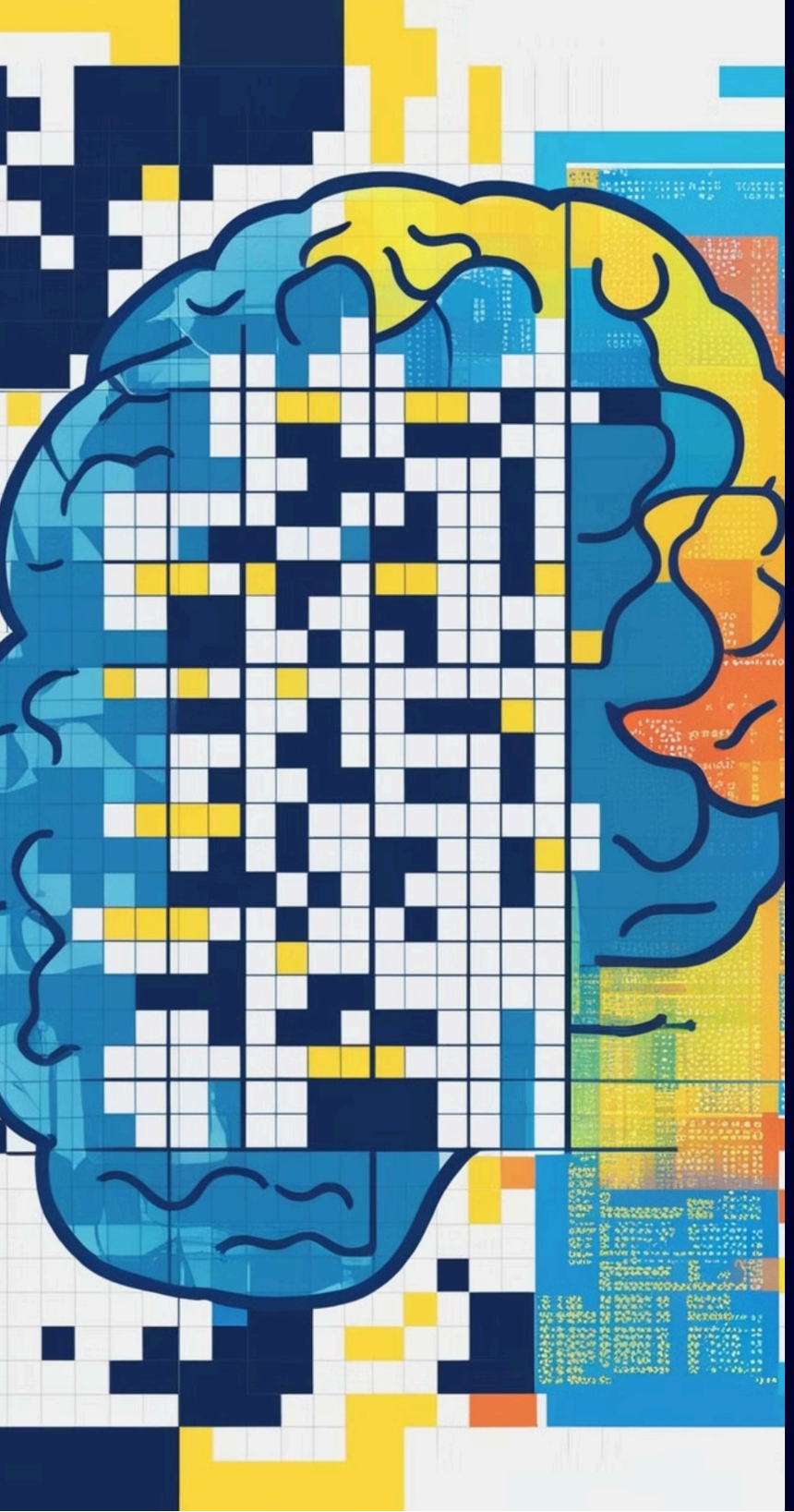
Challenges and Limitations

Stack Overflow

Deep recursion may exhaust system memory, leading to stack overflow.

Computational Intensity

Recursion can be computationally intensive for extremely complex puzzles.



Conclusion

1

Elegant Solution

Recursion offers a powerful and elegant way to solve Sudoku puzzles.

2

Algorithm Power

It demonstrates the importance and versatility of algorithms in problem-solving.

3

Programming Concepts

It enhances understanding of key programming concepts like recursion and backtracking.