AI MSE REPORT

Problem statement :- Sudoku Solver

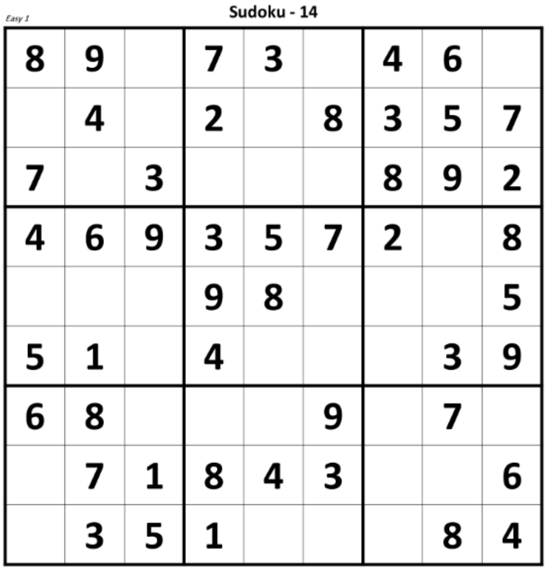
Name : RADHEY PAL
Roll No : 202401100400149
Branch : CSEAIML
Sec : C

INTRODUCTION PAGE

- This report explains the functionality of the Sudoku solver program, which uses a backtracking algorithm to solve Sudoku puzzles. The program consists of multiple functions, each playing a key role in solving and displaying the puzzle.

- Sudoku is a popular number puzzle game played on a 9×9 grid, where the objective is to fill the grid so that each row, column, and 3×3 sub-grid contains the numbers 1 to 9 without repetition.

- This project implements a Sudoku solver using a **backtracking algorithm**, which systematically explores possible solutions by filling empty cells while ensuring valid placements.



Easy 1

Sudoku - 14

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

METHODOLOGY

How It's Work:

- The function solve_sudoku scans the board for an empty cell (represented by 0).
- It iterates through numbers 1 to 9 and checks if placing the number in the empty cell is valid using is_valid.
- If the number is valid, it places the number and calls solve_sudoku recursively to solve the rest of the board.

Why this method is use:

- Backtracking explores all possible number placements in a structured manner, ensuring that no potential solution is overlooked.

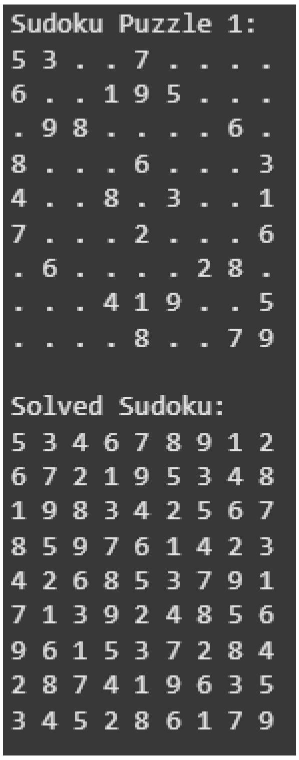
- Sudoku has strict placement rules, and backtracking quickly finds a valid solution by eliminating invalid choices early, reducing unnecessary computations.
- If an invalid number is placed, the algorithm **undoes (backtracks)** the last move and tries the next valid option, ensuring the solution remains correct.

CODE

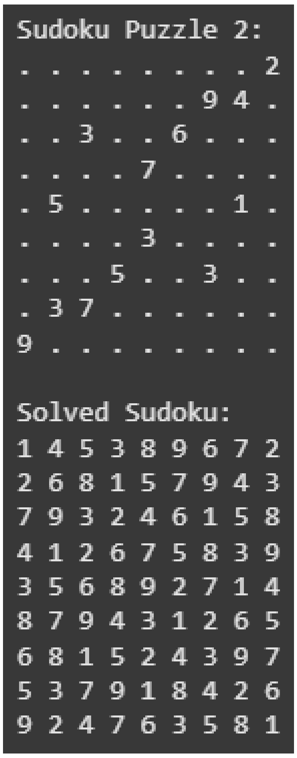
```
# Function to check if a number can be placed in a given cell
def is_valid(board, row, col, num):
    """
    This function checks whether 'num' can be placed at board[row][col].
    It ensures that 'num' is not already present in the same row, column, or 3x3 sub-grid.
    """
    # Check if 'num' exists in the current row or column
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False

    # Check if 'num' exists in the 3x3 sub-grid
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False

    return True

# Function to solve the Sudoku puzzle using backtracking
def solve_sudoku(board):
    """
    This function solves the Sudoku puzzle using a backtracking approach.
    It tries placing numbers 1 to 9 in empty cells and backtracks if no valid option is found.
    """
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:  # Find an empty cell
                for num in range(1, 10):  # Try numbers 1 to 9
                    if is_valid(board, row, col, num):
                        board[row][col] = num  # Place the number
                        if solve_sudoku(board):
                            return True  # If the board is solved, return True
                        board[row][col] = 0  # Reset the cell and backtrack
                return False  # No valid number found, backtrack
    return True  # Puzzle is solved

# Function to print the Sudoku board in a readable format
def print_board(board):
    """
    This function prints the Sudoku board with '.' representing empty cells.
    """
    for row in board:
        print(" ".join(str(num) if num != 0 else '.' for num in row))

# Sample Sudoku puzzles (0 represents empty cells)
puzzles = [
    [
        [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]
    ],
    [
        [0, 0, 0, 0, 0, 0, 0, 0, 2],
        [0, 0, 0, 0, 0, 0, 9, 4, 0],
        [0, 0, 3, 0, 0, 6, 0, 0, 0],
        [0, 0, 0, 0, 7, 0, 0, 0, 0],
        [0, 5, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 3, 0, 0, 0, 0],
        [0, 0, 0, 5, 0, 0, 3, 0, 0],
        [0, 3, 7, 0, 0, 0, 0, 0, 0],
        [9, 0, 0, 0, 0, 0, 0, 0, 0]
    ],
    [
        [8, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 3, 6, 0, 0, 0, 0, 0],
        [0, 7, 0, 0, 9, 0, 2, 0, 0],
        [0, 5, 0, 0, 0, 7, 0, 0, 0],
        [0, 0, 0, 0, 4, 5, 7, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 3, 0],
        [0, 0, 1, 0, 0, 0, 0, 6, 8],
        [0, 0, 8, 5, 0, 0, 0, 1, 0],
        [0, 9, 0, 0, 0, 0, 4, 0, 0]
    ]
]

# Solve and display each puzzle
for index, puzzle in enumerate(puzzles):
```

```
print(f"\nSudoku Puzzle {index + 1}:")
print_board(puzzle)
if solve_sudoku(puzzle):
    print("\nSolved Sudoku:")
    print_board(puzzle)
else:
    print("\nNo solution exists.")
```

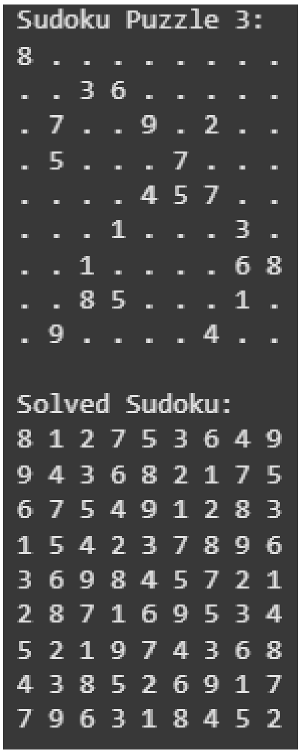
OUTPUTS



```
Sudoku Puzzle 1:
5 3 . . 7 . . . .
6 . . 1 9 5 . . .
. 9 8 . . . . 6 .
8 . . . 6 . . . 3
4 . . 8 . 3 . . 1
7 . . . 2 . . . 6
. 6 . . . . 2 8 .
. . . 4 1 9 . . 5
. . . . 8 . . 7 9

Solved Sudoku:
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
```



```
Sudoku Puzzle 2:
. . . . . . . . 2
. . . . . . 9 4 .
. . 3 . . 6 . . .
. . . . 7 . . . .
. 5 . . . . . 1 .
. . . . 3 . . . .
. . . 5 . . 3 . .
. 3 7 . . . . . .
9 . . . . . . . .

Solved Sudoku:
1 4 5 3 8 9 6 7 2
2 6 8 1 5 7 9 4 3
7 9 3 2 4 6 1 5 8
4 1 2 6 7 5 8 3 9
3 5 6 8 9 2 7 1 4
8 7 9 4 3 1 2 6 5
6 8 1 5 2 4 3 9 7
5 3 7 9 1 8 4 2 6
9 2 4 7 6 3 5 8 1
```



```
Sudoku Puzzle 3:
8 . . . . . . . .
. . 3 6 . . . . .
. 7 . . 9 . 2 . .
. 5 . . . 7 . . .
. . . . 4 5 7 . .
. . . 1 . . . 3 .
. . 1 . . . . 6 8
. . 8 5 . . . 1 .
. 9 . . . . 4 . .

Solved Sudoku:
8 1 2 7 5 3 6 4 9
9 4 3 6 8 2 1 7 5
6 7 5 4 9 1 2 8 3
1 5 4 2 3 7 8 9 6
3 6 9 8 4 5 7 2 1
2 8 7 1 6 9 5 3 4
5 2 1 9 7 4 3 6 8
4 3 8 5 2 6 9 1 7
7 9 6 3 1 8 4 5 2
```

REFRENCE

- **Sudoku Puzzle Rules** – International Sudoku Rules, Retrieved from www.sudoku.com
- **Backtracking Algorithm** – Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- **Sudoku Solving Techniques** – Norvig, P. (2012). *Solving Every Sudoku Puzzle*, Retrieved from norvig.com