Project Title: Sudoku game
Student Name: Pushpendar Kanaram Choudhary
Class: D6AD-B
Roll No: 12

Description of the Game:

The game initializes a 9x9 Sudoku board with some pre-filled numbers and empty spaces (represented by 0). Players can make moves by entering the row, column, and the number they want to place on the board.The game checks if the move is valid by verifying whether the number can be placed in the given row, column, and 3x3 subgrid. The game includes an **undo** feature, which allows players to revert their last move.There is also a **solve** feature, which uses a recursive backtracking algorithm to attempt solving the puzzle. The game ends when the puzzle is solved (i.e., all cells are filled correctly).

Data Structures Used:

1. 2D Array (board[][]):

- This 9x9 array represents the Sudoku board, with numbers 1-9 filling cells and 0 indicating an empty space.
- It stores the initial puzzle setup and updates as players make moves or when the solver fills the cells.
- The array helps check if a number can be placed at a specific position, ensuring no duplicate numbers in rows, columns, or 3x3 subgrids.
- Functions like DisplayBoard() read this array to visually output the current state of the board.
- The solve() function modifies this array during the backtracking process to fill the puzzle.

2. Stack (moveHistory):

- The stack stores the history of player moves as an array [row, column, previous value], with each push representing a new move.
- This structure follows the Last In, First Out (LIFO) principle, making it ideal for the undo function.
- When a player makes a move, the previous value in the cell is stored in the stack, and the new move is made on the board.
- When undoMove() is called, the stack pops the latest move, restoring the board's previous state.

- It ensures that players can correct their moves without altering earlier steps, providing flexibility in gameplay.

3. Recursion (solve()):

- The solve() method uses recursive backtracking to solve the puzzle by filling empty cells.
- It scans the board for the first empty cell (0) and tries placing numbers 1 through 9, checking if they are valid.
- If a valid number is found, it places the number and recursively calls itself to fill the next cell.
- If a dead end is reached (i.e., no valid number works), the function backtracks by resetting the cell to 0 and trying the next possible number.
- This approach systematically explores all possible solutions, making recursion an efficient method for solving Sudoku.

Logic of the Game:

Board Initialization: A 9x9 grid is initialized with some pre-filled numbers and empty cells (represented by 0).

Move Input and Validation: The player inputs the row, column, and number they want to place. The game checks if the number can be placed by ensuring:

The number doesn't already exist in the same row, column, or 3x3 subgrid (checked by the isValid() function).

Stack for Undo: Each move is recorded in a stack, allowing players to undo their last move by popping from the stack and restoring the previous state of the cell.

Backtracking Solver: If the player types "solve," the game uses recursive backtracking (solve() function) to fill the empty cells by trying numbers 1-9. If a number works, it moves to the next empty cell. If no number works, it backtracks and resets the previous cell, trying a different number.

Completion Check: The game continuously checks if all cells are filled to declare the puzzle solved.

Program:

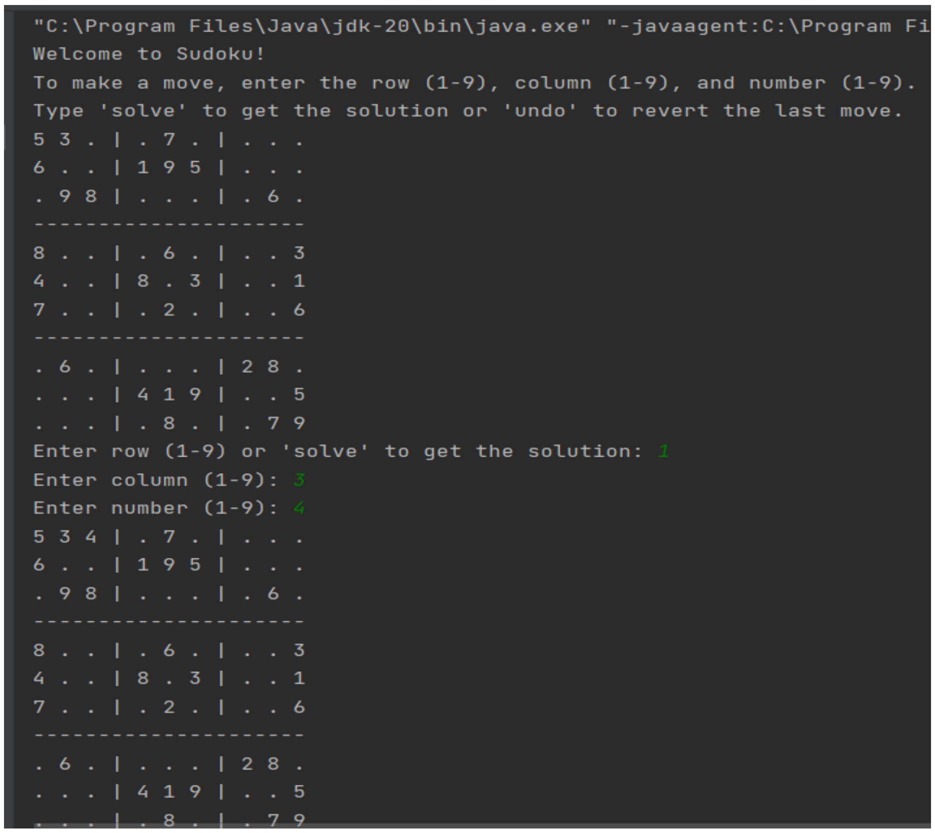
```
import java.util.Scanner;
import java.util.Stack;

public class CustomSudoku {
  static int [][] board;
```

```
 static Stack<int[]> moveHistory = new Stack<>();
public CustomSudoku() {
  board = new int[][]{
          {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}
   };

 }
 public Boolean isValid(int row,int col,int num){
    for(int i=0;i<9;i++){
       if(board[row][i]==num){
          return false;
       }
    }
    for(int i=0;i<9;i++){
       if(board[i][col]==num){
          return false;
       }
    }
    int startrow=row/3*3;
    int startcol=col/3*3;
    for(int i=startrow;i<startrow+3;i++){
       for(int j=startcol;j<startcol+3;j++){
          if(board[i][j]==num){
             return false;
          }
       }
    }
    return true;
 }
 public boolean isComplete(){
    for(int i=0;i<9;i++){
       for(int j=0;j<9;j++){
          if(board[i][j]==0){
             return false;
          }
       }
    }
```

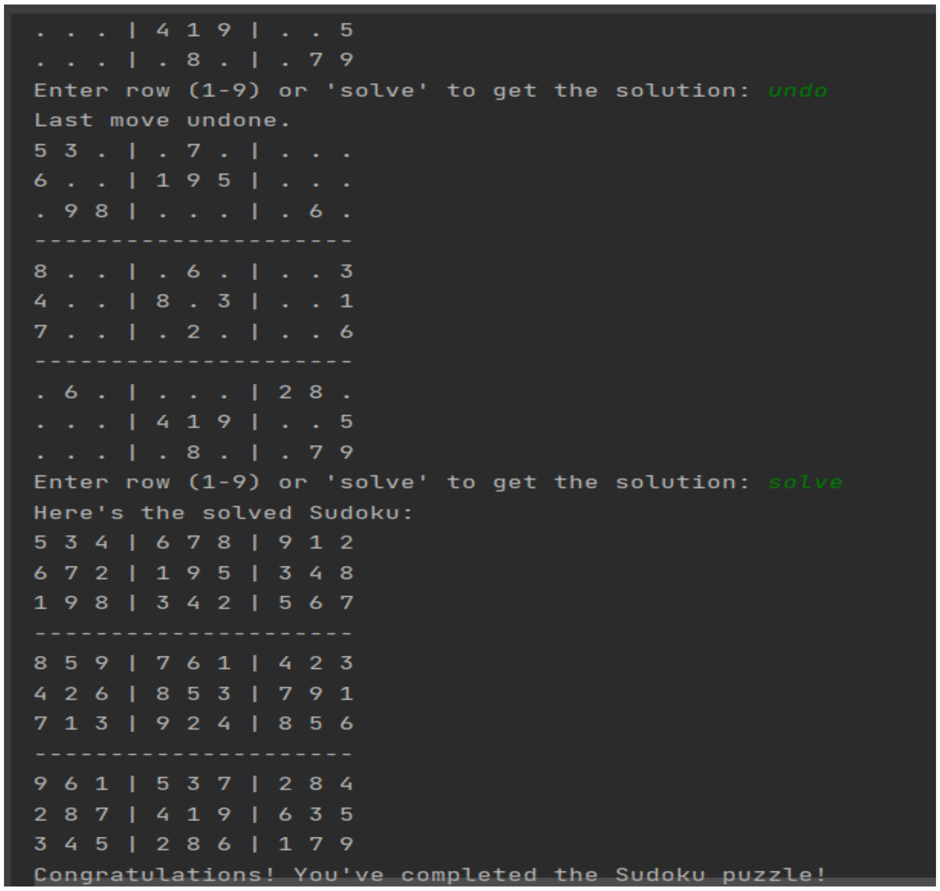
```
return true;
    }
    public void DisplayBoard(){
        for(int i=0;i<9;i++){
            if(i%3==0 && i!=0){
                System.out.println("---------------------");
            }

            for(int j=0;j<9;j++){
                if(j%3==0 && j!=0){
                    System.out.print("| ");
                }
                if(board[i][j]==0){
                    System.out.print(". ");
                }else{
                    System.out.print(board[i][j] + " ");
                }
            }
            System.out.println();
        }
    }
   static public  void undoMove() {
        if (!moveHistory.isEmpty()) {
            int[] lastMove = moveHistory.pop();
            int row = lastMove[0];
            int col = lastMove[1];
            int previousValue = lastMove[2];
            board[row][col] = previousValue; // Restore the previous value
            System.out.println("Last move undone.");
        } else {
            System.out.println("No moves to undo.");
        }
    }
    private boolean solve() {
        int n= board.length;
        int r=-1;
        int c=-1;
        boolean empty=true;
        for(int i=0;i<9;i++){
            for(int j=0;j<9;j++){
                if(board[i][j]==0){
                    r=i;
                    c=j;
                    empty=false;
                    break;
                }
            }
            if(!empty){
                break;
            }
        }
        if(empty){
            return true;
        }
        for(int i=0;i<=9;i++){
            if(isValid(r,c,i)){
                board[r][c]=i;
                if(solve()){
                    return true;
                }else{
                    board[r][c]=0;
                }
            }
        }
        return false;
    }

    private void setValue(int row, int col, int num) {
        if(isValid(row,col,num)){
            board[row][col]=num;
        }else{
            System.out.println("Invalid move! try Again.");
        }
    }




    public static void main(String[] args) {
    CustomSudoku game=new CustomSudoku();
        Scanner sc=new Scanner(System.in);
        System.out.println("Welcome to Sudoku!");
        System.out.println("To make a move, enter the row (1-9), column (1-9), and number
    (1-9).");
        System.out.println("Type 'solve' to get the solution or 'undo' to revert the last move.");
        while(!game.isComplete()){
            game.DisplayBoard();
            System.out.print("Enter row (1-9) or 'solve' to get the solution: ");
            String input = sc.next();
            if (input.equalsIgnoreCase("solve")){
            if(game.solve()){
                System.out.println("Here's the solved Sudoku:");
                game.DisplayBoard();
            } else {
                System.out.println("Unable to solve the Sudoku.");
                System.out.println(" OR please Undo your wrong Steps to Get sudoku
Solved");
            }
            break;
        }
        if (input.equalsIgnoreCase("undo")) {
            undoMove();
            continue;
        }
        int row=Integer.parseInt(input)-1;
        System.out.print("Enter column (1-9): ");
        int col = sc.nextInt() - 1;  // Convert to 0-based index
        System.out.print("Enter number (1-9): ");
        int num = sc.nextInt();
        moveHistory.push(new int[]{row, col, board[row][col]});
        game.setValue(row, col, num);
        }
    if (game.isComplete()) {
        System.out.println("Congratulations! You've completed the Sudoku puzzle!");
        game.DisplayBoard();
    }


    }


}
```

Output:

Wining screenshot:

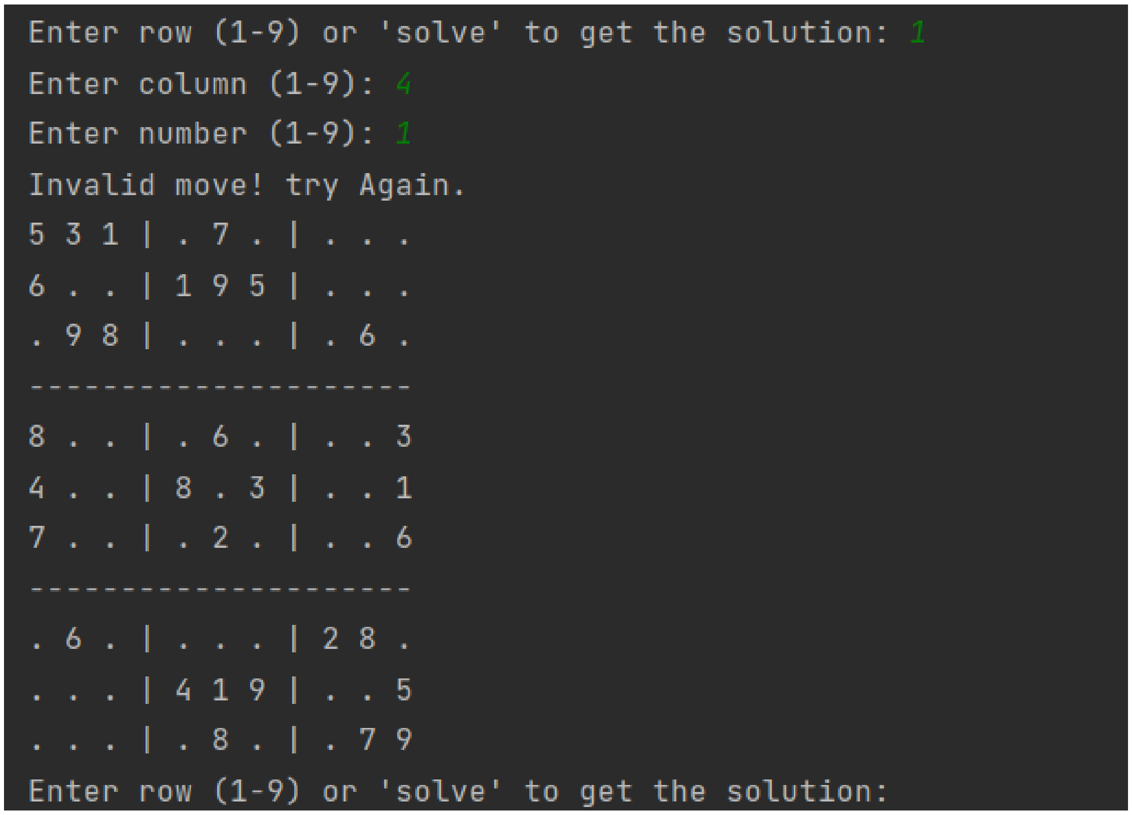


```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:C:\Program Fi
Welcome to Sudoku!
To make a move, enter the row (1-9), column (1-9), and number (1-9).
Type 'solve' to get the solution or 'undo' to revert the last move.
5 3 . | . 7 . | . . .
6 . . | 1 9 5 | . . .
. 9 8 | . . . | . 6 .
---------------------
8 . . | . 6 . | . . 3
4 . . | 8 . 3 | . . 1
7 . . | . 2 . | . . 6
---------------------
. 6 . | . . . | 2 8 .
. . . | 4 1 9 | . . 5
. . . | . 8 . | . 7 9
Enter row (1-9) or 'solve' to get the solution: 1
Enter column (1-9): 3
Enter number (1-9): 4
5 3 4 | . 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
```



```
. . . | 4 1 9 | . . 5
. . . | . 8 . | . 7 9
Enter row (1-9) or 'solve' to get the solution: undo
Last move undone.
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
Enter row (1-9) or 'solve' to get the solution: solve
Here's the 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
Congratulations! You've completed the Sudoku puzzle!
```

Loosing Screenshot:



```
Enter row (1-9) or 'solve' to get the solution: 1
Enter column (1-9): 4
Enter number (1-9): 1
Invalid move! try Again.
5 3 1 | . 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
Enter row (1-9) or 'solve' to get the solution:
```

Conclusion:

This report provides a structured overview of the above Sudoku game, including the logic and purpose of the data structures used.