<https://leetcode.com/problems/sudoku-solver/>

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

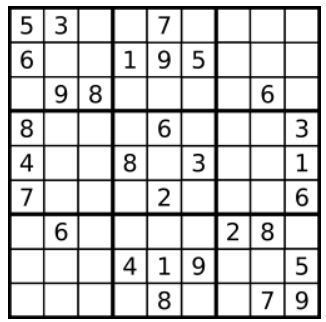
Each of the digits 1-9 must occur exactly once in each row.

Each of the digits 1-9 must occur exactly once in each column.

Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

The '.' character indicates empty cells.

**Example 1:**



Input: board = [["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"]]

Output: [["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"],

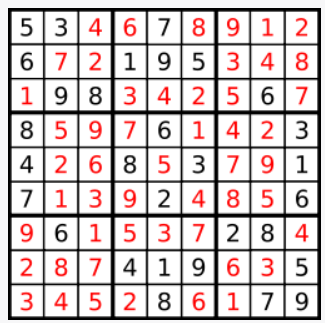
        ["7","1","3","9","2","4","8","5","6"],

        ["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"]]

Explanation: The input board is shown above and the only valid solution is shown below:



**Constraints:**

board.length == 9

board[i].length == 9

board[i][j] is a digit or '.'.

It is **guaranteed** that the input board has only one solution.

**Attempt 1: 2023-01-06**

**Solution 1: Backtracking (10 min)**

class Solution {

    public void solveSudoku(char[][] board) {

        helper(board);

    }

    private boolean helper(char[][] board) {

        for(int i = 0; i < 9; i++) {

            for(int j = 0; j < 9; j++) {

                if(board[i][j] == '.') {

                    // Try to fill a char(try from '1' to '9')

                    for(char c = '1'; c <= '9'; c++) {

                        if(isValid(board, i, j, c)) {

                            // Attempt with c

                            board[i][j] = c;

                            if(helper(board)) {

                                return true;

                            }

                            // Backtracking

                            board[i][j] = '.';

                        }

                    }

                    return false;

                }

            }

        }

        return true;

    }

    private boolean isValid(char[][] board, int row, int col, char c) {

        for(int i = 0; i < 9; i++) {

            // If same char(=c) already exists in given row then invalid

            if(board[row][i] == c) {

                return false;

            }

            // If same char(=c) already exists in given column then invalid

            if(board[i][col] == c) {

                return false;

            }

            // If same char(=c) already exists in given 3x3 sub-boxes of the grid then invalid

            // "3 \* (row / 3)" and "3 \* (col / 3)" help to find which 3x3 sub-box in board

            // "i / 3" and "i % 3" help to locate which cell in sub-box

            if(board[3 \* (row / 3) + i / 3][3 \* (col / 3) + i % 3] == c) {

                return false;

            }

        }

        return true;

    }

}

Time Complexity: O(9^m)

Try 1 through 9 for each cell. The time complexity should be 9 ^ m (m represents the number of blanks to be filled in), since each blank can have 9 choices.

Space Complexity: O(9^m), the recursion stack

**Refer to**

<https://leetcode.com/problems/sudoku-solver/solutions/15752/straight-forward-java-solution-using-backtracking/comments/15800>

Try 1 through 9 for each cell. The time complexity should be 9 ^ m (m represents the number of blanks to be filled in), since each blank can have 9 choices. Details see comments inside code.

    public void solveSudoku(char[][] board) {

        doSolve(board, 0, 0);

    }

    private boolean doSolve(char[][] board, int row, int col) {

        for (int i = row; i < 9; i++, col = 0) { // note: must reset col here!

            for (int j = col; j < 9; j++) {

                if (board[i][j] != '.') continue;

                for (char num = '1'; num <= '9'; num++) {

                    if (isValid(board, i, j, num)) {

                        board[i][j] = num;

                        if (doSolve(board, i, j + 1))

                            return true;

                        board[i][j] = '.';

                    }

                }

                return false;

            }

        }

        return true;

    }

    private boolean isValid(char[][] board, int row, int col, char num) {

        int blkrow = (row / 3) \* 3, blkcol = (col / 3) \* 3; // Block no. is i/3, first element is i/3\*3

        for (int i = 0; i < 9; i++)

            if (board[i][col] == num || board[row][i] == num ||

                    board[blkrow + i / 3][blkcol + i % 3] == num)

                return false;

        return true;

    }

**Refer to**

<https://leetcode.com/problems/sudoku-solver/solutions/15752/straight-forward-java-solution-using-backtracking/comments/15787>

Don't need to check whether the a cell in the row, col or region is not dot. Just check these cells are not c is enough. Since c will not be a '.'

Define region start row and region start col variables make the code a bit more readable and reduce 8 times duplicate computing in each call.

    private boolean isValid(char[][] board, int row, int col, char c){

        int regionRow = 3 \* (row / 3);  //region start row

        int regionCol = 3 \* (col / 3);    //region start col

        for (int i = 0; i < 9; i++) {

            if (board[i][col] == c) return false; //check row

            if (board[row][i] == c) return false; //check column

            if (board[regionRow + i / 3][regionCol + i % 3] == c) return false; //check 3\*3 block

        }

        return true;

    }

**Refer to**

[L36.Valid Sudoku (Ref.L37,L2133)](note://WEBa794bfed438a7ef4e6df229bd1d854d1)