Sudoku Solver

Problem Statement:

Given a 9×9 incomplete sudoku, solve it such that it becomes valid sudoku. Valid sudoku has the following properties.

1. All the rows should be filled with numbers(1 – 9) exactly once.
2. All the columns should be filled with numbers(1 – 9) exactly once.
3. Each 3×3 submatrix should be filled with numbers(1 – 9) exactly once.

Note: Character **'.'** indicates empty cell.

Example:

`Input:`

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

`Output:`

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

`Explanation:`

```
 The empty cells are filled with the possible numbers. There can exist many such
arrangements of numbers. The above solution is one of them. Let's see how we can fill the
cells below.
```

Solution

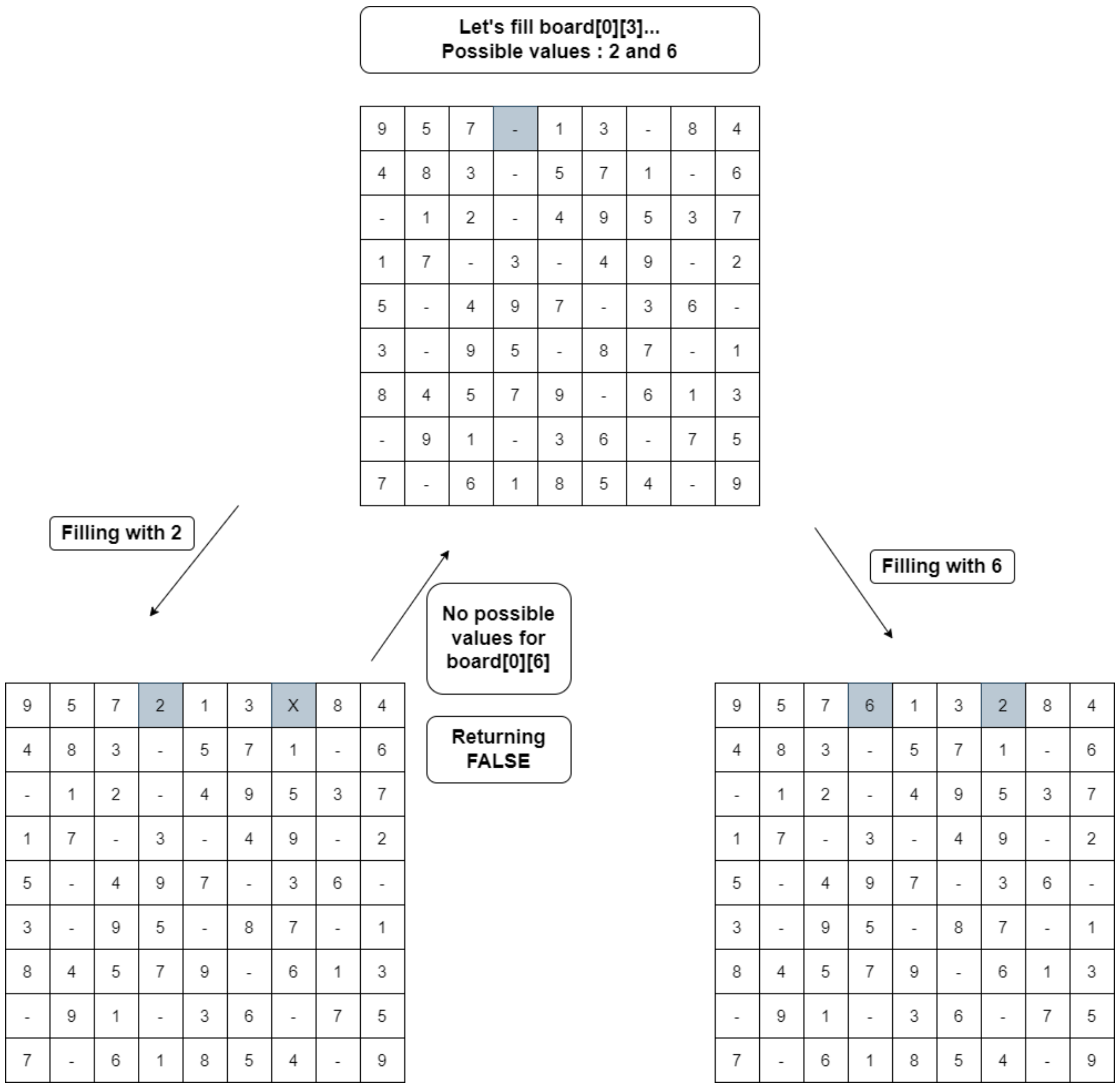
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Intuition:

Since we have to fill the empty cells with available possible numbers and we can also have multiple solutions, the main intuition is to try every possible way of filling the empty cells. And the more correct way to try all possible solutions is to use recursion. In each call to the recursive function, we just try all the possible numbers for a particular cell and transfer the updated board to the next recursive call.

Approach:

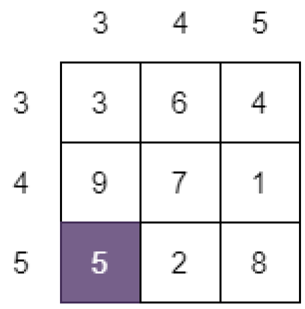
- Let's see the step by step approach. Our main recursive function(solve()) is going to just do a plain matrix traversal of the sudoku board. When we find an empty cell, we pause and try to put all available numbers(1 – 9) in that particular empty cell.
- We need another loop to do that. But wait, we forgot one thing – the board has to satisfy all the conditions, right? So, for that we have another function(isValid()) which will check whether the number we have inserted into that empty cell will not violate any conditions.
- If it is violating, we try with the next number. If it is not, we call the same function recursively, but this time with the updated state of the board. Now, as usual it tries to fill the remaining cells in the board in the same way.
- Now we'll come to the returning values. If at any point we cannot insert any numbers from 1 – 9 in a particular cell, it means the current state of the board is wrong and we need to backtrack. An important point to follow is, we need to return false to let the parent function(which is called this function) know that we cannot fill this way. This will serve as a hint to that function, that it needs to try with the next possible number. Refer to the picture below.


Let's fill board[0][3]...
Possible values : 2 and 6
Filling with 2
No possible values for board[0][6]
Returning FALSE
Filling with 6

- If a recursive call returns true, we can assume that we found one possible way of filling and we simply do an **early return**.

Validating Board

- Now, let's see how we are validating the sudoku board. After determining a number for a cell(at i'th row, j'th col), we try to check the validity. As we know, a valid sudoku needs to satisfy 3 conditions, we can use three loops. But we can do within a single loop itself. Let's try to understand that.
- We loop from 0 to 8 and check the values – board[i][col](1st condition) and board[row][i](2nd condition), whether the number is already included. For the 3rd condition – the expression (3 * (row / 3) + i / 3) evaluates to the row numbers of that 3×3 submatrix and the expression (3 * (col / 3) + i % 3) evaluates to the column numbers.
- For eg, if row= 5 and col= 3, the cells visited are



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

It covers all the cells in the sub-matrix.

Code:
```
class solve {

    public boolean helper(char[][] board)

    {

        for(int i=0;i<9;i++)

        {

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

            {

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

                {

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

                    {

                        if(isPossible(i,j,board,c))

                        {

                            board[i][j]=c;

                            if(helper(board))

                            {

                                return true;

                            }

                            else

                            {
                                board[i][j]='.';
                            }
                        }
                    }
                    return false;
                }
            }
        }
        return true;
    }
    public boolean isPossible(int row,int col,char[][] board,char c)
    {
        for(int i=0;i<9;i++)
        {
            if(board[row][i]==c)
                return false;
            if(board[i][col]==c)
                return false;
            if(board[3*(row/3)+i/3][3*(col/3)+i%3]==c)
                return false;
        }
        return true;
    }
    public void solveSudoku(char[][] board) {
        helper(board);
    }
}
```

Time Complexity: O($9^{(n \wedge 2)}$), in the worst case, for each cell in the n^2 board, we have 9 possible numbers.

Space Complexity: O(1), since we are refilling the given board itself, there is no extra space required, so constant space complexity.