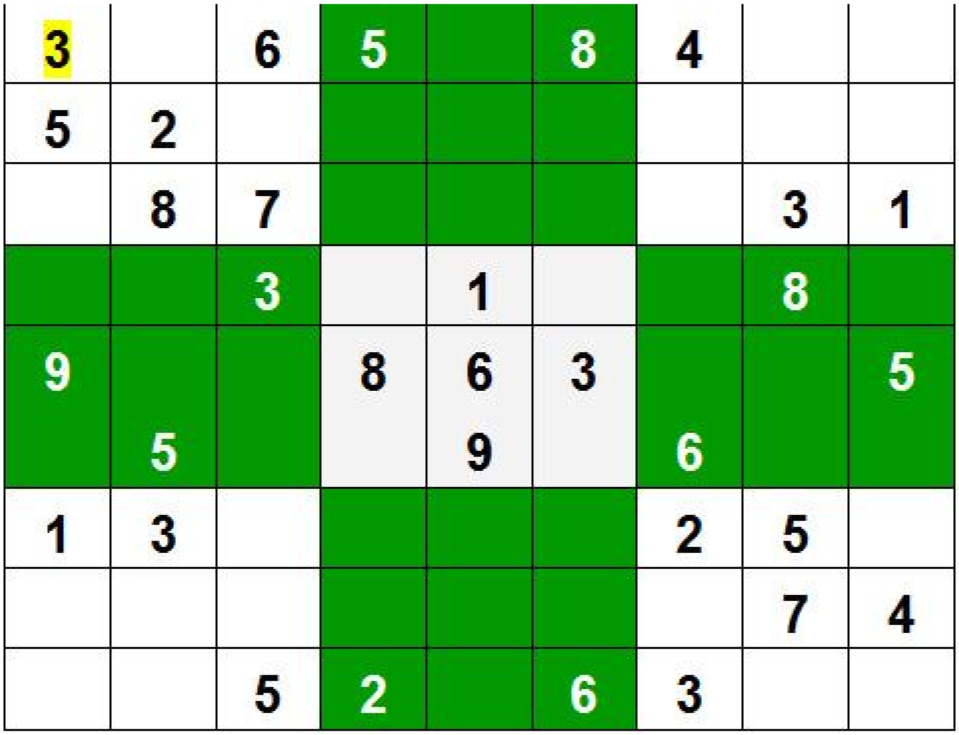
Sudoku Solver Project

Link to the project : https://peaceful-payne-9d98a0.netlify.app/

Problem statement

Given an N X N subgrid where N is a perfect square. Each subgrid is therefore of size square root of N.

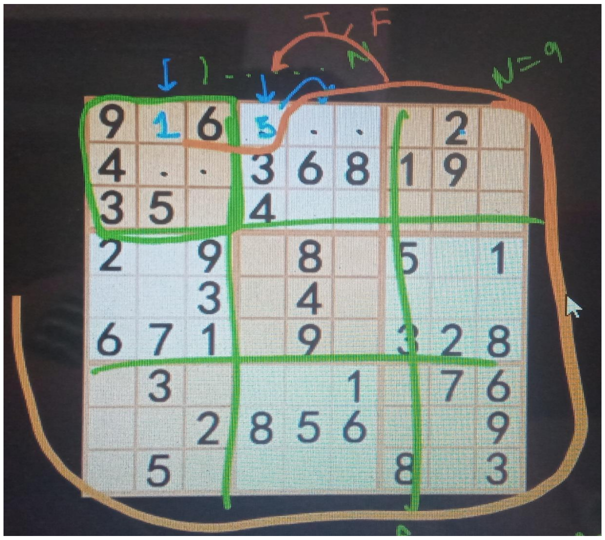


3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

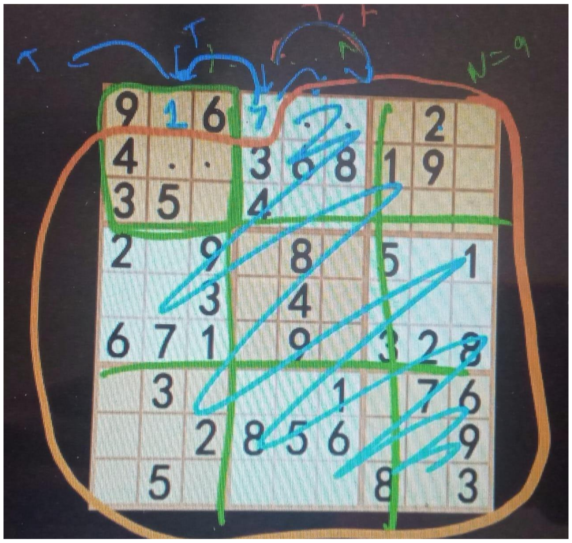
We need to fill the blocks with the numbers between 1-9, such that every row has a number between 1 to 9, every column has a number between 1-9 and every subgrid has a number between 1-9.

Approach

We search for the first non empty cell in the row and fill it with a number that does not occur in the corresponding row,column or sub grid. Sudoku can be solved by one by one assigning numbers to empty cells.Before assigning a number, check whether it is safe to assign. Check that the same number is not present in the current row, current column and current 3X3 subgrid. After checking for safety, assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesn't lead to a solution, then try the next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, return false.



Let’s say we placed 5 in the cell as shown. This might be the right solution or a wrong solution, after 5 is placed in the cell the problem is reduced to a subproblem of subgrid as shown in the figure (highlighted with orange). Now this sub grid would return true or false indicating whether 5 was the correct place or not.

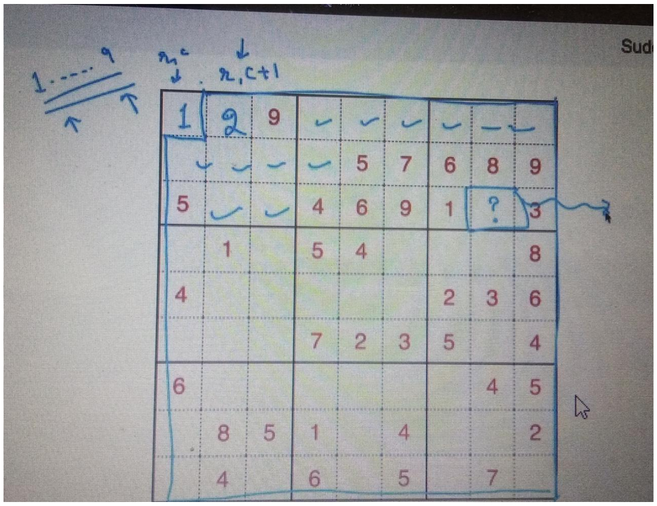


Let’s say it was **incorrect** i.e. false is returned from the subproblem (recursion)**:**

So we place 7 in the cell, the subgrid returns true to 7 which further returns true to the non empty cell (recursively) and finally true is returned to the main function.

Hence to summarize:

1. Work on the current cell
2. Recursively call the subproblem
3. If sub problem returns true, True is returned to main
4. If the subproblem returns false then **Backtrack and update the current cell.**



The subproblem of size (row,column+1)

Explanation of the Code:

1. **solveSudoku function**

solve.onclick -> whenever the solve button is pressed, solveSudoku function is called. A 2-d array is passed to the function as an argument. We will be using **solveSudokuHelper** function.

Note : Explanation in the comments

```
function solveSudokuHelper(board, sr, sc) {
// when you have reached the last row, it's time to display the solved board hence call
the changeboard function and then return
   if (sr == 9)
  {
      changeBoard(board);
      return;
   }
//While moving along the row, we have reached the last cell of the column, in that case
we need to redirect it to the next row and then return whatever value is returned for that
sub problem(true or false)
   if (sc == 9) {
      solveSudokuHelper(board, sr + 1, 0) // change the row
      return;
   }

//condition for pre filled cell, i.e. already filled cell, then skip that column. The cell which
is not pre filled is 0
```

```
    if (board[sr][sc] != 0) {
        solveSudokuHelper(board, sr, sc + 1);
        return;
    }

// there is a 0 in the current location. Try to put all the possible values from 1 to 9 in the cell and
check if it works or not recursively.

    for (var i = 1; i <= 9; i++) {

//check if it is possible to place the value, i  at the current cell using isPossible function.
        if (isPossible(board, sr, sc, i)) {
            board[sr][sc] = i;
            solveSudokuHelper(board, sr, sc + 1); //if it is safe then ,make a call to the
remaining part of the matrix
//if false is returned then we need to remove our assumed value and make it 0 again
            board[sr][sc] = 0; //backtrack
        }
    }
}
// Suppose if we try to fill all the values in the cell from 1 to 9 and none of them works,
then this function would return false to the previous cell i.e. backtrack and then update
the previous cells
```

2. **isPossible** function.
   Note : Explanation is given in the comments

```
function isPossible(board, sr, sc, val) {
//check if the value does not occur in the current row or column
    for (var row = 0; row < 9; row++) {
        if (board[row][sc] == val) {
            return false;
        }
    }

    for (var col = 0; col < 9; col++) {
        if (board[sr][col] == val) {
            return false;
        }
    }
// check if the value in the current cell occurs in the subgrid or not. The starting
coordinates of the subgrid for a given cell can be found using the formula:
// sx = (x/3)*3 and sy = (y/3)*3
// where sx and sy are the x and y coordinate of the starting point of the subgrid
and x and y are the given coordinates for which sx and sy needs to be found.
Once we get the starting coordinates of the subgrid, we can traverse 3 places in
the row and 3 places in the column
//below 2 lines are equivalent to sx = (x/3)*3 and sy = (y/3)*3

    var r = sr - sr % 3;
    var c = sc - sc % 3;

    for (var cr = r; cr < r + 3; cr++) {
        for (var cc = c; cc < c + 3; cc++) {
            if (board[cr][cc] == val) {
                return false;
            }
        }
    }
//if the value does not occur in the corresponding row, column or subgrid return
true
    return true;

}
```

Entire Source code

```
var arr = [[], [], [], [], [], [], [], [], []]
var temp = [[], [], [], [], [], [], [], [], []]

for (var i = 0; i < 9; i++) {
    for (var j = 0; j < 9; j++) {
        arr[i][j] = document.getElementById(i * 9 + j);

    }
}
```

```
function initializeTemp(temp) {

    for (var i = 0; i < 9; i++) {
        for (var j = 0; j < 9; j++) {
            temp[i][j] = false;

        }
    }
}


function setTemp(board, temp) {

    for (var i = 0; i < 9; i++) {
        for (var j = 0; j < 9; j++) {
            if (board[i][j] != 0) {
                temp[i][j] = true;
            }

        }
    }
}


function setColor(temp) {

    for (var i = 0; i < 9; i++) {
        for (var j = 0; j < 9; j++) {
            if (temp[i][j] == true) {
                arr[i][j].style.color = "#DC3545";
            }

        }
    }
}

function resetColor() {

    for (var i = 0; i < 9; i++) {
        for (var j = 0; j < 9; j++) {


            arr[i][j].style.color = "green";
        }
    }
}

var board = [[], [], [], [], [], [], [], [], []]


let button = document.getElementById('generate-sudoku')
let solve = document.getElementById('solve')

console.log(arr)
function changeBoard(board) {
    for (var i = 0; i < 9; i++) {
        for (var j = 0; j < 9; j++) {
            if (board[i][j] != 0) {

                arr[i][j].innerText = board[i][j]
            }

            else
                arr[i][j].innerText = ''
        }
    }
}


button.onclick = function () {
    var xhrRequest = new XMLHttpRequest()
    xhrRequest.onload = function () {
        var response = JSON.parse(xhrRequest.response)
        console.log(response)
        initializeTemp(temp)
        resetColor()

        board = response.board
        setTemp(board, temp)
        setColor(temp)
        changeBoard(board)
    }
    xhrRequest.open('get', 'https://sugoku.herokuapp.com/board?difficulty=easy')
    //we can change the difficulty of the puzzle the allowed values of difficulty are easy, medium, hard and random
    xhrRequest.send()
}

//to be completed by student
function isPossible(board, sr, sc, val) {
    for (var row = 0; row < 9; row++) {
        if (board[row][sc] == val) {
            return false;
        }
    }

    for (var col = 0; col < 9; col++) {
        if (board[sr][col] == val) {
            return false;
        }
    }

    var r = sr - sr % 3;
    var c = sc - sc % 3;

    for (var cr = r; cr < r + 3; cr++) {
        for (var cc = c; cc < c + 3; cc++) {
            if (board[cr][cc] == val) {
                return false;
            }
        }
    }
    return true;

}

//to be completed by student
function solveSudokuHelper(board, sr, sc) {
    if (sr == 9) {
        changeBoard(board);
        return;
    }
    if (sc == 9) {
        solveSudokuHelper(board, sr + 1, 0)
        return;
    }

    if (board[sr][sc] != 0) {
        solveSudokuHelper(board, sr, sc + 1);
        return;
    }

    for (var i = 1; i <= 9; i++) {
        if (isPossible(board, sr, sc, i)) {
            board[sr][sc] = i;
            solveSudokuHelper(board, sr, sc + 1);
            board[sr][sc] = 0;
        }
    }
}

function solveSudoku(board) {
    solveSudokuHelper(board, 0, 0)
}

solve.onclick = function () {
    solveSudoku(board)

}
```