Jasmin Johal

CS 325: Portfolio Project

Rules:

For the final project, I chose to implement the Sudoku puzzle. I created a GUI playable game via HTML/CSS/JavaScript that is accessible on the webpage https://jjrx.github.io/cs340_sudoku/index.html. It follows the traditional rules of Sudoku. To solve the puzzle, each cell in an nxn grid of n rows and n columns must be filled in with a number from 1 to n (inclusive). The grid is also split into n boxes, each with n cells. Each row, each column, and each box must contain every number from 1 to n. Therefore, the game is constrained such that a number can only appear once in a given row, column, or box.

In my implementation, $n = 9$. The board is hard-coded in and the user can enter in integers from 1 to 9 into each cell of the 9x9 grid that isn't hard-coded. Upon filling in all cells, the user can press a button to trigger a verification algorithm that checks if his/her solution is valid. The code and analysis for the verification algorithm is outlined below:

Verification Algorithm Code:

```javascript
function checkForDuplicates(val, i, j, grid, attribute) {
  if (attribute == 'boxes') {
    var cell = document.getElementById(`${i},${j}`);
    var boxClass = cell.className.split(" ")[1];
    var key = boxClass;
  } else if (attribute == 'rows') {
    var curRow = String(i);
    var key = curRow;
  } else if (attribute == 'cols') {
    var curCol = String(j);
    var key = curCol;
  }

  if (key in grid[attribute]) {
    if (grid[attribute][key].includes(val)) {
      updateAlert('Invalid solution.');
      return false;
    } else {
      grid[attribute][key].push(val);
    }
  } else {
    grid[attribute][key] = [val];
  }

  return true;
}
```

```
function checkIfSolved(n) {
  var grid = {
    boxes: {},
    rows: {},
    cols: {}
  };

  for (var i = 0; i < n; i++) {
    for (var j = 0; j < n; j++) {
      var val = sampleTable[i][j];

      if (val == 0) {
        updateAlert('Invalid solution.');
        return false;
      }

      if (!checkForDuplicates(val, i, j, grid, 'boxes')) {
        return false;
      }

      if (!checkForDuplicates(val, i, j, grid, 'rows')) {
        return false;

      }
      if (!checkForDuplicates(val, i, j, grid, 'cols')) {
        return false;
      }
    }
  }

  updateAlert('Solved!');
  return true;
}
```

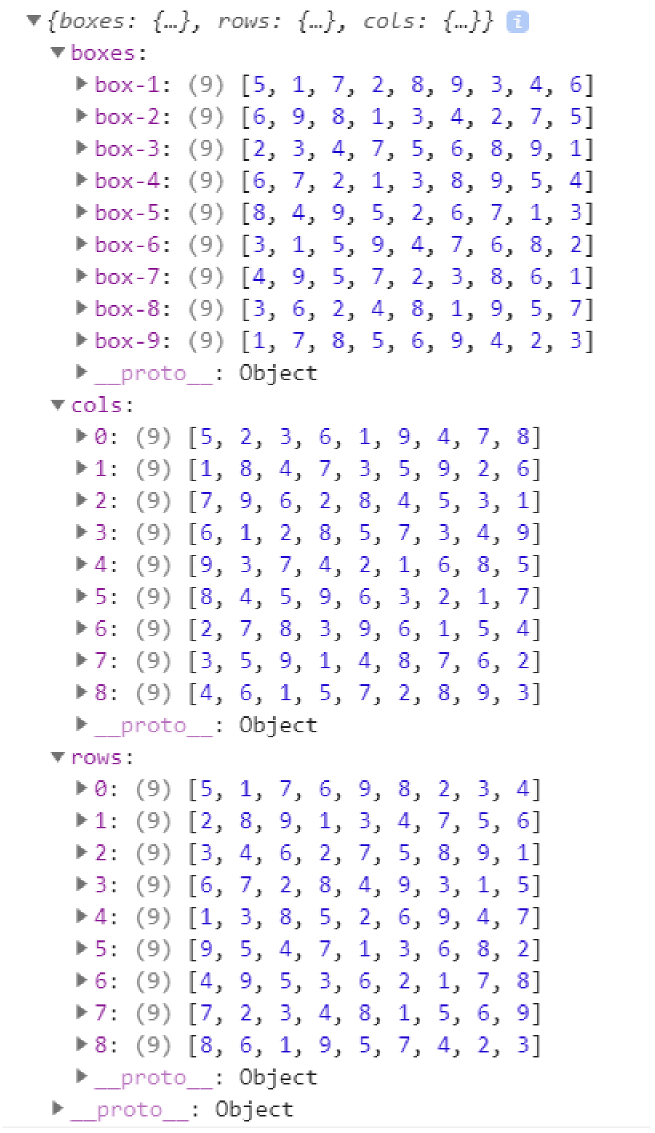
Verification Algorithm Analysis:

The verification algorithm checks that the user-inputted grid obeys the following constraints as required by the Sudoku game:

- There are no duplicate numbers in any row.
- There are no duplicate numbers in any column.
- There are no duplicate numbers in each of the 9 3x3 boxes.

Note that Sudoku also requires that all numbers in the grid are integers between 1 and 9 (inclusive). However, my application checks all user input to make sure that the user is entering a valid integer so the verification algorithm does not need to handle this constraint; it is handled by another piece of the application.

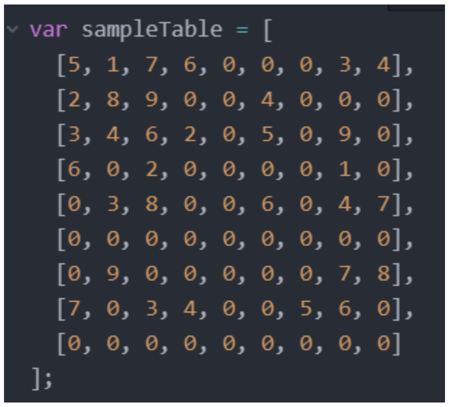
The main verification algorithm is implemented in my application via the function `checkIfSolved`. `checkForDuplicates` is a helper function that is referenced three times within `checkIfSolved`, once for each attribute type (rows, columns, and boxes). `checkIfSolved` returns `true` if the solution is valid; it returns `false` otherwise. It works by creating a dictionary named `grid` with 3 dictionaries nested inside: `boxes`, `rows`, and `cols`. Each of the inner dictionaries is designed to hold all attributes of that type. When the entire grid has been iterated over, the `grid['boxes']` dictionary has a key for each of the 9 3x3 boxes in the grid, the `grid['rows']` dictionary has a key for each of the 9 rows, and the `grid['cols']` dictionary has a key for each of the 9 columns. The corresponding values are lists of all the values in that attribute. For example, `grid['rows']['0']` would be a list of all the numbers in the first row while `grid['cols']['3']` would be a list of all the numbers in the fourth column. Below is a screenshot of the `grid` dictionary logged to the console after checking a user's solution:



```
▼{boxes: {…}, rows: {…}, cols: {…}}
  ▼boxes:
    ▶box-1: (9) [5, 1, 7, 2, 8, 9, 3, 4, 6]
    ▶box-2: (9) [6, 9, 8, 1, 3, 4, 2, 7, 5]
    ▶box-3: (9) [2, 3, 4, 7, 5, 6, 8, 9, 1]
    ▶box-4: (9) [6, 7, 2, 1, 3, 8, 9, 5, 4]
    ▶box-5: (9) [8, 4, 9, 5, 2, 6, 7, 1, 3]
    ▶box-6: (9) [3, 1, 5, 9, 4, 7, 6, 8, 2]
    ▶box-7: (9) [4, 9, 5, 7, 2, 3, 8, 6, 1]
    ▶box-8: (9) [3, 6, 2, 4, 8, 1, 9, 5, 7]
    ▶box-9: (9) [1, 7, 8, 5, 6, 9, 4, 2, 3]
    ▶__proto__: Object
  ▼cols:
    ▶0: (9) [5, 2, 3, 6, 1, 9, 4, 7, 8]
    ▶1: (9) [1, 8, 4, 7, 3, 5, 9, 2, 6]
    ▶2: (9) [7, 9, 6, 2, 8, 4, 5, 3, 1]
    ▶3: (9) [6, 1, 2, 8, 5, 7, 3, 4, 9]
    ▶4: (9) [9, 3, 7, 4, 2, 1, 6, 8, 5]
    ▶5: (9) [8, 4, 5, 9, 6, 3, 2, 1, 7]
    ▶6: (9) [2, 7, 8, 3, 9, 6, 1, 5, 4]
    ▶7: (9) [3, 5, 9, 1, 4, 8, 7, 6, 2]
    ▶8: (9) [4, 6, 1, 5, 7, 2, 8, 9, 3]
    ▶__proto__: Object
  ▼rows:
    ▶0: (9) [5, 1, 7, 6, 9, 8, 2, 3, 4]
    ▶1: (9) [2, 8, 9, 1, 3, 4, 7, 5, 6]
    ▶2: (9) [3, 4, 6, 2, 7, 5, 8, 9, 1]
    ▶3: (9) [6, 7, 2, 8, 4, 9, 3, 1, 5]
    ▶4: (9) [1, 3, 8, 5, 2, 6, 9, 4, 7]
    ▶5: (9) [9, 5, 4, 7, 1, 3, 6, 8, 2]
    ▶6: (9) [4, 9, 5, 3, 6, 2, 1, 7, 8]
    ▶7: (9) [7, 2, 3, 4, 8, 1, 5, 6, 9]
    ▶8: (9) [8, 6, 1, 9, 5, 7, 4, 2, 3]
    ▶__proto__: Object
  ▶__proto__: Object
```

The algorithm then sets up a nested for loop to loop over each cell in the 9x9 grid (outer for loop to iterate over the rows and inner for loop to iterate over the columns).

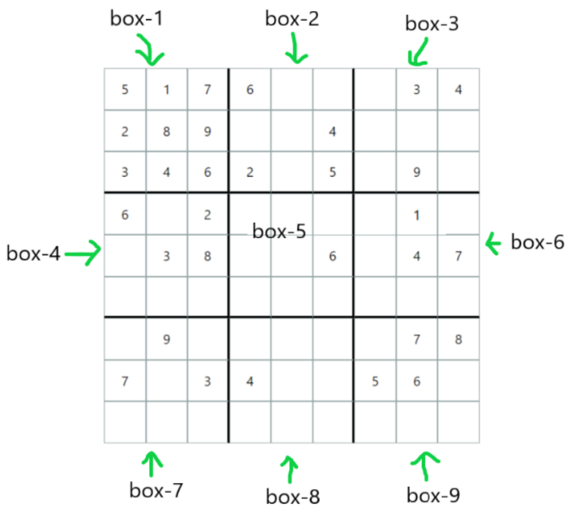
In my application, the 9x9 Sudoku grid is represented by a list of lists. It is stored in a variable called `sampleTable`. Each of the nine lists (which represent the nine rows) holds nine integers, which represent the nine columns. So each cell in the grid is accessible by `sampleTable[i][j]` where `i` is the row of cell and `j` is the column.



```
var sampleTable = [
    [5, 1, 7, 6, 0, 0, 0, 3, 4],
    [2, 8, 9, 0, 0, 4, 0, 0, 0],
    [3, 4, 6, 2, 0, 5, 0, 9, 0],
    [6, 0, 2, 0, 0, 0, 0, 1, 0],
    [0, 3, 8, 0, 0, 6, 0, 4, 7],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 9, 0, 0, 0, 0, 0, 7, 8],
    [7, 0, 3, 4, 0, 0, 5, 6, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0]
];
```

Returning back to the code, we extract the current cell and store it in the variable `val`. We then perform a simple check: if `val` equals 0, the solution is invalid. A 0 in the grid represents an empty cell. Every cell must be filled in for a solution to be valid so a 0 automatically indicates that the solution is incomplete. If there are no empty cells, we proceed to check for other constraints (as outlined above).

The first `checkForDuplicates` call is on the `‘boxes’` attribute. It first retrieves the desired cell element by the `document.getElementById` method since the program assigns each cell a unique ID based on its position in the location in the grid. For example, the first cell would have ID `‘0,0’` while the last element would have ID `‘8,8’` (since it’s located in the 8th row and 8th column). The program also assigns each cell a class based on what 3x3 box it is in. Here is a diagram of how the grid is divided into 9 boxes and how each of the box classes are named:



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

So, the first cell would have class `‘box-1’` while the last cell would have class `‘box-9’`. *Note that these ID and class assigning steps occur in the `createTable` function, which is not shown in the code above.*

After the function extracts the box class, it checks if this box class already exists in `grid[‘boxes’]`. If not, it creates a key in `grid[‘boxes’]` and sets it value to a list with the current cell's value. For example, let's suppose we are on the first cell which holds value 5. So, `i = 0` and `j = 0`. Since this is the first value we've seen in the grid, the dictionary `grid` would be empty when we call `checkForDuplicates`. Therefore, we would add the key-value pair `‘box-1’: [5]` to `grid[‘boxes’]`. On the next iteration, when we are checking the second cell (`i = 0`, `j = 1`, and `val = 1`), we extract its box class and note that it is `‘box-1’`. `grid[‘boxes’][‘box-1’]` already exists so then we would go on to check if the value already exists in `grid[‘boxes’][‘box-1’]`. Currently, `grid[‘boxes’][‘box-1’] = [5]`. Since the current `val` is 1 and $1 \neq 5$, we haven't violated any constraints so we append 1 to `grid[‘boxes’][‘box-1’]`. Now, `grid[‘boxes’][‘box-1’] = [5,1]`. Suppose instead that the `val` of the second cell was 5. Since 5 already exists in `grid[‘boxes’][‘box-1’]` via the first cell, we have violated a constraint. There cannot be duplicate numbers in a 3x3 grid so the `checkForDuplicates` call would return false, which would cause `checkIfSolved` to return false. The user solution is invalid.

Note that we also perform `checkForDuplicates` calls on `rows` and `cols`. We perform similar checks for rows and columns to ensure two of the same number do not exist in any given row or column. If the entire grid has been traversed and no duplicates have been found, we can be sure the solution is valid so we return `true`.

As we've demonstrated, the algorithm is a brute force algorithm since it walks through each cell in the grid and stores all seen values in the `grid` dictionary. It checks to make sure a given cell's value hasn't already been seen in its corresponding 3x3 box, row, or column. The time complexity of the brute force verification algorithm is $O(n^2)$, where n is the number of rows/columns/boxes in the Sudoku grid. The time complexity is $O(n^2)$, because in the worst case, the algorithm traverses the entire grid and performs constant operations (e.g. lookups) at every single cell to assess if the constraints are met. The grid is traversed via a double nested for loop since there are n^2 cells in an nxn grid so it follows that the time complexity is $O(n^2)$.

<u>NP-Complete?</u>

The decision problem for Sudoku is ‘given a Sudoku instance, does it have any solutions?’[1].

To demonstrate that this problem NP-complete, it must satisfy two requirements:

(1) It is in NP.
(2) Every problem $A \in NP$ must be polynomial-time reducible to it.

We have already shown that property (1) is true since we created a certifier (verification algorithm) that runs in polynomial time. Specifically, the runtime complexity for our verification algorithm is $O(n^2)$, where n is the number of rows/columns/boxes in an nxn grid. The verification algorithm performs constant time operations while checking every cell in the nxn grid so it follows that the runtime is $O(n^2)$. Since a solution to the decision problem can be verified in polynomial time, it is NP. Proving (2) is outside the scope of this class but Sudoku has been shown to be NP-Complete[2].

References:

[1] Kendall, G. and Parkes, A. and Spoerer, K. (2008). A Survey of NP-Complete puzzles. ICGA Journal. 31. 13-34. 10.3233/ICG-2008-31103.

[2] Yato, T. and Seta, T. (2003). Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, Vol. 86, No. 5, pp. 1052–1060.