Coursework: Sudoku assignment

Problem Solving for Computer Science

The 11 tasks in this assignment make up the Sudoku coursework assignment. Download and open the folder stored in `sudoku.zip` then change the directory in your CLI to this folder. The tasks in this assignment consist mainly of completing JavaScript functions in the file called `sudoku.js`. Two tasks consist of written work that will be completed in the file `sudoku.txt`.

Note that you do not have any tests to run with `npm test`. You will only test your code through the examples given in the tasks.

There are 55 marks available in total for this assignment

1 Background: Sudoku and Pseudoku

A Sudoku puzzle consists of 9-by-9 grid of squares, some of them blank, some of them having integers from 1 to 9. A typical Sudoku puzzle will then look something like this:

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

To solve this puzzle, all the squares must be filled with numbers from 1 to 9 such that the following are satisfied:

1. every row has all integers from 1 to 9 (with each appearing only once)
2. every column has all integers from 1 to 9 (with each appearing only once)
3. every 3-by-3 sub-grid, or block (with bold outlines around them going from top-left to bottom-right) has all integers from 1 to 9

In this coursework, we won't be generating and solving Sudoku puzzles exactly, but a simplified version of Sudoku puzzles, which I will call Pseudoku puzzles – pronounced the same. In a Pseudoku puzzle, we now have a 4-by-4 grid of squares, some of them blank, some of them having integers from 1 to 4. A typical Pseudoku puzzle will look like this:

	4	**1**	
		2	
3			
	1		**2**

To solve this puzzle, all the squares must be filled with numbers from 1 to 4 such that the following are satisfied:

1. every row has all integers from 1 to 4 (with each appearing only once)

2. every column has all integers from 1 to 4 (with each appearing only once)
3. every 2-by-2 sub-grid, or block (with bold outlines around them going from top-left to bottom-right) has all integers from 1 to 4

These three conditions will be called the Pseudoku conditions. For the above Pseudoku puzzle, a solution is:

2	4	1	3
1	3	2	4
3	2	4	1
4	1	3	2

The goal of the whole Sudoku assignment is to produce a program that can generate Pseudoku puzzles. It is important to emphasise that a Pseudoku puzzle is specifically a 4-by-4 puzzle as above, and not 9-by-9, or any other size. So when we refer to Pseudoku puzzles, we are specifically thinking of these 4-by-4 puzzles.

2 Generating Pseudoku puzzles

You are going to try and produce code that algorithmically generates a Pseudoku puzzle. This algorithm starts with an array of four elements, with all the integers 1 to 4 in any particular order, e.g. `[1, 2, 3, 4]` or `[4, 1, 3, 2]`. In addition to this array, the program also starts with an integer `n`, which is going to be the number of blank spaces in the generated puzzle. This whole process will be modular, where multiple functions combine to produce the puzzle.

The big picture of the algorithm behind the code is to construct a solved Pseudoku puzzle by duplicating the input array mentioned earlier. Then from the solved puzzle, the algorithm will remove numbers and replace them with blank entries to give an unsolved puzzle. These are the main steps in the algorithm:

1. Get the input array called `row` and number `n`
2. Create a two-dimensional array of four rows called `puzzle`, where each row of `puzzle` is itself the array `row`
3. Cyclically permute the bottom three rows of `puzzle` so that `puzzle` satisfies the Pseudoku conditions
4. Remove values in elements of `puzzle` to leave blank spaces, and complete the puzzle

The first three steps of this algorithm involve manipulating JavaScript arrays, using queues and using the Linear Search algorithm multiple times. Step 4 will bring everything together and call a function that can randomly pick elements to make blank.

As mentioned, we will start with a completed puzzle stored in a two-dimensional array called `puzzle` where every element (row) is an array with four elements (giving four columns). If we take the completed puzzle from earlier

2	4	1	3
1	3	2	4
3	2	4	1
4	1	3	2

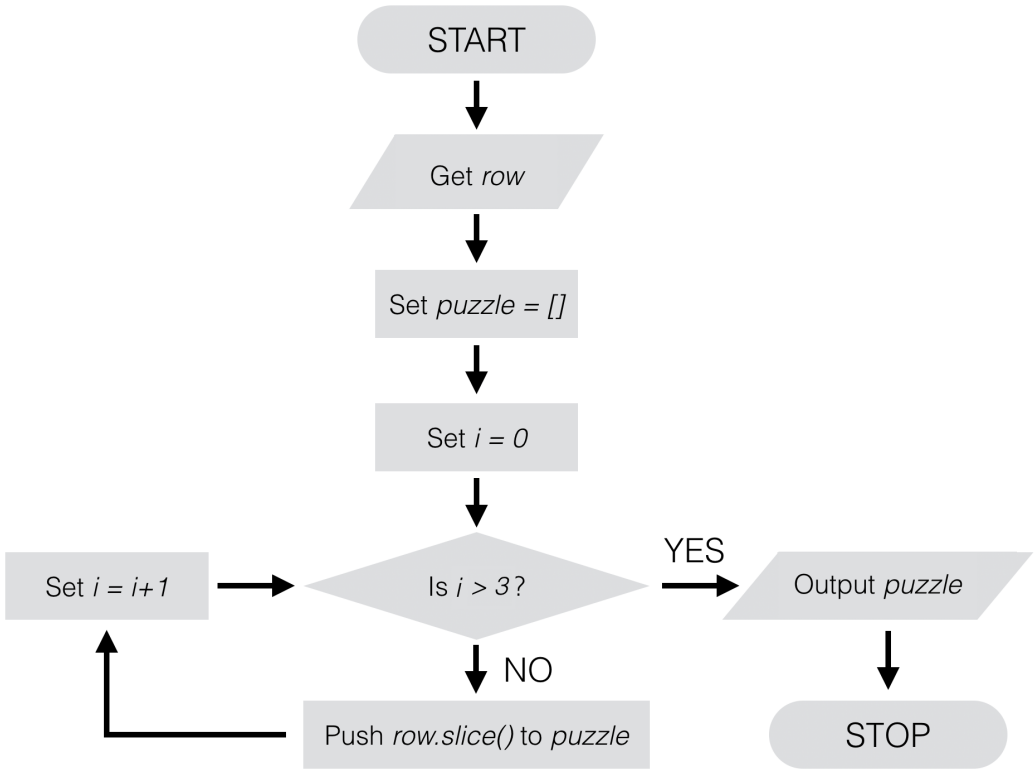
The array representing this completed puzzle will be:

```
[[2, 4, 1, 3], [1, 3, 2, 4], [3, 2, 4, 1], [4, 1, 3, 2]]
```

The first three steps of the algorithm generate such an array from the first row `[2, 4, 1, 3]`. The first eight tasks in this assignment focus on this process.

3 Getting started

Your first task is to write a function that will implement step 2 in the algorithm described in Section 2. That is, you will complete a function that has the argument array `row` and pushes this array to an empty array four times. This is the flowchart for this process:



Task 1: Complete the function `makeRows(row)` in *sudoku.js* that has the argument `row`. Alter the body of the function `makeRows(row)` so that it implements the flowchart above. *To get full marks, you need to use a loop*. Hint: Make sure you push `row.slice()`, and not just `row`.

Testing: Use these two lines of code to test the function:

```
var row = [1, 2, 3, 4];
console.log(makeRows(row));
```

The array `[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]` should be printed to the console.

[4 marks]

At this point it is worthwhile to point out a useful function that will help us visualise our Pseudoku puzzles: this is the function `visPuzzle`, which is at the bottom of the file `sudoku.js`. `visPuzzle` takes an array called `puzzle` as an argument and returns a string that will give a picture of the puzzle. For example, try the code below:

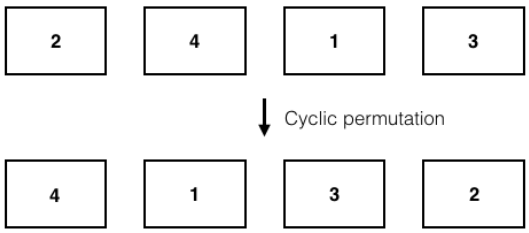
```
var row = [1, 2, 3, 4];
var puzzle = makeRows(row);
console.log(visPuzzle(puzzle));
```

The following should appear in the console:

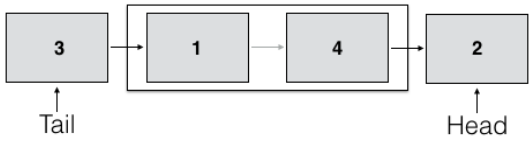
```
-----------------
| 1 | 2 | 3 | 4 |
-----------------
| 1 | 2 | 3 | 4 |
-----------------
| 1 | 2 | 3 | 4 |
-----------------
| 1 | 2 | 3 | 4 |
-----------------
```

4 Cyclic permutation of rows

The returned array from `makeRows(row)` will not satisfy the Pseudoku conditions since, for example, the first column will not have all numbers from 1 to 4. The algorithm for generating Pseudoku puzzles will cyclically permute the values in the bottom three rows until the Pseudoku conditions are satisfied. A cyclic permutation of each row by one element will shift all values of the elements one place to the left with the value at the end going to the other end. For example, for the array `[2, 4, 1, 3]` , if we cyclically permute all elements one place to the left we will have `[4, 1, 3, 2]`, as in the following picture:

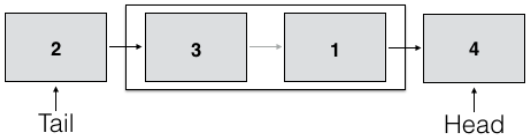


Given an array `row` and a number `p`, which is an integer between 0 and 3 (inclusive) we want to write a function to cyclically permute the values in `row` by `p` elements to the left. An elegant way to do this is to use the queue abstract data structure. All values in the array will be enqueued from left to right into an empty queue, e.g. `[2, 4, 1, 3]` as above should give a queue that looks like this:



We can see that the leftmost element in the array is at the head of the queue, with the tail storing the rightmost element in the array.

To cyclically permute all values one place to the left we enqueue the value stored at the head of the queue and then dequeue the queue. This process will then give the following queue:



To cyclically permute the values further we can just repeat this process of storing enqueuing the head value and dequeuing multiple times. When we have finished this process, we then just push the values stored in the queue to an array, which can be done by reading the head, pushing that to the array, and dequeuing as many times as needed.

A very similar process for cyclic permutations was covered in Lab 3, so the above should feel familiar. In the next task, the goal is to write a function that will take an array called `row` and cyclically permute its values to the left by `p` elements. In the file `sudoku.js`, below `makeRows`, you will see a constructor for the `Queue` object that implements the queue abstract data structure. You will need to use this object in the next task.

Task 2: Complete the function `permuteRow(row, p)` that has the arguments `row` and `p`. Alter the body of the function `permuteRow(row, p)` so that it returns `row` but with all its values cyclically permuted by `p` elements to the left. *To get full marks, you need to use the methods in the `Queue` object.*

The function should reproduce the process described above of enqueuing the values in the array into an empty queue, permute the queue, then push the values in the queue to an empty array. This final array is the one that will be returned.

Testing: Use these two lines of code to test the function:

```
var row = [1, 2, 3, 4];
console.log(permuteRow(row, 2));
```

The array `[3, 4, 1, 2]` should be printed to the console.

[7 marks]

The function `permuteRow`, once completed, will only cyclically permute one array. The next task is to take an array *puzzle* and apply `permuteRow` to each of the bottom three row arrays in *puzzle*. That is, given `puzzle` and numbers `p`, `q`, and `r`, the bottom three rows of `puzzle` will be cyclically permuted `p`, `q`, and `r` places to the left respectively.

Task 3: Complete the function `permutePuzzle` with arguments `puzzle`, `p`, `q` and `r`. Alter the body of the function `permutePuzzle` so that it returns `puzzle` but with the bottom three rows by `p`, `q` and `r` elements to the left. *To get full marks, you need to call the function `permuteRow`.*

Think how each row `puzzle[i]` should be assigned the array returned by `permuteRow(puzzle[i])` for $1 \leq i \leq 3$.

Testing: Use these lines of code to test the function:

```
var row = [1, 2, 3, 4];
var puzzle = makeRows(row);
console.log(permutePuzzle(puzzle, 1, 2, 3));
```

The following array should be printed to the console:

```
[[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]]
```

[4 marks]

5 Checking the Pseudoku column conditions

The next step in implementing the algorithm is to write functions to decide if the Pseudoku conditions are satisfied by a two-dimensional array. If we start with the output of the function call `makeRows(row)`, then all

of the row conditions are satisfied as long as *row* has the numbers 1 to 4 appearing only once. However, the column conditions might not be satisfied: only one number appears in each column (four times). Here we will write two functions that will automate this process of checking if all columns of `puzzle` have all numbers from 1 to 4.

In order to test whether all numbers from 1 to 4 appear in a column, we will use the *Linear Search algorithm* repeatedly. In particular, first we construct an array out of the four values in a column, and then we check if all integers from 1 to 4 appear in that array. You can find an implementation of the Linear Search algorithm in `sudoku.js`: the function called `linearSearch` that takes the arguments `array` and `item`, and it returns `true` if `item` is contained in `array`, and `false` otherwise.

To illustrate this method with an example, given the two-dimensional array `puzzle` of the form:

```
[[1, 2, 3, 4], [2, 3, 4, 1], [2, 3, 4, 1], [4, 1, 2, 3]]
```

To test that all integers from 1 to 4 appear in the first column, first an array called `check` with four elements is created where `check[i]` = `puzzle[i][0]` for $0 \leq$ `i` ≤ 3. From the example above, `check` will be:

```
[1, 2, 2, 4]
```

Then for each integer `k` from 1 to 4, we call `linearSearch(check, k)`, and if it returns `false` for any `k`, then the Pseudoku conditions are not satisfied. In the example `check` above we see that 3 is not there, and so the conditions will not be satisfied.

In the next task, you will write a function that implements the procedure given above: it should create an array of all the column entries for all particular column, and then check that array for all numbers from 1 to 4 using Linear Search. Then in the task after that, you will complete a function that does this process for all columns in the array `puzzle`.

Task 4: Complete the function `checkColumn` with arguments `puzzle` and number `j` between `0` and `3` (inclusive). Alter the body of the function `checkColumn` so that it returns `true` if all integers from 1 to 4 appear in the column `j` of `puzzle`, and `false` otherwise. *To get full marks, you need to call the function `linearSearch`.*

Create an array that stores all elements of column `j` of `puzzle`, and then call `linearSeach` four times on this array to search for all integers from 1 to 4.

Testing: Use these lines of code to test the function:

```
var puzzle = [[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]];
console.log(checkColumn(puzzle, 1));
puzzle = [[1, 2, 3, 4], [2, 3, 4, 1], [2, 3, 4, 1], [4, 1, 2, 3]];
console.log(checkColumn(puzzle, 2));
```

The following should be printed to the console:

```
true
false
```

[7 marks]

Task 5: Complete the function `colCheck` that has the argument `puzzle`. Alter the body of the function `colCheck` so that it returns `true` if all columns in `puzzle` return `true` for the function `checkColumn`, and `false` otherwise. *To get full marks, you need to call the function `checkColumn`.*

Testing: Use these lines of code to test the function:

```
var puzzle = [[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]];
console.log(colCheck(puzzle));
puzzle = [[1, 2, 3, 4], [2, 3, 4, 1], [2, 3, 4, 1], [4, 1, 2, 3]];
console.log(colCheck(puzzle));
```

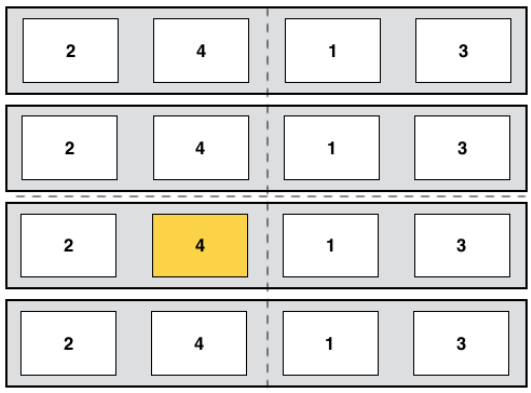
The following should be printed to the console:

```
true
false
```

[4 marks]

6 Checking the Pseudoku sub-grid conditions

The next set of conditions to check is to see if all integers from 1 to 4 appear in the 2-by-2 sub-grids of an array. We need a convenient way to refer to the sub-grids. We will use a coordinate system of (`row`, `col`) for the two-dimensional array `puzzle` as produced by `makeRows`: `row` is the row index, and `col` is the column index. Both of these indices go from 0 to 3. Consider the following two-dimensional array picture:



The coordinates of the element in yellow are (2, 1), for example. Using this coordinate system to refer to the 2-by-2 sub-grids, the top-left sub-grid will consist of the elements with co-ordinates (0, 0), (0, 1), (1, 0) and (1, 1): the top-left element is at (0, 0) and the bottom-right is at (1, 1). We can now use this to make a new array from the sub-grid elements. This is done by specifying the coordinates of the top-left element and the top-right element, given by (`row1`, `col1`) and (`row2`, `col2`) respectively.

In the JavaScript file you will see the function `makeGrid`, which takes five arguments `puzzle`, `row1`, `row2`, `col1` and `col2`. This function makes an array (called `array`), which contains the elements of the 2-by-2 sub-grid defined by the coordinates (`row1`, `col1`) and (`row2`, `col2`). The goal is to decide if all 2-by-2 sub-grids in an array *puzzle* satisfy the Pseudoku sub-grid conditions, i.e. that all integers from 1 to 4 appear in all of the sub-grids.

In the next two tasks you will do a very similar process for the sub-grids that you did for the columns.

Task 6: Complete the function `checkGrid` with arguments `puzzle`, `row1`, `row2`, `col1` and `col2`. Alter the body of the function `checkGrid` so that it returns `true` if all integers from 1 to 4 appear in the sub-grid

returned by `makeGrid(puzzle, row1, row2, col1, col2)`. *To get full marks, you need to call the function `makeGrid` and `linearSearch`.*

Testing: Use these lines of code to test the function:

```
var puzzle = [[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]];
console.log(checkGrid(puzzle, 0, 1, 2, 3));
puzzle = [[1, 2, 3, 4], [3, 4, 1, 2], [4, 1, 2, 3], [4, 1, 2, 3]];
console.log(checkGrid(puzzle, 0, 1, 0, 1));
```

The following should be printed to the console:

```
false
true
```

[6 marks]

Task 7: Complete the function `checkGrids` that has the argument `puzzle`. Alter the body of the function `checkGrids` so that it returns `true` if all four 2-by-2 sub-grids in `puzzle` return `true` for the function `checkGrid`, and `false` otherwise. *To get full marks, you need to call the function `checkGrid`.*

Testing: Use these lines of code to test the function:

```
var puzzle = [[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]];
console.log(checkGrids(puzzle));
puzzle = [[1, 2, 3, 4], [3, 4, 1, 2], [4, 1, 2, 3], [2, 3, 4, 1],];
console.log(checkGrids(puzzle));
```

The following should be printed to the console:

```
false
true
```

[4 marks]

7 Producing the final puzzle

We now have all the ingredients to generate a solved puzzle given a row array called `row`. The next task will involve generating the initial array `puzzle` from `row` using `makeRows(row)`, trying all cyclic permutations (using `permutePuzzle(puzzle, p, q, r)` for all combinations of `p`, `q` and `r`) to see if the returned array returns `true` for both `checkGrids` and `colCheck`.

Task 8: Complete the function `makeSolution` that has the argument `row`. Alter the body of the function `makeSolution` so that it returns an array which is a solved Pseudoku puzzle where the top row is equal to `row`. *To get full marks, you need to call the functions `checkGrids`, `colCheck`, `permutePuzzle`, and `makeRows`.*

Testing: Use these lines of code to test the function:

```
var row = [1, 2, 3, 4];
console.log(makeSolution(row));
```

A correct, fully solved Pseudoku puzzle (without any blank spaces) should be printed to the console.

[5 marks]

All of the methods above will just produce a solved Pseudoku puzzle. In order to produce a proper Pseudoku puzzle, numbers will need to be removed from the output of `makeSolution` and replaced with the single-character string `" "` consisting of a blank space. To complete the algorithm for generating Pseudoku puzzles, in addition to the input array `row`, we have the integer `n`, which will stipulate the number of blank entries in the final puzzle.

In the JavaScript file you will see the function `entriesToDel` with argument `n`. This function randomly chooses `n` entries of a 4-by-4 two-dimensional array and returns an array containing the co-ordinates for these entries – each co-ordinate is stored in an array `[row, col]`. Every time the function is called, it is very likely to produce a completely new set of co-ordinates. For example, if the following code is run:

```
console.log(entriesToDel(5));
```

An example of an array printed to the console could be:

```
[[2, 1], [3, 0], [3, 1], [0, 1], [1, 3]]
```

So `entriesToDel` gives us a list of co-ordinates where we will replace the numbers with `" "`. In the next task, the goal is to loop through an array produced by `entriesToDel` and set the respective element's value to be `" "`.

Task 9: Complete the function `genPuzzle` that has the arguments `row` and `n`. Alter the body of the function `genPuzzle` so that it returns an array which is a solved Pseudoku produced by `makeSolution(row)`, but with `n` elements storing the value `" "`. *To get full marks, you need to call the functions `entriesToDel`, and `makeSolution`.*

Testing: Use these lines of code to test the function:

```
var row = [1, 2, 3, 4];
console.log(genPuzzle(row, 5));
```

A correct, fully solved Pseudoku puzzle with five blank spaces should be printed to the console. Use the function `visPuzzle` to give a complete visualisation of what is returned by `genPuzzle`.

[4 marks]

8 Analysing the algorithm

In the next two tasks, we will analyse the algorithm in this assignment. Go to the file `sudoku.txt` – this is where you will write the answers to these tasks.

The algorithm to generate Pseudoku puzzles outlined here *will not produce all possibly valid Pseudoku puzzles*, and the next task concerns this.

Task 10: Give an example of a valid Pseudoku puzzle with only one blank entry that *cannot* be generated by the algorithm outlined in this assignment. Briefly explain why it cannot be generated by this algorithm.

[4 marks]

Task 11: Describe a method that can be used to generate Pseudoku puzzles that cannot be generated by the method in this assignment. You can use a flowchart, or small amounts of code, to explain how your method would deviate from the one in this assignment.

[6 marks]

9 Additional unmarked task

In the folder `sudoku` that you downloaded, you will also notice an html file called `index.html`. This will produce a very basic webpage that when a button is clicked, a Pseudoku puzzle is randomly generated. If you have fully working code that produces puzzles, then everything should work. One open-ended task is to make the webpage look good. You could introduce some CSS so that things don't look so basic. Can you think of a way of allowing someone who visits the webpage to enter the numbers into the puzzle. Can you think of a way to check their solution?