Worksheet 4: Sudoku assignment

Problem Solving for Computer Science

There 10 tasks worth 55 marks in total in this coursework assignment. Download and open the folder stored in `sudoku.zip` then change the directory in your command line to this folder. The tasks in this assignment consist mainly of completing JavaScript functions in the file called `sudoku.js`. The two last tasks consist of written work that should be completed in a text-based file, e.g. doc, docx, txt, rtf or pdf.

Just as with the previous worksheets, you can use `npm test` to test your code. In this case you can only use `npm test` to test your functions from the first five tasks. *Each of the tests for each task corresponds to a mark for that task.*

DO NOT change the names of the existing functions in `sudoku.js` and DO NOT create new functions *outside* of the template functions in `sudoku.js`. Doing so could affect how your work is graded using tests and so affect your mark.

You will submit (separately) ONLY your completed file `sudoku.js` and your written answers in a text file.

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 main goal of the Sudoku assignment is to write 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

We will implement an algorithm that generates Pseudoku puzzles. 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. In Worksheet 2 we went through two-dimensional arrays, so if you have not completed that worksheet you may have difficulties.

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

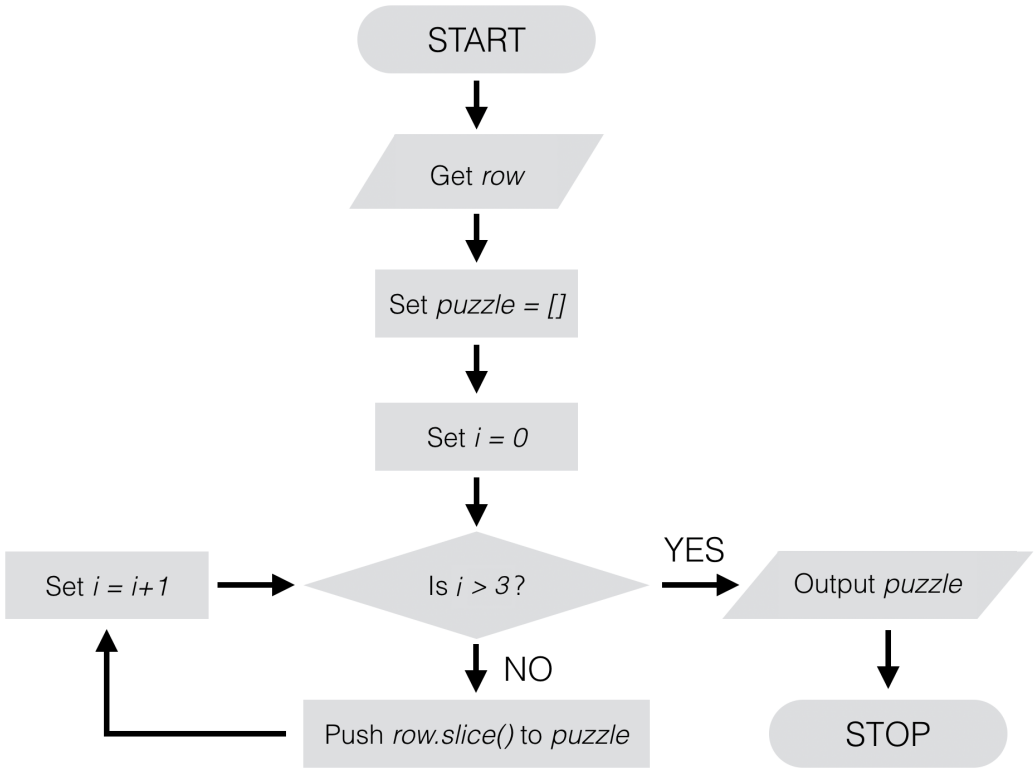
The array representing this completed puzzle will be:

```
[[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]`.

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` in *sudoku.js* that has the argument `row`. Alter the body of the function `makeRows` 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. After this use `npm test` to see if you have passed the tests to get full marks.

[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:

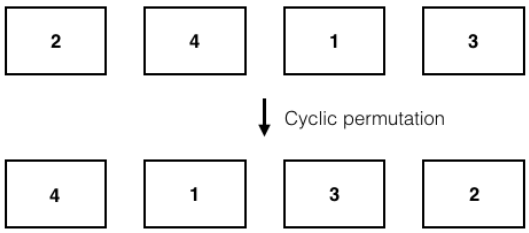
```
var row = [1, 2, 3, 4];
var puzzle = makeRows(row);
console.log(visPuzzle(puzzle));
```

The following should appear in the console:

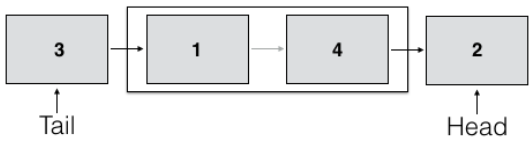
```
-----------------
| 1 | 2 | 3 | 4 |
-----------------
| 1 | 2 | 3 | 4 |
-----------------
| 1 | 2 | 3 | 4 |
-----------------
| 1 | 2 | 3 | 4 |
-----------------
```

4 Cyclic permutation of rows

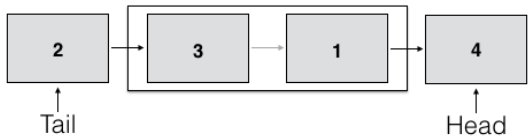
The returned array from `makeRows` will not satisfy the Pseudoku conditions since, for example, the first column will not have all numbers from 1 to 4. The algorithm for generating Pseudoku puzzles will cyclically permute the values in the bottom three rows until the Pseudoku conditions are satisfied. 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:



To cyclically permute all values one place 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 multiple times this process of enqueuing the head value and dequeuing. 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 Worksheet 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. Once completed, use `npm test` to see if you passed all seven tests for this task.

[7 marks]

The function `permuteRow`, once completed, will only cyclically permute elements of a one-dimensional array. Below the function `permuteRow` you can see the fully complete function `permutePuzzle`, which calls `permuteRow`. The function `permutePuzzle` takes a two-dimensional array puzzle array as argument and permutes row 1 by `p` places to the left, and rows 2 and 3 by `q` and `r` places to the left respectively. This function will be useful later on, but once you have completed `permuteRow`, you can try the following code to see what permutePuzzle does:

```
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]]
```

You can see here that the bottom three rows of `puzzle` have been cyclically permuted by 1, 2 and 3 elements respectively.

5 Checking the Pseudoku column conditions

The next step 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. To do this, we should:

1. Construct an array called `check` from the four values in a column
2. Check if all integers from 1 to 4 appear in `check` using the Linear Search algorithm.

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]]
```

First we create the array `check` with four elements where each element is `check[i] = puzzle[i][0]` for $0 \leq \texttt{i} \leq 3$. From the array `puzzle`, `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 search that array for all numbers from 1 to 4 using Linear Search. Then in the task after that, you will complete a function that applies this procedure for all columns in the array `puzzle`.

Task 3: 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 the function `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
```

[5 marks]

Task 4: 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:

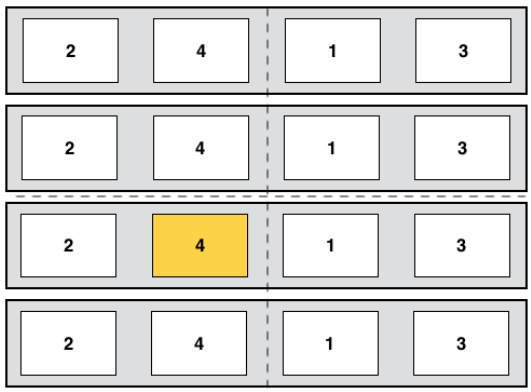
```
true
false
```

[4 marks]

Remember that you can also use `npm test` to see if you passed all of the tests for the two tasks above.

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. To refer to the sub-grids we use the convention rows and columns for two-dimensional arrays. In particular, we use pairs (`row`,`col`) where `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 bottom-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 implement a very similar process of checking sub-grids to that of checking the columns.

Task 5: 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
```

[5 marks]

Note that you can check your function for Task 5 with `npm test`, and this is the last function you can test in this way. From now on you will have to use only the testing code described in the tasks as a guide.

Task 6: 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
```

[5 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 7: 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, some numbers will need to be removed from what is returned from `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`, and the function `genPuzzle` with arguments `row` and `n`. The function `entriesToDel` 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]`. The function `genPuzzle` takes what is returned by `entriesToDel` and `makeSolution` to produce a Pseudoku puzzle.

To summarise what is happening in the two functions:

1. The function `entriesToDel` produces a list of co-ordinates that tell us where to replace the numbers in a puzzle with `" "`
2. The function `genPuzzle` will call `makeSolution(row)` to generate a solved puzzle called `solution`
3. For every element of the array returned by `entriesToDel` giving a specific row and column co-ordinate, set the corresponding element of `solution` to be `" "`

In the next two tasks, the goal is to analyse and correct the method implemented in the function `entriesToDel`.

Task 8: There is a problem with the current implementation in the function `entriesToDel`. It will not always produce `n` distinct co-ordinates to be turned blank in a puzzle. Your task is to work out why this is happening and correct the function `entriesToDel` so that it always produces an array with `n` elements consisting of two-element arrays storing the co-ordinates of an element. However, in your corrected function every element of the output should produce *distinct* co-ordinates.

Testing: To test your function you should print to the console `entriesToDel(n)` for many different values of `n` and check whether `n` distinct co-ordinates are being generated.

[5 marks]

Task 9: In this task you will begin your written work. In a text file briefly explain what the original problem is with the function `entriesToDel` [2 marks], and explain how you fixed the problem [3 marks].

[5 marks]

8 Analysing and improving the algorithm

The algorithm to generate Pseudoku puzzles outlined here is limited. For one thing it *will not produce all possibly valid Pseudoku puzzles*, among other issues. Think about why this algorithm is fundamentally limited, or could be improved. In the same text file as in Task 9 you should complete the following task.

Task 10: In the written work, you should describe and very briefly explain the limitations of the algorithm in this assignment [5 marks]. Then you should outline another algorithm that overcomes the limitations of the algorithm in this assignment [5 marks]. Maximum word count for this task: 600 words.

[10 marks]