Worksheet 7

Problem Solving for Computer Science

In this worksheet, the basic objectives are:

1. To implement a Pseudoku solving algorithm
2. To implement elements of the algorithm using recursion

Setting up

Go to Worksheet 7 resources in Week 10 in learn.gold and download `lab7.zip`. As with previous worksheets, we will only need to work with `lab7.js` inside the folder.

1. Open `lab7.js` in your chosen IDE
2. Open your command line interface and change your directory to `lab7`

We will be using exactly the same methods for testing code as in previous worksheets. That is, first printing things to the console and then running `npm test`.

Solving Pseudoku puzzles

In your Sudoku assignment, your goal was to complete code to generate Pseudoku puzzles. The goal of this worksheet to have a function which will take a Pseudoku puzzle in array form as an argument called `puzzle`, and return a solved puzzle if it can be solved. The argument `puzzle` will contain elements that have values from 1 to 4, as well as a number of elements with the space string `" "`. If `puzzle` cannot be solved then `solvePuzzle` will return the string `"There is no solution!"`.

How `solvePuzzle` works is by doing the following:

1. Make an array that is a list (of the row and column indices) of all the elements in `puzzle` that store `" "` – this will be performed by the function `findBlanks`
2. If there are `b` elements of `puzzle` that have the value `" "`, generate an array (with `makeCandidate`) of length `b` called `candidate` that contains only integers from 1 to 4 – this is a candidate solution for our Pseudoku puzzle
3. Copy `puzzle` to an array called `copy` and replace every value `" "` with a different numerical value of `candidate` made in step 2
4. The array `copy` will then be checked by the function `puzzleCheck` to see if it gives a valid solution
5. Steps 2 to 4 will be repeated until a correct solution is found

Let's look at an example of a Pseudoku puzzle and its solution to see how this will work. Consider the following Pseudoku puzzle:

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

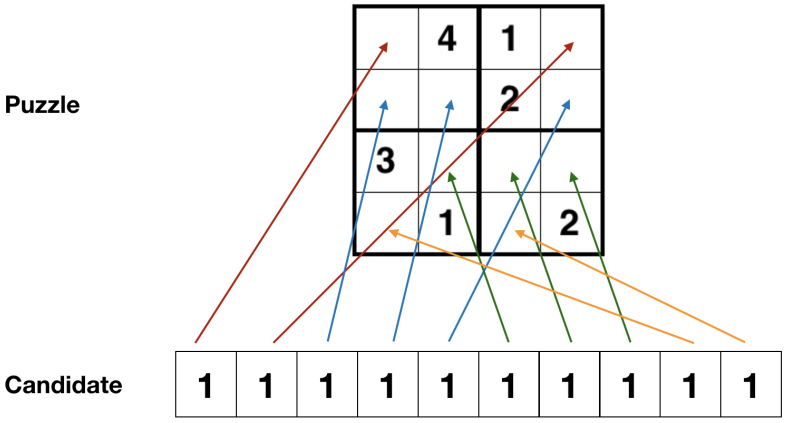
The puzzle array will look like

```
var puzzle = [[" ", 4, 1, " "], [" ", " ", 2, " "], [3, " ", " ", " "], [" ", 1, " ",
    2]];
```

There are 10 appearances of the `" "` value, so a candidate solution will be an array of length 10. Here is an example of a candidate for a solution:

```
var candidate = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
```

For this candidate solution we will assign each element of `candidate` to one of the elements of `puzzle` that has the value `" "`. So the value in `candidate[0]` could be assigned to `puzzle[0][0]`, or the value in `candidate[3]` could be assigned to `puzzle[1][0]`, or the value in `candidate[10]` could be assigned to `puzzle[3][2]`. In this way we need to assign values in `candidate` to the relevant elements of `puzzle` and then see if this is a solution to the Pseudoku puzzle. Here's a figure showing how we can methodically assign values in `candidate` to entries of `puzzle`.



Hopefully it is clear to you that this is not a good candidate solution because it will not satisfy the Pseudoku conditions! Here is another candidate solution written as an array like before

```
var candidate = [2, 3, 1, 3, 4, 2, 4, 1, 4, 3];
```

Now the values in this array, when assigned to elements of `puzzle` in the same way, will give a solution to the Pseudoku puzzle like this:

```
var puzzle = [[2, 4, 1, 3], [1, 3, 2, 4], [3, 2, 4, 1], [4, 1, 3, 2]];
```

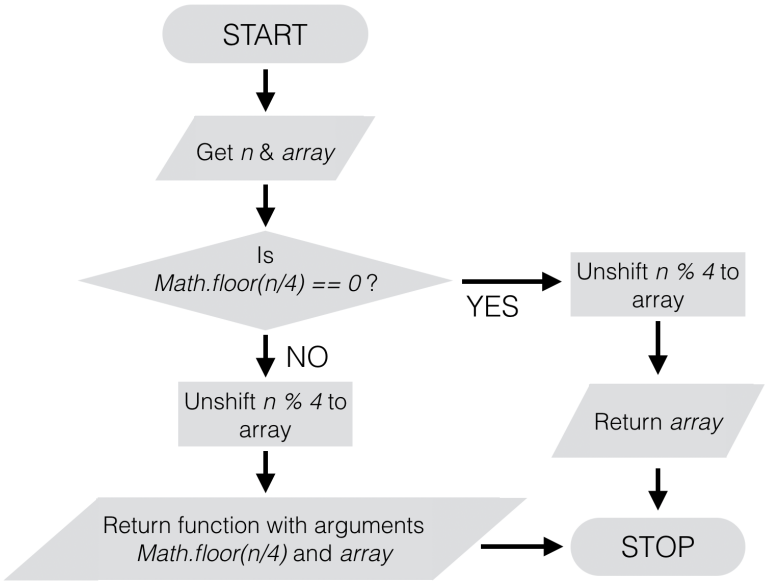
Before coding anything, think about the general problem of being given a Pseudoku puzzle array with entries having the value `" "`. Think about what information you need to generate all possible candidate solutions.

Making candidate solutions

To make a candidate solution we will use a trick of converting a number `n` in decimal into an array storing that number in base 4, and then add 1 to all entries. Then we can add additional ones to it to get the right length, if necessary. For example, the number 18 as an array in base 4 is `[1,0,2]` since 18 is equal to $4^2 + 2$ – this array will then give a candidate solution of `[2,1,3]`, since 1 has been added to all elements.

In the file `lab7.js` you will see an incomplete function called `baseFour`. When completed, this function if given an integer number `n` and an empty array as its arguments, should return an array storing `n` in base 4. Your first task is to complete this function using *recursion* to get it working correctly.

Task 1: Complete the function `baseFour(n, array)` that has the arguments `n` and `array`. Alter the body of the function so that it returns an array storing the number `n` in base 4. The code should implement the following flowchart:



The base case of `Math.floor(n/4)` being zero is already written in the function, so you need to make a recursive function call.

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

```
console.log(baseFour(16, [0,0]));
console.log(baseFour(35, []));
console.log(makeCandidate(35, 4));
```

The following should be printed to the console:

```
[1, 0, 0, 0, 0];
[2, 0, 3];
[1, 3, 1, 4];
```

Here in the first array can see that the initial array `[0, 0]` is passed to the next recursive call. Note also the third array is the second array but with 1 added to all the values, and an additional 1 out the front - this is because the desired length of the candidate is 4 (the argument `numBlanks`). This is how we will make candidates. Try `npm test` to see if your code is working correctly.

Replacing blank entries

In the next task you will complete the function `findBlanks` to implement step 1 of the solution method for Pseudoku puzzles. `findBlanks` should make an array of the coordinates of all elements that store `" "`. Again, we will do this using recursion.

Task 2: Complete the function `findBlanks(puzzle, coordinates, blanks)` in `lab7.js`. Alter the body of the function so that it returns an array storing the coordinates all blank entries in the array `puzzle`.

The base case is already implemented in the function. However, if the variable `col` is equal to three, the function should recursively call `findBlanks(puzzle, [row+1, 0], blanks)`. Otherwise, the function should also recursively call `findBlanks` but with `coordinates` being `[row, col+1]`.

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

```
var puzzle = [[" ", 4, 1, " "], [" ", " ", 2, 4], [4, " ", " ", " "], [" ", 1, " ", 2]];
console.log(findBlanks(puzzle, [0, 0], []));
puzzle = [[" ", 4, 1, " "], [" ", " ", 2, " "], [3, " ", " ", " "], [" ", 1, " ", 2]];
var blanks = findBlanks(puzzle, [0, 0], []);
var candidate = [1, 1, 1, 1, 1, 1, 1, 1, 1];
console.log(fillInBlanks(puzzle, candidate, blanks));
```

The following should be printed to the console:

```
[[0, 0], [0, 3], [1, 0], [1, 1], [2, 1], [2, 2], [2, 3], [3, 0], [3, 2]]
[[1, 4, 1, 1], [1, 1, 2, 1], [3, 1, 1, 1], [1, 1, 1, 2]]
```

Note that the argument `coordinates` is initially `[0, 0]` since the function first inspects the first element of the puzzle. Once it has done this, it passes on the next element's coordinates to the recursive call of `findBlanks`. Finally check your function with `npm test`.

Solving the puzzle

Next in `lab7.js` you will see the function `puzzleCheck`, which will compactly check that the argument `puzzle` satisfies all the Pseudoku conditions: it will return `true` if the conditions are satisfied, and `false` otherwise. In the next task you should complete the function `solvePuzzle`.

Task 3: Complete the function `solvePuzzle` with the argument `puzzle`. Alter the body of the function so that it returns a solved puzzle, or `"There is no solution!"` if there is no solution to the argument `puzzle`.

Within the loop, the function should generate a candidate using `makeCandidate`, then fill in the blanks with `fillInBlanks` (note that the argument `blanks` is already defined). Finally, the function should check if the filled in puzzle satisfies the Pseudoku conditions: if so, return the filled in version of the array. REMEMBER: the functions mentioned above will have multiple arguments.

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

```
var puzzle1 = [[1, 2, 3, 4], [3, 4, 1, 2], [4, 1, 2, 3], [2, 3, 4, " "]];
console.log(solvePuzzle(puzzle1));
var puzzle2 = [[" ", 4, 1, " "], [" ", " ", 2, 4], [4, " ", " ", " "], [" ", 1, " ",
    2]];
console.log(solvePuzzle(puzzle2));
var puzzle3 = [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [" ", " ", " ", " "]];
console.log(solvePuzzle(puzzle3));
```

The following should be printed to the console:

```
[[1, 2, 3, 4], [3, 4, 1, 2], [4, 1, 2, 3], [2, 3, 4, 1]]
[[2, 4, 1, 3 ], [1, 3, 2, 4], [4, 2, 3, 1], [3, 1, 4, 2]]
There is no solution!
```

Finally check your function with `npm test`.

Advanced Material: Backtracking

The previous method for solving Pseudoku puzzles is not that inefficient as it will generate lots of candidate solutions that are not going to be any good. For example, if we have a puzzle such as the following:

```
var puzzle = [[2, 4, 1, 3], [1, 3, 2, 4], [" ", " ", " ", " "], [3, 1, 4, 2]];
```

One of the solutions to this row will be a permutation of the numbers 1 to 4, e.g. `[4, 2, 3, 1]`. There are $4! = 24$ possible permutations, but the approach above will try $4^4 = 256$ possible combinations, including things like `[1, 1, 1, 1]`. We will now try a new technique that more methodically solves puzzles: backtracking. This is a general technique for solving hard problems.

In the Worksheet 7 resources page in Week 10, you should see a video called "Backtracking" explaining backtracking with an example. Watch the video and then look at the functions `backtrackRowsCheck`, `backtrackColsCheck` and `backtrackGridsCheck`. These functions check that the Pseudoku conditions are satisfied even if some of the elements are blanks: they check for repetitions of numbers in rows, columns and two-by-two subgrids respectively.

Below these functions, the incomplete function `backTracking` should implement the backtracking algorithm using recursion and a stack. You can see that the function should call itself within its body: after one element is assigned a number, the function is called on the rest of the puzzle. The input argument `stack` should be a stack that keeps track of the blank entries in the puzzle in case a previous element is assigned an incorrect number. In the final task in the worksheet you should complete this function by completing three recursive calls `backTracking`.

Advanced Task: Complete the function `backTracking` with the arguments `puzzle`, `coordinates` and `stack` - the argument `coordinates` is a two-element array storing the current row and column entry being considered. Alter the body of the function so that it returns a solved puzzle, or `"There is no solution!"` if there is no solution to the argument `puzzle`.

There are three incomplete recursive functional calls of `backTracking`, where the arguments need to be carefully chosen. Use the comments in the code for inspiration.

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

```
var stack = new Stack();
var puzzle = [[' ', 4, 1, ' '], [' ', ' ', 2, 4], [4, ' ', ' ', ' '], [' ', 1, ' ', 2]];
console.log(backTracking(puzzle, [0,0], stack));
```

The following should be printed to the console:

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

You can't check the function with `npm test`, but you can try many different Pseudoku puzzles as arguments to `backTracking`.