Worksheet 2

Problem Solving for Computer Science

In this worksheet, the basic objectives are:

1. To describe, manipulate and generate two-dimensional arrays
2. To complete functions that computes averages of array columns

Setting up

Will we use almost the same programming environment as in Worksheet 1, except we're going use a different folder (keep following). If you don't know how to set up the programming environment, please watch "Introduction to Worksheet 1" from Week 2. Now go to Week 3 in learn.gold and download `lab2.zip`. From the zip file you will get a folder called `lab2`. In the folder we will only need to work with `lab2.js`.

1. Open `lab2.js` in your chosen IDE, e.g. Brackets
2. Open your command line interface and change your directory to `lab2`

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

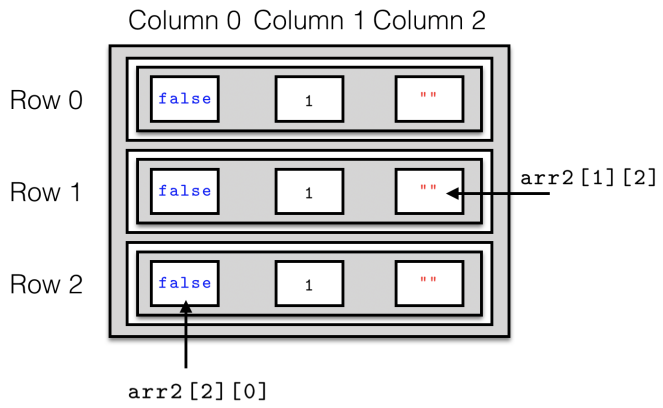
Two-dimensional arrays

We can store arrays within arrays: each element of an array can also be an array, which has its own elements. Consider the following code:

```
var arr = [false, 1, ""];
var arr2 = [arr, arr, arr];
```

Since each element of `arr2` is an array, when we access `arr2[1]` we are accessing another array of three elements. Since `arr2[1]` is an array, then we can access its elements using the square brackets, such that `arr2[1][1]` gives us the second element of the array `arr2[1]`, which is the second element of the array `arr2`. So the value `1` is being stored at `arr2[1][1]`.

A really nice way of visualising an array within an array is on a 2-dimensional grid where every row is an array. For example, the array `arr2` above can be visualised in the following way:



Instead of just indices, we can talk in terms of rows and columns. This is why it is 2-dimensional: there are two degrees of freedom where you can go from top-to-bottom along the rows, or from side-to-side along the columns.

In our example of `arr2`, we access individual values in the array using `arr2[i][j]`, where `i` is the element of the array `arr2`, and `j` is the element of the array `arr2[i]`. Now imagine that `i` represents rows of a grid, and `j` represents the columns of a grid. So `arr2[1][2]` is the value stored at row 1 and column 2, as in the figure above.

Using `slice()`

It is useful to create new JavaScript arrays when copying the data from another array. For example, we might want a second array to contain manipulated data from the first array but not lose the data from this first array. One problem with JavaScript arrays is that when we assign an array to a variable, we assign a *reference* to the array, and not the values in the array themselves. For example, if we take the code from above and amend `arr` in the following way:

```
var arr = [false, 1, ""];
var arr2 = [arr, arr, arr];
arr[0] = true;
console.log(arr2);
```

what we will get printed to the console is `[[true, 1, ""],[true, 1, ""],[true, 1, ""]]`, this is because the elements of `arr2` store references to the same array `arr`, so if you amend `arr` you amend all elements of `arr2`. To copy correctly just the values in an array we use `slice()`. So if we want to copy all the values of an array called `arr` to an array called `arr2` we should have used:

```
var arr = [false, 1, ""];
var arr2 = [arr.slice(), arr.slice(), arr.slice()];
```

What this does is copy the values into a completely new array, which is assigned to variable `arr2`. If we then alter `arr` it will not affect `arr2`. It is a good idea that if we only want to copy values and not references, we should use `slice()`, which we will use later.

The IMDb problem

The Internet Movie Database (IMDb) is a website that collects user reviews and ratings for films. Users give films a rating between 0 and 10 stars. An average of all these ratings is then calculated to give the film a rating. For our purposes the average is the sum of all the ratings divided by the number of ratings. For example, if a film got 23 ratings, and 10 of them are 5, 10 are 6, 2 are 7 and 1 is 8, so the average is

$$\frac{(10 \times 5) + (10 \times 6) + (2 \times 7) + 8}{23} = 5.74 \text{ (2 d. p.)} \tag{1}$$

In this lab, the tasks involve generating two-dimensional arrays that will store the ratings of several films by many users: each row corresponds to a single user, and every column corresponds to a single film. The array can be visualised in the following way:

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

The website IMDb will take such data and compute the average of the ratings for each movie.

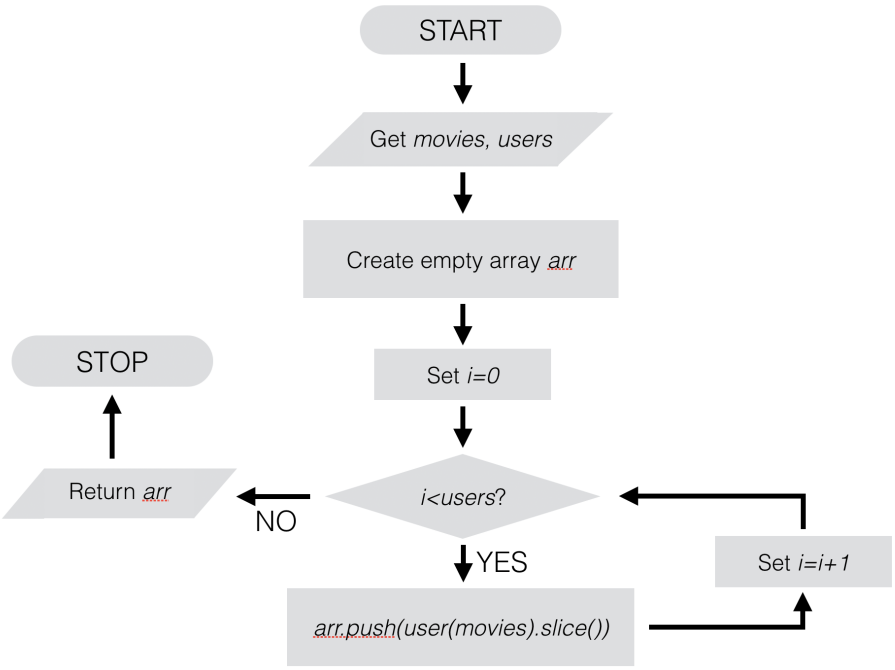
To help with visualising these two-dimensional arrays, in `lab2.js` you will find a function called `visArray`. Given a two-dimensional array as an argument, this function will give a picture that looks like the one above.

Task One: Generating IMDb Arrays

The function `user` in the file `lab2.js` takes a number called `movies` as its argument. This function will return an array with a number of elements equal to `movies` – in each element there is a randomly assigned number between 0 and 10.

In this task you need to complete the function `genUsers` which takes the arguments `users` and `movies`: these are both numbers that represent the number of rows and columns in a two-dimensional array. Once completed the function `genUsers` should return a two-dimensional array with the number of rows and columns being `users` and `movies` respectively. In each row, the array should be an array returned by `user(movies)`.

The following flowchart describes this process:



To develop your function, try the following code:

```
console.log(visArray(genUsers(5,4)))
```

This should print something that looks like the array above, but likely with different numbers.

Task Two: Computing the averages

In the next task, you will complete the function `findAverage`, which takes a two-dimensional array called `array` as its argument. This function should return an array called `averages` where every element `averages[i]` should be equal to the average of the elements in column `i` of the argument `array`.

Within the function, the code should loop over the rows to add up all elements in each column, then divide this number by the number of rows.

To develop your function, try the following code:

```
var arr1 = [1, 2, 3, 4];
var arr2 = [2, 3, 4, 5];
var array = [arr1.slice(), arr2.slice()];
console.log(findAverage(array));
```

The following should be printed in the console:

```
[1.5, 2.5, 3.5, 4.5]
```

Advanced Tasks: Incomplete user ratings

In the previous tasks we had that all elements had numbers stored in them. In this case all users had given ratings for all of the same films. Clearly this is unrealistic since not all users have seen the same films. The function `genIncompleteUsers` with arguments `users` and `movies` will generate a two-dimensional array with the number of rows and columns being `users` and `movies` respectively. The array returned by `genIncompleteUsers` will contain random numbers between 0 and 10, but also might contain the string `" "`, which is the string consisting of a space. The string `" "` signifies that a particular user has not rated a particular film. Use `visArray` for a few examples of `genIncompleteUsers` to see what is happening

Advanced Task One

In the next task, complete the function `isComplete`, which takes a two-dimensional array called `array` as its argument. This function should return `false` if any of the elements in `array` store the string `" "`, and return `true` otherwise.

You should think about looping over the rows and columns and then return `false` if any of the entries are equal to `" "`. If none of the elements store `" "`, then return `true`.

Advanced Task Two

In the final task of this lab, the hardest task, the goal is to complete a function to compute the averages of each column, like `findAverage`, but for incomplete arrays like those produced by `genIncompleteUsers`.

Complete the function `findIncompleteAverages`, which takes as its argument, a two-dimensional (possibly incomplete) array called `array`. It should return an array that is as long as the number of columns in `array`: in each element it should store the average of all the numbers in the relevant column of `array`, ignoring the empty strings. If there are no numbers in any column then the average will be the value `NaN`.

A suggestion for how to do this is the following: use a counting variable called `count` and a variable called `sum`. Now, to compute the average for a column, loop over the rows in each column and for every element that is not `" "`, add its numerical value to `sum`, and increase `count` by one. After all the rows have been inspected, the average can be computed by dividing `sum` by `count`. This will give `NaN` if both are zero.

To develop your function, try the following code:

```
var arr1 = [1, " ", 3, 4, " "];
var arr2 = [" ", 2, " ", 5, " "];
var array = [arr1.slice(), arr2.slice()];
console.log(findIncompleteAverages(array));
```

The following should be printed in the console:

```
[1, 2, 3, 4.5, NaN]
```