Worksheet 5

Problem Solving for Computer Science

In this worksheet, the basic objectives are:

1. To implement a searching algorithm that looks for multiple, unique items
2. To implement a sorting algorithm to simplify a search
3. To compare the worst-case time complexity between two solution methods

Setting up

In learn.gold, in Worksheet 5 resources in Week 7, download `lab6.zip`. As with previous worksheets, we will only need to work with `lab6.js` inside the folder.

1. Open `lab6.js` in your chosen IDE
2. Open your command line interface and change your directory to the folder `lab6`

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

The Lottery Problem from Lecture 4

At the end of Lecture 4, we had the problem of being given lottery tickets with a member's number and their birthday on, and then we had to generate a list of members that did not share their birthday. There were two solution methods given at the beginning of Lecture 5: the first one involved adapting the Linear Search Algorithm, and the second method first sorted the list of members' tickets and then compared neighbours. In this lab we will implement both solution methods.

Generating Lists of Members

In the file `lab6.js` you will see two functions at the top: `genDay` and `genBirthdays`. The first function will generate a random birthday, and the function `genBirthdays(n)`, when given an argument `n`, will return an array of length `n`, where each element might look something like this: `["003", 25, 12]`. The first element is the membership number of a member, followed by the day of their birthday and then the month: in the example above, the membership number is `"003"`, and their birthday is 25th December. Try the following line of code (and run `node lab6` in the terminal) to see an example of such an array:

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

You should see printed to the console an array with five elements, and each element stores an array with three elements. Here is an example of such an array:

```
[ [ "0", 22, 8 ],
  [ "1", 11, 4 ],
  [ "2", 16, 10 ],
  [ "3", 22, 8 ],
  [ "4", 16, 10 ] ]
```

The goal of this lab is generate an array of all the membership numbers for the unique birthdays in an array generated by genBirthdays.

For the example above, the array of members with unique birthdays is `[ "1" ]`. We will go through two solution methods to achieve this.

First Solution Method

The first algorithm to solve this problem will use the Linear Search algorithm, which is implemented by the function `linearSearch` in the JavaScript file. The function `linearSearch` takes two arrays as arguments: the first is the array being searched, the second array stores a member's ID and their birthday. This implementation of Linear Search searches the larger array for a birthday.

We can now describe the first solution method. Given an array called `birthdays` representing the list of members, the steps involved are the following:

1. Create empty array called `uniqueBirthdays`, which will store all unique birthdays
2. For each element `birthdays[i]` in `birthdays`, create an array called `restOfArray`, which stores all values stored in `birthdays` *other than* `birthdays[i]`
3. If `linearSearch(restOfArray, birthdays[i])` returns `true` then the birthday is not unique and we do nothing
4. If `linearSearch(restOfArray, birthdays[i])` returns `false` then push the membership number in `birthdays[i]` to the array `uniqueBirthdays`
5. After checking all elements `birthdays[i]`, return `uniqueBirthdays`

Now go to function `find(birthdays)`. Here you will see that steps 1, 2 and 5 are already implemented in the function. You will need to complete the function to implement steps 3 and 4.

Task 1: Complete the function `find(birthdays)` that has the argument `birthdays`. Alter the body of the function so that it returns all the membership numbers that have a unique birthday.

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

```
var birthdays = [ [ "0", 22, 8 ],
  [ "1", 11, 4 ],
  [ "2", 16, 10 ],
  [ "3", 22, 8 ],
  [ "4", 16, 10 ] ];
console.log(find(birthdays));
```

The array `["1"]` should be printed to the console. You can also try various arrays of members using `genBirthdays`. Run `npm test` to see if your code passes all the tests.

Sorting Arrays

The second algorithm involves first sorting the array `birthdays` so that members born in January go at the front of the array and members born in December go at the end. For example, given the array

```
[ [ "0", 22, 8 ],
  [ "1", 11, 4 ],
  [ "2", 22, 8 ],
  [ "3", 16, 10 ],
  [ "4", 20, 4 ] ]
```

After sorting according to the birthdays we should get:

```
[ [ "1", 11, 4 ],
  [ "4", 20, 4 ],
  [ "0", 22, 8 ],
  [ "2", 22, 8 ],
  [ "3", 16, 10 ] ]
```

If you go to `lab6.js` you will see a function `swap` that swaps the values stored at elements in an array. Then you will see the function `sortDays`, which will sort the array according to the day using Insertion Sort. Try the following lines of code:

```
var birthdays = [ [ "0", 22, 8 ],
  [ "1", 11, 8 ],
  [ "2", 16, 10 ],
  [ "3", 22, 11 ],
  [ "4", 8, 10 ] ];
console.log(sortMonth(birthdays));
```

The following array should be printed to the console when you run `node lab6`

```
[   [ "4", 8, 10 ],
  [ "1", 11, 8 ],
  [ "2", 16, 10 ],
  [ "0", 22, 8 ],
  [ "3", 22, 11 ] ]
```

The days in the birthdays are in order, but the months are not. The next function `sortMonth`, when completed, should resolve this issue. It should implement a version of Insertion Sort or Bubble Sort, whichever you wish.

Task 2: Complete the function `sortMonth` that has the argument `birthdays`. Implement a version of Insertion Sort or Bubble Sort to sort the elements of `birthdays` according to the month value in the elements.

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

```
var birthdays = [ [ "0", 22, 8 ], [ "1", 20, 4 ], [ "2", 11, 4 ], [ "3", 16, 8 ] ];
console.log(sortMonth(sortDays(birthdays)));
```

The following array should be printed to the terminal:

```
[ [ "2", 11, 4 ],
  [ "1", 20, 4 ],
  [ "3", 16, 8 ],
  [ "0", 22, 8 ] ]
```

You can also try various arrays of members using `genBirthdays`. Run `npm test` to see if your code passes all the tests.

Second Solution Method

The second solution method to the problem of producing an array of members with unique birthdays is:

1. First sort the input array `birthdays` so that the birthdays are in chronological order
2. Create empty array called `uniqueBirthdays`, which will store all unique birthdays
3. For each element `birthdays[i]`, compare the immediate neighbouring elements in the array `birthdays`, so that if `birthdays[i]` has the same birthday value as at least one of the neighbours, do nothing
4. If an element does not share its birthday with any of its neighbours, push the membership number to the array `uniqueBirthdays`
5. After checking all elements in the input array, return `uniqueBirthdays`

This method should require only one loop after the sorting step at the beginning. Go back to `lab6.js` and you will see the function `findSorted`. In the next task your goal is to complete this function to implement this second algorithm.

Task 3: Complete the function `findSorted` that has the argument `birthdays`. The function should return an array of strings, where each string is a membership number for a unique birthday. Alter the body of the function so that it implements the second solution method above. Make sure you call the sorting functions `sortDays` and `sortMonth`, and only use one loop in total for steps 3 and 4. Use the existing code to help you.

Testing: Use these lines of code to test the function first (it's the same as for the first task:

```
var birthdays = [ [ "0", 22, 8 ],
  [ "1", 11, 4 ],
  [ "2", 16, 10 ],
  [ "3", 22, 8 ],
  [ "4", 16, 10 ] ];
console.log(find(birthdays));
```

The array `["1"]` should be printed to the console. You can also use the following code:

```
var birthdays = genBirthdays(15);
console.log(findSorted(birthdays));
```

This will randomly generate a random array of members and birthdays, and by inspection check that `findSorted` works. Use `npm test` for a final test.

Analysing the Algorithms

Which solution method is the best? To analyse this, we can try to find the worst-case time complexity of both of these algorithms. In the file `lab6.js` you will see two variables `answer1` and `answer2`. In the comments there are a couple of questions asking about the worst-case time complexity of the two algorithms. Think about how many iterations there are in each algorithm and proceed to the final task.

Task 4: Look at Question 1 and Question 2 at the bottom of `lab6.js`. To answer these questions uncomment the relevant value of `answer1` and `answer2`. Test your answers with `npm test`.