Worksheet 6: Primes assignment

Problem Solving for Computer Science

The 4 tasks in this assignment make up the Primes coursework assignment. Download and open the folder stored in `primes.zip` then change the directory in your CLI to this folder. The tasks in this assignment consist of completing JavaScript functions in the file called `primes.js`.

When you submit your work *ONLY SUBMIT `primes.js` and nothing else*.

The first three tasks can be tested with `npm test`, which will give you a mark out of 11 for the first three tasks. You can only test the fourth function through the examples given in the task.

There are 15 marks available in total for this assignment

IF YOU GET CAUGHT IN AN INFINITE LOOP WHEN RUNNING YOUR CODE PRESS CTRL+C

1 Background

If you do not care about the motivation for generating random prime numbers, skip this section and go to Section 2

When you make a purchase online with your credit or debit card, you are asked to communicate your card details to someone through a website. To secure this information so that no malicious third party can get your details, the data will be encrypted using an encryption scheme called the Rivest-Shamir-Adleman (RSA) cryptosystem. It is used for e-mail, credit card transactions and any time you need to exchange sensitive data over the internet.

A prime number is an integer (greater than 1) that can be perfectly divided (i.e. leaving no remainders) by only itself and 1. The first few prime numbers are 2, 3, 5, 7, 11 and so on [1]. There are infinitely many primes and they appear everywhere in mathematics as the basic building blocks of numbers. Not only this, they are *very useful* in computer science.

The RSA cryptosystem uses the product of two random prime numbers as the basic ingredient for encryption. It is really hard for standard computers to factor a large number into its prime factors, and thus get back the two original random prime numbers. However, it is very easy to multiple two numbers together. The idea of RSA is to use the multiplication of two numbers to encrypt information, which is easy,; but it will be hard for someone to decrypt this information since they will need to find out what the original two prime numbers are.

In this assignment the goal is to generate randomly a prime number of a particular length. That is, if we specify how many digits (in base 10) a prime number should have, then a function should produce a random prime number with that many digits.

2 Random Generation of Prime Numbers

In this assignment prime numbers of length `len` will be randomly generated in the following way:

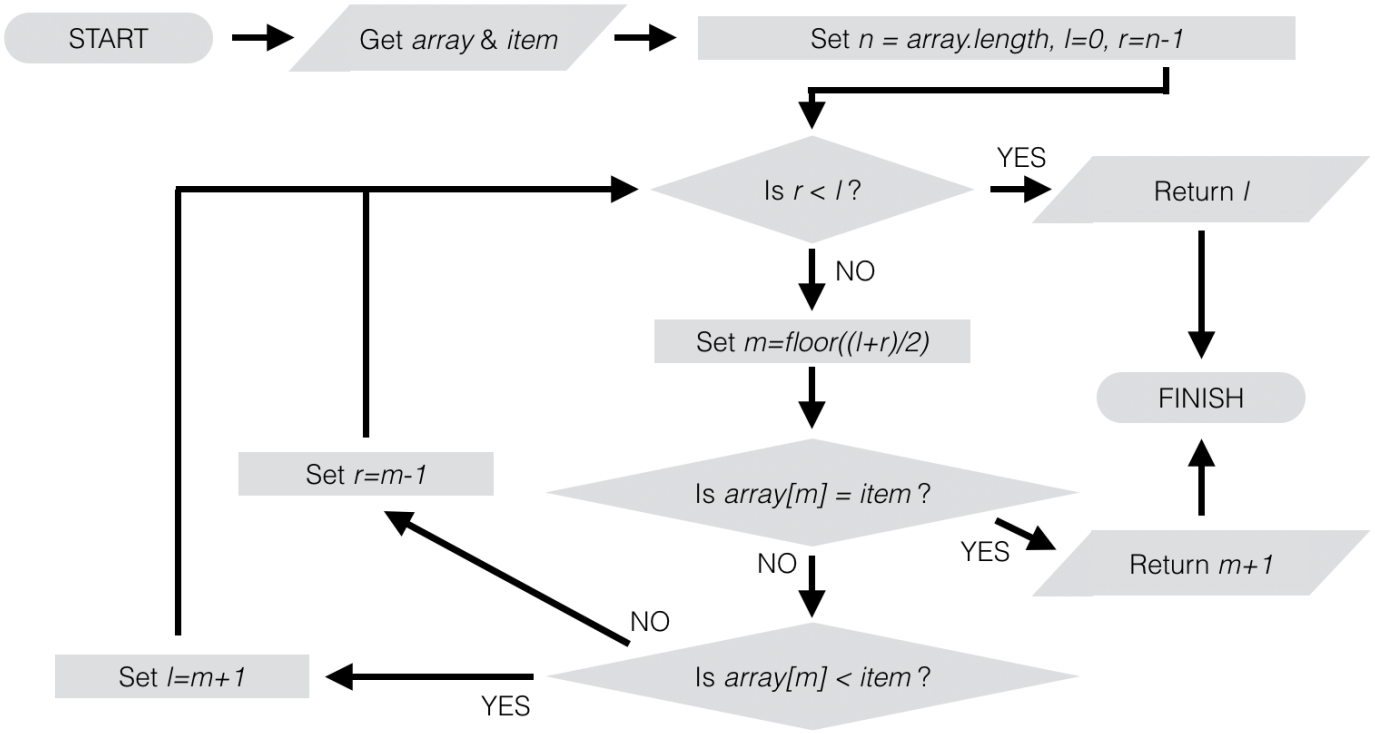
1. Get the length `len` and generate an array called `primes` storing all prime numbers up to (and possibly including) $10^{\texttt{len}} - 1$
2. Find the index `j` of the first element with a value larger than $10^{\texttt{len-1}}$ (using the Binary Search algorithm)
3. Randomly generate a number `i` between (and including) `j` and `n - 1`, where `n = primes.length`
4. Return `primes[i]`

[1] More prime numbers can be found here: https://www.mathsisfun.com/numbers/prime-numbers-to-10k.html

To implement this algorithm you will first complete a function that will be used to implement step 2, then a function to help implement step 1 and finally a function to implement steps 3 and 4.

3 Binary Search Algorithm

The first task in the assignment is to complete the function `minBinarySearch`, which will implement a form of the Binary Search Algorithm. Instead of returning the index where a value called `item` is stored, it will return the index of the first element larger than `item`. Consider the following flowchart:



This flowchart describes this slightly modified version of the Binary Search Algorithm.

Task 1: Complete the function `minBinarySearch` in *primes.js* that has the arguments `array` and `item`. Uncomment the `while` loop and its body and replace `MISSING1`, `MISSING2` and `MISSING3` so that it implements the flowchart above.

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

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

The value 2 should be printed to the console. Use `npm test` to check that your function works.

[5 marks]

4 Sieve of Eratosthenes

We will now go through the method for generating all prime numbers up to (and possibly including) a specified integer. This method uses the Sieve of Eratosthenes.

This method first creates a list of all integers from 2 to the integer `n`, then it goes through the integers `i` from 2 to the ceiling of $\sqrt{\texttt{n}}$, and looks for multiples of `i` in the list starting from 2`i`. For all multiples of `i` we mark that integer in the list as being non-prime. Once we have finished looking at all values of `i`, all unmarked integers in the list will be the prime numbers.
Let's go through an example. Write out the following list of integers:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Now let's go through the process with `i = 2`, and go through all the multiples of 2 from 4 onwards until you have marked all the multiples of two (the even numbers) up to and including 12

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~

Now we set `i = 3`, and then go through all the multiples of three and mark them if they haven't been marked already:

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~

Here we could set `i = 4`, but it is already marked along with all multiples of 4 (since they are also multiples of 2), so we will skip to the next value `i = 5`. However, 5 is larger than (the ceiling of) the square root of 12, so we can stop here. As you can see, all of the unmarked numbers in the list are the first five prime numbers.

We can turn this process into a program that has the following steps:

1. Get the integer `n` and generate an array (called `array`) with each element storing a two-element array `[i, true]` where `i` goes from 2 to `n`
2. Start a loop over integers `i` from 2 upto and including `Math.ceil(Math.sqrt(n))`
3. If `array[i-2][1]` is `false` move on to the next integer `i`
4. If `array[i-2][1]` is `true`, start a loop over integers `j` from 2 upto and including `n / i`
5. Set `array[(i*j) - 2][1]` to be `false`
6. Generate and return an array `out` of all integers `array[i][0]` where `array[i][1]` stores `false`

If you go to the function `genPrimes` in `primes.js` you will see that steps 1 and 6 are already implemented in the function. In the next task you need to implement steps 2 to 5.

Task 2: Complete the function `genPrimes` in *primes.js* that has the argument `n`. In the middle of the function body implement steps 2 to 5 above.

Testing: Use the following line of code to test the function:

```
console.log(genPrimes(12));
```

The array `[2, 3, 5, 7, 11]` should be printed to the console. Use `npm test` to check your function.

[3 marks]

5 Generating Random Primes

The two previous tasks are useful for the first two steps of the algorithm for generating random primes of a particular length. Your next task is to complete the function `randomPrime(len)` that will go through the steps of the

algorithm on the first page of this lab sheet.

Task 3: Complete the function `randomPrime` in *primes.js* that has the argument `len`. The function should go through the four steps of the algorithm for generating random primes. To get full marks the function should call both `minBinarySearch` and `genPrimes`.

Testing: Use the following line of code multiple times to test the function:

```
console.log(randomPrime(2));
```

A prime number between 11 and 99 (inclusive) should be printed to the console. Use `npm test` to check your function.

[3 marks]

6 Primes and Semiprimes

As mentioned in the first section, the RSA cryptosystem uses the product of two primes. A number that is the product of primes is called a semiprime. The next task relates to this and involves completing the function `semiPrimes(n)`.

Task 4: Complete the function `semiPrimes` in *primes.js* that has the argument `n`, where `n` can be a non-negative integer. The function should do the following:

- If `n` is a semiprime, return an array with two elements: each element should be one of the prime factors of `n`
- If `n` is prime, return `n`
- Otherwise, return `false`

Testing: Use the following lines of code:

```
console.log(semiPrimes(6));
console.log(semiPrimes(5));
console.log(semiPrimes(8));
```

The following should be printed to the console:

```
[2, 3]
5
false
```

Note that in the first line `[2, 3]` or `[3, 2]` could be returned depending on how the function works.

[4 marks]