Problem Solving for Computer Science

Lab 6: week of 4th March 2019

The work assigned in this lab sheet is not assessed but will be useful for your next piece of coursework

1 Background

1.1 Encryption

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. We tend to trust the organisation to whom we send these details, but how do you know that your details won't be intercepted by someone else? After all, the internet has made us all interconnected with each other, so it's possible to intercept all kinds of messages between various parties.

The resolution is that these messages can be intercepted, but they are processed in such a way that the intercepted messages will look like nonsense to the person doing the interception. This processing is called encryption: you start with the information you want to send, called the *plaintext*, and then it is converted into a typically larger piece of information, called the *ciphertext*. To recover the plaintext from the ciphertext, a *key* is needed. Without the key, it should be extremely difficult to recover the plaintext.

In the second coursework assignment (excluding the quizzes) you will implement a widely used encryption scheme called the Rivest-Shamir-Adleman (RSA) cryptosystem. It is used for e-mail, credit card transactions and pretty much any time you need to exchange sensitive data over the internet.

1.2 Generating Prime Numbers

A prime number is an integer number (greater than 1), which 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, as you will have covered in your Fundamentals of Computer Science course. 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 generation of prime numbers as the basis for encryption: the key in the scheme is the product of these prime numbers. You do not need to worry about the details of RSA for now. The basic idea is that it is really hard to factor a large number into its prime factors, and so we make it that any interceptor of a message needs to factor a large number into its primes to obtain the plaintext. All we need then is a method for generating prime numbers with which we can take the product.

In this lab we will go through a method for generating prime numbers

To do this we will use the method known as the **Sieve of Eratosthenes**. We will implement this method to give all primes from 2 until a given integer `n`, where `n` does not need to be prime.

This method creates a list of all integers from 2 to the integer `n`, then it goes through the integers `i` from 2 to $\sqrt{n}$ and looks for multiples of `i` in the list, i.e. `ij` where `j`≥ 2 is an integer such that `ij`$\leq$`n`. 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 with pen and paper. 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 values of `ij` from 4 onwards until you have marked all the multiples of two (the even numbers) to get all primes up to 12

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

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 will algorithmically reproduce this process and then generalise it.

1.3 Your lab folder

Create a folder called *pscs6* – you will be using this folder for the next two lab sessions. Go to learn.gold section "Lab sheets", and download the file *primes.js*. Once the file has been downloaded, make sure it is in your newly created folder called *pscs6*.

Open the file called *primes.js* and have a look at its contents. In it you will see some code at the top with an array of 11 elements, where each element stores both a number and a Boolean. The number just stores the integer between 2 and 11, and the Boolean will be used to "mark" the non-prime numbers in the array. Run the file *primes.js* through your console to see what is printed to the console. You should see that all non-prime numbers are assigned the Boolean value `false`, and all primes are assigned the value `true`.

In addition to this code at the top we have three empty function templates for three functions: `genIntegers(n)`, `genPrimes(n)`, `isPrime(m)`. Once you feel like you have understood the code in *primes.js*, move on to the tasks. Look up on MDN any functions that are not familiar to you, e.g. `Math.ceil` or `Math.sqrt`.

2 Tasks

In the next task you will write a function that generalises the piece of code at the top of *primes.js*. In particular this function will take an input parameter `n` and return an array where each element is a number-Boolean pair just like the code in *primes.js*. Just as with this code, the Boolean value should be `true` if the number in the element is a prime, and `false` otherwise.

Before attempting the task, comment out all lines where there appears `console.log`

Task 1: Write a function called `genIntegers(n)`, which takes as an input parameter an integer number `n` and returns an array where each element is of the form `[i,x]` where `i` is an integer and `x` is a Boolean [2]

Goal: To write a function that will return an array listing all integers from `2` to `n` and alongside this integer, a Boolean that dictates whether the integer is prime (`true`) or not prime (`false`)

Method: The function should be a generalisation of the piece of code in *primes.js* already, but now to the case of numbers other than 12

Test: Print the output of the function to the console for a few values of `n` – you should see the array returned by the function in all its glory

---

[2] I do not mean that every element is literally equal to `[i,x]`, just that the form of the element is a number-Boolean pair with particular values stored there.

This function will just return all integers as part of its output. The number of primes will be less than all the integers so it will be far more useful to us to just have an array which lists just the prime numbers up to (and possibly including) the number `n`; this will be your next task.

Task 2: Write a function called `genPrimes(n)`, which takes as an input parameter an integer number `n` and returns an array where each element is an integer greater or equal to 2

Goal: To write a function that will return an array listing all prime numbers from 2 to `n`

Method: The function will call `genIntegers(n)` and then use the information returned by this function to push integers to an array. For example if an element of the array returned by `genIntegers(n)` contains `true` then push the number stored in that element to another array, otherwise do nothing.

Test: Print the output of the function to the console for a few values of `n` – you should only see an array of primes (look up the prime numbers online to make sure this works)

Excellent! Now you should have a method for generating prime numbers. We can use this method to test whether a number `m` is indeed a prime number. First we produce the array of all prime numbers from 2 up to (and obviously including) `m`, and then we will search the array to see if `m` appears in it. But first, a question:

When searching the array of all primes up to and including `m` to see if `m` appears, is it better to start at the beginning of the array, or the end?

Discuss the answer to this question with your colleagues. Once you think you have a sensible answer, move on to the next task.

Task 3: Write a function called `isPrime(m)`, which takes as an input parameter an integer number `m` and returns a Boolean

Goal: To write a function that will return `true` if `m` is a prime number, and return `false` if `m` is not

Method: The function will call `genPrimes(m)` and then search the array produced by this function to determine if `m` is prime.

Test: Print the output of the function to the console for a few values of `m` – check that the numbers are prime or otherwise by comparing to a resource online.

After you have attempted Task 3, have a think about whether you are searching in an efficient manner. For example, do you need to search the whole array produced by `isPrime(m)`, or can you get away with just looking at fewer elements? Once you have thought about this and amended your code, have a think about the following question:

Can you think of a simpler way than the above process to test if an integer is prime?

Discuss this with your colleagues to test your understanding. You could write a new function if you wish, or you could move on to the next section.

3 Advanced: Random Generation of Prime Numbers

In RSA, the first step is to randomly generate two prime numbers. In JavaScript the function `Math.random()` will return a random number between `0` (inclusive) and `1` (exclusive). We have used this function before in code for generating random arrays for searching and sorting. If you wish, go to learn.gold and look at *sort.js* and *search.js* in Weeks 4 and 5 to refresh your memory.

In this section the goal is to randomly generate a prime number between 2 and some desired number `n`. For example, if `n = 12` then the method will randomly generate one of the following five integer numbers: 2, 3, 5, 7, 11.

There are two possible approaches for doing this: generate the array of all prime numbers between 2 and `n` and then randomly select one of these numbers – we will call this **Method 1**; the other approach is to randomly generate an integer between 2 and `n` and then check whether it is prime – we will call this **Method 2**.

We can implement Method 1 by using `genPrimes` and then randomly choosing one of its entries. For Method 2 we just randomly generate an integer and then use `isPrime` to check if it is prime. You will now implement both of these.

Advanced Task 1: Write a function called `methodOne(m)`, which takes as an input parameter an integer number `m` and returns an integer between 2 and `m`

Goal: To write a function that will return a randomly generated prime number between 2 and `m`

Method: The function will call `genPrimes(m)` and then randomly pick an element from the array and output the value stored there.

Test: Print the output of the function to the console for a few values of `m` – check that the numbers are prime.

Advanced Task 2: Write a function called `methodTwo(m)`, which takes as an input parameter an integer number `m` and returns an integer between 2 and `m`

Goal: To write a function that will return a randomly generated prime number between 2 and `m`

Method: The function will randomly generate an integer between 2 and `m`, then check if it is prime by calling `isPrime`: if it is prime, then the number is returned; if it is not prime, another random integer is generated and checked to be prime, and so on.

Test: Print the output of the function to the console for a few values of `m` – check that the numbers are prime.

On your own have a think about which method seems the most practical to you. Is it possible that the methods never stop and produce a prime? To answer this question you might to think about the probabilities of randomly generating a prime integer. Is it possible to improve either of the methods to get something in fewer time-steps?