Problem Solving for Computer Science

Lab 7: week of 11th March 2019

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

If you haven't yet, read through the previous week's lab script (including the Advanced section) to get some background on the generation of prime numbers. You do not need to solve the tasks in last week's lab to do this lab, but when you have time, go through the solutions. In this lab we will improve upon last week's approach to generating random prime numbers (in the Advanced section).

1 Outline and lab folder

In this lab, we will do the following:

- Implement a method for randomly generating prime numbers of a particular length
- Implement a much better method for randomly generating prime numbers of particular length
- Implement a function that converts text into numbers

In the last lab session you should have set up a folder called *pscs6* – you will use this folder for today's lab session as well. Go to learn.gold section "Lab sheets", and download the file *primegen.js*. Once the file has been downloaded, make sure it is in the folder *pscs6*.

Open the file called *primegen.js* and have a look at its contents. You will see functions which are carried over from the last lab's solutions, but with different names: `primesLab6(n)`, which implements the Sieve of Eratosthenes; `primesLength(len)`, which will output all primes number with `len` digits; `methodOne(len)`, which will randomly produce a prime number with `len` digits. You will also see an imcomplete function `minBinarySearch(array,item)`, which will completed by you.

In the file you will also see another two templates for functions `isPrime(n)`, `genPrime(len)` respectively; part of `isPrime(n)` has been filled in. The tasks in this lab will involve the completion of these functions as well.

Finally you should also see an array called `sub`, and an incomplete function called `encrypt(string)`. In the section on encryption you will complete this function.

2 Random Generation of Prime Numbers

In last week's lab we were using the Sieve of Eratosthenes to generate all prime numbers up to a certain number. *In your coursework, we will be generating prime numbers with a particular number of digits*. That is, we specify a positive number (greater than two) `len`, which will dictate the number of digits in our prime number. If we set `len` to be 2, then a randomly generated prime of this length could be 17.

To do this we are going to use a form of the `Binary Search` algorithm to search our array of primes to look for the smallest prime larger than a particular value [1]. In particular, if we want prime numbers with `len` digits, then we look for the smallest prime larger than 10^{len-1}. Once we have the *index* of the element containing this prime, we will make a smaller array, with which to generate randomly our prime number. Your task is to complete the function that will search the array of primes.

Task 1: Complete the function `minBinarySearch(array,item)`, which takes an array as input parameter and returns a number (which is an index of the array)

[1] We can use the `Binary Search` algorithm because our array of prime numbers is sorted.

Goal: To have a completed function that will implement the `Binary Search` algorithm on `array`, but it will return the index of the element that has the smallest value larger than `item`

Method: You will replace `MISSING1` and `MISSING2` with code to get the function working correctly

Test: Print `console.log(primesLab6(50));` and `console.log(minBinarySearch(primesLab6(50),20));` to the console to see if the function is correctly finding the index of the element which has the smallest value larger than `20`.

When this is all completed you now have a method for randomly generating prime numbers with a particular number of digits: this is done by calling `methodOne(len)` where `len` specifies the number of digits. Try ever increasing values of `len` until the process slows down or fails completely. You should see it slow down pretty soon as you increase `len`. In the next section we will implement a much quicker method.

3 Random Generation of Prime Numbers: an improved version

This method of generating all prime numbers up to a certain number can get pretty memory-consuming. We would like a method for randomly producing prime numbers without having to generate all prime numbers up to another number. This will then make us prefer Method Two from last week's lab script: first we randomly generate an integer number and then check if it is prime. Now the method to check whether a number, called `n`, is a prime is not going to be the Sieve of Eratosthenes (as in last week's lab), but something a bit simpler, which we will call the **Brute Force Primality Test**:

1. Get the non-negative integer number `n`
2. If the integer is less than 2, return `false`
3. Set `j=2`
4. Check if `j` is greater than or equal to `n` – if it is, then return `true`
5. Check if `n` mod j is 0 (which would imply that `n` is perfectly divisible by j) – if it is 0 then return `false`, otherwise increase `j` by 1 and go to step 4

This approach will just look at all of the integers up to (but not including) `n` and see if they perfectly divide `n`. If a number does perfectly divide `n` then it is not a prime and `false` will be returned. If a number has passed all of these checks then it is a prime number.

Note that we can actually have fewer checks of the variable `n`. Assume that `n` is not prime, then it can be written as a product of two integer numbers, `j` and `k`, i.e. `n = j*k`, where `j` is less than or equal to the square root of `n`, and `k` is greater than or equal to the square root of `n` – convince yourself of this fact, and discuss it with your colleagues if you're not sure. So, if `n` is not prime then it is perfectly divisible by any integer number `j` where $2 \leq j \leq \sqrt{n}$. In summary, we can improve the method above in step 4 of the **Brute Force Primality Test** by checking whether `j` is greater than the $\sqrt{n}$, and if it is return `true`: this will be the **Improved Brute Force Primality Test**.

Before you start on the next task, have a think on your own what the "Big O" complexity of this algorithm is in terms of the input parameter `n`. Once you have an answer, discuss it with your colleagues to check your understanding. The next task is now to complete `isPrime(n)` to give an implementation of the Improved Brute Force Primality Test.

Task 2: Complete the function `isPrime(n)`, which takes as an input parameter an integer number `n` and returns a Boolean

Goal: To write a function that will implement the Improved Brute Force Primality Test, and return `true` if `n` is a prime number, and `false` otherwise

Method: The function will use a `for` loop to iterate over integers from 2 to $\sqrt{n}$ (inclusive) to see if `n` is perfectly divisible by this integer

Test: Print the output of the function to the console for a few values of `n` being both prime and not prime – check that `true` is returned for the prime numbers only

---

Now we want to use this test to randomly generate prime numbers. As mentioned, we will follow Method Two from the previous lab script in the Advanced section. The procedure for generating a prime of a particular length is the following:

1. Get the non-negative integer number `len`
2. Generate a random integer `x` between $1 \times 10^{len-1}$ (including this number) up to but not including 1×10^{len}
3. Check if `x` is prime – if it is, then return `x`, otherwise go to step 1

You will now implement this procedure in the following task

Task 3: Complete the function `genPrime(len)`, which takes as an input parameter an integer number `len` and returns a number

Goal: To write a function that will randomly generate a prime number with `len` digits, and return this prime number

Method: The function will use the approach described above of randomly generating an integer `x` with `len` digits in it, and then check if it is prime, if it is not, it will generate a new random integer

Hint: Refer to Method Two from the lab script of Lab 6

Test: Print the output of the function to the console for a few values of `len` to check that there are the correct number of digits and the number is prime either by looking it up online, or running it through `isPrime(n)`

This method of generating random prime numbers should be pretty robust. Try and see for what values of `len` this method slows down or fails.

4 Encryption and substitution

And now for something completely different. In the RSA cryptosystem all of the encryption and decryption will be done on the level of numbers, but we will want to send words to each other, not just numbers. *So we need to take some text and convert it into a number*. We are going to do this using a simple substitution. In particular, we take each character in a word and convert it to a number and then concatenate these numbers together. To do this we need a table that will tell us which characters correspond to which numbers.

Here is such a table of characters and their corresponding numbers:

character	number	character	number
a	1	n	14
b	2	o	15
c	3	p	16
d	4	q	17
e	5	r	18
f	6	s	19
g	7	t	20
h	8	u	21
i	9	v	22
j	10	w	23
k	11	x	24
l	12	y	25
m	13	z	26
	27	?	28
!	29		

Note that the character corresponding to number 27 is the space character (i.e. the string `" "`). Now each character in a word or phrase will be converted into a number, and then these numbers will be concatenated into a longer number. For example, the text "hello!" will be converted into the number 8512121529.

If you go to *primegen.js* you will see an array that contains the information from the table in the array itself. The characters are strings stored in the elements of the array, and the index (plus one) of this character is the corresponding number in the conversion from letters to numbers. In the partially completed function `encrypt(string)`, it takes a string as an input parameter and inspects every character, then looks up `sub` to see where that character appears in `sub`. Once the relevant element of `sub` is found, it is concatenated with a string called `output`.

We need to convert the string `output` into a number and then return this number. This can be done with the built-in function `parseInt(string)` where `string` is the string you want to convert into an integer number. We now have the ingredients for the next task.

Task 4: Complete the function `encrypt(string)`, which takes as an input parameter a string `string` and returns a number

Goal: To have a function that will convert a piece of text in the form of a string `string` into an encrypted number (of the number type, not a string)

Method: The function when completed will go through every character in `string` and find the corresponding number, concatenate all of these numbers into a string, and then convert this string into a number, and then return this number

Test: Print the output of `encrypt(string)` to the console for a few words and check that the conversion was created correctly (by using the table above). Also use `typeof` to check that you are correctly returning a number and not a string.

---

5 Advanced: The Enigma machine

In the Second World War, a sophisticated machine called the Enigma Machine was used to encrypt messages among the German Navy. A great deal of effort was put into decrypting the messages generated by these machines. One the greats in the history of computer science is Alan Turing, who spent the Second World War helping to decrypt these messages (made famous in the Imitation Game). I recommend you go to Bletchley Park to find out for yourself.

The Enigma machine would start with a method of substituting one letter for another, much like how we are substituting numbers for letters in the previous section. So, in the initial configuration, the letter A might be substituted for the letter G. But one of the fiendish elements of the Enigma machine was a set of rotors that rotated every time a letter was entered into the machine. Every rotation created a new substitution of one letter for another. Before a letter was entered, A might have been substituted for G, but after a letter is entered into the machine, a rotor would rotate and now A might be substituted for the letter B. This changing of the substitution made decryption incredibly hard at the time, and special purpose machines were developed to go through the possible substitutions of letters. In addition to this, a lot of linguistic intuition and sloppiness by the German Navy were needed to make progress.

In this section I would like you to apply an Enigma-like encryption scheme to take a word and convert it to a number. Start with the conversion of characters to numbers as discussed in section 4, and after every time a character is converted to a number, cyclically permute the array `sub` by one element. Revisit your Sudoku coursework for the cyclic permutation function. So within the body of the for loop of `encrypt(string)` that inspects every character in `string` add in a cyclic permutation of `sub` by one element.

When you have finished altering `encrypt(string)` try a few words and see what is produced by the modified function. See if you can generate words where two characters correspond to the same number. Have a think about what you would need to do to decrypt these numbers and turn them back into words. Now imagine if you didn't have the original table (or array `sub`) describing the initial substitution. How would you go about solving this problem?