Coursework Assignment: Encryption

Lab 8: week of 18th March 2019

The 8 tasks in this lab are for your second coursework assignment called the Encryption assignment. You do not need to complete all 8 tasks on this sheet by today; this lab session and the next are reserved for working on this assignment. You may work on the assignment at home, but you will not obviously get help from the lab assistants outside of lab.

The deadline for your submission is **29th March 2019 at 6pm**. When submitting your work you will need to submit one JavaScript file through the Encryption assignment in the Coursework section of learn.gold: one file called *rsa.js*.

You will get partial credit for attempting these tasks so do not give up completely if things are not working

1 Your lab folder

For this submission, create a folder called *encryption*. Go to learn.gold section "Coursework", and from the folder called "Encryption" download the file *rsa.js*. Once the file has been downloaded, make sure they are in your newly created folder called *encryption*.

Open the file called *rsa.js*. Then you will see four functions that we will be using, which correspond to technical mathematical calculations:

- Function `modulo(x, y, z)`, which takes positive integers x, y and z as inputs and returns

$$x^y \bmod z$$

- Function `totient(p,q)`, which takes positive integers p and q as inputs and returns *Carmichael's totient function* for those values[1]
- Function `genExp(n)`, which takes positive integer n as input and returns what we will call the *encryption exponent* e
- Function `genInverse(a,m)`, which takes two positive integers a and m as inputs, and returns the *modular multiplicative inverse* of a mod m

All of these functions will be used at different points in this assignment: **you will not need to understand these functions**: this coursework will test your ability to call functions, even if you have not written them nor understand them.

Below these functions you will see seven function templates that will be completed by you. You will also an array called `table` and a partially completed function. You will complete all of these eight functions.

2 The RSA cryptosystem

The Rivest-Shamir-Adleman (RSA) cryptosystem is used throughout online communications so it is useful to learn *what* is going on when it is used, if not understand completely *why* it works. It will also give you experience of the following:

- Using functions as "black box" subroutines
- Performing tasks with large data, i.e. big numbers

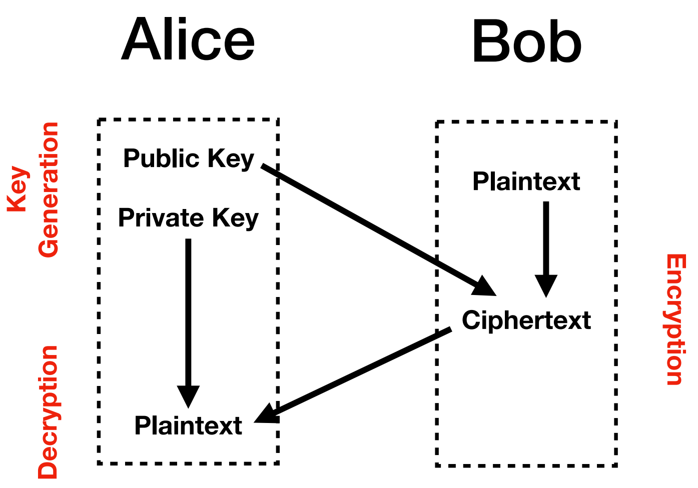
[1]This will be used in the RSA scheme but there no need to know exactly what this is

- Converting data and data types representing the same underlying information

So how does it work? We have two parties: Alice and Bob. Bob wants to send Alice some important but sensitive data in the form of a number. Bob thus needs to encrypt his data so that anyone other than Alice cannot decrypt it. Remind yourself of encryption from the last two lab sheets. To achieve this encryption, Alice sends Bob a **public key**, and Bob uses this public key to encrypt his data and then send it to Alice. To decrypt the data Alice uses a **private key** to which only she has access. So to summarise:

- Alice generates a private key and a public key
- Alice sends the public key to Bob
- Bob uses the public key to encrypt his data
- Bob sends his encrypted message to Alice
- Alice uses the private key to decrypt the message

These steps are represented pictorially below:



The idea is that messages can be decrypted **only if someone has access to both the public and the private key**. Any eavesdropper listening in on messages between Alice and Bob only has access to the public key, and would need a powerful computer to compute Alice's private key.

In the assignment, your JavaScript file will simulate the things that Alice and Bob do. First you will generate public and private keys, then use these keys to encrypt numbers and decrypt numbers.

RSA uses some deep facts about numbers, so to appreciate why it works you will need to know some number theory, but we won't worry about that. Instead of giving a lengthy introduction to RSA, we will work through all of the stages where you will complete the tasks.

2.1 Generating Private and Public Keys

The first step in RSA is for Alice to generate a public and a private key; the former will be communicated to Bob, and the latter will be kept a secret. Both the public and private key are generated from **two randomly generated prime numbers** of a particular length. Therefore, we need a method for doing this, and this will be your first task.

Task 1: Write function `genPairPrimes`, which takes as input parameter a non-negative number `len`, and returns an array of two elements where each element is a number with `len` digits

Goal: To write a function that will return two randomly generated prime numbers in an array, where each prime number has `len` digits

Method: The function `genPairPrimes` should use another function to randomly generate an integer number with `len` digits and then check if this integer is prime (again possibly using another function). The function `genPairPrimes` can then twice call the function that randomly generates primes.

Test: Call the function for a few small values of `len` and then check whether these are indeed prime numbers (by looking up the numbers online)

Hint: It is recommended that you look at the lab sheet for Lab 7, especially sections 2 and 3. Also, you are free to create your own functions that you can call within `genPairPrimes`.

[6 marks]

Now we have a method for randomly generating two prime numbers of a particular length, we have the basis with which to generate the public and private keys. The first step in generating both the private and public keys is to calculate *Carmichael's totient function* for these two prime numbers. We already have a function that does this: `totient(p,q)` where `p` and `q` will be our randomly generated prime numbers.

However, the implementation of `totient(p,q)` is currently flawed. You can see that it calls the function `gCD`, but this function is currently empty; this function should compute the **greatest common denominator** of two numbers. Your next task is to complete this function.

Task 2: Write function `gCD`, which takes as input parameters two non-negative numbers `a` and `b`, and returns a number

Goal: To write a function that will return the greatest common denominator of two numbers `a` and `b`, which is the largest number that *perfectly divides* `a` and `b`

Method: The function `gCD` should use a *non-recursive* implementation of the Euclidean algorithm to find the great common divisor: a recursive implementation might not be able to handle large numbers.

Test: Call the function for a few small values of `a` and `b` to see if it producing the greatest common divisor

Hint: It is highly recommended that you look through your lecture notes from the first two lectures

[3 marks]

Once this second task is completed we can now implement `totient(p,q)` correctly. We can now move on to generating the public key.

2.1.1 Generating the Public Key

In RSA, the public consists of two numbers: the product of two randomly generated prime numbers, and what we call the *encryption exponent*, which is a random number between 1 and Carmichael's totient function on the two primes. The function `genExp` performs this task of generating the encryption exponent if we plug in the output of `totient`. Your next task is to write a function that will generate the public key.

Task 3: Write function genPublicKey, which takes as input parameters two non-negative numbers prime1 and prime2, and returns an array of two elements where each element stores a number

Goal: To write a function that will return the public key in the RSA scheme given two numbers: the public key will be an array where the first element stores the product of prime1 and prime2, and the second element stores the encryption exponent

Method: The function genPublicKey should call both genExp and totient. The input parameter to genExp should be what is returned by totient(prime1, prime2). Thus, the function genPublicKey should return a two-element array where the first element is (prime1 * prime2), and the second element is what's returned by genExp(totient(prime1, prime2)).

Apply: Call the function using the following code to generate a public key:

```
var primes = genPairPrimes(3);
var publicKey = genPublicKey(primes[0], primes[1]);
console.log(publicKey);
```

This will print the public key to the console.

[3 marks]

2.1.2 Generating the Private Key

Now we have generated the public key, we can now generate the private key in the next task. The private key also consists of two numbers: the first is the product of the two prime numbers used to generate the public key, and the second is the *modular multiplicative inverse* of the encryption exponent modulo Carmichael's totient of these two primes. Again, you do not need to worry about what these mean mathematically, you just need to know that the *modular multiplicative inverse* is computed by the function genInverse. In the next task you will use these functions to generate the private key.

Task 4: Write function genPrivateKey, which takes as input parameters three non-negative numbers product, exponent and totient, and returns an array of two elements where each element stores a number

Goal: To write a function that will return the private key in the RSA scheme given product, the encryption exponent and Carmichael's totient of the two primes: the private key will be an array where the first element stores product, and the second element stores modular multiplicative inverse of the encryption exponent modulo the totient.

Method: The function genPrivateKey should call genInverse. The array returned by the function should have two elements: the first element will just be equal to product and the second element will be what is returned by genInverse(exponent, totient).

Apply: Call the function using the following code to generate a private key (note the overlap with the code from task 3 - you do not need to declare primes and publicKey again):

```
var primes = genPairPrimes(3);
var publicKey = genPublicKey(primes[0], primes[1]);
var privateKey = genPrivateKey(primes[0] * primes[1], publicKey[1], totient(primes[0],
    primes[1]));
console.log(privateKey);
```

This will print the private key to the console. You might want to comment out or delete the last line when you're finished testing for all ultimate secrecy.

[3 marks]

Now you have generated the public and private keys in the declared variables `publicKey` and `privateKey`, we can be begin the encryption tasks.

2.2 Encryption

As mentioned in the previous lab sheet, it is all very good communicating numbers using RSA, but what if we want to send words? We need a means to turn those words into numbers. The next task will be to do just this.

In *rsa.js* you will see an array that is storing characters as strings, along with some empty elements. This array represents the following table:

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

In particular, if a letter corresponds to a number n then `table[number - 1]` will store that letter. This is how we will convert from letters to numbers. The reason we have skipped 10 and 20 in the table is that we will use the number 0 to separate out letters in a word. For example, the word `"hi"` will give the number `804`, since according to the table `h` will be converted to `8` and `i` will be converted to `4`, with the number `0` there to "break up" the letters. This will make it easier to convert numbers back into words. So to convert `"hi"` into a number, we could use the following code (do not write this in your file):

```
var word = "hi";
var number = "";
number = number + table[7] + 0 + table[3];
number = parseInt(number);
```

The variable `number` will now store the number `804`.

Clearly it would be rather annoying to have to look up a table every time you wanted to convert a word into a number. In the next task you will write a function to do this.

Task 5: Write function `encode`, which takes as input parameters a string called `string`, and returns a number

Goal: To write a function that will return the number corresponding to a word represented as `string`, which results from the conversion in the table above, and a zero in between every letter. So, for example for the input `"sup"` will be converted to `7013021`, which comes from `7 + 0 + 13 + 0 + 21`.

Method: The function `encode` should scan the word in `string` for letters in the array `table`, and then produce the index (plus one); this will be repeated for every letter, and then after each letter, but before the last, the number `0` should be concatenated, as pointed out in the example above. Finally, this concatenation of numbers will result in a string, and then this string should be converted to a number data type.

Apply: Call the function with the following code:

```
var word = "hi";
var message = encode(word);
console.log(message);
console.log(typeof message);
```

This will print the number encoding `"hi"` to the console, and then it should say that the type of `message` is a number. Comment out or delete the lines printing to the console for extra secrecy!

Hint: Go back to the lab sheet for lab 7, and then the section "Encryption and substitution" – consult the JavaScript file for this lab session.

[3 marks]

Now we have a means of encoding words into numbers, and the public key. We are now in a position to do Bob's encryption of his number, or plaintext, into ciphertext using the public key. In RSA, the ciphertext is another number c, which is calculated to be

$$c = p^e \bmod n, \tag{1}$$

where p is our plaintext (the number Bob wants to encrypt), e is the encryption exponent, and n is the product of the two prime numbers: n is the first part of the public key, and e is the second part of the public key. Your next task is to do this encryption.

Recall that the function `modulo(x, y, z)` will compute x^y mod z, so this will be useful in the next two tasks.

Task 6: Write function `encrypt`, which takes as input parameters a number called `number` and an array `publicKey`, and should return a number

Goal: To write a function that will return the number c from Equation (1) using the number corresponding to the plaintext (p) and the public key (which contains n and e).

Method: The function should obtain n and e from the first and second elements of the public key respectively in the input parameter `publicKey`. To calculate the expression in Equation (1), **use the function `modulo`, and not the modular arithmetic syntax in JavaScript** – the latter will be far too slow for large numbers.

Apply: Call the function with the following code (note the overlap with the last task, so you should not write the same lines of code twice):

```
var word = "hi";
var message = encode(word);
message = encrypt(message, publicKey);
console.log(message);
```

This will print the ciphertext to the console: this should be different from the message if everything worked well.

[3 marks]

Excellent! You now have a method for encrypting words into numbers using RSA. It just remains to decrypt this number back into the original word.

2.3 Decryption

You will now see a function called `decrypt`, which should take the ciphertext in the form of a number with the privateKey and produce the plaintext in number form. In RSA, the plaintext p can be obtained from the ciphertext c through calculating the following:

$$p = c^v \bmod n, \tag{2}$$

where n is the product of the two primes and v is the modular multiplicative inverse in the private key. In other words, n and v are the first and second elements of the private key. So, as you can see, the decryption is very similar to the encryption. The next task is to write the function for decryption.

Task 7: Write function `decrypt`, which takes as input parameters a number called `message` and an array `privateKey`, and should return a number

Goal: To write a function that will return the number p from Equation (2) using the number corresponding to the ciphertext (c) and the private key (which contains n and v).

Method: The function should obtain n and v from the first and second elements of the private key respectively in the input parameter `privateKey`. To calculate the expression in Equation (2), **use the function `modulo`, and not the modular arithmetic syntax in JavaScript**.

Apply: Call the function with the following code:

```
var plain = decrypt(message, privateKey);
console.log(plain)
```

This will print the plaintext to the console.

[3 marks]

You have now carried out the RSA cryptosystem. Well done! The one thing that remains is to convert the plaintext number back into a word. The function `convertToText`, when completed, will do this for us. Your final task is to complete this function.

Task 8: Complete the function `convertToText` so that it takes a number as input parameter and returns a string

Goal: To have a function that converts numbers back into the original word that was encoded in task 1.

Method: First the variable `string` is currently undefined but needs to be the input `number` as a string (with quotations). Secondly, the function is not returning anything, so the string storing the word should be returned.

Apply: Call the function with the following code:

```
var plain = convertToText(plain);
```

If everything went well this should print whatever the variable `word` was storing.

[1 mark]

We now have a method for converting a word into a number, encrypting that number, decrypting it and converting back into a word. Try this method with a few different words and lengths of prime numbers to see when it stops working. You should see that the length of the prime numbers generated needs to vary with the size of the word, also that for **words with more than two letters the code might stop working**.

In practice, for good security, prime numbers with over 600 digits are used in current implementations of RSA. This implementation here will struggle to do that, but the principles are exactly the same. So bear this is mind before you start sending each other your PINs or passwords using RSA!