The supremacy of quantum algorithms: Lesson 5 – RSA: the unbreakable code?

In this lesson, we will focus on the RSA encryption algorithm and classical methods to break it. The RSA algorithm was designed in 1977 by Rivest, Shamir and Adleman and remains one of the most successful forms of encryption to this day. If you have ever performed a financial transaction over the internet, it is likely that the security codes you used to do that will have been encrypted, and your identity verified, using the RSA system.

This lesson will look at the RSA methodology and show how, in the classical world, it is practically unbreakable.

Fermat's Little Theorem

At the heart of the RSA system is a beautiful mathematical result known as Fermat's Little Theorem:

$$\boldsymbol{x^{p-1} \equiv 1\ (mod\ p)}$$

This states that any positive integer, **x**, when raised to the power of a prime number, **p**, minus one will give a remainder of one when the result is divided by **p**. The **mod** function here stands for modulo, which means "the remainder after division by" (in this case, **p**).

For example, take x = 3 and p = 5:

$$3^{p-1} = 3^4 = 81$$

$$81 / p = 81 / 5 = 16\ remainder\ 1$$

Exercise 1

Check Fermat's Little Theorem holds for the following values of x and p:

(a) x = 2, p = 3; (b) x = 2, p = 5; (c) x = 4, p = 5; (d) x = 4, p = 7; (e) x = 5, p = 11.

Programming issue

The final example in Exercise 1 hints at a potential problem in implementing the RSA system, namely that the power function grows very quickly. Even with very small prime numbers, we will quickly get overflow errors if we simply calculate the power function in full. However, we can avoid this problem by using the fact that raising a number to a power in modulo arithmetic is the same as raising the remainder to the same power.

For example, taking x = 3 and p = 5 again, we can see that:

$$3^4 = 9^2$$

But

$$9 \equiv 4\ (mod\ 5)$$

So rather than calculating 9^2 under the modulo operator, we can just calculate 4^2 and easily see that $4^2 = 16 \equiv 1\ (mod\ 5)$ as expected.

We can therefore write a Python routine to calculate the power function in modulo arithmetic (also known as modular exponentiation), using a loop to multiply **x** by itself repeatedly but each time replacing the result with its remainder under division by **p**.

Exercise 2

Implement the following Python code (note that the **%** symbol implements the modulo function in Python):

```
def powerMod(a, b, c): # raises a to the power of b modulo c
    res = a % c
    for i in range(0,b-1):
        res = a * res % c
    return res

x=5
p=11
print(powerMod(x,p-1,p))
```

Use the code to deduce which of the following are not prime numbers (because Fermat's Little Theorem does not hold for them):

(a) 1367; (b) 1369; (c) 7527; (d) 7531; (e) 7537; (f) 15723; (g) 15727; (h) 18791.

The RSA algorithm

RSA relies on a variant of Fermat's Little Theorem that says the following:

$$\boldsymbol{x^{(p-1)(q-1)} \equiv 1\ (mod\ pq)}$$

where **p** and **q** are both primes.

For example, if x = 2, p = 3 and q = 5:

$$2^{(p-1)(q-1)} = 2^{2 \cdot 4} = 2^8 = 256$$

$$256\ /\ pq = 256\ /\ 15 = 17\ remainder\ 1$$

Multiplying both sides of the identity by **x** gives:

$$\boldsymbol{x^{(p-1)(q-1)}.x \equiv x\ (mod\ pq)}$$

$$\boldsymbol{x^{(p-1)(q-1)+1} \equiv x\ (mod\ pq)}$$

This means that x raised to the power of (p-1)(q-1) + 1 will return x (mod pq).

The RSA algorithm uses this fact to create a two-step process for first encrypting and then decrypting a message. If we can find two numbers **e** and **d** such that

$$\boldsymbol{ed \equiv 1\ (mod\ (p-1)(q-1))}$$

then

$$\boldsymbol{x^{ed} \equiv x\ (mod\ pq).}$$

We now define **e** to be our encryption key and **d** to be our decryption key and **x** to be the message we wish to encrypt.

Step 1 – encryption

We raise **x** to the power **e** to generate an encrypted message **y**.

$$\boldsymbol{x^{e} \equiv y\ (mod\ pq)}$$

The strength of the RSA system is built on the fact that knowledge of **y**, **e** and **pq** is not sufficient to retrieve **x**. There is no way to find the **e**-th root in modulo arithmetic.

This can easily be seen from Fermat's Little Theorem itself:

$$1^4 \equiv 2^4 \equiv 3^4 \equiv 4^4 \equiv 1\ (mod\ 5)$$

So if we attempted to take the fourth root of 1 (mod 5) we would not be able to tell whether the original number was 1, 2, 3, or 4.

Step 2 – decryption

Once we have encrypted our message as **y**, the only way to decrypt the message is by raising to the power of **d (mod pq)**. Using power rules, we can see that:

$$\boldsymbol{y^{d} \equiv (x^{e})^{d} \equiv x^{ed} \equiv x\ (mod\ pq)}$$

The RSA algorithm: summary of process

Sender	General public	Receiver
	What they know	
x (the message to be encrypted) **e** (the encryption key) **pq** (the modulus to be used) They do not know **p** or **q** separately.	**y** (the encrypted message) **e** (the encryption key) **pq** (the modulus to be used) They do not know **p** or **q** separately.	**y** (the encrypted message) **d** (the decryption key) **pq** (the modulus to be used) They know **p** and **q** separately, which allows them to calculate **d**.
	What they do	
Calculate encrypted message **y = x^e (mod pq)**	Knowledge of **e** and **pq** is insufficient to be able to decrypt **y**.	Calculate decrypted message **x = y^d (mod pq)**

Example

Choose p = 89, q = 151, pq = 13439. If we pick e = 7543, the corresponding decryption key is d = 7.

Choose a four-digit code to encrypt, e.g. 8571 and set up the code as follows:

```
def powerMod(a, b, c): # raises a to the power of b modulo c
    res = a % c
    for i in range(0,b-1):
        res = a * res % c
    return res

plainText=8571
pq=13439
e=7543

encryptedMessage=powerMod(plainText,e,pq)
print(encryptedMessage)
```

This should return the encrypted value: 1134.

Now we can verify the system works by using the decryption key of 7 to decrypt the message as follows:

```
def powerMod(a, b, c): # raises a to the power of b modulo c
    res = a % c
    for i in range(0,b-1):
        res = a * res % c
    return res

cipherText=1134
pq=13439
d=7

decryptedMessage=powerMod(cipherText,d,pq)
print(decryptedMessage)
```

This code now returns the decrypted message of 8571.

Exercise 3a

Use pq=13439 and e = 5077 to encrypt the following message: QUANTUM.

Change each letter into a number and encrypt each in turn (assume A = 1, B = 2, C = 3, etc.)

Exercise 3b

Use pq=13439 and d = 13 to decrypt your message.

The RSA algorithm: summary

The RSA system has two parts: a public key and a private key. The public key is comprised of **e** and **pq**. The private key is comprised of **d** and the separate numbers **p** and **q**.

The public key may be safely distributed in the knowledge that it may be used to encrypt messages, but not decrypt them. For example, a bank wishing to verify a transaction from a customer will publish its public key allowing the customer to encrypt some secret information (for example a PIN or the CVC number on a credit card). The customer can then encrypt this information using $x^e \equiv y \ (mod \ pq)$ knowing that if a hacker intercepts the encrypted message **y,** they will not be able to retrieve the secret information **x** even if they also have access to the encryption key **e**. Only the bank, who holds the private key **d**, will be able to decrypt the message, checking the secret information and therefore being able to verify that the person making the transaction is who they say they are.

Breaking the RSA algorithm using a classical computer

To break RSA, we need to know **d**, but to work out **d** requires knowledge of **(p-1)(q-1)**. Although **pq** is part of the public key, it is not immediately obvious from this what **(p-1)(q-1)** is. In order to find it we must factorise **pq**, but if **p** and **q** are two large primes, factorizing **pq** is a computationally difficult problem. The following exercise illustrates the challenge:

Exercise 4

Write a program to factorise the number 900,239,055,271,631. Time how long it takes.

Conclusion

Factoring is an example of an intractable problem in computing. We have a means of solving the problem (trial division), but as the numbers grow, the number of computations required grows too quickly for any computer to run in a reasonable time. This means it is possible to find a product of primes which no classical computer would ever be able to factorise, regardless of how much processing power it had. In this example, the two primes were 8 digits long. In a real-life implementation of RSA, the primes used would typically be 300 digits long each. Even the fastest supercomputers are unable to factorise numbers of this length in any time that could threaten the security of RSA encryption systems. For this reason, RSA remains a gold standard of internet security, on which many of the world's online banking systems continue to depend.

However, quantum computing offers a method that could reduce the complexity of the factoring problem down to a speed that could make RSA breakable in real-time. In the next lesson, we will see how **Shor's Algorithm** could threaten the security of the RSA method and how this potential threat is driving a revolution in cryptography.