The supremacy of quantum algorithms: Lesson 6 – Breaking the unbreakable? Shor's Algorithm

In this lesson, we will look out how a quantum algorithm devised by Peter Shor in 1994 can be used to crack the RSA encryption method. Although quantum computers are not yet powerful enough to apply this in a large-scale implementation, Shor's algorithm is nevertheless driving a revolution in cryptography to ensure that the encryption routines the internet depends on remain secure into the future.

Period finding

Shor's algorithm exploits a weakness in the RSA system due to the fact that modular exponentiation is periodic, i.e. if we raise our plaintext (or ciphertext) by successive powers under modulo arithmetic, eventually we will return to our original number and the pattern will repeat itself.

For example, raising 2 to successive powers modulo 5 gives:

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

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

$2^3 = 8 \equiv 3\ (mod\ 5)$

$2^4 = 16 \equiv 1\ (mod\ 5)$

$2^5 = 32 \equiv 2\ (mod\ 5)$

So the pattern goes: 2, 4, 3, 1, 2, 4, 3, 1, and continues in this cycle. The first power where we get $1\ (mod\ 5)$ tells us the **period** of the cycle, in this case 4.

Exercise 1: Finding the period of an encrypted RSA number

For this exercise we will use an RSA system with pq = 77 (p = 7, q = 11) and encryption key = 7. We will use the code for modular exponentiation from the unit "RSA: the unbreakable code":

```
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

pq=77
e=7
a=13
print(powerMod(a,e,pq))
```

If we start with a plaintext number of 13, this should return the encrypted number 62. The strategy for breaking RSA will be to start from an encrypted number (which in real-life we would be able to intercept) and work out its period.

Use the modular exponentiation function to work out what the period of 62 is $(mod\ 77)$, i.e. what is the first power of 62 that is equal to $1\ (mod\ 77)$.

TAKEN FROM QUANTUM COMPUTING IN STEM EDUCATION
A project by Science on Stage Germany

How the period helps break RSA

If we were to encrypt the number 16 using this key (e = 7), we would have a ciphertext of 58. 58 has a period of 15. The full list of powers of 58 is [58, 53, 71, 37, 67, 36, 9, 60, 15, 23, 25, 64, 16, 4, 1]. (Note: the period ends at 1 because 16 and 77 have a highest common factor of 1.)

There are two important mathematical facts concerning the period that help us to break RSA:

(1) 16 (which was the original plaintext) is a member of the set of powers of 58.

(2) 16 also has a period of 15 and running the code for 16 shows that the set of numbers generated is exactly the same as 58, only in a different order: [16, 25, 15, 9, 67, 71, 58, 4, 64, 23, 60, 36, 37, 53, 1].

So if we can work out the period of a piece of ciphertext, we will also know the period of the piece of plaintext that generated it.

Let the period be **r**. We can now find a number **d'**, such that:

$$\boldsymbol{ed' \equiv 1 \ (mod\ r)}$$

In our example, e = 7 and r = 15, so we could choose **d'** to be 13, as 7 x 13 = 91 ≡ 1 $(mod\ 15)$.

If we now raise our ciphertext to the power of **d'** it will return the original plaintext directly and we will have done this only with access to information in the public domain (**e** and **pq**), so we will have broken the code! This is because:

Ciphertext $y \equiv x^e \ (mod\ pq)$
So $y^{d'} \equiv x^{ed'} \ (mod\ pq)$
But $x^{ed'} \equiv x^{1+kr} \ (mod\ pq)$
And $x^{kr} \equiv 1 (mod\ pq)$ because any multiple power of the period will return 1
So $x^{ed'} \equiv x \ (mod\ pq)$

Using our code with the ciphertext 58 and d' = 13 we can verify that:

$$\boldsymbol{58^{13} \equiv 16 \ (mod\ 77)}$$

We have managed to find out the plaintext using a method that doesn't rely on knowing what it was originally!

Unfortunately, finding the period is not easy using classical methods. As we have seen in these examples, although modular exponentiation is periodic, the powers follow no clear pattern (unlike a periodic function such as a sine curve). This means we have no alternative but to either guess which power first returns 1 or evaluate all successive powers until we find it. For numbers of the size used in real-life implementations of RSA, this cannot be done by a classical computer in a practical timeframe.

Quantum computers to the rescue!

For a classical computer, finding the period of modular exponentiation is still an intractable problem. However, there is an ingenious method that can be employed by a quantum computer that can extract information about periodic functions very quickly by use of a mathematical method known as the Quantum Fourier Transform. Shor's algorithm works using the following steps:

(1) We encode the modular exponentiation function using quantum logic gates on a number of qubits sufficient to cover the size of the RSA key (**pq**);

(2) We use Hadamard gates on all qubits to create an equally-weighted superposition to which the function is applied (this is the same strategy that we have seen in all our quantum algorithms);

(3) We perform measurements on the outputs of this function, which collapses the input qubits to a superposition of states which are spaced out by the period of the function;

(4) We apply the Quantum Fourier Transform to these inputs and measure again to return information about the period itself.

It should be noted that step (4) will only give us a probabilistic guess at what the period is, but this information would already be sufficient, in a real-world context, to enable us to break the code in a realistic timeframe (by checking the vastly reduced number of high-probability guesses on a classical computer).

Practical example

In the following example, we will build a modular exponentiation function for the key **pq** = 15, using successive powers of the number 7. The table on the left shows the calculation for each power of 7 in turn. The table on the right shows these numbers converted into binary, showing the mapping we will attempt to implement from input qubits to output qubits:

x	7^x mod 15
1	7
2	4
3	13
4	1
5	7
6	4
7	13
8	1
9	7
10	4
11	13
12	1
13	7
14	4
15	13
16	1

Input	Output
$x_3x_2x_1x_0$	$x_7x_6x_5x_4$
0001	0111
0010	0100
0011	1101
0100	0001
0101	0111
0110	0100
0111	1101
1000	0001
1001	0111
1010	0100
1011	1101
1100	0001
1101	0111
1110	0100
1111	1101
0000	0001

We now use four qubits, $|x_0\rangle$, $|x_1\rangle$, $|x_2\rangle$, $|x_3\rangle$, to represent the input binary number with digits $x_3x_2x_1x_0$ and four qubits, $|x_4\rangle$, $|x_5\rangle$, $|x_6\rangle$, $|x_7\rangle$, to represent the output binary number with digits $x_7x_6x_5x_4$.

We then use combinations of NOT, CNOT and Toffoli (CCNOT) gates to map the inputs to the outputs.

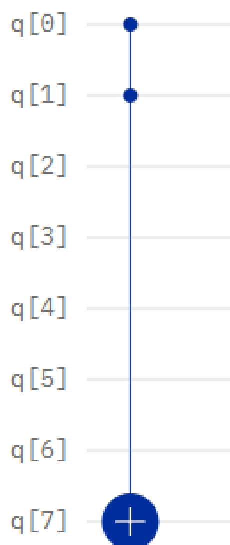
First, we note that the output is determined by the final two digits of the input in all cases. This means that qubits $|x_2\rangle$ and $|x_3\rangle$ do not need to be connected to the output qubits at all and can be ignored.

We can then construct our circuit by considering how $|x_0\rangle$ and $|x_1\rangle$ affect each of the output qubits in turn.

Let's start by looking at $|x_7\rangle$ (the largest digit of the output number) and extracting the relevant digits from the table above to see how it is affected by $|x_0\rangle$ and $|x_1\rangle$:

x_1x_0	x_7
01	0
10	0
11	1
00	0

We can see that it is only $|1\rangle$ when both $|x_0\rangle$ and $|x_1\rangle$ are $|1\rangle$, and $|0\rangle$ in all other cases (in the classical world, this would be an AND operation). This is the same as the action of a Toffoli gate (which is a CNOT gate that switches the state of a target qubit when the state of two control gates is $|1\rangle$). Using a quantum composer, this can be represented as follows:

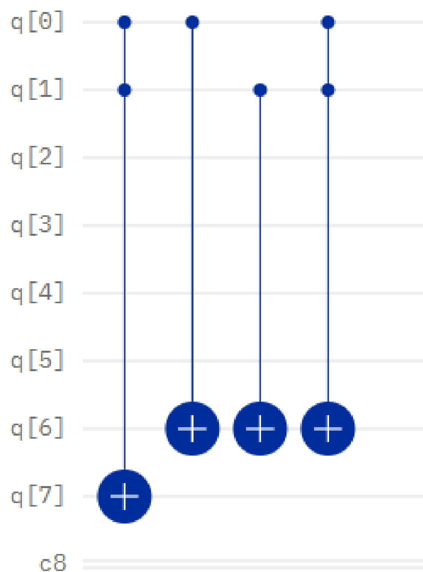


For $|x_6\rangle$:

x_1x_0	x_6
01	1
10	1
11	1
00	0

TAKEN FROM QUANTUM COMPUTING IN STEM EDUCATION
A project by Science on Stage Germany

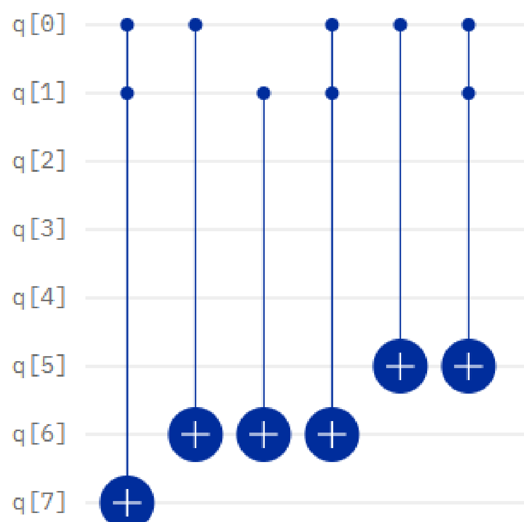
The output is $|1\rangle$ when either $|x_0\rangle$ or $|x_1\rangle$ is $|1\rangle$, or both are $|1\rangle$ (a classical OR operation). We can use a CNOT gate on each of the two input qubits to switch the output qubit to $|1\rangle$ in the case where only one of the inputs is $|1\rangle$. However, if both inputs are $|1\rangle$, the effect of the two CNOTs will be to switch the output qubit back to $|0\rangle$. To deal with this case, we need to add an additional Toffoli gate, to give the following:



For $|x_5\rangle$:

$x_1 x_0$	x_5
01	1
10	0
11	0
00	0

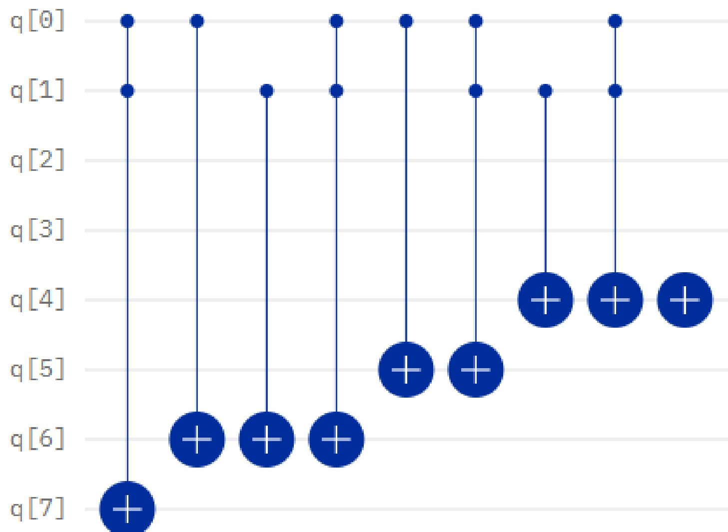
The output is $|1\rangle$ only when $|x_0\rangle$ is $|1\rangle$ and $|x_1\rangle$ is $|0\rangle$. To achieve this, we first use a CNOT gate connecting $|x_0\rangle$ to $|x_5\rangle$ and then a Toffoli gate connecting $|x_0\rangle$ and $|x_1\rangle$ to $|x_5\rangle$ to correct the output back to $|0\rangle$ in the case where both inputs are $|1\rangle$:



Finally, for $|x_4\rangle$:

$x_1 x_0$	x_4
01	1
10	0
11	1
00	1

We observe that the output $|x_4\rangle$ is $|1\rangle$ in all cases except when only $|x_1\rangle$ is $|1\rangle$. This can be created by applying a CNOT gate connecting $|x_1\rangle$ to $|x_4\rangle$ and a Toffoli gate connecting $|x_0\rangle$ and $|x_1\rangle$ to $|x_4\rangle$, followed a single-qubit NOT gate on $|x_4\rangle$:



This circuit will now replicate the action of $7^x \pmod{15}$ for all input values of x. We will then be able to apply further quantum functions to extract information about the period of the function.

Exercise 2: Build a circuit for another modular exponential function

Following the example provided above, build the circuit for $8^x \pmod{15}$. Start by completing the following tables and use the mappings of the combinations of bits to derive the circuitry of the qubits (noting that $|x_0\rangle$ and $|x_1\rangle$ are again the only two qubits to determine the output):

x	$8^x \ mod\ 15$
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	

Input	Output
$x_3x_2x_1x_0$	$x_7x_6x_5x_4$
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	
0000	

Breaking the code

Now we have a way of building the function using quantum logic gates, we can extract the information we need by applying four further steps:

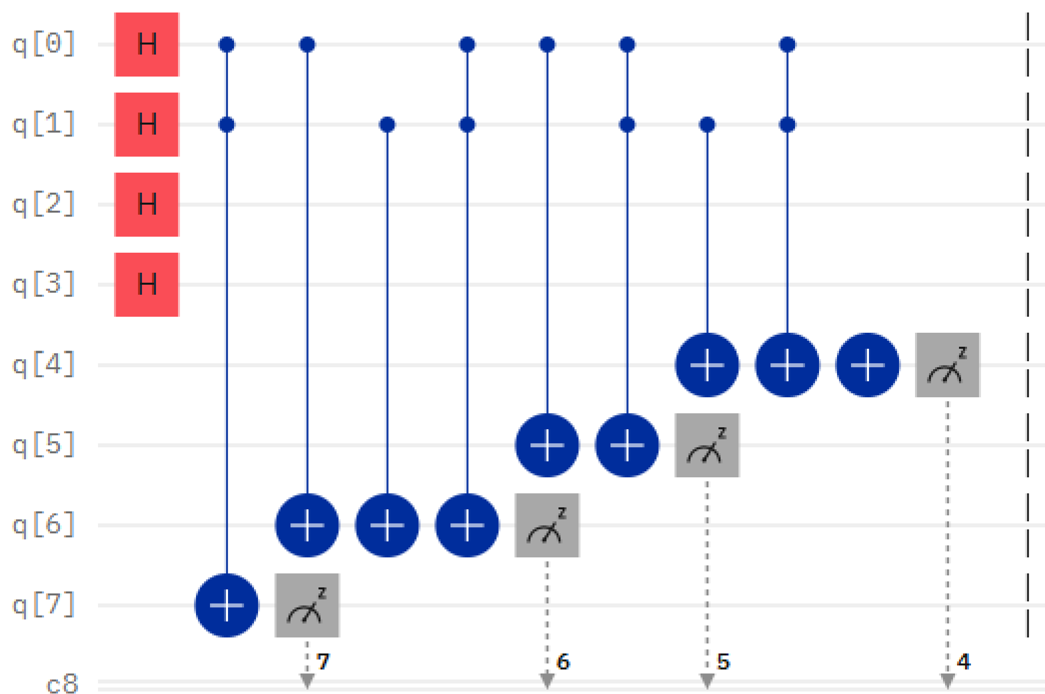
(1) We apply Hadamards to all of the input qubits. This allows us to create a superposition state of all the outputs of the modular exponentiation function we have chosen.

(2) Apply a measurement to all of the output qubits. This will now leave our input qubits in a particular superposition of states which are spaced by the period of the modular exponentiation function.

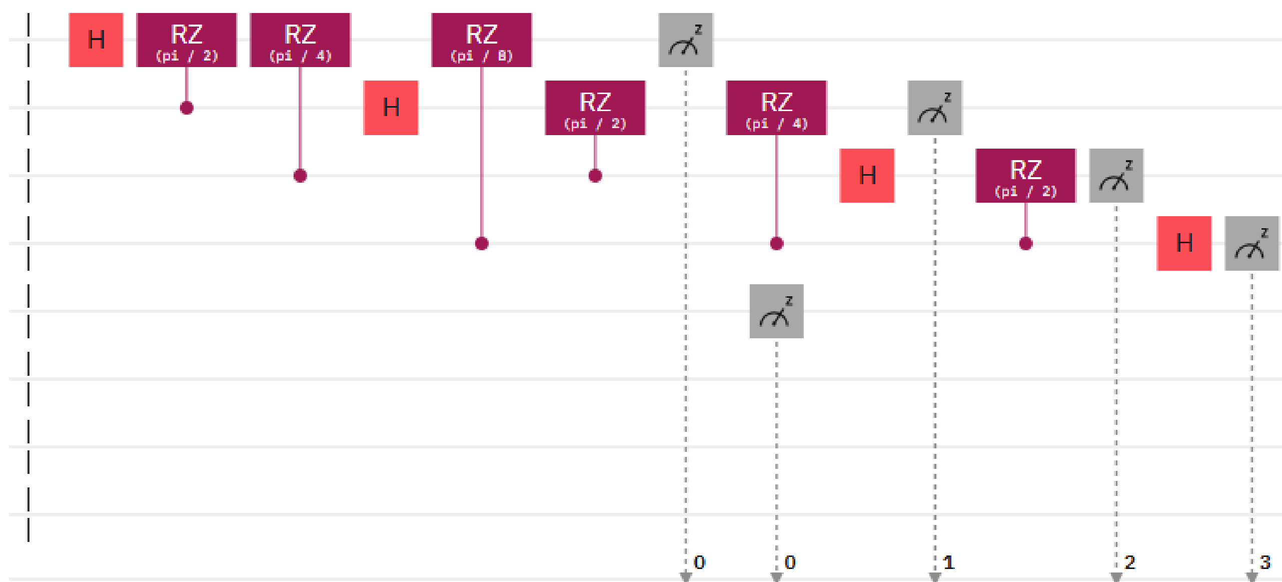
(3) Apply the Quantum Fourier Transform to generate a state whose probability density function is determined by the period of the original modular exponentiation function.

(4) Apply a measurement to the input qubits and run the algorithm multiple times, then use the outcomes with highest frequency to deduce what the period is from knowledge of the probability density function.

The full circuit which does this is shown here for the case of $7^x \pmod{15}$. For ease of viewing this is broken into two parts. First the Hadamards, modular exponentiation function and measurement of the output qubits:



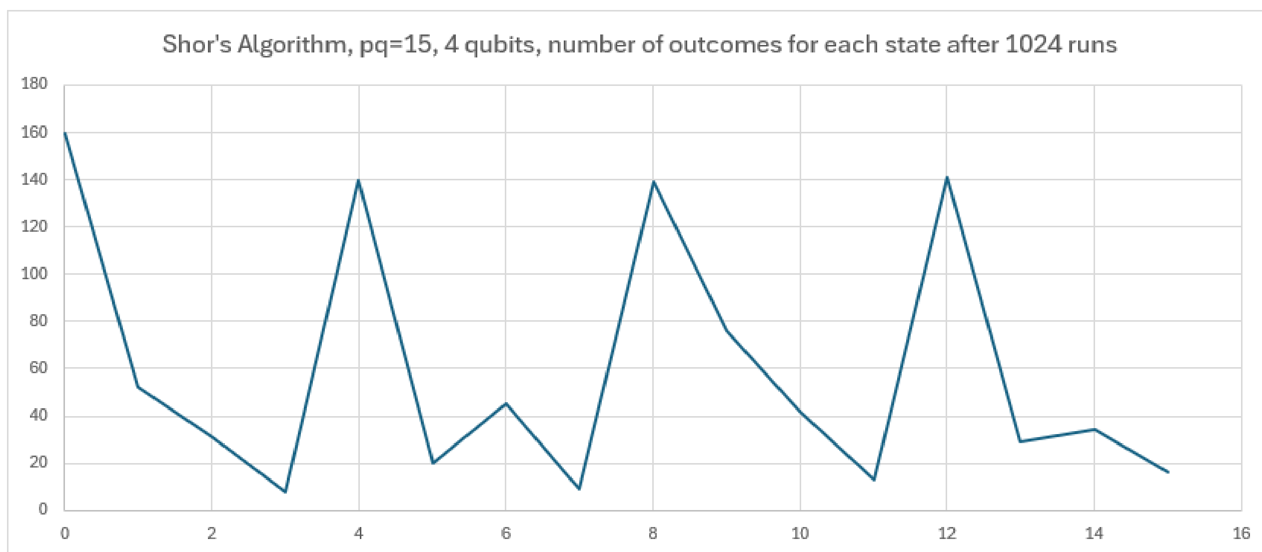
Then this is followed by the Quantum Fourier Transform and the final measurement of the input qubits:



The Quantum Fourier Transform itself is performed by a combination of Hadamard gates and conditional rotation gates as set out above.

The probability density function for the final state of the input qubits will have peaks at multiples 2^n / r, where n is the number of qubits and r is the period of the modular exponentiation function. So, by looking at the positions of the peaks from our running of the algorithm we can derive an estimate for the period r.

Running the circuit for $7^x \pmod{15}$ 1024 times yielded the following results:



We can clearly see from this experiment that the peaks occur at multiples of 4. Given that $2^n = 2^4 = 16$, we can deduce from our experiment that the period of our function is also 4. In this simple example, this was clear from the mapping exercise we did at an earlier stage. However, with larger keys than this we would be in a situation where we would not necessarily be able to observe the period of the function when we input it into our circuit. In these cases, Shor's algorithm would provide us with the only way of deducing this information in a practically-computable period of time.

Exercise 3: Verify that we have broken RSA in this case

Using the key pq = 15, plaintext = 7, and encryption key = 3:

(a) Compute the ciphertext

(b) Use the period (= 4) to find d'. (Recall d' is chosen such that $e.d' \equiv 1 \ (mod\ r)$)

(c) Check that raising the ciphertext to d' returns the original plaintext

Congratulations! You have broken RSA encryption!

Afterword

Shor's Algorithm has not yet been implemented in a way that might threaten any real-world implementation of RSA. Our example relied heavily on the fact that the modular exponentiation function was easy to construct (which was a consequence of it having a short period). In a real-world example, there would be no simple pattern for mapping the input qubits to the output qubits, and so far no-one has managed to invent a mechanism for building the circuitry for a general modular exponentiation function. However, the prospect that Shor's Algorithm might one day be able to break RSA has already been sufficient to drive the development of a whole new field of "Post-Quantum Cryptography", designed to insure against the eventuality that Shor's Algorithm does one day become practicable.