Shor’s algorythm

Project for Quantum Computing, winter 2023/24

Radosław Lech, 242245

Contents

1. Introduction

Shor's algorythm is a quantum solution to the factorization problem (used e.g. in cracking the RSA cipher), conceived by an American mathematician Peter Shor in 1994. For the sake of this project this problem can be stated as follows:

Given a number N, which is a product of two primes p and q (both of them are secret), find p and q.

Despite extensive research, no polynomial-time classical algorithms are known for general integer factorization. Classical algorithms have time complexities that are sub-exponential but still grow rapidly with the size of the input. This lack of efficient algorithms contributes to the inherent difficulty of the factorization problem.

Factoring large numbers efficiently with Shor's Algorithm requires a quantum computer with a sufficient number of qubits. The number of qubits needed scales with the size of the number to be factorized. While small-scale demonstrations of Shor's Algorithm have been achieved in laboratories, scaling up the number of qubits while maintaining low error rates is a significant technical challenge.

In the following report I provide a general introduction to general introduction to the algorythm and I discuss how it is possible to realize on a quantum computer.

2. Shor's algorythm

General formulaton of the algorythm is pretty straightforward.

1. Choose a number *a* such that *1 < a < N*
2. Compute the greatest common divisor (gcd) of a and N
3. If gcd(a,N) > 1 then gcd(a,N) = p, which means success! Unfortunately the chance of
4. If gcd(a,N) = 1 we look for period r of a^r *mod N* (a^0 mod N is always, so we are looking for the smallest *r* for which a^r *mod N* is again 1)
5. If r is odd -> go back to step 1.
6. If $a^{r}/_{2} \bmod N = N - 1$ -> go back to step 1.

Points 3-4 constitute the most challenging part. After the period r has been found, we need postprocessing to extract p and q. We know that:

$$a^r - 1 \bmod N = 0$$

$$a^r - 1 = kN; \;\; k = 1,2,3, \dots$$

$$a^r - 1 = kpq$$

$$\left(a^{\frac{r}{2}} - 1\right)\left(a^{\frac{r}{2}} + 1\right) = kpq$$

$\left(a^{\frac{r}{2}} - 1\right) = cp; \left(a^{\frac{r}{2}} + 1\right) = dq$

Each of these terms shares one nontrivial factor with N, and now we find it simply with Euclid algorythm:

$$p = gcd\left(a^{\frac{r}{2}} - 1, N\right)$$

$$q = gcd\left(a^{\frac{r}{2}} - 1, N\right)$$

The Shor's algorythm, as presented in the previous section, raises the most important question:

- How to find the period r of modular exponentiation from point 4.?

- How to perform modular exponentiation on a quantum circuit?

These are the questions I will be tackling in the next sections.

3. Period finding algorythm

The task is to construct a quantum gate (let us call it U) that will perform modular exponentiation on a state vector $|x\rangle$
Important: state $|x\rangle$ exists in superposition, so we are performing the exponentiation on all possible values of x at once. The first step is to choose the base a. In our case a = 2.

The general situation can be sketched like this:

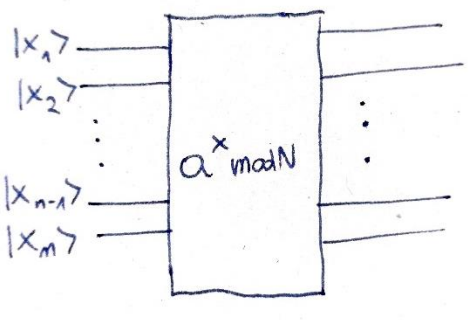


Figure 1. Modular exponentiation gate

Now, as x can be expressed in binary as:

$$x = x_1 x_2 \dots x_n$$

$$a^x = a^{x_1 x_2 \dots x_n} = a^{x_n + 2x_2 + \dots + 2^{n-1} x_1} = a^{x_n} a^{2x_{n-1}} a^{4x_{n-2}} \dots a^{2^{n-1} x_1}$$

This way, the big gate from Figure 1. Has been separated into smaller gates controlled by single qubits. Each of them is just a double application of the previous one. The gate that performs a single modular exponentiation can be called U:

$$U|v\rangle = |va \bmod N\rangle$$

$$U^2|v\rangle = |va^2 \bmod N\rangle$$

Thus, depending on the precision we want to obtain (*m* digits), we apply m gates U, U^2, U^4, ... $U^{2^{(m-1)}}$.

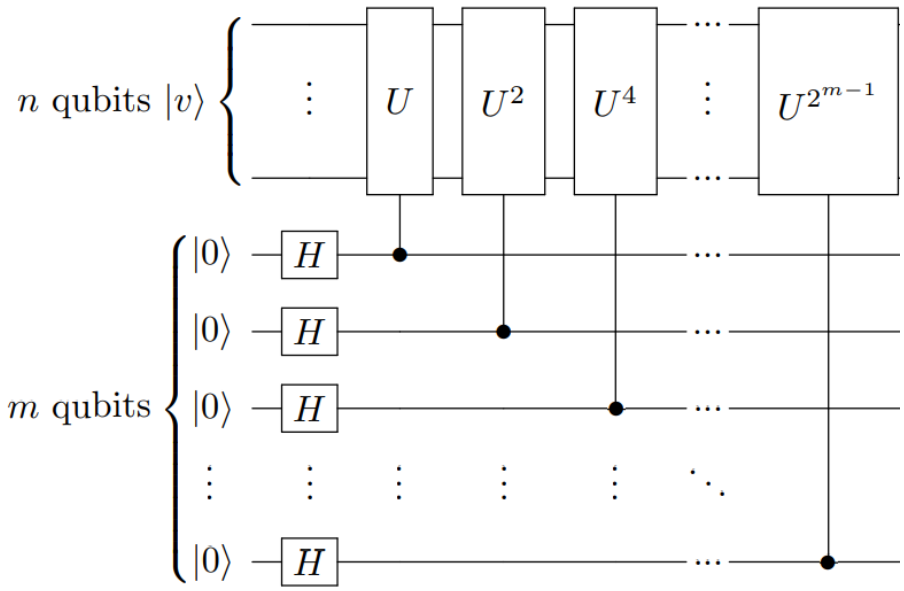


Figure 2. Modular exponentiation gate divided into m gates acting on the |v> vector, controlled by the qubits of the period register. Source: Thomas Wong - Introduction to Classical and Quantum Computing, 2022

The operation I just described creates a signal with period r, which then can be figured out using Quantum Fourier Transform (Section 5).

Now we need to construct the input state vector. This is done by assuming that we have equal superposition of all possible states, which is equal to one (for proof go to *T. Wong – Introduction to Classical and Quantum Computing*, 2022, Section 7.8.3.):

$$\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}|v_s\rangle = \frac{1}{r}r\left|a^0 \bmod N\right\rangle = |1 \bmod N\rangle.$$

The only thing we need to do is to prepare *n* qubits, where

```
n = int(np.floor(np.log2(N)))+1
```

and intitialize the least significant qubit with 1.

4. Quantum Fourier Transform

Quantum Fourier Transform is just a quantum equivalent of Fast Fourier Transform, performer on discrete samples of signal to find its frequencies.

While in FFT we compute the impulse response as:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i \frac{kj}{N}} x_j \tag{1}$$

In QFT, the input statevector is mapped:

$$|x\rangle \xrightarrow{QFT} \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i \frac{kx}{N}} |k\rangle \tag{2}$$

We will make the most use of the formula (2) when we expand it:

$$\frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i \frac{kx}{N}} |k\rangle = \frac{1}{\sqrt{N}} \Big(\big|00 \dots 0\rangle + e^{2\pi i \frac{x}{N}} \big| 00 \dots 1\rangle + e^{2\pi i \cdot 2 \frac{x}{N}} |0 \dots 10\rangle + \cdots + e^{2\pi i \cdot (N-1) \frac{x}{N}} |N-1\rangle \Big) \tag{3}$$

x can be represented in binary as $x_1 x_2 \dots x_n$ and $N = 2^n$, dividing x by N means simply shifting the dot to the left by n digits:

$$\frac{x}{N} = \frac{x_1 x_2 \dots x_n}{2^n} = 0.x_1 x_2 \dots x_n$$

At this point we can notice that expanding $\big(|0\rangle + e^{2m}|1\rangle\big)\big(|0\rangle + e^{m}|1\rangle\big)$ gives:

$$\big(|0\rangle + e^{2m}|1\rangle\big)\big(|0\rangle + e^{m}|1\rangle\big) = |00\rangle + e^{m}|01\rangle + e^{2m}|10\rangle + e^{3m}|11\rangle$$

So the eq.(3) can be contracted to the form:

$$|\tilde{x}\rangle = (1) = \frac{1}{\sqrt{N}} \big(|0\rangle + e^{2^{n-1} 2\pi i \cdot 0.x_1 x_2 \dots x_n} |1\rangle\big)\big(|0\rangle + e^{2^{n-2} 2\pi i \cdot 0.x_1 x_2 \dots x_n} |1\rangle\big) \dots \big(|0\rangle + e^{2\pi i \cdot 0.x_1 x_2 \dots x_n} |1\rangle\big) = \big(|0\rangle + e^{2\pi i \cdot 0.x_n} |1\rangle\big)\big(|0\rangle + e^{2\pi i \cdot 0.x_{n-1} x_n} |1\rangle\big) \dots \big(|0\rangle + e^{2\pi i \cdot 0.x_1 x_2 \dots x_n} |1\rangle\big) \tag{4}$$

The first bracket corresponding to the first (least significant) qubit, the second bracket to the second qubit, and so on. Now we obtained a form of the eq.(1) that is applicable to the quantum circuit. E.g.:
$\big(|0\rangle + e^{2\pi i \cdot 0.x_1 x_2 \dots x_n} |1\rangle\big)$ corresponds to the (n-th) most significant cubit, and

$$0.x_1 x_2 \dots x_n = x_1 \cdot \frac{1}{2} + x_2 \cdot \frac{1}{2^2} + \cdots x_n \cdot \frac{1}{2^n},$$

that is we need to apply a relative phase rotation:
- by π between the n-th and 1st qubit
- by π/2 between the n-th and 2nd qubit
- by π/4 between the n-th and 3rd qubit

And so forth.

Having dealt with the theory, this is what QFT looks like in a circuit:

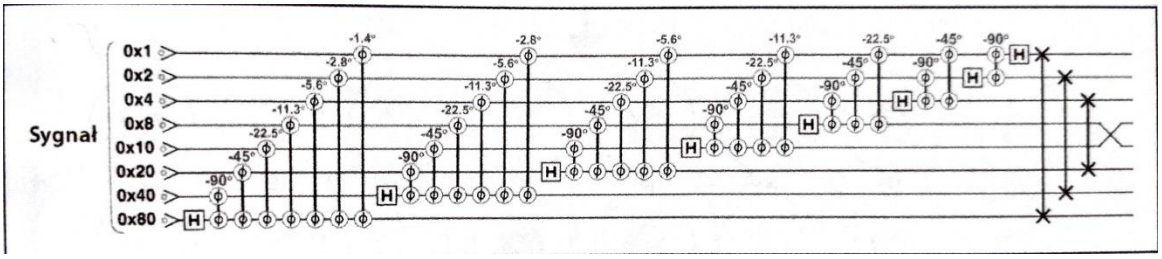


Figure 3. QFT circuit. Source: Johnston, Harrigan, Gimeno-Segovia - Programming Quantum Computers. Essential Algorithms and Code Samples

And a simple function in Python (using qiskit library) looks like this:

```
def QFT(m):
    qc = QuantumCircuit(m)
    for qubit in reversed(range(m)):
        qc.h(qubit)
        print(f'qubit = {qubit}')
        for i in range(1,qubit+1):
            qc.cp(math.pi/2**i,qubit,qubit-i)
            print(f'cphase gate on qubits {qubit},{qubit-i}')
    for i in range(1,m-1):
        qc.swap(i-1,-i)
    return qc
```

5. Solution with explanation for a =2

As stated in the previous sections, the main part of the problem is devising a modular exponentiation gate. The most pesimistic realization is that gate is hardly scalable for any arbitrary value of a.

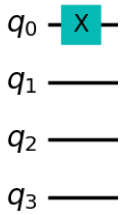
Instead of giving a general explanation, I will provide a concrete example of the gate (for predefined value of a and N) and explain why it works.

The quantum computer used for period finding needs to have two registers: one that will store the superposition of our eigenvectors (which from now on will be called *work register*), and the other one called *period register*, which will store the value of *x*.

The circuit consists of two parts. First one performs modular exponentiation of our superposition $|v_s\rangle$ from Section 1. This operation is controlled by the qubits of the precision register. U^1 gate (single multiplication by 2) is activated when 0-th qubit of the precision register is 1, U^2 gate (multiplication by 4) is activated when 1st qubit of the period reg. is active, and so on.

Second part of the circuit performs Quantum Fourier Transform on the period register to find out the period.

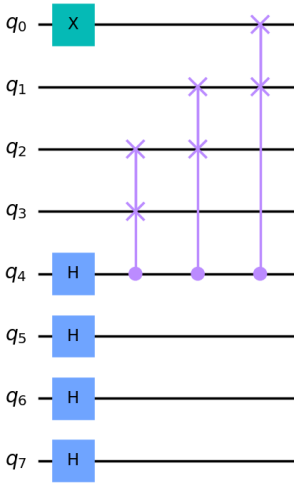
1. We initialize the zero-th qubit of the work register with one



2. We initialize the precision register with Hadamard gates, creating an equal superposition of all possible states
3. Since a=2, modular exponentiation a^x mod N means simply multiplying by 2 x times. In binary representation it is obtained simply by moving the 1 to the left:

$$0001 \times 2 = 0010$$
$$0010 \times 2 = 0100$$
Etc.

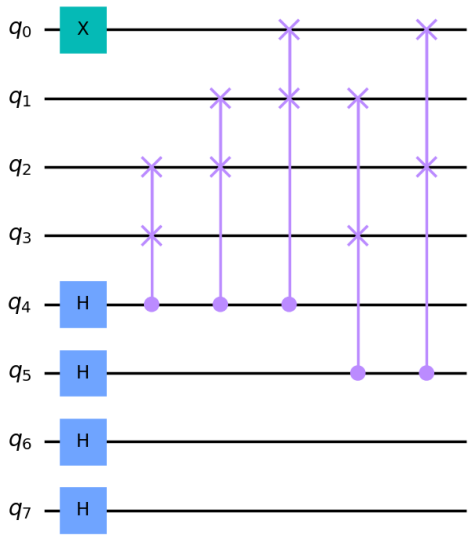
That is why this part of the code is implemented by introducing SWAP gates:



Analogically, multiplication by 4 is obtained by shifting '1' two digits to the left:

$$0001 \times 4 \bmod 15 = 0100$$
$$0100 \times 4 \bmod 15 = 0001$$
$$0010 \times 4 \bmod 15 = 1000$$
$$1000 \times 4 \bmod 15 = 0010$$

And thus, it can be implemented in the circuit as SWAP gates spanned across 2 cubits:



4. Now we need to implement the QFT on all four qubits of the period register.

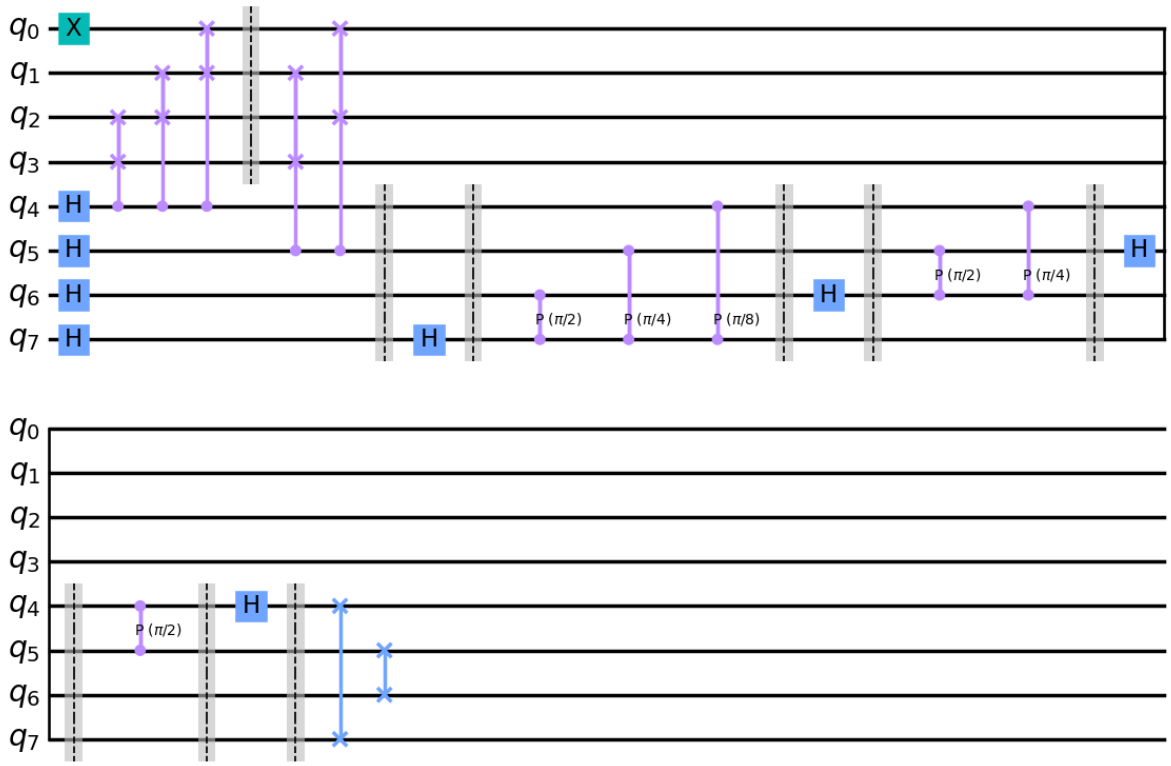
```
#INITIALIZATION WITH 1
qc.x(0)
#creating superposition
for i in range(n,qc.num_qubits):
    qc.h(i)

# U^1 gate
for i in reversed(range(1,n)):
    qc.cswap(n,i,i-1)
qc.barrier([i for i in range(n)])

# U^2 gate
for i in reversed(range(2,n)):
    qc.cswap(n+1,i,i-2)


qft = QFT(m)
qc.compose(qft,[i for i in range(n,n+m)],inplace=True)
```

This is the resultant circuit:



6. We obtained the period r, but we still need to do some classical postprocessing to extract our primes p and q.

```
M = 2**m
# Find the period r using the continued fraction expansion
potential_rs = set()
for measured_value in hist:
    measured_value_int = int(measured_value, 2)
    for denominator in range(2, M):
        fraction = Fraction(measured_value_int,
M).limit_denominator(denominator)
        if fraction.denominator not in potential_rs and abs(fraction -
Fraction(measured_value_int, M)) < 1/M:
            potential_rs.add(fraction.denominator)



factors = set()

for r in potential_rs:
    if r % 2 == 0 and a**(r//2) % N != N - 1:  # additional check to ensure
a^(r/2) is not congruent to -1 mod N
        possible_factor_1 = GCD(a**(r//2) - 1, N)
        possible_factor_2 = GCD(a**(r//2) + 1, N)
        if 1 < possible_factor_1 < N:
            factors.add(possible_factor_1)
        if 1 < possible_factor_2 < N:
            factors.add(possible_factor_2)
```

```
if factors:
    print(f"The non-trivial factors of {N} are: {factors}")
else:
    print(f"No non-trivial factors found. The period might not be correct
or 'a' might not be a good choice.")
```

The above code comes from https://www.sharetechnote.com/html/QC/QuantumComputing_Shor.html.

Result:

The non-trivial factors of 15 are: {3, 5}

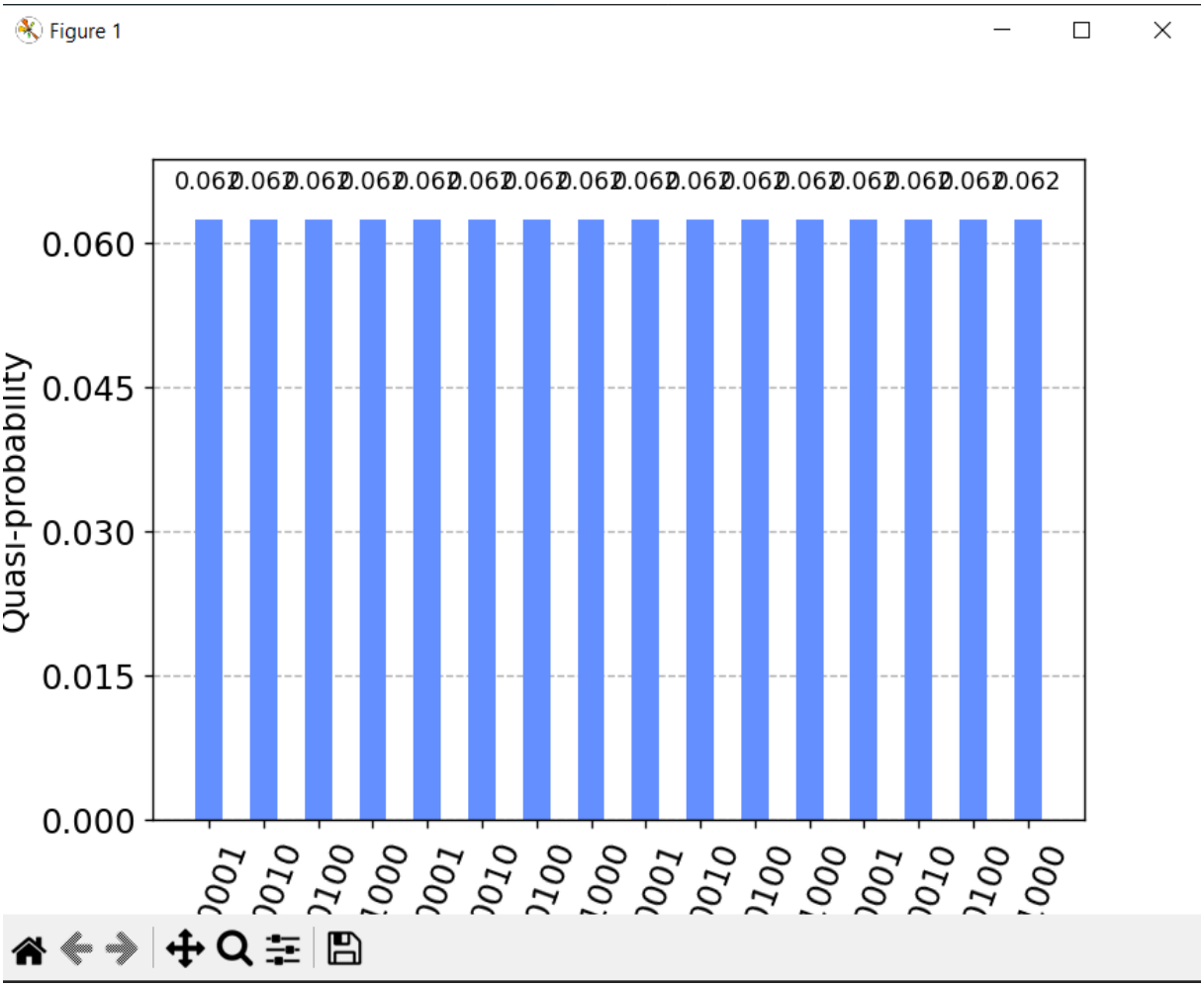


Figure 4. Histogram obtained for N=15 and a =2, after applying QFT

As presented in Figure 4. histogram contains too many values for such small N. It means that in the classical postprocessing we are still left with every possible number to conisider.

But! The real difference happens when we start to play with much bigger numbers.

a = 2 is a pretty fortunate choice,because it is coprime with all odd numbers, so I stick to it, not to modify the circuit too much.

I set **N = 3029, and m = 8** for better precision. Below is the histogram for this setting:

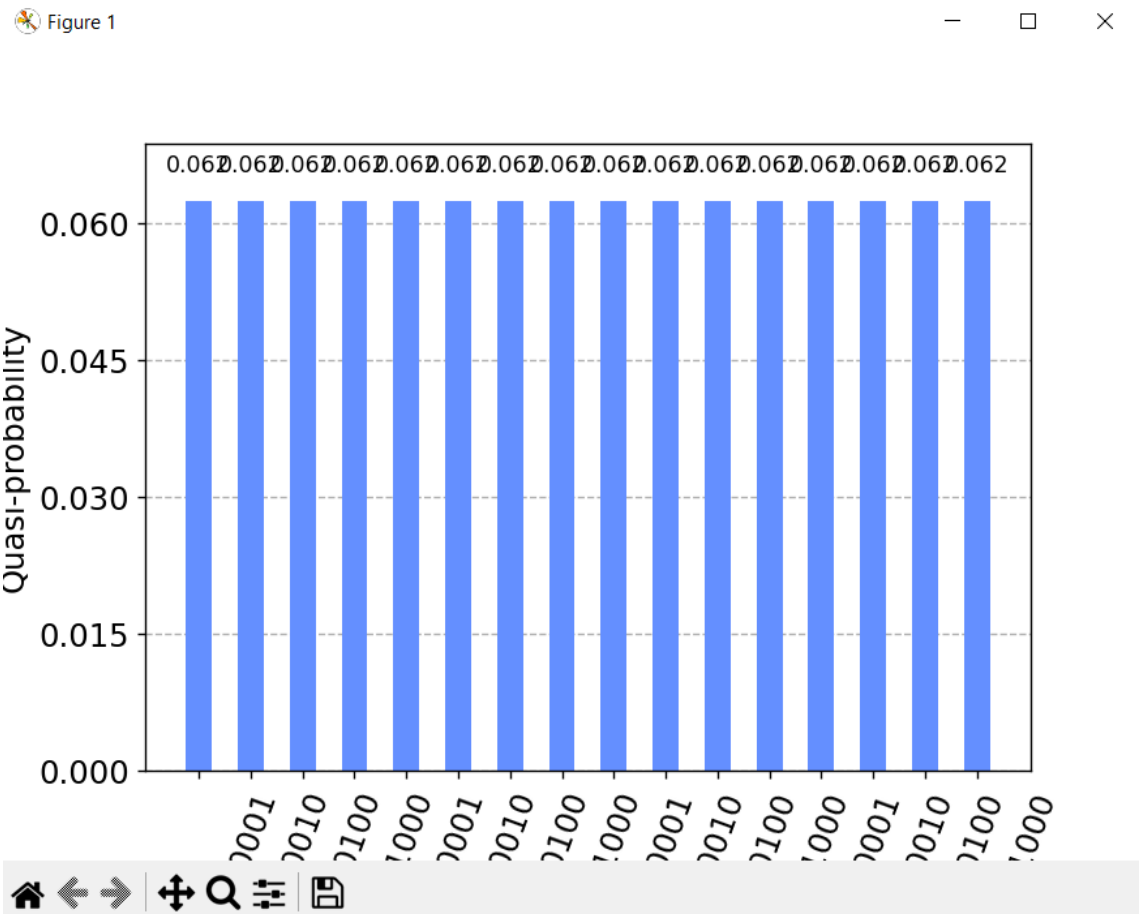
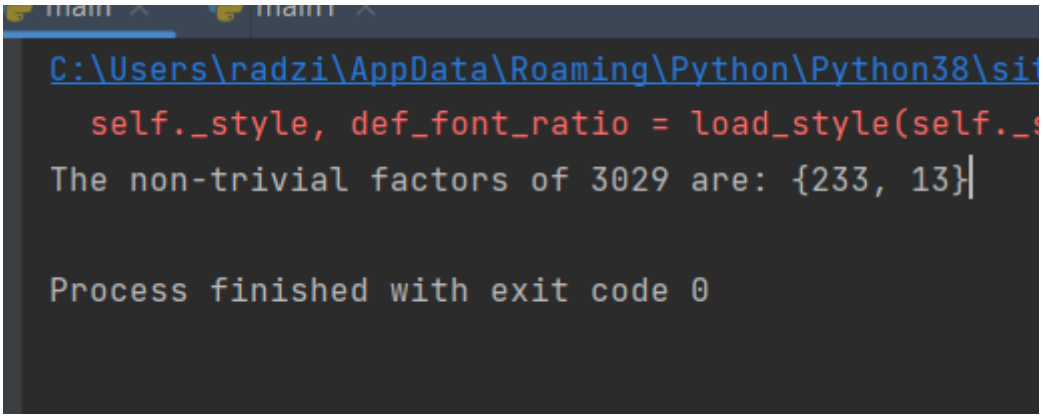


Figure 5. Histogram obtained for N=3029 and a =2, after applying QFT

And the result:



As we can see, it againg yields only 16 values. It is definitely not too hard a task for a classical computer, to look through all of them in search of p and q.

For comparison, if I dismiss the QFT part of the circuit, this is the histogram I obtain:

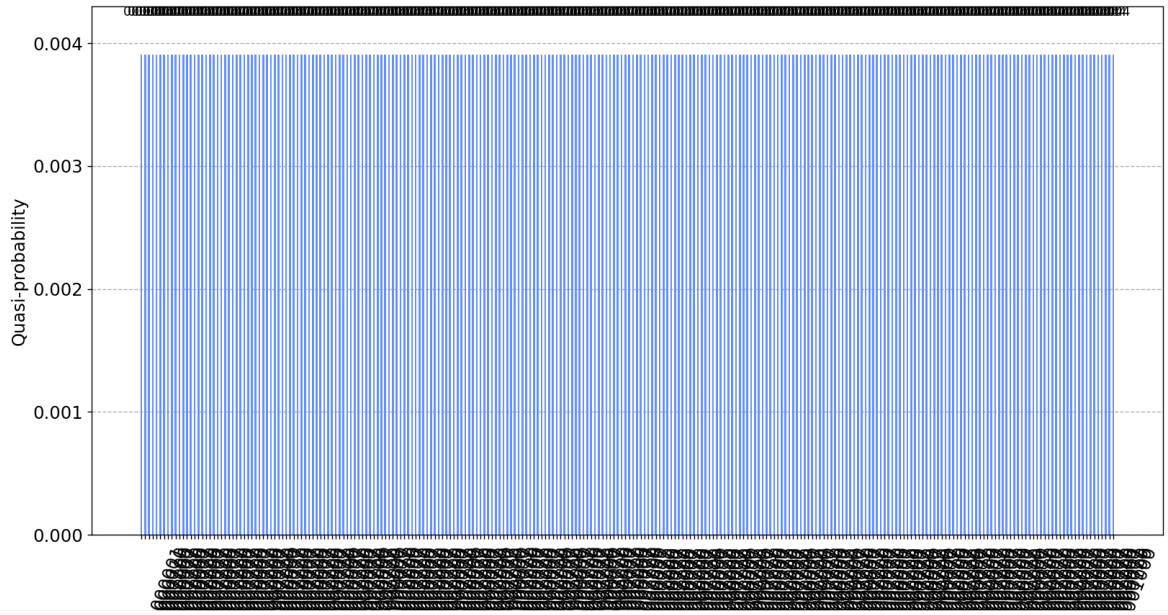


Figure 6. Histogram of the statevector that only went through the modular exponentiation gate, without applying QFT

Reassuming, the QFT part does the job and the code works!

Bibliography

[1] Monz T. et al., *Realization of a scalable Shor algorithm*, August 3, 2015, https://arxiv.org/pdf/1507.08852.pdf

[2] Wong T.G., *Introduction to Classical and Quantum Computing*, 2022

[3] Johnston E.R., Harrigan N., Gimeno-Segovia M., *Programming Quantum Computers. Essential Algorithms and Code Samples,* 2019 O'Reilly Media

[4] https://cnot.io/quantum_algorithms/qft/

[5] https://github.com/qiskit-community/qiskit-community-tutorials/blob/master/algorithms/shor_algorithm.ipynb

[6] https://web.archive.org/web/20230201165207/https:/www.qmunity.tech/tutorials/shors-algorithm

https://www.sharetechnote.com/html/QC/QuantumComputing_Shor.html

[7] https://github.com/Qiskit/textbook/blob/main/notebooks/ch-algorithms/quantum-phase-estimation.ipynb