POLITECNICO DI TORINO
Master Degree Course in Computer Engineering

Master Degree Thesis

Shor's Algorithm

Supervisors
prof. Antonio Lioy
dott. Ignazio Pedone

Candidate
Vito Medici

April 2022

Contents

Chapter 1

Introduction

The possibility to develop a new type of algorithms that solve some known hard problems in polynomial time given by quantum computing created a growing interest in it. Its applications can be found in many fields, from the financial [1] to the chemical, so a large number of companies are investing resources in its development. Important players like IBM [2] and Google [3] have already begun the development of quantum computers to execute these algorithms and solutions to let users from all over the world use them have been created, such as SDKs quantum programming languages. The growth of these technologies is expected to be very fast, as we can see from figure 1.1, since in two years it estimates to improve by an order of magnitude the information that a quantum device is able to store and manage.

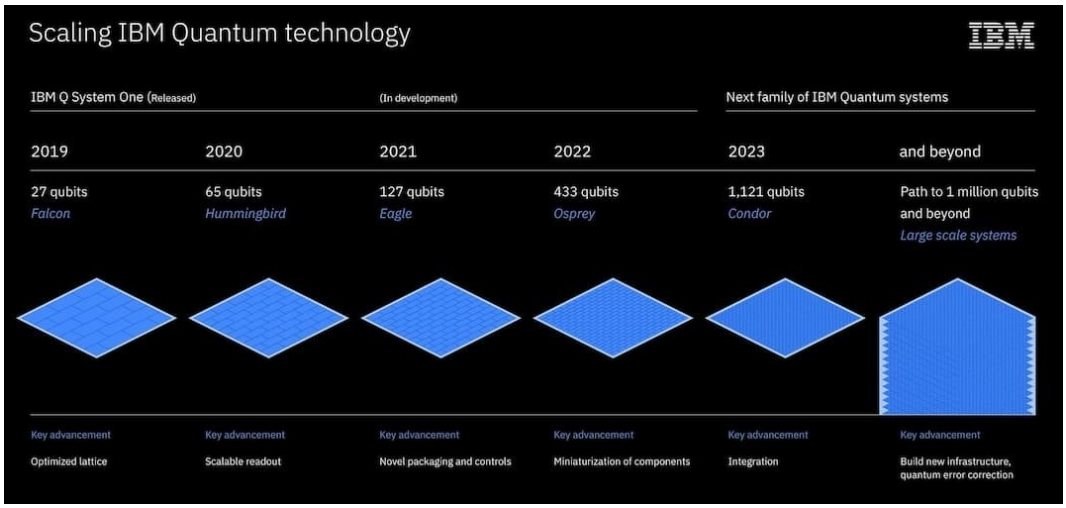


Figure 1.1. IBM roadmap for scaling quantum technology. Source:[4]

A particular quantum algorithm, developed by Shor, solves the problems that are used as a mathematical basis for various public-key cryptography algorithms. The threat caused by this algorithm is certified by the NIST call for proposals for public-key quantum-resistant algorithms [5] and by the development of quantum cryptography solutions. NIST is in fact selecting new classical algorithms for key exchange, key encapsulation, and digital signature, that should be

chosen in 2023, which are believed to be resistant to quantum algorithms as none has been found to break them. Quantum cryptography follows the other path, developing solutions to cryptographic problems with quantum computing. Important examples are Quantum Key Distribution [6] and Quantum Digital Signature. In this work, we will not concentrate on these fields, as we want to evaluate how distant we are from the day when quantum algorithms will be able to break the algorithms we use.

Shor's algorithm [7] uses some of the basic blocks of quantum computing to calculate the period of a modular function in polynomial time, which permits finding the prime factors or calculating the discrete logarithm of a number. In this way, it is possible to break both RSA and Elliptic curves cryptography. To understand why the possibility to break these two algorithms brought this great popularity and attention to the quantum world, we can think of all the applications that we use every day in our life. For example, both are used in the key exchange protocol and to perform digital signatures in TLS 1.3 [8], and elliptic curves also in the cryptocurrency field with Bitcoin and Ethereum [9].

However, the actual device capabilities are far from being able to execute Shor's algorithm to break the version of RSA or of elliptic curves used, as many problems in the realization of them must be taken into consideration. In fact, nowadays quantum computers are greatly affected by noise and have a very limited capability in terms of memory.

In this work, we want to evaluate how distant we are from an implementation able to break the cryptographic algorithms we use, understanding the capabilities of the needed device, based on the methods that IBM plans to use for its future quantum computer architectures. We also consider various enhancements to the basic version that have been published over the years, which are able to reduce its complexity. In order to do this, we study Shor's algorithm, both in the case of integer factorization and discrete logarithm problem, the way to implement it, and how we can execute it in a simulator or an actual quantum device.

We started from the Qiskit [10] implementation of Shor's algorithm to write a parametrized library that gives as output the integer factors of a biprime number, adding some functions to slightly increase the success probability of this algorithm. We proved the correctness of this library with a small example of decryption of an RSA-encrypted message, simulating the quantum part of the algorithm in the IBM qasm simulator. In addition, we provide a library for the realization of most of the gates for the circuit to break the Elliptic Curve Cryptography in Qiskit.

The work is divided in the following way:

- Chapter 2 introduces the basics of quantum computing, with its advantages and techniques.
- Chapter 3 lists some of the most important quantum algorithms, providing a high-level description of them.
- Chapter 4 talks about the different types of physical realization of quantum devices, who develop them, and how it is possible to use the solutions given by the manufacturers.
- Chapter 5 introduces the problems that must be taken into consideration when a user wants to execute a quantum algorithm on a quantum computer.
- Chapter 6 is useful to recall the basics of asymmetric cryptography.
- Chapter 7 provides a first analysis about how the quantum world affects the cryptographic algorithms we use nowadays, focusing on Grover's algorithm.

- Chapter 8 gives a detailed analysis of Shor's algorithm to break RSA, with all the necessary blocks to implement it, and an estimation of the resources that a quantum device would need to be a threat.
- Chapter 9 contains an explanation of the more complex passages of our implementation of the circuit for factorization.
- Chapter 10 makes the analysis performed on the previous two chapters for Shor's algorithm against the elliptic curves.
- Chapter 11 contains the tests we made to analyze our implementations and the resources needed to implement them.

The analysis proved that quantum computers are still not a threat to nowadays cryptography, so why the necessary measures to contrast it are already being chosen and, in the next few years, applied? The NIST call for proposals for post-quantum cryptography algorithms can be seen as an exaggeration looking at the current quantum hardware vendors' roadmaps, but quantum physics is in rapid growth, and we do not know when a certain technology in this field can be produced: in other words, we can not rely on something like the Moore law, so we should be provident and well prepared to these threats. In addition, we must remember that an important message can be sniffed and stored by an attacker for many years, waiting for the needed quantum device to break the security of the message: in this context, the solution adopted must guarantee the wanted security even for 15 years, so it is not possible to rely on algorithms such as RSA with certainty for that amount of time.

Chapter 2

Quantum Computing

The development that has been done so far in quantum computing allowed the discovery of a series of possible applications in many different topics, like cryptography, finance, optimization, and machine learning, so it is of particular interest to understand the basic foundations of this emerging field.

2.1 Quantum Computing overview

Quantum computing is the branch of the computing theory that exploits the properties of the quantum states, such as the superposition, the interference, and the entanglement, permitting to obtain a huge speed-up in the computational complexity of some typical hard problems, like the integer factorization, that even supercomputers will never solve with the currently known algorithms [11]. These problems require the analysis of a great number of combinations, that in classical computing will be performed sequentially. In quantum computing, we create multi-dimensional spaces that are able to represent these combinations together, so that the analysis can be performed in a parallel way, and then we translate this representation into a logical and understandable one.

Nowadays, there are many ways to build a quantum computer, since there are many ways to control a quantum state useful for computation. In the quantum circuit model, the computation is a sequence of gates and measurements that act on a set of quantum information units, called qubit (i.e., the corresponding of the bits for classical computers): we will use this one for this work since is the prevailing one nowadays [12].

Since quantum computing is based on quantum states, it is not an easy task to understand the quantum algorithms and the quantum protocols, mostly because the information carried by the qubit is not "0" or "1" like the one carried by the bit, but it is probabilistic. The first property that we will study is the superposition: as for classical waves, a quantum state can be seen as the linear combination of other quantum states [13]. When this new quantum state is measured (i.e., translated into a logical representation), it will result in a "1" with a probability p, and in a "0" with a probability $1 - p$. Because of this, we will need to learn a new formalism to represent the qubit states, which will be presented in 2.2. However, the characteristic of "probabilistic information" permits to "play" with many intermediate results at the same time, since the superposition allows to perform calculations on many states represented in the same qubits, giving an exponential speed-up to the quantum algorithms. Then, using the interference

effects we will be able to reduce the probability of getting "wrong answers", while the probability of "good answers" will increase, so we will be able to obtain our wanted result with a good probability.

To obtain an understandable result from the quantum computation, we can measure the qubits involved, but this will cause the state to lose its properties, decading into one of its states, since we can obtain just one answer from one measurement, and not all the possible different answers corresponding to the possible different states. In fact, when a quantum state that stores the linear combination of different ones is observed by the external world, it reduces to one of them: this state is said to "collapse"[14].

2.2 Qubit representation

To express a qubit, we need to first explain the concept of statevector. The statevectors are vectors where all the possible values of the state of a system are represented, along with their probability. This comes quite handy with qubits since they can only express the probability of the values 0 and 1. So, we can represent a state that, if measured, gives with 100% probability the output "0", and a state that gives with 100% probability the output "1", but also states that give a "1" or a "0" with a determined probability.

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \tag{2.1}$$

These 2 states form an orthonormal basis, that we can use to express all the possible quantum states, but before going into that field, let's concentrate on the symbol used to express the states that will give with probability 100% the 0 and the 1, called Dirac notation.

The Dirac (or "bra-ket") notation uses angular brackets to describe quantum states, introducing the following symbols:

- ket: $|a\rangle = \begin{bmatrix}a_1\\a_2\end{bmatrix}$
- bra: $\langle b| = \begin{bmatrix}b_1^* & b_2^*\end{bmatrix}$
- bra-ket: $\langle a|b\rangle = a_1 b_1^* + a_2 b_2^* = \langle a|b\rangle^*$
- ket-bra: $|a\rangle\langle b| = \begin{bmatrix}a_1 b_1^* & a_1 b_2^*\\ a_2 b_1^* & a_2 b_2^*\end{bmatrix}$

As we said, being that $|0\rangle$ and $|1\rangle$ form an orthonormal basis, we can express every quantum state as a linear combination of those two. In fact, $\langle 0|1\rangle = 0$ and we can normalize any quantum state (a normalized quantum state is a state such that $\langle\psi|\psi\rangle = 1$). So it is possible to use the notation in equation 2.2, where α and β are complex numbers called probability amplitudes.

$$\begin{gathered}|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\\beta\end{bmatrix}\\ \alpha, \beta \in \mathbb{C}\end{gathered} \tag{2.2}$$

The $\{|0\rangle,|1\rangle\}$ is not the only basis possible, since there are infinite ones. Two of the most important ones are $\{|+\rangle,|-\rangle\}$ and $\{|+i\rangle,|-i\rangle\}$, which states can be written in the $\{|0\rangle,|1\rangle\}$ basis as:

$$|+\rangle := \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{2.3}$$

$$|-\rangle := \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.4}$$

$$|+i\rangle := \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle) \tag{2.5}$$

$$|-i\rangle := \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle) \tag{2.6}$$

2.2.1 Measurements

The α and β values in the equation 2.2 can be used, according to the Born rule [15], to recover the probability to obtain a 0 or a 1 with a measurement in the $\{|0\rangle,|1\rangle\}$ basis.

$$p(0) = |\alpha|^2 \quad p(1) = |\beta|^2 \tag{2.7}$$

This method is called projective measurement [16, sec. 2] since we choose an orthonormal basis to describe and measure the quantum state. This measurement can be done on any orthonormal basis, so we can find the probability of having any state $|\psi\rangle$ giving as output of the measurement in the basis $\{|x\rangle,|x^{\perp}\rangle\}$ the state $|x\rangle$ with the equation.

$$p(|\psi\rangle) = |\langle x|\psi\rangle|^2 \tag{2.8}$$

This kind of "gadget" seems to give us the possibility to know at each step of an algorithm the states of the qubits, but this is not true. In fact, observing a qubit change its state irreversibly: if the result of the measurement is 0, α will become equal to 1 and β equal to 0, while in the other case α will become equal to 0 and β equal to 1. This transforms the quantum computation into a nearly classical one, so we will not be able to use the important quantum computing properties in section 2.3.

In addition, this makes it impossible to measure the global phase of the complex number. We can take as an example the probability of a state to be $i\,|1\rangle$. Performing the measurement as in equation 2.8 of a state $|x\rangle$, we can see in 2.9 how this probability is equal to the one of measuring $|1\rangle$.

$$p(i\,|1\rangle) = |\langle x|\,(i\,|1\rangle)|^2 = |i\,\langle x|1\rangle|^2 = |\langle x|1\rangle|^2 \tag{2.9}$$

However, projective measurements are not enough to describe all the types of measurements that we can make on the quantum states, so we need some kinds of generalization of it, like the Positive Operator-Valued Measures (POVMs) [16, sec. 3].

Another important effect of the measurement is the possibility, dealing with entangled states, to measure a qubit and collapse the state of the other entangled qubits, but we will see this more in detail in section 2.3.2.

2.2.2 Bloch sphere representation

Let's take a step back, looking at equation 2.2: even if α and β are complex numbers, we can not measure the global phase of these two number, but only the difference between the two phases, so we can write that equation as

$$\begin{gathered} |\psi\rangle = \alpha\,|0\rangle + e^{i\phi}\beta\,|1\rangle \\ \alpha,\beta,\phi \in \mathbb{R} \end{gathered} \tag{2.10}$$

Being that the state has to be normalized, we know that $\sqrt{\alpha^2 + \beta^2} = 1$, so we can use a trigonometric variable in order to express both α and β, such that the equation become

$$|\psi\rangle = cos(\frac{\theta}{2})\,|0\rangle + e^{i\phi} sin(\frac{\theta}{2})\,|1\rangle \qquad \theta, \phi \in \mathbb{R} \tag{2.11}$$

Having the quantum state written in this form, it is clear to see that it can be easily represented in a sphere of radius 1, called "Bloch sphere". Figure 2.1 shows the position of the most used quantum states of a qubit in the Bloch sphere, as well as the meaning of the angles in the equation 2.11.

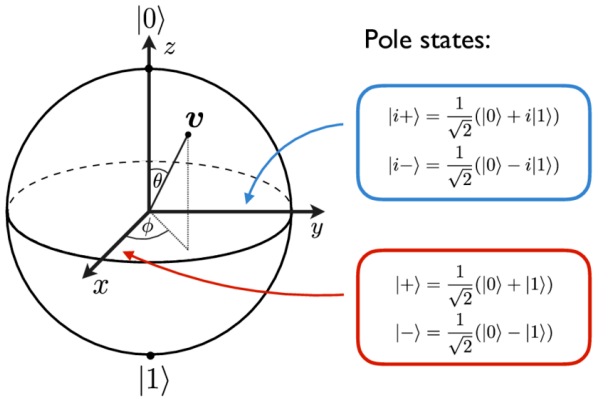


Figure 2.1. Bloch sphere. Source: [17]

Observing the figure, we can understand the meaning of an important piece of nomenclature:

- the Z-measurement is a projection done in the z-axis, so in the $\{|0\rangle,|1\rangle\}$ basis;
- the X-measurement is a projection done in the x-axis, so in the $\{|+\rangle,|-\rangle\}$ basis;
- the Y-measurement is a projection done in the y-axis, so in the $\{|+i\rangle,|-i\rangle\}$ basis.

2.2.3 Multiple qubit states

We have seen so far the methods to represent a quantum state of a single qubit, but in our algorithms, we will need many of them. Although we will not see in this paragraph the properties that can be gained by using many qubits together, the way to represent quantum states of multiple qubits is the same both in these cases and in the case of completely separated qubits.

As for the one-qubit case, we can express multi-qubit states using a state vector, but this time it will contain 2^n amplitude values, where n is the number of qubits to represent:

$$|x\rangle = x_{00}\,|00\rangle + x_{01}\,|01\rangle + x_{10}\,|10\rangle + x_{11}\,|11\rangle = \begin{bmatrix} x_{00} \\ x_{01} \\ x_{10} \\ x_{11} \end{bmatrix} \tag{2.12}$$

Also in this case we can use the Born rule to obtain the probability of each state to be measured, as can be seen in equation 2.13.

$$p(|00\rangle) = |\langle 00|x\rangle|^2 = |x_{00}|^2 \tag{2.13}$$

For example, if we perform a Z-measurement on the state $|\psi_1\rangle$ of equation 2.14, we will have as output 00 and 10 in the 37,5% of the cases, and 01 and 11 in the 12,5%.

$$|\psi_1\rangle = \frac{1}{2\sqrt{2}}(\sqrt{3}\,|00\rangle + |01\rangle + \sqrt{3}\,|10\rangle + |11\rangle) \tag{2.14}$$

In order to express the state of two or more separated qubits into a single statevector, we have to apply the tensor product between the two or more statevectors. For example, if we have the qubits a, b and c, their collective state will be

$$|abc\rangle = \begin{bmatrix} a_0b_0c_0 \\ a_0b_0c_1 \\ a_0b_1c_0 \\ a_0b_1c_1 \\ a_1b_0c_0 \\ a_1b_0c_1 \\ a_1b_1c_0 \\ a_1b_1c_1 \end{bmatrix} \tag{2.15}$$

Even if the Bloch sphere can describe very well a state of a single qubit, it is not suitable for a quantum state with two or more qubits, since every qubit would need a different sphere.

2.2.4 Q-sphere

IBM introduced, to represent a multi-qubit state, the Q-sphere, which uses the size of the blobs to indicate the amplitude probability of that state, and the color of them to indicate the phase [18], as can be seen in figure 2.2.

The basis states are equally distributed in the surface of the sphere, with the state with all zeros $(|0\rangle^{\otimes n})$ in the north pole and the state with all ones $(|1\rangle^{\otimes n})$ in the south pole.

2.3 Qubit properties

As we already mentioned, the quantum operations are done in order to exploit the peculiar quantum properties of the qubits. We already saw in detail the effects of the superposition, so we can focus on the interference and the entanglement.

2.3.1 Interference

Until now we have described the quantum superposition as the capacity for a qubit to have the probabilistic characteristic, but thinking just at this explanation, we may ask how it is different from having a state that we simply know to be "1" or "0" with a certain probability: the *interference* effect answers this question, but brings with it a side-effect called decoherence, described in section 2.3.3.

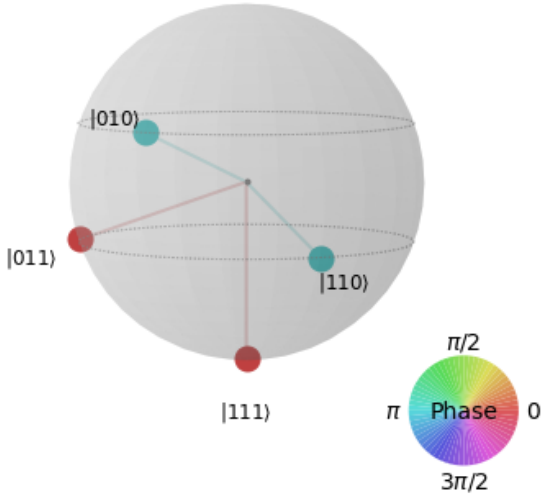


Figure 2.2. Q-sphere representing $|\psi\rangle = -|010\rangle - |110\rangle + |011\rangle + |111\rangle$. The sphere is always shown together with a legend that permits to understand the phase of the various states. The probability amplitude of each of them is represented by the size of the blob.

As we said in 2.2, we can express a quantum state using different bases, that are mathematically correlated. If we have a quantum state $|+\rangle$, the measurement in the $\{|0\rangle,|1\rangle\}$ basis results in a "1" and in a "0" the 50% of the times, and the same consideration can be done for $|-\rangle$.

Now we can look at the state $|\psi\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle)$.

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle) = \frac{1}{2}(|1\rangle + |0\rangle + |1\rangle - |0\rangle) = |1\rangle \tag{2.16}$$

If we think about it classically, the measurement in the $\{|0\rangle,|1\rangle\}$ basis should give the same outputs as before. However, looking at equation , we can see how the two $|0\rangle$ terms are eliminated.

This effect is called interference [20] and can be understood by remembering that quantum states are complex numbers that can be represented as waves. In classical waves, the amplitude is a physical property, while in quantum states, it represents the probability of obtaining with a measurement a certain state. However, the interference effect acts in a similar way in the two cases, changing the amplitudes when more waves or states interact.

2.3.2 Entanglement

When we have two or more qubits, we can have situations where they are strictly correlated. The simplest example are the Bell states, listed in the equations below.

$$\begin{aligned} |\phi_1\rangle &= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \qquad |\phi_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \\ |\phi_3\rangle &= \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \qquad |\phi_4\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle) \end{aligned} \tag{2.17}$$

As explained in section 2.2.3, measuring $|\phi_1\rangle$ in the $\{|0\rangle,|1\rangle\}$ basis will give the state $|00\rangle$ with probability 50%, and the state $|11\rangle$ with probability 50%. However, being that these are in

superposition, measuring the state of one of them, will automatically tell us the state of the other: if we measure $|0\rangle$ for the first qubit, we can be certain that the other qubit will also be in state $|0\rangle$, so it will collapse as well. In these situations, we can not write the two states separately since it is not possible to attribute to them a pure state: we call them *entangled states* [19].

If it is not possible to associate a state vector to one of the entangled states, it is still possible to represent it with a density matrix, as we will describe in section 2.5.

The most direct and intuitive example of the use of the entanglement is the teleportation algorithm, but we will use this property for nearly every algorithm in quantum computing.

2.3.3 Decoherence

There are cases where a quantum state seems to not be affected by interference, due to the interactions with the system: we call this effect *decoherence* [21]. This comes from the difficulty of realizing an isolated environment, which makes it impossible to avoid interactions of a qubit with the external world.

For example, in physical quantum computers, we may have an unintended measurement for an unwanted entanglement between two qubits. Assume that we have the state $|\psi_1\rangle = |-\rangle$ and that it is affected by an unintended measurement in the $\{|0\rangle,|1\rangle\}$ basis: in this situation we lose the interference effect and the resultant quantum state will be either $|\psi_2\rangle = |0\rangle$ or $|\psi_2\rangle = |1\rangle$ with 50% probability each, without any superposition effect.

This possibility poses a real challenge to the development of quantum computers since every qubit maintains its state only for a window of time. Because of this, every qubit in a quantum device is characterized by the so-called *decoherence time*, which gives us a measure of how much time a qubit can maintain in average the state.

2.4 Quantum gates

Quantum gates are the basic block of the quantum circuits since they permit the transformation between all the possible states.

Every gate acting on a quantum state performs an evolution governed by Schrödinger's equation [22], and every transformation applied to this equation can be expressed as a unitary matrix. The unitarity of the quantum gates gives us an important property called *reversibility*. If a matrix U is unitary, then its inverse U^{-1} is equal to its conjugate transpose $U^{\dagger}$, so we can obtain equation 2.18, and the sum of the square of every element in a fixed row or column is always one, as expressed in equation 2.19.

$$U^{\dagger}U = \mathbb{1} \tag{2.18}$$

$$\sum_{i=1}^{n} |U_{ij}|^2 = 1 \qquad \sum_{j=1}^{n} |U_{ij}|^2 = 1 \tag{2.19}$$

These equations tell us that a quantum gate is always reversible and that if it is applied on a normalized quantum state the result will be another normalized quantum state.

They are usually expressed as matrices, but since they are just a rotation of the quantum state, they can be represented in the Bloch sphere.

2.4.1 Single qubit gates

The basic single-qubit gates are the Pauli gates and the Hadamard gate, but we can construct every rotation, as we will see with the U-gate.

The Pauli gates perform a rotation of π around an axis of the Bloch Sphere, so we will have the X-, the Y-, and the Z-gate.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0| \tag{2.20}$$

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = -i\,|0\rangle\langle 1| + i\,|1\rangle\langle 0| \tag{2.21}$$

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = |0\rangle\langle 0| - |1\rangle\langle 1| \tag{2.22}$$

The Hadamard gate is the one that permits the create the superposition of $|0\rangle$ and $|1\rangle$. Its matrix representation is given by

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.23}$$

so it performs the operations described in the transormations below.

$$\begin{aligned} H\,|0\rangle = |+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \qquad H\,|+\rangle = |0\rangle \\ H\,|1\rangle = |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \qquad H\,|-\rangle = |1\rangle \end{aligned} \tag{2.24}$$

A digression should be done for the Hadamard applied on n qubits.

$$|x\rangle \left\{ \begin{array}{l} |x_0\rangle \text{—} \boxed{H} \text{—} |y_0\rangle \\ |x_1\rangle \text{—} \boxed{H} \text{—} |y_1\rangle \\ \vdots \quad \text{———} \\ |x_n\rangle \text{—} \boxed{H} \text{—} |y_n\rangle \end{array} \right\} |y\rangle \; = H^{\otimes n}\,|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{k\in\{0,1\}^n} (-1)^{k*x}\,|k\rangle \tag{2.25}$$

This equation means that $|y\rangle$ is the superposition of all possible 2^n bit strings, so the probability of measuring any number between 0 and $2^n - 1$ is the same: we will use this method in many circuits, as well as in the one for the Shor's Algorithm.

The U-gate is used in order to describe a generic rotation, as it has the form

$$U(\theta, \phi, \lambda) = \begin{bmatrix} cos(\frac{\theta}{2}) & -e^{i\lambda} sin(\frac{\theta}{2}) \\ e^{i\phi} sin(\frac{\theta}{2}) & e^{i(\phi+\lambda)} cos(\frac{\theta}{2}) \end{bmatrix} \tag{2.26}$$

For example, if we want to describe the Hadamard gate, we can use $U(\frac{\pi}{2},0,\pi)$.

2.4.2 Multi qubit gates

The interaction between the different qubits (i.e. the entanglement) is created with the CNOT-gate, that is an X-gate controlled by another qubit. If the control qubit is $|1\rangle$, the X-gate is applied in the target qubit:

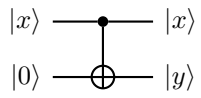


Figure 2.3. General application of the CNOT gate.

In this case $|y\rangle$ will remain $|0\rangle$ if $|x\rangle$ was $|0\rangle$, while it will become $|1\rangle$ otherwise. With this gate, we can create the Bell state described in section 2.3.2, using the simple circuit in figure 2.4

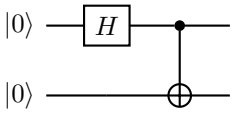


Figure 2.4. CNOT gate with a qubit in superposition as control qubit.

After the Hadamard gate, the first qubit will be in the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, so the application of the CNOT-gate will create the collective state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. We can see that by measuring the target qubit, we can be sure of the value of the control qubit since the target will change to state $|1\rangle$ only if the control is in this state.

Other gates can be constructed using this one, like the Toffoli and the Swap.

The Toffoli is a 3-qubit gate where two qubit are used as control for the U-gate in the target qubit. It can be constructed using just 2-qubit controlled rotations.

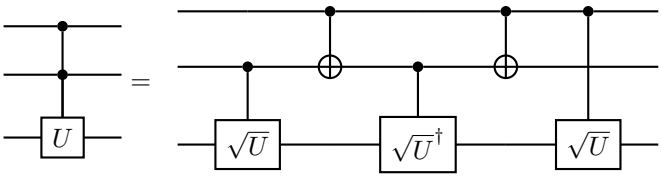


Figure 2.5. Decomposition of a Toffoli using 2-qubit gates.

The swap is a 2-qubit gate that just exchange the state of 2 qubits.

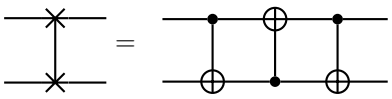


Figure 2.6. Decomposition of a Swap gate using 2-qubit gates.

2.4.3 Phase kickback

Given the discussion done in the last section, one may think that the control qubit is completely not affected by the application of the CNOT-gate. However, there is a trick that permits to bring a change on the phase of the state of the control qubit that is used as a basis for most of the quantum algorithms that we will see: this is called *phase kickback* [23]. To understand this method, let's look at this simple circuit identity of the following figure.

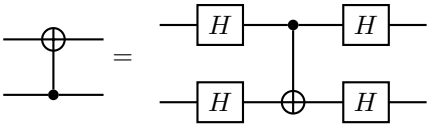


Figure 2.7. Simple circuit that takes advantage of the phase kickback.

If both the control and the target qubit are in the state $|+\rangle$ before applying the CNOT-gate, really nothing happens, as expressed in the equation below.

$$|++\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \rightarrow \frac{1}{2}(|00\rangle + |01\rangle + |11\rangle + |10\rangle) \tag{2.27}$$

As we can see, the probability amplitudes stays unchanged, but if the control qubit is in state $|+\rangle$ and the target is in state $|-\rangle$, the equation becomes:

$$|+-\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \rightarrow \frac{1}{2}(|00\rangle - |01\rangle + |11\rangle - |10\rangle) = |--\rangle \tag{2.28}$$

The phase of the control qubit change, since it passes from state $|+\rangle$ to state $|-\rangle$, thanks to the relative phase on the target qubit. This method is used in many quantum algorithms because it enables to encode the output of a circuit in the phase of the control qubits so that it can be easily measured, as we will see in chapter 3.

2.4.4 Universality

A set of quantum gates is called universal when any possible operation can be reduced to a finite sequence of these gates. Actually, a quantum circuit can always be expressed as a single unitary, so the universality is the capacity to implement any possible unitary.

The most common universal set of gates is given by the Clifford gates and the T gate. The Clifford gates are those that permit the permutation of the Pauli operator. The set formed by the CNOT, the H, the S-gate (that is actually $U(0, -\frac{\pi}{2},0)$, so it is called also $\sqrt{Z}$), and the T-gate ($U(0, -\frac{\pi}{4},0)$, so it is called also $\sqrt[4]{Z}$) is the most common universal one.

The Hadamard and the S-gate, being Clifford gates, can be combined with the Pauli operators to have the effect of different operations. For example, we can look at these identities:

$$HXH = Z \qquad HZH = X \qquad SXS^\dagger = Y \qquad SYS^\dagger = -X \qquad SZS^\dagger = Z$$

Using a universal set of quantum gates gives us the possibility to reproduce any classical computation in a quantum computer, so we may think about a way to reproduce a function $f(x)$ in a reversible way.

Gate	Effect
X	Rotation of π around the x axis of the Bloch Sphere
Y	Rotation of π around the y axis of the Bloch Sphere
Z	Rotation of π around the z axis of the Bloch Sphere
H	$H(\|0\rangle) = \|+\rangle\,, \quad H(\|1\rangle) = \|-\rangle\,, \quad H(\|+\rangle) = \|0\rangle\,, \quad H(\|-\rangle) = \|1\rangle$
$U(\theta, \phi, \lambda)$	Most general gate
S	Rotation of $\frac{\pi}{2}$ around the z axis of the Bloch Sphere
T	Rotation of $\frac{\pi}{4}$ around the z axis of the Bloch Sphere
CNOT	Performs an X-gate on the target qubit if the state of the control qubit is $\|1\rangle$
Toffoli	Performs an X-gate on the target qubit if the state of the two control qubits is $\|1\rangle$
Swap	Swap the state of two qubits using 3 CNOTs

Table 2.1. List of described gates.

2.5 Density Matrix

Until now, we have seen states that are pure or in superposition. However, we can have cases where a qubit has a state with a certain probability and another state with another probability without being in superposition. For example, if Alice sends the state $|+\rangle$ to Bob, using a channel that has a percentage of error p, Bob will receive state $|+\rangle$ with probability $1-p$ and state $|-\rangle$ with probability p: we can see that these states are not in superposition, and we cannot express them using statevectors because they are not linear combinations of basis states. The states that consist of statistical ensembles of different quantum states are called *mixed states*, and we can represent them using the density matrix.

The mixed states can not be expressed with the bra-ket notation, since this brings the concept of state vectors, but instead, they must be expressed as a list of possible states. For example, suppose that we have a Hadamard gate that is applied to a qubit initialized to state $|0\rangle$. However, the initialization the 10% of the times fail so, after the application of the Hadamard, we can have:

1. with 90% probability the state $|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$;
2. with 10% probability the state $|\psi_2\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$.

In this case, the situation can only be represented as an ensemble of states, together with the corresponding probability.

$$\{|\psi_1\rangle\,, |\psi_2\rangle\} = \{\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle\}$$
$$\{p_1, p_2\} = \{\frac{9}{10}, \frac{1}{10}\}$$

The density matrix is built summing the outer product of every possible state with itself ($|\psi\rangle\langle\psi|$) multiplied by its probability. In the situation of the previous example, the density matrix is described in equation 2.29

$$\rho = \frac{9}{10}|\psi_1\rangle\langle\psi_1| + \frac{1}{10}|\psi_2\rangle\langle\psi_2| = \frac{9}{10}\begin{bmatrix}\frac{1}{2} & \frac{1}{2}\\ \frac{1}{2} & \frac{1}{2}\end{bmatrix} + \frac{1}{10}\begin{bmatrix}\frac{3}{4} & \frac{\sqrt{3}}{4}\\ \frac{\sqrt{3}}{4} & \frac{1}{4}\end{bmatrix} = \begin{bmatrix}\frac{21}{40} & \frac{\sqrt{3}+18}{40}\\ \frac{\sqrt{3}+18}{40} & \frac{19}{40}\end{bmatrix} \tag{2.29}$$

This failure probability on the initialization of the qubits is always considered in real systems since the error rates are high in modern architectures.

The importance of the density matrix representation relies on the possibility to calculate the application of a gate, so a unitary matrix, directly on it. For example, if we apply a Y-gate to the resulting mixed state of equation 2.29, it can be done as in equation 2.30.

$$\rho' = Y\rho Y^\dagger = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} \frac{21}{40} & \frac{\sqrt{3}+18}{40} \\ \frac{\sqrt{3}+18}{40} & \frac{19}{40} \end{bmatrix} \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = \begin{bmatrix} \frac{19}{40} & -\frac{\sqrt{3}+18}{40} \\ -\frac{\sqrt{3}+18}{40} & \frac{21}{40} \end{bmatrix} \tag{2.30}$$

Another feature that the density matrix has, as said in section 2.3.2, is the possibility to represent separately entangled states: in fact, if a unitary transformation is applied to the density matrix of one of the entangled qubits, the density matrix of the other ones will not be affected [19].

2.6 Classic computation on a Quantum computer

A quantum oracle U_f is a black-box that permits to perform a classical function $f(x)$, so it can be described as:

$$U_f \left|x, y\right\rangle = \left|x, y \oplus f(x)\right\rangle \tag{2.31}$$

We need two registers in order to ensure the reversibility of the operation: the first one will encode the input, while the output will be represented in the second one. Additionally, we cannot encode the output in the same register of the input because they can have different size. As we can see from equation 2.31, $U_f = U_f^\dagger$, since $y \otimes f(x) \otimes f(x) = y$.

For example, we can build the function $f(x) = x + 3$ with an input register of 2 qubits $|x\rangle$ and an output register of 3 qubits $|y\rangle$, which will result in the truth table of table 2.2.

x_1	x_0	y_2	y_1	y_0
0	0	0	1	1
0	1	1	0	0
1	0	1	0	1
1	1	1	1	1

Table 2.2. Truth table of the function $x + 3$.

To construct the circuit, we can find a boolean function for every bit of the output [24], so we can build it as in figure 2.8.

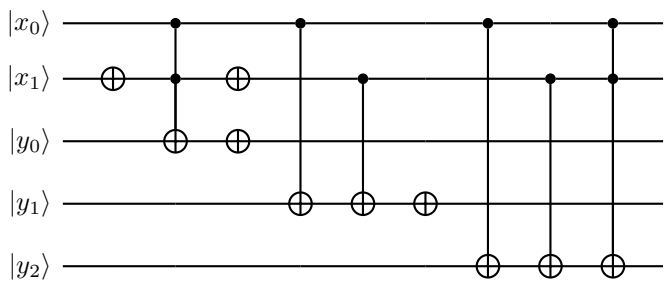


Figure 2.8. Circuit for the oracle of the function $f(x) = x + 3$.

Another type of oracle is the phase oracle P_f, that permits to apply a phase on the input based on it. An example of that can be

$$P_f |x\rangle = (-1)^{f(x)} |x\rangle \tag{2.32}$$

This type of oracle can be useful if applied to a superposition or as a controlled operation. For example:

$$P_f |+\rangle = \tfrac{1}{\sqrt{2}}((-1)^{f(0)} |0\rangle + (-1)^{f(1)} |1\rangle) = \tfrac{(-1)^{f(0)}}{\sqrt{2}}(|0\rangle + (-1)^{f(1)-f(0)} |1\rangle) = (-1)^{f(0)} Z^{f(0)-f(1)} |+\rangle$$

And, being $Z = Z^\dagger$, so $Z^{f(0)-f(1)} = Z^{f(1)-f(0)}$, this oracle actually changes the phase depending on $f(0)$ and applies a Z-gate.

We will see an application of this oracle in section 3.1 with the Deutsch-Jozsa algorithm.

2.6.1 Ancillae qubits

There are some cases where the classical function, in order to be transformed into a reversible one, needs more qubits than the actual needed as output. We will see in the next chapters of this work various examples of this kind of oracles since more complex operations usually need them, in some cases also to lower the number of operations performed.

We can use an X-gate controlled by 5 qubits as an example, which without the use of ancillae qubits would need many operations to be performed. We can lower this count using just 3 ancillae and 4 Toffoli gates, as done in figure 2.9.

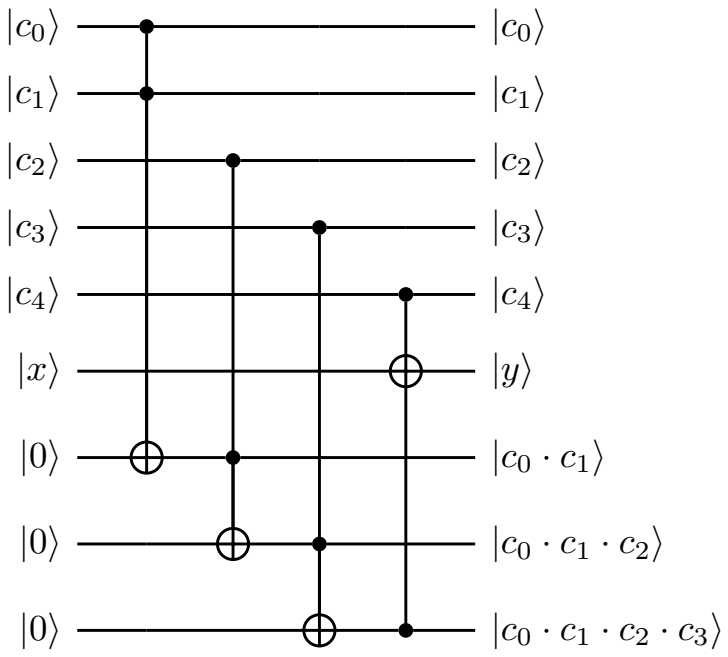


Figure 2.9. Implementation of an X-gate controlled by 5 qubits using 3 ancillae.

There are two types of ancillae qubits since some algorithms require qubits initialized to a particular state like $|0\rangle$ and others not. In the first case, we will refer to them as *clear ancillae*, while in the second one as *dirty ancillae*. The important thing to note is that both of these need to be returned to their original state at the end of the oracle. In order to do so, it is common to copy the output of the function to another ancilla register, so that the inverse of the oracle can be performed and the original states preserved. This permits, in a long computation, to use ancillae qubits for different purposes, or even to use, as dirty ancillae, qubits that store the output of other functions.

2.7 Logical qubits

We talked so far of a qubit as if it was perfect, without any error, but this is completely not true, since the current quantum technologies are far to be perfect. In fact, every gate has a percentage of error, the decoherence time of the qubits does not permit to have long circuits, and even the measurements sometimes fail. We can refer to a "perfect" qubit as a logical qubit, while to a real qubit as a physical qubit. If we run our algorithms using physical qubits, even in the most recent quantum devices, we will obtain random results, since the error percentage is too high: we need a way to make logical qubits out of physical ones. Due to the no-cloning theorem, we cannot create copies of an unknown quantum state, so Peter Shor formulated a Quantum Error Correction (QEC) code [25] to store the information of one qubit into nine entangled ones. The type of QEC techniques that are based on the repetition of the logical information is called repetition codes. However, since many errors can happen in the same logical qubit, we need an increasing number of physical qubits for a single logical one if we want to gain more fidelity: the distance code d is a number defined such that the maximum number of errors that can be simultaneously corrected are $\lfloor \frac{d-1}{2} \rfloor$.

Some important techniques must be considered: we need a way to detect and correct the error (syndrome measurement), a way to retrieve the logical information from many physical qubits (decoding), and a way to transform every logical operation so that it can operate on the entangled structure (logical operation [26], made by many physical operations).

In order to better understand them, we will take as an example the Shor code in the rest of this section to understand the steps needed to be taken into account, but we will expand this subject in section 5.4, where an example of a modern error correction code is given.

2.7.1 Encoding

In the Shor code, the basis states $|0\rangle$ and $|1\rangle$ are encoded creating an entanglement between 9 qubits using the following transformations.

$$|0\rangle \rightarrow \frac{1}{2\sqrt{2}}(|000\rangle + |111\rangle) \otimes (|000\rangle + |111\rangle) \otimes (|000\rangle + |111\rangle) = |0_L\rangle$$
$$|1\rangle \rightarrow \frac{1}{2\sqrt{2}}(|000\rangle - |111\rangle) \otimes (|000\rangle - |111\rangle) \otimes (|000\rangle - |111\rangle) = |1_L\rangle$$

This encoding is created using the circuit in figure 2.10, that protect against an error the information on the first qubit $|\psi\rangle$.

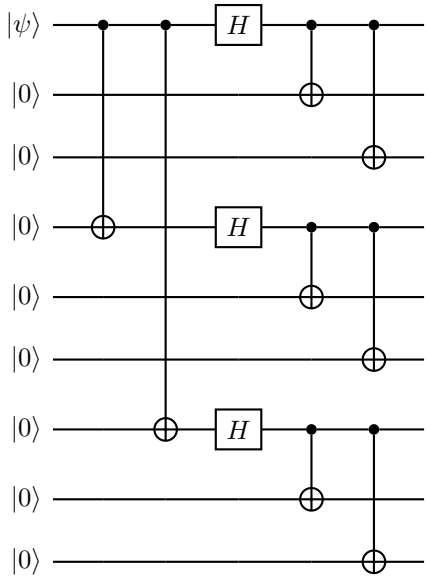


Figure 2.10. Encoding operation of the Shor code. Source: [25]

In this way, the state $|\psi\rangle$ is encoded in the state $|\psi\rangle = \alpha\,|0_L\rangle + \beta\,|1_L\rangle$.

2.7.2 Syndrome measurements

The syndrome measurement is a measurement done on many qubits that, without affecting the logical quantum information encoded, can detect an error and apply an operation to correct the state. The output of the measurement, called syndrome tells which is the affected qubit and what is the error that occurred. The effect of the noise is the same as a rotation around an axis of the Bloch sphere, so we can refer to this as a Pauli error, so the QEC algorithm can use the same Pauli operator to correct it. The errors are in practical two: the bit flip, which is like an X-gate, and the sign flip, which is like a Z-gate. However, if these two happen together, they will become like a Y-gate.

To protect against bit-flip errors, we use parity measurements. In the Shor code, this is done acting on 3 chunks of 3 qubits, correcting a maximum of 1 bit-flip error each. The parity measurement acts on 2 consecutive qubits, checking if they have the same state (resulting in a $|+\rangle$) or not ($|-\rangle$). The measurements are done to the first and the second qubit (Z_1Z_2) and to the second and the third one (Z_2Z_3). The correction to apply is retrieved using this truth table:

Z_1Z_2	Z_2Z_3	Correction
+	+	I
-	+	X_1
-	-	X_2
+	-	X_3

Table 2.3. Correction to apply in regard of the measurements.

To correct the phase errors, we act in the same way, with the difference that this time the measurement is done in the $\{|+\rangle,|-\rangle\}$ basis and that it is done considering as single qubit one of the chunks of the bit flip correction, acting again in the single chunk if an error was detected according to table 2.3.

2.7.3 Decoding

The decoding process is needed in order to retrieve the logical information from the encoded entangled state. In the Shor code, this is done using a circuit that permits to recreate the information on the first qubit, leaving the other qubits to state $|0\rangle$, retrieving in this way all the original states of the qubits before the encoding. Since the operation is the inverse of the encoding, the needed circuit is the inverse of the one in 2.10.

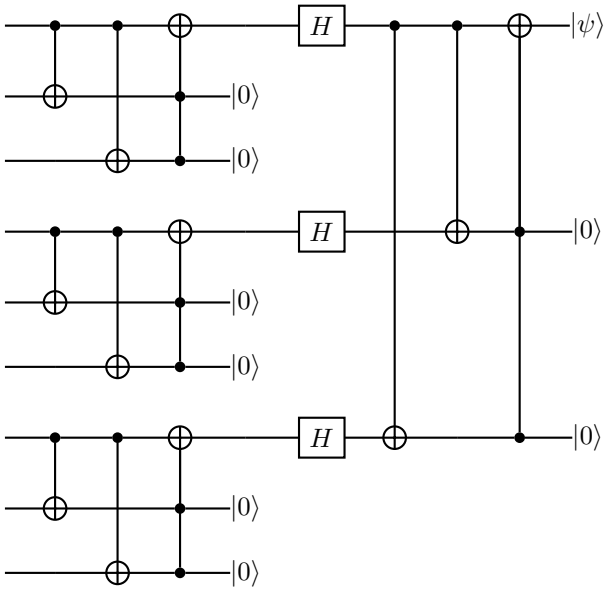


Figure 2.11. Decoding operation in the Shor code. Source: [25]

2.7.4 The surface code

This basic background will help us to understand the way QEC is applied to modern architectures, that are designed to be optimized for a specific surface code. In the surface codes [27] the physical qubits interact only with their nearest neighbors, and we will see how much this property is important in quantum computers in section 5.3. The surface code is implemented in a 2-d lattice of physical qubits, which can be either data qubits or measurement qubits. The measurement qubits are used to maintain the state of the data ones, so they are of two types: some perform X-measurements (X-syndrome qubits) and some perform Z-measurements (Z-syndrome qubits). We will see a type of surface code in section 5.4, called heavy hexagon code, that we used to hypothesize the number of physical qubits needed for Shor's algorithm implementation.

Chapter 3

Quantum Algorithms

In this chapter, we study the most important and ground-breaking algorithms that are found in the literature. We will see their possible use in our world and the way they can be implemented in a quantum computer describing the high-level representation of the circuit.

This is important since many algorithms are used as basic blocks for the subsequent ones: for example, Shor's algorithm starts from the Quantum Phase Estimation idea and uses the Quantum Fourier Transform.

3.1 Deutsch-Jozsa Algorithm

This algorithm [28] was the first example of a quantum algorithm that is faster than the best classical algorithm for a problem, so it was one of the important milestones of quantum computing. Given a function f which takes as input a string of bits, returns 0 or 1, and is guaranteed to be balanced (50% of the times outputs 1 and 50% of the times 0) or constant (always 0 or always 1), the problem is to determine if f is balanced or constant. The classical algorithm would need to check $2^{n-1}+1$ inputs in the worst-case scenario, while the quantum algorithm solves this problem with just one call of the function. However, it is not considered as an exponential speed-up because the probability that we have to do many checks in the classical algorithm is very low.

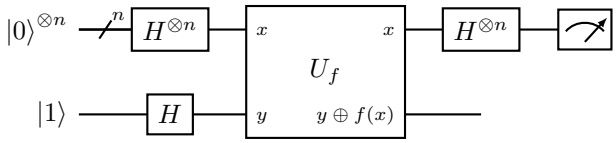


Figure 3.1. Implementation of the Deusch-Josza Algorithm.

After the application of the Hadamard gates, following equation 2.25, we have the superposition of all the possible states in the top register and the state $|-\rangle$ in the qubit. Applying the oracle the resulting state will be as in equation 3.1.

$$|\psi\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle \left(|f(x)\rangle - |1 \oplus f(x)\rangle\right) = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle \left(|0\rangle - |1\rangle\right) \tag{3.1}$$

This equation can be understood by recalling section 2.4.3 for the phase kickback, and that $f(x)$ can only be equal to 0 or 1.

At this point, we can look only yo the top register. The Hadamard gates are applied to the state, transforming it into the state in the equation below.

$$|\psi\rangle = \frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}[\sum_{y=0}^{2^n-1}(-1)^{x\cdot y}|y\rangle] = \frac{1}{2^n}\sum_{y=0}^{2^n-1}[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}]|y\rangle \tag{3.2}$$

Following the Born rule described in equation 2.8, we can calculate the probability to measure the state $|0^{\otimes n}\rangle$, that is the one obtained as done in the following one.

$$p(|0^{\otimes n}\rangle) = |\langle 0^{\otimes n}|\psi\rangle|^2 = |\frac{1}{2^n}\sum_{y=0}^{2^n-1}[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}]\langle 0^{\otimes n}|y\rangle|^2 \tag{3.3}$$

The component $\langle 0^{\otimes n}|y\rangle$ result 1 only in the case $|y\rangle = |0^{\otimes n}\rangle$, 0 otherwise, so the probability to measure this state will be one if the function is constant and 0 if it is balanced, as expressed in equation 3.4.

$$|\frac{1}{2^n}\sum_{y=0}^{2^n-1}[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}]\langle 0^{\otimes n}|y\rangle|^2 = |\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|^2 = \begin{cases}1, & \text{if } f \text{ is constant}\\ 0, & \text{if } f \text{ is balanced}\end{cases} \tag{3.4}$$

since $\sum_{x=0}^{2^n-1}(-1)^{f(x)} = \begin{cases}+2^n, & \text{if } f(x)=0\\ -2^n, & \text{if } f(x)=1\\ 0, & \text{if } f(x) \text{ is balanced}\end{cases}$

The algorithm works because, when the oracle is balanced, before the measurements we obtain a state that is surely orthogonal to $|0^{\otimes n}\rangle$, thanks to the phase kickback.

3.2 Bernstein-Vazirani algorithm

This algorithm [29] solves a problem that is similar to the Deutsch-Jozsa one, but the function, instead of being balanced or constant, performs the bit-wise product of the input for the string s ($f(x) = s \cdot x \pmod 2$): the problem lies in finding this string. Classically, the function must be run n times, since we would have to try all the inputs with just one bit to 1, so that we find the bits of the string s one by one, while the quantum algorithm needs only one call to $f(x)$. The quantum circuit is the same as the one we have seen for the Deutsch-Jozsa algorithm, but the quantum oracle changes, providing the transformation in equation 3.5.

$$|x\rangle \rightarrow (-1)^{s\cdot x}|x\rangle \tag{3.5}$$

In this way, after the application of the oracle, the state of the input register becomes the one in equation 3.6.

$$\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}(-1)^{s\cdot x}|x\rangle \tag{3.6}$$

From this, we can obtain s simply by applying the Hadamard gates and measuring the state.

3.3 Simon's Algorithm

This algorithm [30] was the first quantum algorithm with an exponential speed-up with respect to its classical counterpart. In the Simon' problem, we have a function f that can perform a one-by-one mapping (we have a unique output for every different input) or a two-by-one mapping (we have a unique output for exactly two different inputs). In the second case, if the inputs x_1 and x_2 have the same output, than $x_1 \oplus x_2 = b$, where b is a hidden bitstring. We have to determine if f is one-to-one or two-to-one and, in the second case, find b. We will see that the algorithm just needs to find b since the one-to-one mapping lies in the case where $b = 000..00$. Classically, we have to check up to $2^{n-1}+1$ inputs, where n is the bit length of b, especially in the case where we have a one-to-one mapping, while using the quantum algorithm we need just a number of runs of the oracle that is linear on n.

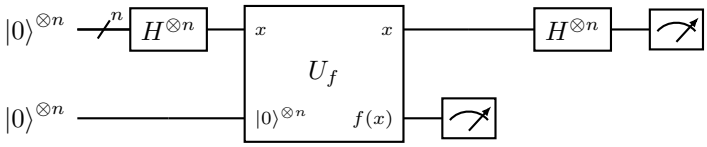


Figure 3.2. Implementation of the Simon's Algorithm.

As done in the other algorithms, we first apply the Hadamard gates following equation 2.25, so that the superposition of all the possible states is created on the first register. Now the oracle applies $f(x)$ on the second register so that the state of the system evolves in the one in the following equation.

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (|x\rangle |f(x)\rangle) \tag{3.7}$$

Performing the measurement of the second register, which can result in either x or $y = x \oplus b$, the first register will collapse to the one of equation 3.8.

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|x\rangle + |y\rangle) \tag{3.8}$$

Now we apply the Hadamard gates in order to return to the $\{|0\rangle, |1\rangle\}$ basis, obtaining the following state.

$$|\psi\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} [(-1)^{x\cdot z} + (-1)^{y\cdot z}] |z\rangle \tag{3.9}$$

In this situation, a state can be measured only if $(-1)^{x\cdot z} = (-1)^{y\cdot z}$, which means:

$$x \cdot z = y \cdot z \quad \Rightarrow \quad x \cdot z = (x \oplus b) \cdot z \quad \Rightarrow$$
$$\Rightarrow \quad x \cdot z = x \cdot z \oplus b \cdot z \quad \Rightarrow \quad b \cdot z = 0 \pmod 2$$

We will in this way measure a string z that will be orthogonal to b, so repeating the circuit a sufficient number of times we will obtain a system of n equations and n variables, that can be resolved by Gaussian elimination:

$$\begin{cases} b \cdot z_1 = 0 \\ b \cdot z_2 = 0 \\ \quad \cdots \\ b \cdot z_n = 0 \end{cases}$$

3.4 Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is one of the building blocks that we will use in the next algorithms, including Shor's one, so it is particularly important to understand it for this thesis work. It permits to bring a state in the $\{|0\rangle, |1\rangle\}$ basis (or computational basis) to the $\{|+\rangle, |-\rangle\}$ basis, denoted as Fourier basis.

$$QFT\,|x\rangle = |\tilde{x}\rangle \tag{3.10}$$

If in the computational basis we store number as binary numbers, in the Fourier basis every qubit, in order to store the state $|x\rangle$ follows equation 3.11, which means that every qubit will perform an increasing rotation around the X-axis to store a bigger x.

$$|\tilde{x}\rangle = \frac{1}{\sqrt{N}} \bigotimes_{k=1}^{n} (|0\rangle + e^{\frac{2\pi i}{2^k} x} |1\rangle) \tag{3.11}$$

For example, using 4 qubits, qubit 0 will rotate of $\frac{\pi}{8}$ every time x increases by 1, qubit 1 of $\frac{\pi}{4}$, qubit 2 of $\frac{\pi}{2}$ and qubit 3 of π. For example, we can see in figure 3.3 how the rotations are applied depending on the input.

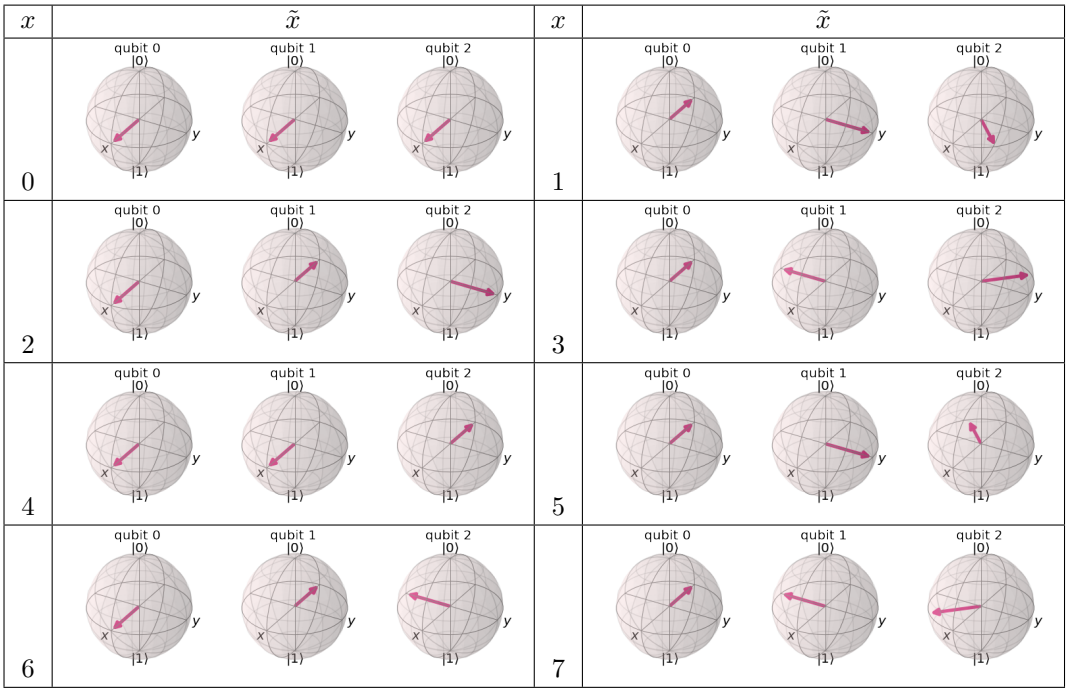


Figure 3.3. Visualization of the QFT rotations on a 3 qubit register, that can encode numbers from 0 to 7.

The circuit to implement the QFT is quite simple, as it uses only Hadamard and controlled rotation (CROT) gates. The CROT gates are defined as:

$$CROT_k = \begin{bmatrix} I & 0 \\ 0 & UROT_k \end{bmatrix}, \text{ s.t. } UROT_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^k}} \end{bmatrix} \tag{3.12}$$

$$CROT_k \left|1x_j\right\rangle = e^{\frac{2\pi i}{2^k} x_j} \left|1x_j\right\rangle \tag{3.13}$$

The circuit of a QFT applied on n qubits is described in figure 3.4.

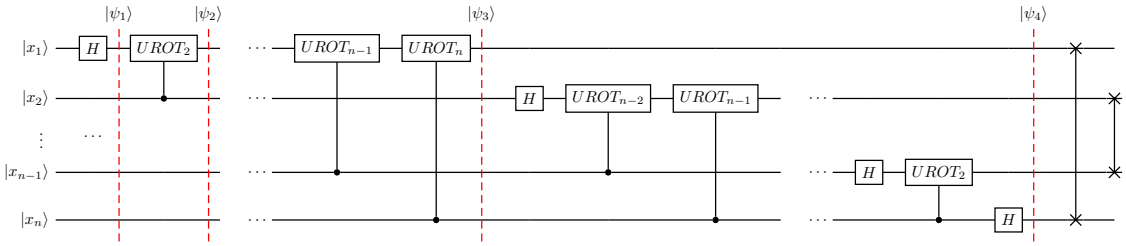


Figure 3.4. Implementation of the QFT.

We start applying the Hadamard gate on $|x_1\rangle$, leaving the system in the following state:

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}[|0\rangle + e^{\frac{2\pi i}{2} x_1} |1\rangle] \otimes |x_2 ... x_n\rangle$$

Then the first UROT gate, controlled by the second qubit, is applied:

$$|\psi_2\rangle = \frac{1}{\sqrt{2}}[|0\rangle + e^{\frac{2\pi i}{2^2} x_2 + \frac{2\pi i}{2} x_1} |1\rangle] \otimes |x_2 ... x_n\rangle$$

After the application of all the CROT gates applied on the first qubit, we have:

$$|\psi_3\rangle = \frac{1}{\sqrt{2}}[|0\rangle + e^{\frac{2\pi i}{2^n} x_n + \frac{2\pi i}{2^{n-1}} x_{n-1} + ... + \frac{2\pi i}{2^2} x_2 + \frac{2\pi i}{2} x_1} |1\rangle] \otimes |x_2 ... x_n\rangle$$

Being that every number x can be written as $2^{n-1}x_1 + 2^{n-2}x_2 + ... + 2^1 x_{n-1} + 2^0 x_n$, we can write the equation below in the form:

$$|\psi_3\rangle = \frac{1}{\sqrt{2}}[|0\rangle + e^{\frac{2\pi i}{2^n} x} |1\rangle] \otimes |x_2 ... x_n\rangle$$

We repeat the procedure for all the qubits, obtaining the following state.

$$|\psi_4\rangle = \frac{1}{\sqrt{2}}[|0\rangle + e^{\frac{2\pi i}{2^n} x} |1\rangle] \otimes \frac{1}{\sqrt{2}}[|0\rangle + e^{\frac{2\pi i}{2^{n-1}} x} |1\rangle] \otimes ... \otimes \frac{1}{\sqrt{2}}[|0\rangle + e^{\frac{2\pi i}{2^2} x} |1\rangle] \otimes \frac{1}{\sqrt{2}}[|0\rangle + e^{\frac{2\pi i}{2} x} |1\rangle]$$

In the end, we just swap the qubits in order to reorder them correctly.

As we will see in the next sections, the importance of the QFT for the quantum world is very high: it permits to convert an information encoded in the phase of a state, that is not observable, into something that is encoded in the $\{|0\rangle, |1\rangle\}$ that can be easily measured [31].

3.5 Quantum Phase Estimation

The first application of the QFT is the Quantum Phase Estimation (QPE) algorithm, which permits to perform an estimation of θ in the equation $U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$, where $|\psi\rangle$ is an eigenvector and $e^{2\pi i\theta}$ is its eigenvalue. In this algorithm, we use the phase kickback method (2.4.3) to encode the phase of the gate U in the Fourier basis to the t control qubits. So, after the application of the inverse QFT, we can measure it in the computational basis to retrieve the vale of θ.

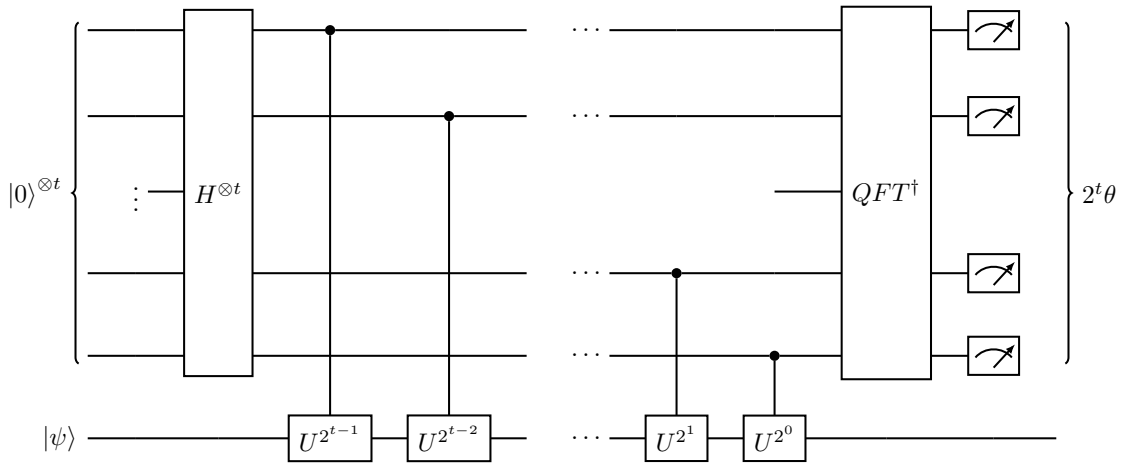


Figure 3.5. Implementation of the QPE Algorithm.

The application of the Hadamard gates creates the superposition of all the possible states in the top register, as described in equation 2.25. Now 2^t U gates are applied, each one of a different eigenvalue, resulting in a transformation of the top register as in equation 3.14. This equation has the same form as the one of equation 3.11, just using $2^t\theta$ instead of x, so we can understand why the inverse QFT is performed.

$$|\psi\rangle = \frac{1}{\sqrt{2^t}}(|0\rangle + e^{2\pi i\theta 2^{t-1}}|1\rangle)\otimes(|0\rangle + e^{2\pi i\theta 2^{t-2}}|1\rangle)\otimes\cdots\otimes(|0\rangle + e^{2\pi i\theta 2^{1}}|1\rangle)\otimes(|0\rangle + e^{2\pi i\theta 2^{0}}|1\rangle) \quad (3.14)$$

Applying the inverse QFT to this state, the register will encode the number $2^t\theta$, so we can measure the qubits to know it.

3.6 Shor's Algorithm

In this section, we will briefly explain the theory of the quantum circuit for Shor's algorithm [7], which permits us to find the order of the function $a^r \pmod N$, i.e. the smallest r s.t. $a^r \equiv 1 \pmod N$, as graphically described in figure 3.6.

Now we just describe the methods that are used, as we will provide a more complete and exhaustive explanation in chapter 8.

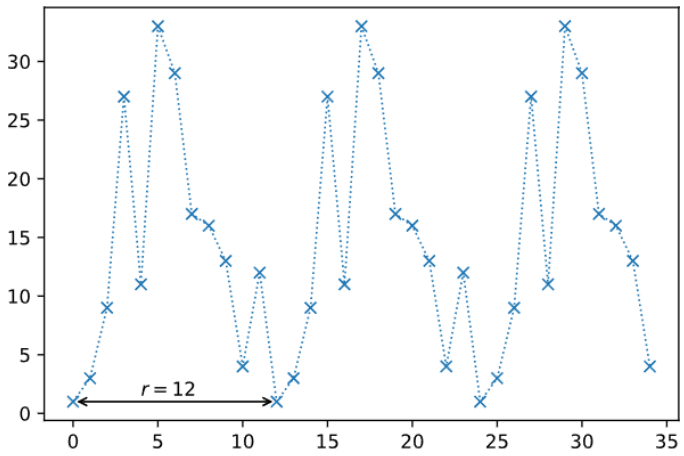


Figure 3.6. Visualization of the period of the modular function $f(x) = 3^x \pmod{35}$. Source: [10]

To solve the order finding problem, Shor used QPE on the matrix U in equation 3.15: using this operator, if we start from state $|1\rangle$, after r iterations we will come back to the state $|1\rangle$ since $U^r |1\rangle = |1\rangle$.

$$U |y\rangle = |ay \pmod{N}\rangle \tag{3.15}$$

Applying successively the operator U, we will obtain a superposition of the states $a^k \pmod{N}$, so an eigenstate of U would be the one described in equation 3.16, but this is not interesting since we can not extract any phase from it. In order to select a good eigenstate, we use one that depends on r and on an integer number s, s.t. $0 \leq s \leq r-1$, so we can have a different eigenstate for each value of s, as we can see in equation 3.17.

$$|u_0\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |a^k \pmod{N}\rangle \tag{3.16}$$

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-\frac{2\pi i s k}{r}} |a^k \pmod{N}\rangle \tag{3.17}$$

$$U |u_s\rangle = e^{\frac{2\pi i s}{r}} |u_s\rangle \tag{3.18}$$

Applying QPE to the unitary operator, will give as a result the angle $\phi = \frac{s}{r}$, so we can use the method of the continued fractions to retrieve r, as we will see in chapter 8.

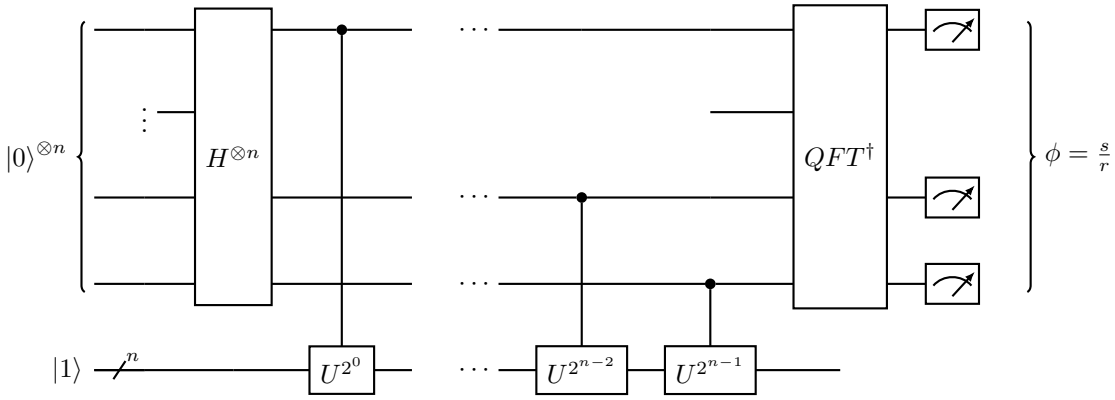


Figure 3.7. Implementation of the Shor's Algorithm.

Actually, in order to apply $U^{2^{n-1}}$, we do not have to repeat U 2^{n-1} times, since a is a classical value and we can use the repeated squaring method to calculate a^{2^x} classically in an efficient way.

$$U^{2^x} |y\rangle = |a^{2^x} y \pmod N\rangle \tag{3.19}$$

3.7 Grover's Algorithm

Grover's algorithm is one of the algorithms that made the interest towards quantum computing grow since it enables to perform research in an unsorted database in $\mathrm{O}(\sqrt{\mathrm{N}})$ time, while classical algorithms need O(N) time. It can also be used for other purposes since it is a general method to speed-up many other algorithms.

It solves the problem where we want to find an item w in a list of N items. To find its location, we will use Grover's amplitude amplification trick, without doing any assumptions on the list. We create an oracle that adds a negative phase to the solution state, as in the following equation.

$$U_f |x\rangle = \begin{cases} |x\rangle & \text{if } x \neq w \\ -|x\rangle & \text{if } x = w \end{cases} = 1 - 2 |w\rangle \langle w| \tag{3.20}$$

We can construct this oracle using phase kickback, as done in the Deutsch-Jozsa Algorithm in section 3.1. As we can see from the third member of the equation, it can be seen as a reflection of the state at $|w^{\perp}\rangle$: this will be helpful since we will study the meaning of the circuit geometrically. Another important piece of the circuit is the oracle U_{f_0}, which performs the function in equation3.21.

$$U_{f_0} |x\rangle = \begin{cases} |x\rangle & \text{if } x = 0 \\ -|x\rangle & \text{if } x \neq 0 \end{cases} = 2 |0\rangle \langle 0|^{\otimes n} - 1 \tag{3.21}$$

At the beginning, we don't know where the element is, so we must create a superposition of all the possible location using Hadamard gates, obtaining the following state.

$$|s\rangle \coloneqq \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \tag{3.22}$$

Now the amplitude amplification will transform the probability amplitudes, reducing the ones of the non-marked items and improving the one of the target, so we run many times this procedure

until we obtain the wanted probability to measure it. This process is explained in detail in the Qiskit textbook.

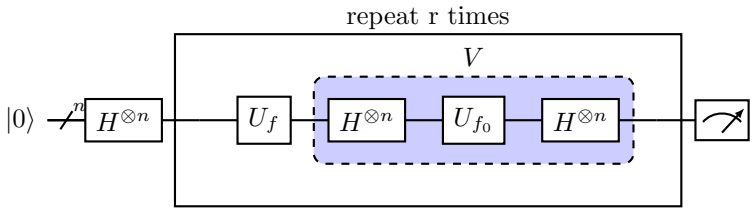


Figure 3.8. Implementation of the Grover's Algorithm.

The V-gate is a reflection of the state at $|s\rangle$, as expressed in the equation below.

$$V \coloneqq H^{\otimes n} \cdot U_{f_0} \cdot H^{\otimes n} = H^{\otimes n} \cdot 2\,|0\rangle\langle 0|^{\otimes n} \cdot H^{\otimes n} - H^{\otimes n} \cdot H^{\otimes n} = 2\,|s\rangle\langle s| - 1 \tag{3.23}$$

In order to see graphically the application of these gates, we must define the plane Σ, that is the one that is formed by $|s\rangle$ and $|w\rangle$, where we can define $|w^{\perp}\rangle$ as in equation 3.24.

$$|w^{\perp}\rangle \coloneqq \frac{1}{\sqrt{2^n - 1}} \sum_{x \neq w} |x\rangle \tag{3.24}$$

In this plane, the state $|s\rangle$ of equation 3.22 will be equal to the one in equation 3.25, that we can express defining θ such that $\sin\frac{\theta}{2} = \frac{1}{\sqrt{N}}$ as in equation 3.26.

$$|s\rangle = \sqrt{\frac{2^n - 1}{2^n}}\,|w^{\perp}\rangle + \frac{1}{\sqrt{2^n}}\,|w\rangle \tag{3.25}$$

$$|s\rangle = \cos\frac{\theta}{2}\,|w^{\perp}\rangle + \sin\frac{\theta}{2}\,|w\rangle \tag{3.26}$$

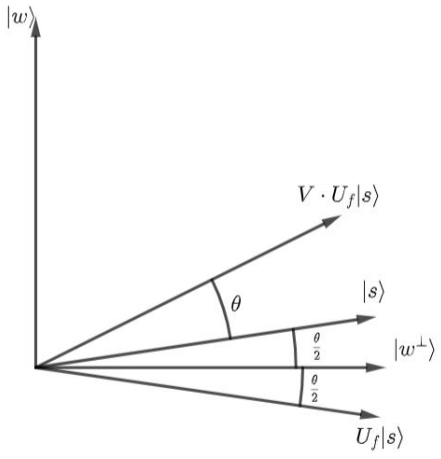


Figure 3.9. Representation of the reflections of the state

We can now follow the protocol step-by-step, also looking at figure 3.9:

1. Apply the Hadamard gate to obtain $|s\rangle$;
2. Perform the reflection at $|w^{\perp}\rangle$;
3. Perform the other reflection at $|s\rangle$;
4. Repeat steps 2 and 3 r times.

In step 4 we rotate the state by $r \cdot \theta$, choosing r s.t. $r \cdot \theta + \frac{\theta}{2} \approx \frac{\pi}{2}$. In this way, r will be:

$$r = \frac{\pi}{2\theta} - \frac{1}{2} = \frac{\pi}{4 \cdot \arcsin \frac{1}{\sqrt{N}}} - \frac{1}{2} \xrightarrow{N \text{ large}} r = \frac{\pi}{4}\sqrt{N} \tag{3.27}$$

We can see from this equation why this algorithm runs in $O(\sqrt{N})$ time. Each time we apply the two oracles, the probability amplitude of the wanted item rises, while the one of the other items goes down.

This algorithm can be easily applied also in the case where we have M marked elements, just defining the state $|w\rangle$ as in the following equation.

$$|w\rangle := \frac{1}{\sqrt{M}} \sum_{i=1}^{M} |w_i\rangle \tag{3.28}$$

In this way, the other states will be:

$$|w^{\perp}\rangle := \frac{1}{\sqrt{N-M}} \sum_{x \notin \{w_1,\ldots,w_M\}} |x\rangle \tag{3.29}$$

$$|s\rangle := \sqrt{\frac{N-M}{N}}\,|w^{\perp}\rangle + \frac{M}{\sqrt{N}}\,|w\rangle = \cos\frac{\theta}{2}\,|w^{\perp}\rangle + \sin\frac{\theta}{2}\,|w\rangle \tag{3.30}$$

The convergence of the algorithm is quicker, as it needs just $O(\sqrt{N/M})$ iterations.

We will study this algorithm more deeply in chapter 7, as it represents a threat for the modern cryptographic hash and symmetric algorithms.

3.8 Quantum Counting

Quantum counting [32] is an algorithm that counts the marked elements of a list. It combines QPE with Grover's algorithm since we just use QPE to find an eigenvalue of a Grover search iteration. Using the $\{|w\rangle,|w^{\perp}\rangle\}$ basis, we can write an iteration of the Grover algorithm as a matrix as done in the equation below.

$$G = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{3.31}$$

This matrix has eigenvectors $\begin{bmatrix} -i \\ 1 \end{bmatrix}$ and $\begin{bmatrix} i \\ 1 \end{bmatrix}$ and eigenvalues $e^{\pm i\theta}$, so when we measure the register after the QPE algorithm we will obtain either $+\theta$ or $-\theta$.

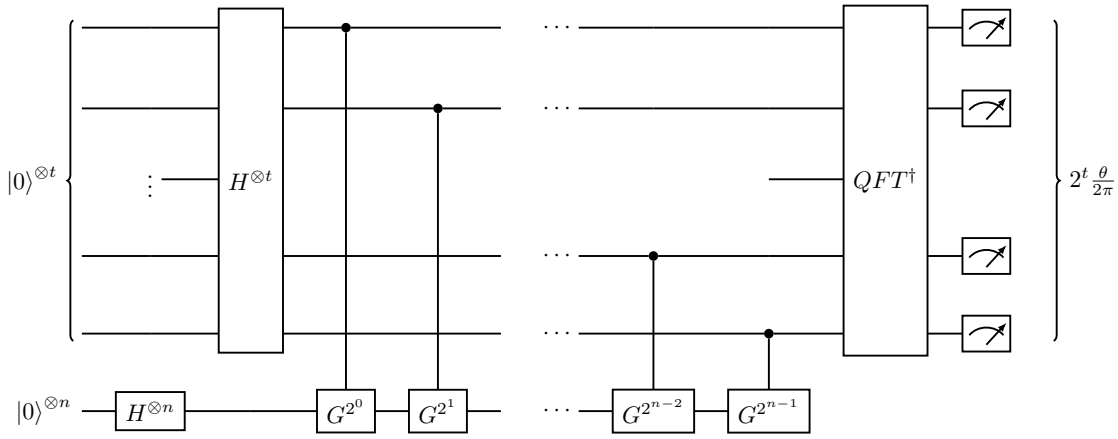


Figure 3.10. Implementation of the Quantum Counting Algorithm.

Now that we know θ, we can obtain M from equation 3.30, as done in the following equation.

$$\sqrt{\frac{N-M}{N}} = \cos\frac{\theta}{2} \rightarrow M = N\sin^2\frac{\theta}{2} \tag{3.32}$$

If in the classical case the number of marked elements can be found only by performing a lookup of the whole database, this algorithm performs the task with a logarithmic complexity on the number of elements of the database.

This application of Grover's algorithm can be used also to check the existence of a solution in the functions, looking if $M \neq 0$.

Chapter 4

Quantum Computers

In May 1981 Richard Feynman gave a talk about the simulation of real physics using computers, proposing the idea of exploring quantum mechanics with a new kind of computer [33]. From that date, scientists put a lot of effort into building quantum devices capable of performing this type of computation and discovering new algorithms that take advantage of them. The continuous research brought to Shor's discoveries of an efficient algorithm for factoring large numbers, that proved the importance of this new field of computation and made the interest for quantum computing grow exponentially, and of the error-correcting codes, that gave the possibility to limit the effect of decoherence.

Today we have a situation that does not permit us to run many useful circuits due to the hardware capabilities needed: we call this period Noisy intermediate-scale quantum (NISQ) era [40]. In fact, the actual devices have a number not superior to a few hundred qubits (intermediate-scale) and are sensitive to the environment noise (noisy), so they do not permit to achieve fault tolerance or quantum supremacy [69], that is the demonstration that a programmable quantum device can solve a problem that no classical computer can solve in any feasible amount of time.

In October 2019 google claimed to have been able to resolve in 200 seconds a task using the Sycamore chip [3], when a supercomputer would take 10000 years, but IBM (that had the best supercomputer available at the time) sustained that with several optimizations that task would need just 2 years ad a half [70].

One year later, in December 2020, a team of the USTC led by Jian-Wei Pan reached quantum supremacy with the Gaussian Boson Sampling (GBS) method [71], solving in 20 seconds what a classical supercomputer would solve in 6×10^8 years of computation (a speed-up on the order of 10^{14}), and stated that important improvements to that record could be done by the development of better hardware [72]. The GBS is a variant of the Boson Sampling, where non-classical light is injected into a linear optical network, and the output is measured by single-photon detectors.

The same team in October 2021 reported that a new quantum computer had reached a speed-up of 10^{24} against the classical supercomputers using the GBS method, while another team (led always by Jian-Wei Pan) was able to improve the Sycamore computation by 3 qubits, so the same classical computation would require a computation time of 2 or 3 orders of magnitude greater [73].

These experiments prove that quantum computing will be able to overcome the classical algorithms probably also soon, providing the possibility to speed up the computations in many fields, such as pharmaceutics, finance, data center, chemistry, and many others. However, it represents

also an important problem for cybersecurity since many cryptographic algorithms rely on the exponential complexity of certain mathematical problems.

As we have seen, we have two distinct fields of research, one logical, related to the algorithms, and one physical, related to the way the devices are built. The current quantum architecture has these two worlds at the top and at the bottom of the stack, whit several layers that permit an algorithm to be implemented in a certain quantum computer.

4.1 Full-stack Quantum Architecture

In order to run a quantum algorithm, we have to develop the components in figure 4.1.

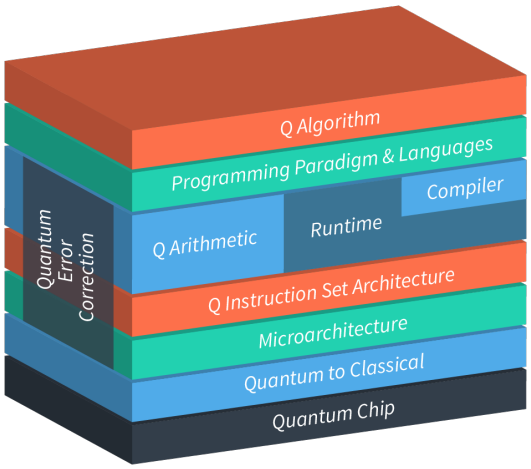


Figure 4.1. Full-stack quantum architecture. Source: [34]

The first layers see the qubits as logical and are independent of the physical realization of the device that will be used. The algorithm is written using a classic programming language, such as C++ or python, or a quantum programming language like Q#. For this purpose, IBM developed Qiskit, an SDK that uses python to describe quantum circuits that can be run in one of their systems or simulators (4.4.1).

The next layers are needed to transform the classic description of the circuit into a representation that can be used by a specific quantum device. In this context, the Quantum Error Correction algorithm used drives the transformation of the logical qubits into physical ones, while the compiler, after the classic logic compiling, translates the logical quantum operations to a series of physical ones [34]. The instructions are now described in a Quantum Instruction Set Architecture, that contains low-level directives like single-qubit gates or measurements. This language can be understood by the underlying quantum microarchitecture, which consists of two parts: one that has a classical processor to execute part of the accelerator logic and the quantum chip that contains the qubits that need to be executed in an analog way [35].

In the next sections we will propose different solutions for the layers in figure 4.1, starting from the different types of quantum hardware and the current manufacturers (4.2), and providing an overview of the current frameworks that can be used as an interface to these devices (4.4).

4.2 Quantum hardware in the NISQ era

Quantum computers are the devices that exploit the quantum properties of section 2.3 to perform the computations. The different types of quantum hardwares are classified regarding the way the qubits are created. Some of the most important technologies are shown in table 4.1.

Technology	Properties	Companies
Photons	They can work at room temperature and have weak interations with the environment and limited fault tolerance.	- Xanadu - Amazon Quantum Solutions Lab
Trapped Ions	They are manipulated by laser light, so there is no need for complicated wiring, and cooled by laser cooling not cryogenics, while the qubit storage times reaches minutes or even hours.	- Honeywell - IonQ
Superconducting qubits	They have both fast gate and operation time, but also fast decoherence time. They must be mantained below 100mK, so they need cryogenic refrigerators.	- Intel - Google - IBM - Rigetti

Table 4.1. Most known technologies to create a qubit, along with characteristics and current developer companies. Source: [37, 38, 39]

To run a circuit on a quantum device, we need to be aware of the hardware capabilities that we have, since current devices have many limitations:

- The number of qubits is at most between 10 and 100;
- The decoherence time does not permit to run long circuits;
- The gates (both singles and doubles) precision has to be considered;
- The topology of the qubits is not all-to-all, so the interaction between two not connected qubits has to be propagated through other qubits; and
- Some quantum gates may not be present.

The number of qubits of the circuit (width) must be lower than the number of qubits of the quantum computer and the number of gates should consider their precision. Another important characteristic of a circuit is the depth, that is the number of gates on the longest path in the circuit since many gates can be done in parallel: if the decoherence time is too small with respect to the depth of the circuit the errors does not permit to retrieve good results.

4.2.1 Manufacturers

Now we will see the properties of the most important quantum manufacturers seen in table 4.1 and their solutions, in order to confront them.

IonQ developed the first Reconfigurable Multicore Quantum Architecture (RMQA)[44], providing a technology that permits to change the topology of the device. This brought to an 11-qubit fully-connected quantum processor, but the number of qubits is expected to pass the one hundred wall in the next future. They provide cloud access with AWS, Azure, Google, and Qiskit, so their hardware is accessible from nearly every provider available.

Rigetti[45] develops superconducting qubit-based quantum processors and supports integration with a wide variety of classical resources through network APIs. These APIs can be used by their SDK, enabling developers to construct their own software. Their most performing machine is called "Aspen 8", a machine with 31 physical qubits and fast gates.

Honeywell[46] built the Honeywell System H1, a 10 qubit device with high-fidelity, fully connected qubits and features like mid-circuit measurement and qubit reuse, that was the first one to obtain a quantum volume (a metric that we will describe in section 4.2.2) of 128. This system has cloud access and is compatible with many frameworks. Mid-Circuit Measurement is a unique feature that allows qubits to be measured during the circuit, while the non-measured qubits maintain their quantum state: we will better understand the importance of this property in section 8.4.1.

The IBM Quantum Experience is a cloud-based solution that permits the users to have free access to real quantum hardware and powerful quantum simulators made available by IBM. The IBM Q project estimate to be able to break the wall of a device with 1000 superconducting qubits for 2023, but nowadays their most powerful quantum device has 27 qubits, but they provide access also to devices with 65 and 127 qubits. They offer a big variety of topologies and different kinds of connectivities between the qubits so that the user can choose the most suitable one for his purpose. They provide also 5 simulators, such as the ibmq_qasm_simulator, that allows simulating quantum systems of a maximum of 32 qubits: we will use it to test our algorithms in the next chapters. The users can interact with the devices using the quantum circuit model of computation that we have seen so far, so we need a way to code our algorithms in circuits that can be passed to the devices.

As quantum computers become more complex and capable of performing bigger circuits, we would need a fast way to understand the capabilities of a quantum device, so we need a measure of its quality.

4.2.2 Quantum computers capability metrics

The Quantum Volume (QV)[41] is the IBM proposal to give a single-number measure to the capabilities of a quantum device. It has been initially thought for near-term quantum computers with less than 50 qubits, but the principals can be expanded also for bigger devices. It quantifies the largest random circuit of equal width and depth that the computer successfully implements with good precision, so it indicates in a fast manner the size of the circuits that it can run. It is defined as in the following equation.

$$\log_2 QV = \arg\max_m \min(m, d(m)) \tag{4.1}$$

so it is the maximum m (width) of the minimum value between m and $d(m)$ (depth) of all the circuit that produced a correct result with good probability. In order to calculate the good result,

the circuits are runned in a quantum simulator, and this is one of the reasons why it will be difficult to use this measure for the future quantum computers.

Date	QV measured (circuit size)	Manufacturer	Device and qubits number
2020, January	32 (5x5)	IBM	Raleigh, 28 qubits
2020, June	64 (6x6)	Honeywell	6 qubits
2020, August	64 (6x6)	IBM	27 qubits
2020, November	128 (7x7)	Honeywell	System Model H1, 10 qubits
2021, March	512 (9x9)	Honeywell	System Model H1, 10 qubits
2021, July	1024 (10x10)	Honeywell	Honeywell System H1, 10 qubits

Table 4.2. Hystorical records for QV, along with manufacturer and device. Source: Wikipedia, free contributors.

If the QV measures the quality of a quantum computer, we need a measure to quantify the performance in terms of speed of it, that is the number of QV circuits that the system can execute per unit of time. IBM proposed a method to measure the Circuit Layer Operations Per Second (CLOPS) [42], and used it to calculate the performances of its systems. They define CLOPS as "the number of QV layers executed per second using a set of parameterized QV circuits, where each QV circuit has $D = \log_2 QV$ layers". They use 100 parametrized template circuits and measure the time needed to run each of them 10 times. A PRNG is used to take the parameters for the 100 parametrized circuit, which can now be executed using 100 shots. The results of the circuits are used as seed for the PRNG to calculate the parameters to re-run the circuits again.

$$\text{CLOPS} = \frac{M \times K \times S \times D}{\text{time\_taken}} \tag{4.2}$$

The clops can now be calculated as in equation 4.2, where M is the number of templates (100), K is the number of parameter updates (10) and S is the number of shots (100).

4.3 Quantum Circuits Simulation

The quantum algorithms described in chapter 3 would need hardware that is not available nowadays if the number of qubits needed starts to increase. Also, as we will see in chapter 5, the quantum architectures are very susceptible to errors and this makes many computations hard to be completed. In this context, how can an algorithm be tested and verified? We have already seen that quantum states and quantum operators can be described as vectors and matrices respectively and that the result of the application of an operator to a state is the multiplication between this two. These elements can be easily described in a classical computer, which therefore can be used as a quantum simulator, not susceptible to errors and where the programmer can easily check step-by-step the evolution of the states. This technique is used for many purposes, for example, it can provide a good base for the development of error correction codes, help to see if a quantum speed-up is really possible for a certain circuit, or verify the correctness of the real quantum hardware output for a certain circuit to test and benchmark it. In addition, it is possible to simulate a certain error, in order to estimate how a circuit will behave in a real quantum architecture. As it does not rely on real hardware, the simulation will not take into account the limited connectivity between qubits typical of the quantum devices but will perform the computations as if qubits are connected all-to-all.

However, since the vectors that represent the qubit states grow exponentially in the number of qubits (2^n), the application of an operator would require a multiplication between a matrix of dimension $2^n \times 2^n$ and a vector of 2^n elements. For this reason, even modern computers can arrive to compute the simulation of a maximum of approximately 30 qubits, while supercomputers with petabytes of memory can simulate up to 50 qubits. A solution that mitigates this problem is based on decision diagrams, that permit to reduce both the memory requirements and the simulation time [43].

4.4 Frameworks for quantum computing

Cirq is the google quantum computing framework solution, a python library for writing and optimizing quantum circuits, and then running them on quantum computers and quantum simulators. This library introduces the concept of Moment, which is a set of gates operating on different qubits that can be operated in parallel: even if constructing circuits in this way is time-consuming, this provides a low-level optimization method easy to understand and to use, as well as a fast method to slice and invert the circuits. The library provides a pure-states and a mixed-states simulator for small circuits, but also an interface for some of the most important quantum devices in the cloud (IonQ, Google Quantum Computing Service, Pasqal, and Alpine).

The Microsoft Quantum Development Kit is the framework for Q#, a quantum programming language provided as a downloadable extension for Visual Studio, and for Azure Quantum, the Microsoft cloud solution that works as an interface for Honeywell and IonQ devices. The framework provides also methods for resource estimation and modules for some specific fields that are of interest in quantum computing.

The Amazon braket framework is another SDK solution that permits to simulate and execute quantum algorithms built on Jupiter notebooks. This framework provides three cloud simulators that can be of help in different situations:

- a state vector simulator (SV1) that takes the full wave function of the quantum state and applies the operations of the circuit to calculate the result for circuit up to 34 qubits;
- a density matrix simulator (DM1) where the user can specify realistic error models and study the impact of noise on the algorithms for circuits up to 17 qubits; and
- a tensor network simulator (TN1) that encodes quantum circuits into a structured graph to find the best way to compute the outcome for circuits up to 50 qubits in size, but with a small depth.

For the execution on real quantum devices, it provides an interface for IonQ, Rigetti, and D-wave.

4.4.1 Qiskit

The Quantum Information Software Kit, or Qiskit, is an open-source development kit that provides a set of tools and APIs to create quantum programs at the level of circuits, with the possibility to optimize them for the wanted quantum device. It also provides an optimized simulator to execute compiled circuits, with functions to reproduce errors that often occur in real quantum computers. It is available in python, but also in Javascript and Swift. The script is then compiled into the OpenQASM (open quantum assembly) language, which is directly given to the selected device.

Qiskit is made up of four principal packages, that can interact with each other in the realization of a quantum circuit, called Qiskit elements:

- Terra permits to create circuits putting every gate step-by-step and to optimize them for a certain device. It is the foundation where the other components lie since it is the most low-level one.
- Aer contains the functions to simulate locally the circuits, providing good representations of noise models.
- Ignis is used to study and benchmark quantum devices, error mitigation algorithms, and error correction codes.
- Aqua contains a collection of algorithms of various topics, such as machine learning, chemistry finance, and optimization. It also contains well-known quantum protocols, such as Shor's algorithm, to allow high-level programming.

In addition, we have the API needed to interact with the IBMQ hardware and simulators, called IBM Quantum Provider.

An important actor that permits to run more complex circuits in quantum devices is the compiler: since the NISQ devices are non-fault tolerant, the compilers have to mitigate as much as possible the operations and the errors that arise from the interactions between the qubits. The compiler of Qiskit is constituted by the transpiler and the scheduler: the first transforms the circuit in order to be optimized for the selected device, for example swapping qubits taking into account the topology of them or analyzing the circuit depth, while the scheduler converts the circuit into a series of pulses to be executed directly to the device. For example, it is possible to perform a CNOT between any qubit of a circuit, but when the program is compiled, the Qiskit compiler converts the single two-qubit gate into a sequence of CNOT gates allowed in the connectivity: the Qiskit compiler, in fact, allows to specify an arbitrary basis gate set and topology or relies on the information given by IBM about their devices to optimize circuits for them. However, it is also possible to write quantum circuits directly in OpenQASM, as Terra provides all the functions to develop them. These operations are effective only knowing as much as possible the characteristics of the quantum device that has to be used: for this reason, the IBM Quantum Experience provides all the necessary data to the users, like the topology of the system, the properties of the single qubits and the error rates of the gates.

Given that there are various frameworks, we have to explain why we choose Qiskit to implement and simulate our circuits for Shor's algorithm. The first reason is the compatibility with the IBMQ hardware and qasm simulator, that we widely used. The qasm simulator permitted us to test our circuits using a decent number of qubits, even if the depth of these was really high. The second reason is the variety of modules that Qiskit Aqua contains, providing also a starting point for Shor's algorithm circuit for the factorization problem. Another reason why we choose to use Qiskit is the possibility to have a big community that helped me to clear every doubt that arose during the development process and a vast documentation helpful to move the first steps in the quantum computing world.

Chapter 5

Quantum algorithms on real devices

The execution of a quantum circuit in a quantum device, as we have seen in section 4.1, needs some actions to be performed, such as the transpiling, the transformation of the circuit in QASM instruction, and the readout of the results. In this chapter, we will explain the steps to create and run more in detail, taking as an example the IBM Q procedure.

5.1 Circuit definition

As explained in chapter 4, one of the methods to define a quantum circuit is using classical languages, such as python, and frameworks that contains functions to describe and execute them, like Qiskit. In this environment, we start from the definition of the quantum and classical registers, which are the sets of qubits and bits needed by the circuit, that are used as inputs for the initialization of the quantum circuit. The qubits are then reset to state $|0\rangle$ and initialized in the wanted quantum state. Now it is possible to apply the wanted quantum gates and measurements. As described in figure 5.1, Qiskit gives us the possibility to insert in the circuit quantum gates controlled by classical bits: we will use these to reduce the number of qubits needed by the Shor's circuit, but they are not yet implemented in the IBM quantum computers so, using the IBM Quantum Experience environment, we can only simulate it. The functions needed to set up the circuit are all gathered in the circuit library[1].

Now we have to choose and define the wanted backend and number of shots and execute the circuit.

5.2 Transpiling

The first step that Qiskit does when a circuit requires the execution on a quantum backend is the compilation, composed of 4 routines: assemble, schedule, transpile, and sequence. While the

[1] see https://qiskit.org/documentation/apidoc/circuit_library.html

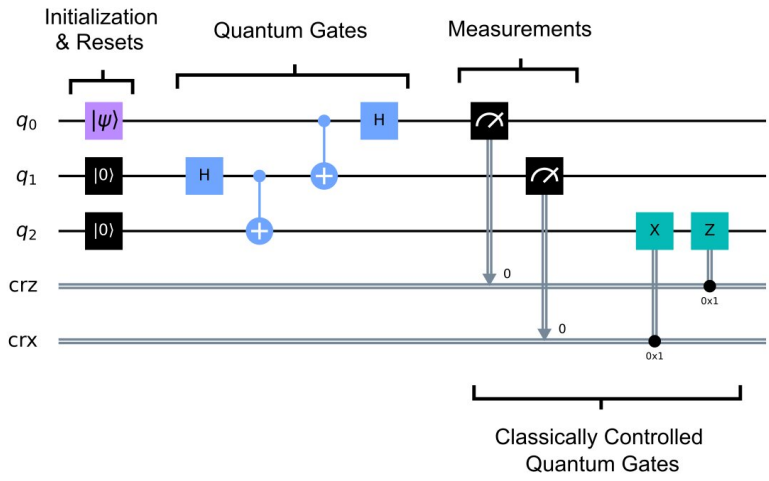


Figure 5.1. Definition steps of a quantum circuit. Source: [10]

assemble, schedule, and sequence phases are just a rewriting of the inputs, a different task is performed by the transpiler.

The basic process of transpiling is the mapping of the logical-level qubits of a circuit into the physical qubits of a specific NISQ device that has limited connectivity, but other actions are usually performed, like the transformation of the gates into a set of physical gates understandable by the device.

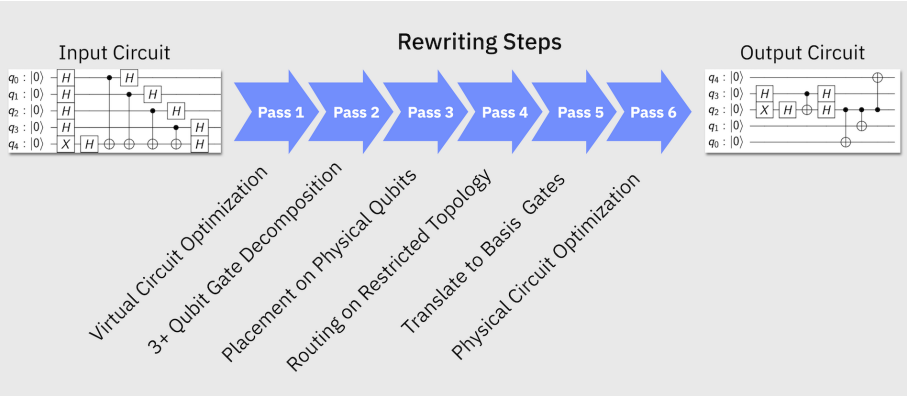


Figure 5.2. Basic building blocks of the IBM Q transpiler. The process has not to be intended as linear, but it may contain different sub-routines or loops. Source: [10]

Every quantum device does not support all the gates that we are present in the literature, but often just a universal set of basis gates due to physical limitations or to the difficulty in implementing multi-qubit gates. For example, the processors developed by IBM support only the

gates CX, ID, RZ, SX, X, so the other gates have to be redesigned into a combination of these five. This process can be quite expensive in terms of depth for a quantum circuit: for example, a SWAP is decomposed into 3 CNOT gates. Another problem resolved by the transpilation is the decomposition of multi-qubit gates using only basis gates, but in this case, the depth increases even more, since a 3-qubit Toffoli-gate is decomposed into up to 6 gates.

5.3 Topology of qubits

The mapping of the qubits is an important step in many types of quantum hardware, including the ones of the IBM that contains superconducting qubits. In section 4.2.1 we underlined the importance of the RMQA of IonQ, which permitted to have a fully-connected quantum device thanks to the properties of the trapped ions, but this is not the case for the superconducting qubits. The topology of the device strongly influences the depth of our circuits, since the communication between two non-adjacent qubits will need the interaction between all the qubits in one path (possibly the shortest) between the two.

As we will see in chapter 8, to construct efficiently the circuit for the period finding algorithm we would need a highly connected topology, so the depth of the compiled circuit will notably increase due to this consideration in an IBM quantum device.

An example is provided in the circuit 5.3, that is referenced to the architecture in figure 5.6 when there is a CNOT gate qubit 0 as control and qubit 7 as target.

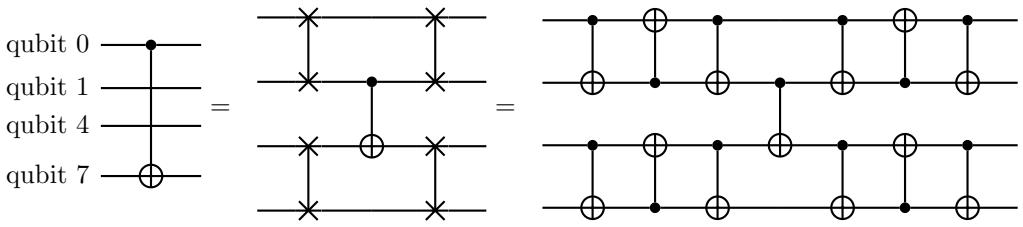


Figure 5.3. Example of scheduling of a CNOT gate applied to non-adjacent qubits, referenced to the architecture in figure 5.6 when there is a CNOT gate with qubit 0 as control and qubit 7 as target. We can see that the depth grows from 1 to 7.

With this circuit we can see how a CNOT gate between two qubits separated by other two qubits means an increase in the depth of 7 instead of 1: in our circuits we would need a highly connected structure, but the heavy hexagon code permits to have a connectivity of degree 3 between even the logical qubits, so we will need some ancillae logical qubits to reduce the distance between all of these.

The subsequent problem that comes from the limited connectivity is the qubit routing, for which we can find a solution in the Cambridge Quantum Computing's t|ket⟩ compiler [51]. The first operation that this compiler does is to slice the circuit into time steps so that the circuit is divided into partitions of gates that can be executed simultaneously. Then an initial mapping of logical qubits and physical qubits is done and a routing algorithm iteratively constructs a new circuit adapted to the desired architecture. The last operation is called SWAP synthesis and clean-up, which is needed just to rewrite the SWAP gates as CNOT structures and to clean useless gates (for example, two consecutive CNOTs).

Another possibility to solve this problem is to add ancillae logical qubits to act as bridges between two qubits that want to communicate. For example, adding a clear connectivity ancillae qubit in circuit 5.3 connected to qubits 0 and 7 will cause an increase in depth to just 3, as we see in circuit 5.4.

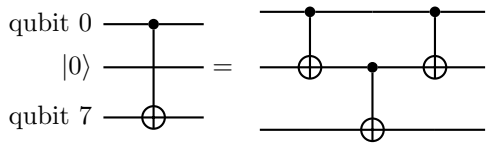


Figure 5.4. Example of scheduling of a CNOT gate applied to non-adjacent qubits, referenced to the architecture in circuit 5.6 when there is a CNOT gate with qubit 0 as control and qubit 7 as target and a clear ancillae qubit connected to these two is added. We can see that the depth grows from 1 to 3.

In a circuit with many qubits the number of ancillae to be added can be chosen with respect to the resulting depth, obtaining a trade-off between depth and width, but this is possible only if we have to develop a quantum computer for a specific purpose.

5.4 Quantum Error Correction

As we have seen in section 4.2, in the NISQ era one of the greatest obstacles to building large-scale quantum computers that can execute long circuits is the noise that affects the qubits and the gates. Quantum computations are prone to errors since each gate is slightly wrong, the qubits involved are disturbed by external forces and measurements may fail. These problems, together with the quantum decoherence, do not permit us to treat the qubits as perfect, and this represents a problem for the algorithms that we have seen until now. To apply them, we need a way to create a qubit that can be considered as ideal, that will be called "logical qubit", starting from physical ones, as we have described in section 2.7.

Taking into account the Error Correction Code is very trivial for the transpiling process, as the physical operations change significantly: in fact, not only the logical gates will be more complex, but many actions have to be performed just to maintain the correct state in the qubits. The transpiling process is an NP-hard problem: adding variables makes the optimal solution exponentially harder to be discovered, so heuristics have been defined for this, and performing the transpiling algorithm multiple times does not give always the same optimized circuit.

5.4.1 Heavy Hexagon Code

In December 2019 IBM proposed the heavy hexagon code [48], a surface/Bacon-Shor code [49] that is mapped into a hexagonal lattice that includes all the ancilla qubits required for fault-tolerant error-correction. This type of code is of particular interest for us since IBM bases its newest (and future) quantum devices on it, to create logical qubits with a minimum effort. In this scheme, represented in figure 5.5, the 60% of the qubits have degree 2, while the 40% have degree 3: the overall degree is, therefore, $\frac{12}{5}$. IBM plans to use cross-resonance gates for its hardware since the researchers achieved gate fidelities around 99%, so this low degree permits to mitigate the issues of frequency collision and crosstalk [50]. The way to operate the CR gates requires that the qubit frequency must be different from all its neighbors and selected from a predefined set.

The possibility that this code gives is to have a set of only three frequencies present in the device, so the frequency collisions are maximally reduced.

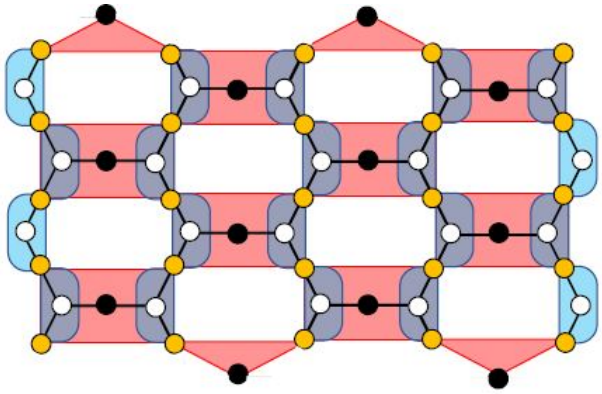


Figure 5.5. actual layout of the d = 5 heavy hexagon code which encodes one logical qubit. The data qubits are represented by yellow vertices, white vertices are the flag qubits and dark vertices represent the ancillae to measure the bit flip errors (red areas) and the sign flip errors (blue areas). Source: [48]

As described in [48, sec. II.A], the total number of qubits n_p for a given distance code d is given by the following equation.

$$n_p = \frac{5d^2 - 2d - 1}{2} \tag{5.1}$$

This number will be important in section 8.3, where we try to estimate the number of physical qubits needed for the integer factorization with Shor's algorithm. The logical error rate will be strongly affected by the distance code, since it will be calculated using the probability that $\lfloor \frac{d-1}{2} \rfloor$ errors occur simultaneously in a physical construction of a logical qubit.

In section 4.2 we have seen how the depth of the circuit is important to understand the needed performance in terms of QV and CLOPS that a certain quantum device needs to implement it. The logical error rate has to be maintained below a threshold dependent on the depth of the wanted circuit so that a computation can be completed with a good percentage of success. Given this assumption, we can estimate the distance code that the ECC should provide and in this way the number of physical qubits needed for every logical one.

For example, we may consider a circuit of depth 10^6 and width 50 and a device that has a physical error rate (PER) of 10^{-4} to calculate the number of physical qubits needed to execute the circuit with fidelity of 90%. At each step, we must assure a probability of error ρ, that is calculated as in equation 5.2.

$$\rho \simeq \frac{1 - 0{,}9}{\text{depth}} = 10^{-7} \tag{5.2}$$

Assuming to consider a distance code $d = 7$, using the formulae 5.1, we need 115 physical qubits for every logical one and a total of 575 for the whole circuit. Given that we can correct up to 3 errors in a logical qubit at each step, we can calculate the error probability on each logical qubit

$\rho_{L,d=7}$ using Binomial distribution[2], as done in the following equation.

$$\rho_{L,d=7} = 1 - \sum_{k=0}^{3} \binom{115}{k} (\text{PER})^k (1 - \text{PER})^{115-k} = 6{,}85 \cdot 10^{-10} \tag{5.3}$$

Having 50 logical qubits, the probability of error at each step $\rho_{d=7}$ is therefore:

$$\rho_{d=7} = 1 - (1 - \rho_{L,d=7})^{50} = 3{,}42 \cdot 10^{-8} \tag{5.4}$$

Given that $\rho_{d=7} < \rho$, we proved that the usage of this distance code is sufficient.

The Falcon and the Hummingbird quantum processors of the IBM, used by many of their quantum computers, are already based on this code, as we can see from the configuration of the ibmq_montreal in figure 5.6. .

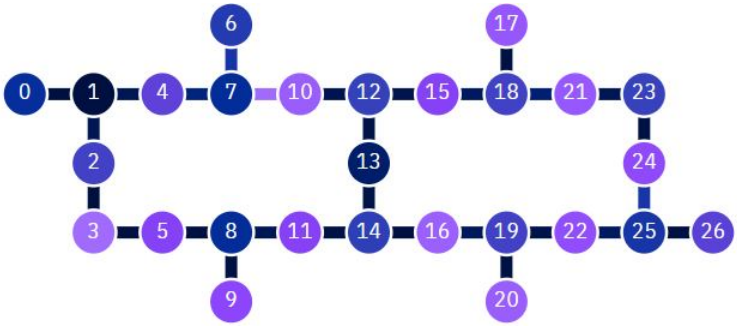


Figure 5.6. Physical topology of the ibmq_montreal, implementing the heavy hexagon code. Source: [2]

In Qiskit we have a useful function to calculate the depth of a circuit before the compiler transformations in the Terra module: "QuantumCircuit.depth()".

5.5 Execution of quantum circuits

The output of the transpiler is a series of gates, that are transformed into instructions directly passed to the microarchitecture of the quantum device, defined using a type of language called Quantum Assembly (QASM). IBM, in order to be used with the Qiskit framework, developed the OpenQASM language[52], which is currently at version 3.0[53] and defines quantum circuits as computational routines consisting of an ordered sequence of quantum gates, measurements, and resets on qubits.

This representation can be also directly hand-written so that we can avoid unwanted transformation of the compiler, using the functions that the Qiskit Terra module contains.

The OpenQASM program is classically compiled to generate a collection of quantum objects (circuits and gates) with no branches and classical control instruction, which is then transformed into low-level physical instruction by a controller that is able to execute them in the quantum

[2]see https://en.wikipedia.org/wiki/Binomial_distribution

device retrieving the results of the measurements. These results are post-processed in a classical computer for further circuit generation in the case of quantum gates controlled by classical bits or for retrieving the final result of the quantum computation.

In Qiskit, one option to visualize the result is the histogram representation, where it is possible to clearly see the most occurred results, using the "plot_histogram" function, but other visualization options are present[3].

[3]see https://qiskit.org/documentation/apidoc/visualization.html

Chapter 6

Public-key cryptography

Shor's algorithm, for which we have briefly analyzed the high-level quantum circuit in section 3.6, permits to break many algorithms that are used as basic foundations for public-key cryptography. To understand these threats, we have to analyze their mathematical basis and study their usage in our current digital world to see the impact that the possibility to break them would have. For these reasons, a review of asymmetric cryptography is provided, focusing on the algorithms that rely on mathematical problems breakable using quantum computers.

6.1 Introduction on asymmetric cryptography

In public-key cryptography, or asymmetric cryptography, every actor of the communication has a pair of keys, that are:

- the public key, that must be distributed to every member of the communication;
- the private key, that is somehow correlated to the public key and that must remain secret.

Using them, it is possible to avoid the problem of key distribution, typical of the symmetric cryptosystems. If a public key is used to cipher the message, then it is possible to decipher it only with the private counterpart of the public key used, and vice-versa: this is possible thanks to some mathematical tricks that we will see in the next sections of this chapter. These keys can be used for two main purposes:

- encryption, where if the sender encrypts the message with the public key of the wanted receiver, then he can be sure that only him can decrypt the message with his personal private key;
- digital signature, where if somebody encrypts a message, or commonly the digest of a document, with his private key, then whoever can be sure that he is the one who encrypted the message since it is possible to decrypt it only using the corresponding public key.

As we said, asymmetric cryptography solves one of the most important problems of symmetric cryptography, but it has a disadvantage: its complexity makes it slow, so it is suitable just for small quantities of bits. This is the reason why we usually create the digest in the digital signature

and we perform the encryption not directly of the message but of a secret bit string to establish a symmetric key, that will be then used to encrypt the message, in a secure way. In addition, to obtain a similar level of security, the keys used in the asymmetric algorithms are an order of magnitude bigger than the keys of the symmetric ones.

Type	Bit length			
Symmetric	40	64	128	256
Asymmetric	256	512	1024	2048

Table 6.1. Comparison between the key lengths with similar resistance for asymmetric and symmetric algorithms.

The security of the algorithms that use this kind of cryptography is given by finding the private key starting from the knowledge of the public key, so it is related to the mathematical foundations of the key generation. The strong cryptographic algorithms are the ones that require to solve a known hard problem in the classic computation theory in order to be broken, like the factorization problem and the discrete logarithm problem.

6.2 Key exchange problem

An important problem of symmetric algorithms is the need for a common key between the parties of the communication. A first solution can be to exchange those keys out-of-band, but most of the current uses of the symmetric algorithms need a more automatic way to do it. A second solution is to use public-key cryptography to encrypt the symmetric key, as we already mentioned, but there are some cases where one party does not want to trust the key created by the other party. A key-agreement algorithm permits, starting from a security parameter known by the two parties, to create separately two keys k_1 and k_2 such that $k_1 = k_2$, in such a way that k can be created only by the two parties together.

6.2.1 Diffie-Hellman

Diffie-Hellman (DH) was presented as a method to perform a key agreement in a public channel in 1976, and is still used nowadays for many practical operations: in TLS, for example, it is used to obtain perfect forward secrecy, an important property that permits to have the assurance that the session keys used in TLS are not compromised even if the long-term keys become compromised in the future. To achieve perfect forward secrecy, we need to have a private key that, if compromised, compromises only the current and future traffic, so it must change continuously. To assure that we use ephemeral mechanisms, we create a one-time key generated on the fly. In this way, the server is not using directly on the messages his certificated private key but creates another key pair, that is signed using the certificated private key. The point now is that the server must create a new key pair for every session, so this has to be done quickly: DH is suitable for this problem, because it is fast, so it became compulsory in TLS 1.3.

The two parties that want to start a communication have no common knowledge at the beginning of the algorithm, so they agree on a prime number p and the so-called generator g. Then each of them chooses a random number ($A, B \in 1, ..., p-1$) and compute the modular exponentiation of the generator g to the power of that number ($g^A \pmod p, g^B \pmod p$), transmitting them to the counterpart. Then, the two parties perform the same computation on the number received,

obtaining the values in the following equations. The whole process is described graphically in figure 6.1.

$$k_A = (g^B)^A \pmod{p} \qquad k_B = (g^A)^B \pmod{p} \tag{6.1}$$

We can easily check that they are equal, so the two parties agreed on using the key $k = k_A = k_B$. The generator is chosen in $GF^*(p)$, that denotes the set of integers $\{1, 2, ..., p-1\}$. In the algorithm, A and B represent the private keys, while g^A and g^B are the public keys. The strength of the keys is therefore based on the discrete logarithm problem (DLP) and the DH problem, described in section 6.2.2.

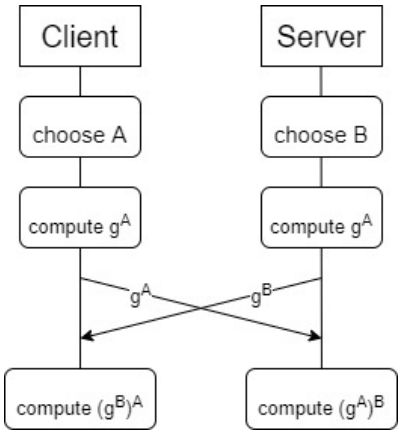


Figure 6.1. Representation of the DH algorithm when a client and a server want to start a communication. The symmetric key that will be agreed is $k = (g^B)^A = (g^A)^B$. All the operations must be intended in arithmetic modulo p.

An active attack that can be performed against DH is a Man in the Middle (MITM). In a MITM, the communication between two parties is intercepted, and sometimes modified, by an attacker. In the DH algorithm, the attacker intercepts and blocks the messages containing g^A and g^B, transmitting to the two victims the number g^C that he created selecting the random number C. In this situation, the two parties that want to establish a symmetric key could not even notice the presence of the attacker and, while they are thinking to talk each other, they are talking with him, with the two different keys $k_A = (g^C)^A$ and $k_B = (g^C)^B$, as described in figure 6.2.

In order to avoid being attackable via a MITM, the two parties should use certificates distributed by a trusted third party (TTP), such as the digital certificates.

6.2.2 Discrete Logarithm Problem

The DLP is based on finding A from the value g^A under a modulus p, as presented in section 6.2.1: if an attacker finds a computationally feasible way to perform it and sniffs $g^A \pmod{p}$ and $g^B \pmod{p}$, he can find the secret key exchanged easily. The most intuitive way would be to perform many exponentiations on g up to $g^A \pmod{p}$ or $g^B \pmod{p}$, but this is exponential in the number of bits of the modulus. Many algorithms speed up the computation, but none of them is polynomial: at best they are exponential in the number of bits of the modulus divided by 2. Examples are the number field sieve, the baby-step giant-step, and the pollard ρ [55, sec. 3].

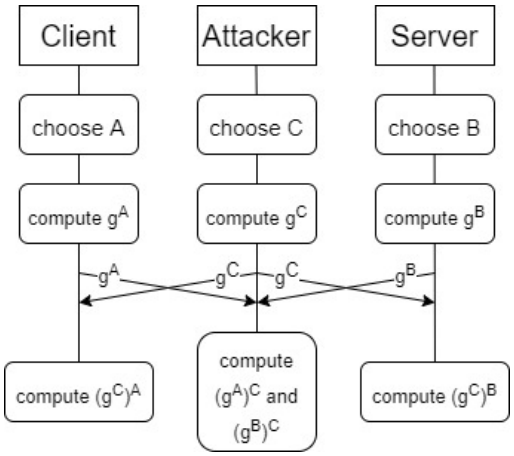


Figure 6.2. Representation of the MITM attack against DH when an attacker intercepts the beginning of the communication between a client and a server. The symmetric keys produced are $k = (g^C)^A = (g^A)^C$, which will be used to encrypt the messages to and from the client, and $k = (g^C)^B = (g^B)^C$, that will be used to encrypt the messages to and from the server. All the operations must be intended in arithmetic modulo p.

The DH problem takes the opposite way: it is based on finding the key $k = (g^A)^B \pmod p$ starting from $g^A \pmod p$ and $g^B \pmod p$. We can see that solving one of these two problems permits to solve the other, but it is not known which is the easiest way to take.

6.3 Public Key Encryption Algorithms

One of the important use of asymmetric cryptography, as already mentioned, is the possibility to encrypt a message with a key and to decrypt it with a different key, related to the first. In this section, we will describe El-Gamal and RSA, two of the most important cryptographic algorithms that, as we will see in chapter 7, are vulnerable against Shor's algorithm.

6.3.1 El-Gamal

The El-Gamal algorithm is widely used and is based on the same mathematical basis of DH (6.2.2): the generation of keys is performed following the same procedure, but everything is done by just one node since the scope of this algorithm is different. At first, a prime number p and a generator $g \in GF^*(n)$ are chosen, where n is the order of the generator, then a random number $A \in GF^*(n)$ is picked, that will be the private key of the key pair, and the public key is computed as $h = g^A \pmod p$.

For the encryption of a message m into the ciphertext C a random number $B \in GF^*(p)$ is chosen as key and equation 6.2 is calculated, while for the decription the message is retrieved as in equation 6.3.

$$C = (g^B \pmod p, m \cdot h^B \pmod p) = (c_1, c_2) \tag{6.2}$$

$$m = \frac{c_2}{c_1^A} = \frac{m \cdot h^B}{(g^B)^A} \tag{6.3}$$

The number B has to be ephemeral, because if two messages m_1 and m_2 are enciphered by using the same ephemeral key B and the attacker knows or guess one message (m_α) then it can read the other (m_β). In fact, the term c_1 of both messages will be the same and m_β can be calculated as in the following equation.

$$\begin{cases} m_\alpha = \frac{c_{2\alpha}}{(c_1)^A} \\ m_\beta = \frac{c_{2\beta}}{(c_1)^A} \end{cases} \Rightarrow m_\beta = \frac{c_{2\beta} \cdot m_\alpha}{c_{2\alpha}} \tag{6.4}$$

Besides from the DLP, the plain El-Gamal algorithm can be attacked using a chosen-ciphertext attack[1] [56]: in fact, having the chipertext $C = (c_1, c_2)$ of message m, it is easy to construct the ciphertext corresponding to the message $s \cdot m$ just computing $C_s = (c_1, s \cdot c_2)$. To avoid this attack, padding is added to the message, so that the receiver can check its integrity.

6.3.2 RSA

The most famous and used cryptosystem for Public-Key Encryption, but used also for digital signature, is RSA. The acronym stands for Rivest Shadir Adleman, which are its three inventors.

The algorithm for the key generation starts choosing two prime numbers of λ bit p and q, where λ is the half of the wanted security level for RSA (for example, in RSA-1024 $\lambda = 512$). Then two numbers are computed starting from p and q: the modulus $N = p \cdot q$ and the value $\phi(N) = (p-1)(q-1)$. Now the public exponent e is selected such that $e \leq \phi(N)$ and $\gcd(e, \phi(N)) = 1$, and finally the value $d = e^{-1} \pmod{\phi(N)}$ is computed. To understand this step, we have to explain two mathematical properties:

- The $\gcd(e, \phi(N)) = 1$ requirement means that e and $\phi(N)$ have to be coprime; and
- The inverse of a number in modular arithmetic is that number such that $x \cdot x^{-1} = 1 \pmod{y}$, and can be calculated efficiently using the Extended Euclidean Algorithm, that we will describe in section 6.3.3.

Now that all the needed values are available, the private key is set as $(\phi(N), d)$ and the public key as (N, e).

In order to perform the encryption of a message m, that has to be lower of the modulus N, into the ciphertext C, we perform the operation of equation 6.5, while for the decryption we follow the one of equation 6.6

$$C = m^e \pmod{N} \tag{6.5}$$

$$m = C^d \pmod{N} \tag{6.6}$$

The decryption equation holds because of the following equation, that can be proved with the Femat's little theorem[2].

$$m = m^{e \cdot d} \pmod{N} \tag{6.7}$$

The Femat's little theorem states equation 6.8, where p is a prime number.

$$a^{p-1} \equiv 1 \pmod{p} \tag{6.8}$$

[1] see https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-5906-5_556

[2] see https://en.wikipedia.org/wiki/Fermat's_little_theorem

In fact, being $e \cdot d = 1 \pmod{\phi(N)}$, we can say that $e \cdot d - 1 = h(p-1) = k(q-1)$, for some integers h and k. In order to prove equation 6.7, we check that this holds separately for mod p in equation 6.9 and mod q in equation 6.10 using Fermat's little theorem.

$$m^{e \cdot d} = m^{e \cdot d - 1} m = m^{h(p-1)} m = (m^{p-1})^h m \xrightarrow{6.8} 1^h m = m \pmod{p} \tag{6.9}$$

$$m^{e \cdot d} = m^{e \cdot d - 1} m = m^{k(q-1)} m = (m^{q-1})^k m \xrightarrow{6.8} 1^k m = m \pmod{q} \tag{6.10}$$

To have a more efficient encryption, e is often chosen to have a short bit length and small Hamming weight (numbers of 1 in the binary form), like 3, 17, and 65537 ($2^{16} + 1$). However, using a too small number like 3 and 17, introduce a vulnerability, as we will see in section 6.3.5.

6.3.3 Extended Euclidean algorithm

The Extended Euclidean algorithm can be used to compute efficiently the gcd between two integers a and b and the coefficients of the identity expressed in equation 6.11.

$$ax + by = \gcd(a, b) \tag{6.11}$$

We know that in RSA d is the inverse of e in arithmetic modulo $\phi(N)$,so having selected e such that it is coprime to $\phi(N)$, we can use equation 6.11 to find d:

$$\phi(N)s + ed = 1 \tag{6.12}$$

The algorithm starts with the coefficients of the first two columns of table 6.2, that evolves following the equations in the third column.

Starting parameters		evolution of the parameters
$r_0 = \phi(N)$	$r_1 = e$	$r_{i+1} = r_{i-1} - q_i r_i$
$s_0 = 1$	$s_1 = 0$	$s_{i+1} = s_{i-1} - q_i s_i$
$t_0 = 0$	$t_1 = 1$	$t_{i+1} = t_{i-1} - q_i t_i$

Table 6.2. Initial parameters and their evolution in the extended euclidean algorithm. q_i is that number such that $0 \leq r_{i+1} < |r_i|$.

The computation stops when $r_{k+1} = 0$, leaving us the following values:

- r_k, that is the gcd between $\phi(N)$ and e, so it will be 1 in our case;
- s_k, that corresponds to s in equation 6.12;
- t_k, that is the d value in the RSA algorithm.

6.3.4 Prime Factorization

Prime factorization is the division of a number into a product of primes. The most intuitive attack that can be performed on RSA is to try to factorize N in order to find p and q. This is a known hard problem of mathematics since there are no classical algorithms that can solve it polynomially. If an attacker is able to compute p and q, then he can easily recover the secret key, doing the same feasible steps of the algorithm. Over the years, RSA factoring challenges

have been made for this problem, to test the security of RSA and decide which bit length use. Nowadays, keys of 512 bits can be attacked in four hours or less, even with solutions available in the cloud [57], while keys of 1024 bits are breakable in some months [58]. The recommendations suggest using keys of 2048 bits, that can be considered secure for the next decade, or keys of 4096 bits [59].

There are some special cases where the factorization becomes simpler. For example, if the two numbers p and q are really close to each other, we can use Fermat's factorization method. We represent N as the difference of two squares:

$$N = a^2 - b^2 \tag{6.13}$$

In RSA, N is the product of two primes, so we can apply the transformation in equation 6.14.

$$N = p \cdot q = (\frac{p+q}{2})^2 - (\frac{p-q}{2})^2 \tag{6.14}$$

We try to guess a pair (a,b) that satisfies 6.13, starting considering a as $\lceil\sqrt{N}\rceil$ and increasing it step-by-step by one. When we find a good pair, we can calculate p and q:

$$\begin{cases} a = \frac{p+q}{2} \\ b = \frac{p-q}{2} \end{cases} \tag{6.15}$$

If p and q are close, the pair a and b will be found quickly: the number of steps that the algorithms does to find them is approximately $\frac{(\sqrt{p}-\sqrt{q})^2}{2}$.

In the general case, the best known classical algorithm to solve the factorization problem is the general number field sieve (GNFS) [60]: this has been used to make the records done during the RSA challenges. It is a random algorithm that has an expected time complexity $O(e^{\sqrt{\frac{64}{9}}(\log N)^{\frac{1}{3}}(\log\log N)^{\frac{2}{3}}})$, it is also very complex and only the set-up phase requires a lot of time, so it is not fast for small moduli, but it permits to speed-up the factorization for large ones.

6.3.5 Attacks on RSA

Apart from the factorization problem, various attacks exploit some vulnerabilities of the basic RSA cryptosystem.

An attack that can be conducted is a chosen-ciphertext attack to retrieve a message m from the correspondant ciphertext C. The attacker starts picking a random number r and computes $h = r^e \cdot C \pmod N$. Now he has to find a way to trick the private key owner in order to decrypt h, obtaining the equation below.

$$h^d = r^{e \cdot d} \cdot C^d = r \cdot C^d = r \cdot m \pmod N \tag{6.16}$$

The attacker can in this way obtain m just multiplying $r \cdot m$ by r^{-1}. The decryption performed by the victim can be possible if the same key pair is used for both signing and encrypting because the attacker can just ask the victim to sign a message instead of decrypting and resending it. So, the two countermeasures to this attack are to use different key pairs for the encryption and the signature and never decrypt a message at request.

Another method to break plain RSA is using a Chosen Plaintext Attack[3]. In this, the attacker creates a dictionary encrypting with the public key of the victim many expected messages, so when

[3]see https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-5906-5_557

he sniffs a ciphertext that matches one of the computed ones, he can be sure about the content of the message. To resolve this problem, padding is added to the message before the encryption, following a secure padding scheme.

As we have said in 6.3.2, using 3 or 17 as public exponent creates a vulnerability in RSA due to the Chinese Remainder Theorem [61]. Supposing that the same message is encrypted with three different key pairs, it is possible for an attacker to sniff the three different ciphertext of equation 6.17.

$$\begin{cases} C_1 = m^3 \pmod{N_1} \\ C_2 = m^3 \pmod{N_3} \\ C_3 = m^3 \pmod{N_3} \end{cases} \tag{6.17}$$

Following CRT, we can obtain m considering a modulus N obtained as $N = N_1 \cdot N_2 \cdot N_3$, that will surely be greater than m^3, and performing the recomposition steps for the CRT. This problem can persist even using 17 as public exponent, so the most used one is 65537.

RSA is present in many chipersuites of TLS up to version 1.2, but the way it is implemented opened the port to a very famous attack called the Bleichenbacher attack. In 1998 Daniel Bleichenbacher described the so-called "million-message attack" [62], based on a vulnerability in the way the RSA encryption was used in TLS. The attacker can perform an RSA private key operation without knowing the server's private key by sending about one million crafted messages and looking for differences in the error codes returned. It requires a lot of time, but it is possible. It has been refined over the years and in some cases, it requires only some thousands of messages, so it is feasible nowadays for a laptop. In 2017 the ROBOT attack made it possible to attack major websites, like facebook.com. Even if you can't copy the private key of the server, you can perform one operation using this key, so you can create a fake server. Due to this vulnerability and to the fact that it does not permit forward secrecy, in TLS 1.3 the RSA key exchange was completely removed as declared in RFC-8446, but it can still be used for digital signatures, as we will see in the next section.

6.4 Digital Signature

When we have to send a message or publish a document and we want to provide proof that we are the author of it, we can provide a digital signature as an attachment to it. This permits to achieve the following security properties:

- Authentication, using a method to bind the author to the private key used as the digital certificates;
- Integrity, since any change in the data signed, invalidates the signature; and
- Non repudiation, that is the impossibility to deny the paternity of a signature, since the signer is the only owner of the private key.

Since the encryption procedure can be slow for large documents or messages, the digital signatures are often performed on its digest, which is calculated using a hash function.

Whit the RSA key generation algorithm we are able to produce a pair of keys, so we may think to use them in order to perform digital signatures. In this case, the sender encrypts the digest of the message using his private key performing the operation in equation 6.18 and transmits it along with the message m, while the receiver verifies the signature with the public key of the

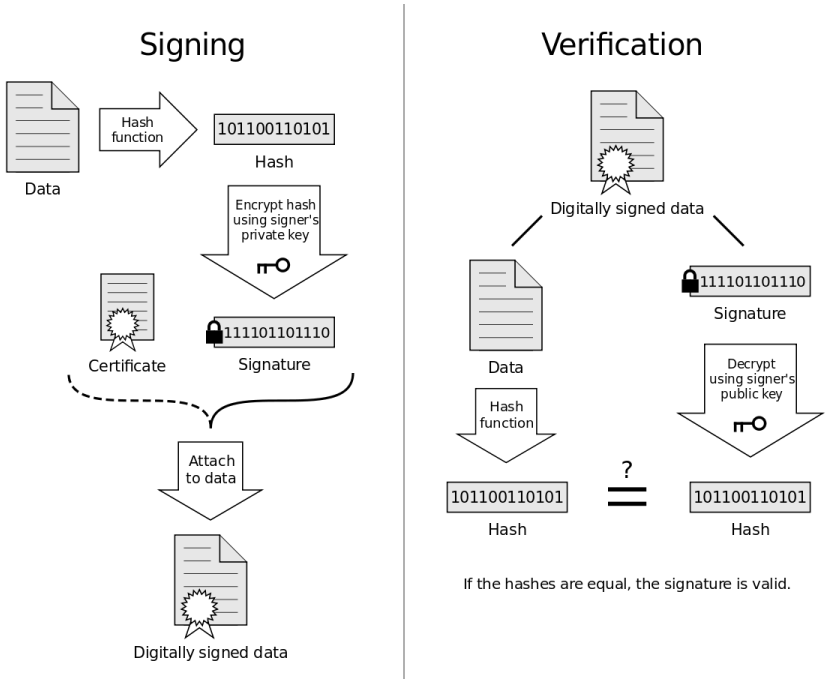


Figure 6.3. Representation of the Digital Signature creation and verification process. The certificate is needed to prove the validity of the public key. Source: [64]

sender, checking if $\sigma^e = m \pmod{n}$. However, producing many signatures can make it easy to discover the private key of the sender, so the signatures are made with a secure padding scheme and with a big modulus.

$$\sigma = \text{Sign}_{sk_B}(m) = m^d \pmod{n} \tag{6.18}$$

Another cryptosystem that can be used to sign messages and documents is the Digital Signature Algorithm (DSA). The key pair generation for a n bit public key length starts finding a prime number p such that $2^{n-1} < p < 2^n$ and selecting a number q, prime divisor of $p-1$, such that $2^{\lambda-1} < q < 2^{\lambda}$, where λ is the size of the private key. The correlation between these two values is described in table 6.3. The following step is to find an element α that generates a subgroup

Type of key	Bit length				
Public key (n)	1024	2048	3072	7680	15360
Private key (λ)	160	224	256	384	512

Table 6.3. Correspondency between key length for the DSA cryptosystem. Source: [63].

with q elements modulo p: this means that q is the order of a modulus p, so it is the smallest number such that $a^q = 1 \pmod{p}$. This is not an hard step since we use an auxiliary number $h \in \{2, ..., p-2\}$ and we compute α as in the equation below.

$$\alpha = h^{\frac{p-1}{q}} \pmod{p} \tag{6.19}$$

Now an element d, that will be the private key, is chosen such that $0 < d < q$, and the value $\beta = \alpha^d \pmod p$ is computed.

The public key will be composed by (p, q, α, β), while the private key, as we said, is d. As we can see, the security of this algorithm is based on the discrete logarithm problem like DH.

The signature is done choosing a random integer k_E (where E stands for ephemeral) such that $0 < k_E < q$ and computing the values r and s as in the following equations.

$$r = \alpha^{k_E} \pmod p \pmod q \tag{6.20}$$

$$s = (\text{hash}(x) + d \cdot r)k_E^{-1} \pmod q \tag{6.21}$$

In these equations, hash(x) is the output of a cryptographic hash function applied on the message x. The key k_E has to be ephemeral for the attack explained in section 6.5.2, where an attacker can find the private key d just looking at two messages and their signatures. The signature will be composed both of r and s, so it will of 320 bits, even if the overall security is only 80 bits.

The verification process is done computating the value v as in equation 6.22, and comparing it with r in arithmetic modulo q: If the two numbers are equal, then we can bu sure of the authenticity of the message.

$$\begin{aligned} w &= s^{-1} \pmod q \\ u_1 &= w \cdot \text{hash}(x) \pmod q \\ u_2 &= w \cdot r \pmod q \\ v &= \alpha^{u_1} \cdot \beta^{u_2} \pmod p \pmod q \end{aligned} \tag{6.22}$$

6.5 Elliptic Curve Cryptography

One of the major problems of asymmetric cryptography is the excessive length of the keys. RSA keys are nowadays at least of 2048 bits, so the computations with this algorithm for decryption and signing (the operations where we have x^d) are heavy and also not always suitable for some systems in the IoT. The same reasoning can be done for the other asymmetric algorithms, so we need a method to lighten the computations obtaining the same cryptographic strength.

The elliptic curves are a mathematical method used to build the parameters of the public-key algorithms, where the operations are executed on the surface of a 2-dimensional curve that permits to reduce the length of the keys (as shown in table 6.4). This happens because the DLP on a 2D surface, called Elliptic Curve DLP (ECDLP) is way more complex than the one studied for the 1D case.

Type	Bit length			
ECC	110	160	224	256
RSA	512	1024	2048	3072

Table 6.4. Comparison between the key lengths with similar resistance for RSA and elliptic curve algorithms.

As we have seen in section 6.2.2, the fastest known classical algorithms for this problem run in $O(\sqrt{n})$, where n is the bit length of the key, so the security level will be half the key size.

The current breaking record of ECDLP is on an EC of 118 bits, done in 2016 by D.J. Bernstein et al. [68, 67] : this has been performed using the pollard ρ algorithm and a cluster of FPGAs. They also demonstrated the scalability of the method, so even bigger modules can be attacked with this method.

An elliptic curve (EC) is the set of solutions to the following equation.

$$f(x, y) = y^2 - (x^3 + ax + b) = 0 \pmod{p} \tag{6.23}$$

In this, a and b are two parameters such that $4a^3 + 27b^2 \neq 0$, together with the point at infinity O, that is the identity element of the group. We can plot an example of a generic elliptic curve, as shown in figure 6.4.

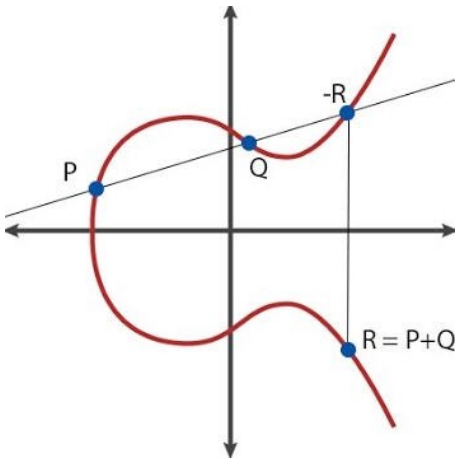


Figure 6.4. Representation of the elliptic curves. We can graphically see the addition of two points (P and Q) as the reflection along the x-axis of the point where the straight line between P and Q intersects the curve in a third point (-R). Source: [65]

In an elliptic curve E, we can perform an addition between two points of the curve obtaining a third point of the curve, as we can see graphically in figure 6.4.

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3) \tag{6.24}$$

This addition is computationally feasible so, any cryptographic algorithm based on the discrete logarithm, like Diffie-Hellman and DSA, can be adapted to work on a elliptic curve. The formula for the addition in equation 6.24 is:

$$\begin{cases} (x_3, y_3) = (\lambda^2 - x_1 - x_2, -(\lambda x_3 + \nu)) \\ \lambda = \frac{y_2 - y_1}{x_2 - x_1} = \frac{x_1^2 + x_1 x_2 + x_2^2 + a}{y_1 + y_2} \\ \nu = y_1 - \lambda x_1 \end{cases} \tag{6.25}$$

If in the modular arithmetic we can define the order r such that $g^r = 1 \mod p$, in the elliptic curves the order n of a so-called generator G is that number such that

$$n \cdot G = \mathrm{O} \tag{6.26}$$

In the elliptic curve cryptography (ECC) we define the curves using six parameters:

$$\mathrm{T} = (p, a, b, G, n, h)$$

where the first three parameters are the ones present in equation 6.23, G is a generator, n is its order and h is an integer such that $h = \frac{|E(\mathbb{F}_p)|}{n}$ (the cardinality of the Galois field[4] of p divided by the order of G, that for the Lagrange Theorem[5] will be an integer).

6.5.1 Elliptic Curve Diffie-Hellman

To reduce the length of the keys used for key agreement purposes, we can apply ECC on DH. The first step that is carried out in the Elliptic Curve Diffie-Hellman (ECDH), is the agreement on a curve E with parameters T= (p, a, b, G, n, h). Now one of the two actors, namely Alice, picks a random number $A \in \{2, ..., \#\mathrm{E} - 1\}$, computes $pA = A \cdot G = (x_A, y_A)$ and transmit it. The same is done by the other actor, namely Bob, so he picks a random number $B \in \{2, ..., \#\mathrm{E} - 1\}$, computes $pB = B \cdot G = (x_B, y_B)$ and transmit it. Now they have all the elements to calculate the shared secret since it $k_A = A \cdot pB = k_B = B \cdot pA$. In this algorithm, Alice's public key will be pA, while her private key will be A and the same for Bob.

In this version of the DH algorithm we have to take the same precautions as the normal one:

- Ephemeral keys are needed to provide forward secrecy; and
- Mutual authentication is necessary to avoid MITM attacks.

There is a particular curve, named Curve25519, that has been designed by D.J. Bernstein specifically for ECDH, and that is one of the fastest in ECC [66]. The name comes from the fact that it is defined modulus $p = 2^{255} - 19$: it has a 256-bit key size and offers a security level of 128 bits. The generator is $G = 9$, because it generates a subgroup with n elements such that $h = \frac{|E(\mathbb{F}_p)|}{n} = 8$ and n is prime.

The support for this curve is mandatory in TLS 1.3, so it is one of the most used cryptographic methods nowadays.

6.5.2 Elliptic curve DSA

If the creator of a message wants to sign it with EC-DSA, the first thing he has to do is to create the keys, following a procedure similar to the one of normal DSA. The signer chooses an elliptic curve T= (p, a, b, G, n, h) suitable for cryptographic purposes and selects a random integer $d \in \{1,2, ..., n - 1\}$ to compute $B = d \cdot G$. The public key will be the tuple (p, a, b, n, G, B), while the secret key will be d.

In order to sign the message x, he chooses a random integer K such that $0 < K < n$, and computes $R = K \cdot G = (x_R, y_R)$, setting $r = x_R$, and s as in equation 6.27, where hash(x) is the

[4] A Galois filed, or finite field, is a set of a finite number of elements where the operators of the multiplication, addition, subtraction, and division are defined. The Galois filed of a number p is formed by all the integers modulo p. https://en.wikipedia.org/wiki/Finite_field

[5] Lagrange's theorem states that for any finite group G, the order of every subgroup of G divides the order of G, where a subgroup is a subset that can be formed with the multiplication. https://en.wikipedia.org/wiki/Lagrange's_theorem_(group_theory)

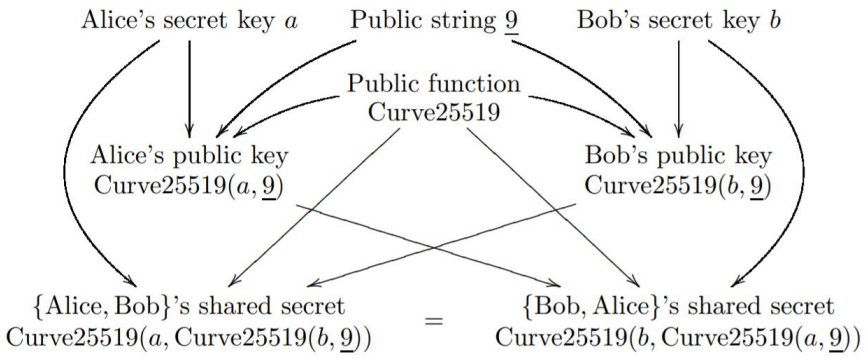


Figure 6.5. Representation of the secret key generation using Curve25519. Source: [66]

output of a cryptographic hash function applied on the message x. If $s = 0$, he has to restart the signature procedure. The signature, as in the DSA case, is formed by the pair (r, s).

$$s = (\text{hash}(x) + d \cdot r)K^{-1} \pmod{n} \tag{6.27}$$

The procedure for the verification of the signature is done performing the following calculations:

$$\begin{aligned} w &= s^{-1} \pmod{n} \\ u_1 &= w \cdot \text{hash}(x) \pmod{n} \\ u_2 &= w \cdot r \pmod{n} \\ P &= (u_1 \cdot G + u_2 \cdot B) = (x_P, y_P) \end{aligned} \tag{6.28}$$

The verification succeeds if $x_P = r$ modulus n, otherwise it fails.

If in the DSA algorithm, to obtain a level of security of 80 bits, we needed a public key of 1024 bits, in the EC-DSA we need a public key of just 160 bits. However, the length of the signature remains 320 bits, 160 for r and 160 for s. As in DSA, the parameter K has to be ephemeral, since signing two different messages with the same K opens the door to a famous attack that was performed against the PlayStation 3.

When the signer calculates s_1 and s_2 using equation 6.27 of the messages x_1 and x_2 with the same K, he will obtain the two values as in the equations below.

$$s_1 = \text{hash}(x_1)K^{-1} + rdK^{-1} \pmod{n} \tag{6.29}$$

$$s_2 = \text{hash}(x_2)K^{-1} + rdK^{-1} \pmod{n} \tag{6.30}$$

An attacker can retrieve the secret key d since the rdK^{-1} term is equal in the two equations, just following the passages reported.

$$\begin{aligned} s_1 - \text{hash}(x_1)K^{-1} &= s_2 - \text{hash}(x_2)K^{-1} \pmod{n} \\ K &= \frac{\text{hash}(x_1) - \text{hash}(x_2)}{s_1 - s_2} \pmod{n} \\ d &= \frac{s_1 K - \text{hash}(x_1)}{r} \pmod{n} \end{aligned}$$

EC-DSA is present in TLS 1.3 and is the basis of the Bitcoin signature: here the hash function used is SHA256, while the curve is called secp256k1, which is defined over a modulus of 256 bits $(2^{256} - 2^{32} - 977)$ [9].

6.6 Quantum threats for asymmetric cryptography

With this digression about asymmetric cryptography, we have studied some of the most common algorithms used nowadays, and we have underlined the fact that they rely on two hard problems: the factorization problem (RSA) and the DLP (DH, El-Gamal, DSA, ECC). We have also seen that there are no classical algorithms that can solve them in polynomial time, but this consideration falls when we point the attention to the quantum world: Shor's algorithm [7], in fact, permits to break in polynomial time both these two problems, so it represents a threat for all the current asymmetric cryptography present in TLS, in the cryptocurrency world, and in most of the Hardware Security Modules. We will see more in detail how this algorithm work in chapters 8 and 10.

Chapter 7

The advent of quantum computing for cryptography

The advent of quantum algorithms caused different consequences in cryptography. As we have seen, due to Shor's algorithm, many asymmetric cryptosystems will become vulnerable in the next years, but this is not the only case that affects the cryptography world: in fact, Grover's algorithm can be used to reduce the computational complexity needed to break the symmetric and the hash algorithms. However, quantum computing brought also new possibilities for important aspects of cybersecurity, like the Quantum Key Distribution, that can play useful roles in the future.

7.1 Asymmetric schemes affected by Shor's algorithm

As we have studied in chapter 6, the most used public-key algorithms are based on the factorization problem or on the DLP. These two problems have a similar mathematical structure and can be broken in polynomial time with Shor's algorithm.

The current estimations for the factorization problem state that a 1000-bit number can be factorized using a quantum computer with 2000 logical qubits, which can be not so far looking at the current roadmaps of quantum devices manufacturers [4]. But the most problematic threat regards the elliptic curves since the small key space makes them easier to be broken. In fact, it is estimated that an elliptic curve defined over a modulus of 160 bits is attackable using only 1000 logical qubits.

As we will see more in detail the quantum circuits for Shor's algorithm in chapters 8 and 10, we will now discuss the security implications of Grover's algorithm and the countermeasures that the cryptographic world can take to face up these threats.

7.2 Grover's algorithm against cryptographic protocols

In addition to Shor's algorithm, there is another quantum algorithm that represents a threat for modern cryptography: Grover's algorithm, which we have introduced in section 3.7, permits to have a quantum speed-up in brute-force attacks, so it applies to all the algorithms where an exhaustive search can be applied meaningfully: symmetric and hash algorithms. In these

algorithms, we can obtain a reduction of the security of a key/digest to half its length, since Grover's computational time is $O(\sqrt{N})$ [74]. For example, AES with keys of 256 bits would have the same security that is currently provided by keys of 128 bits.

However, AES256 is still believed quantum-resistant because the current quantum computers capability is far enough from the resource estimations that are currently taken into consideration, but the use of AES128 is discouraged.

7.2.1 Resource estimation for AES256

We will now see the passages for the construction of Grover's algorithm circuit for AES256, to understand the estimations that have been done[1]. From section (3.7), we know that in the Grover algorithm we need $\frac{\pi}{4}\sqrt{N}$ application of the block formed by U_f and V. To study this circuit we will often recall at references [75] and [76].

We can start with the construction of the V gate, that is actually an inversion of the phase of the basis state $|0\rangle$ on the upper k qubits, as we can see in the following circuit.

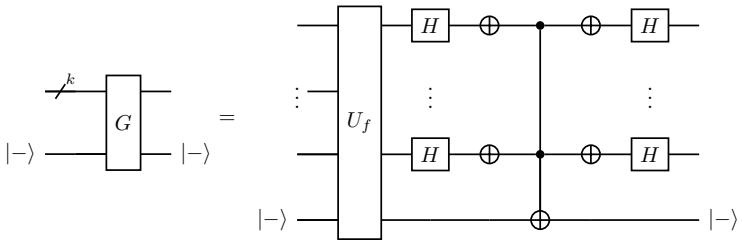


Figure 7.1. Expansion of the V gate in Grover's algorithm. Source: [75]

In the U_f gate we need to develop a circuit that with a given key $|K\rangle$ in input is able to understand if this is equal to the secret key or not, so we have to check the encryption of a plaintext with the corresponding known (plaintext, ciphertext) pair. To perform this encryption, we will follow the construction of the AES algorithm in a reversible way step-by-step.

A first consideration that has to be taken regards the uniqueness of the solution: since the blocks in AES are 128 bits long, we would need two (plaintext, ciphertext) pairs to have good possibilities of having a unique key as result, as shown in [76, sec. 2.2]. From this, comes that an attacker would need two pairs to perform the attack.

Now we can start to implement the building blocks of the AES operations, looking firstly at the specifications of NIST [77] and then at their implementations [75, sec. 3.2]. The AES operations are performed on a two-dimensional array of bytes called "state", which consists of four rows of bytes, each containing 4 bytes. The AES256 key is decomposed into 14 round keys via the Key Expansion method, and each of these keys is used in one of the 14 rounds that are performed, each of them is composed of the following basic blocks: the AddRoundKey, the MixColumns, the ShiftRows, and the SubBytes.

In the AddRoundKey operation, the round key is added to the state by a simple bitwise XOR operation, so we just need 128 CNOT operations in our quantum circuit to implement it.

[1]The estimations in this chapter always refer to logical qubits for simplicity.

The MixColumns operates on the state column-by-column, transforming every one of them with a matricial operation. This operation can be conducted in place using 277 CNOT gates and a total depth of 39.

In the ShiftRows transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes: the second row is shifted of one position, the third of two, and the fourth of three. To obtain this result in our circuit we do not need to insert any gate, since we will just need to act on different qubits subsequently to emulate this shifting.

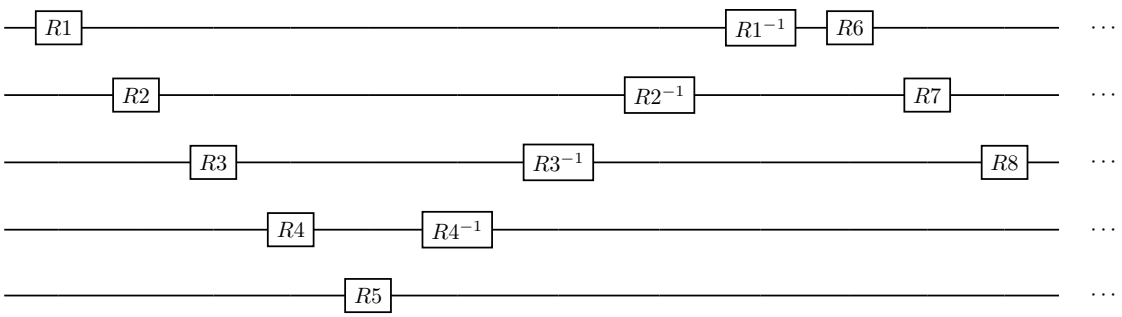
The SubBytes is the most expansive operation to perform in our quantum circuit: while in the FIPS specification is just a classical table look-up, called S-box, that transforms every byte of the state into a new one, we have to explicitly calculate the result of this operation. To realize it, we will need 9 ancilla qubits, 9695 T-gates, and 12631 Clifford gates.

Before using the key, we need to transform it and divide it into many sub-keys. The key expansion for AES256 takes the key K and generates an expanded key of 60 words, that are divided into 15 round keys. The first 8 words of the expanded key are filled with K, while the other words are created performing the following transformation to the previous word:

- The function RotWord, that is just a cyclic permutation;
- The S-box seen in the SubBytes operator; and
- The operation Rcon, which adds a round-constant value to the first byte of each word.

The resources needed for this part of the circuit are 416 storage qubits and 96 ancillae qubits, while we obtain an overall depth of 16408 [75, sec. 3.2.2].

After the creation of the round keys, the first operation is the XORing between the first round key and the input, so we need 256 qubits to store the input to do this operation reversibly, then we start with the application of the rounds. Every round needs 16 SubBytes calculations: if we want to do them in parallel, we need 384 auxiliary qubits, while an increase in depth of 8 SubBites cycles permits us to spend only 24 auxiliary qubits. Since SubBytes is not done in place, every round requires 128 qubits, so the computation would need $128 * 14$ qubits, in addition to the number of qubits to store the original key, but we can reduce this number by reverting some rounds during the computation. As we can see in figure 7.2, we need just 1336 qubits for one round.



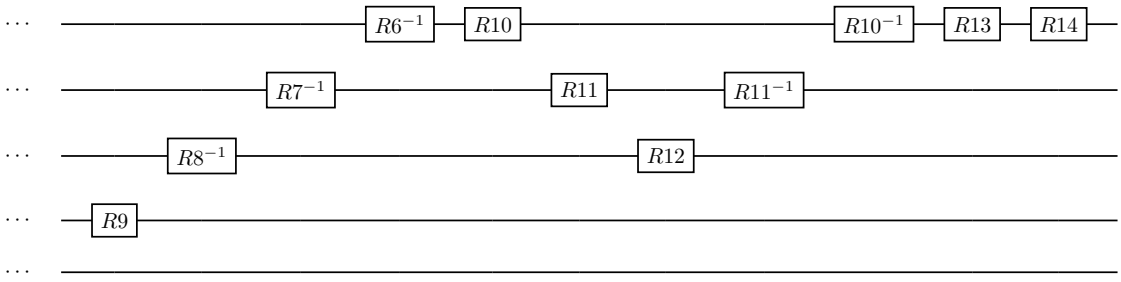


Figure 7.2. Order of application of the AES rounds and their inverse for AES256. Source: [75]

As we have said, we need 2 (plaintext,ciphertext) pairs for the computation, so we have 2 AES circuits in parallel: from this comes that we need $1336 * 2 + 1$ qubits for the whole circuit, whit a full depth on the order of 2^{145} [76, sec. 6.2], that is an acceptable level of depth for the NIST specification for post-quantum cryptography [5].

7.2.2 Resource estimation for hash functions

In cryptography, a property that is often important is integrity, which is the assurance that a particular message has not been modified and is the intended one. This can be obtained using a cryptographic hash function, that is a type of hash function (that maps strings of any length into strings of a fixed length) that has four additional properties:

- It is quick to compute the hash for any message;
- It is computationally infeasible to find a message with a wanted hash value;
- It is computationally infeasible to find two messages with the same hash value; and
- A small change to the message causes significant changes in the hash value, making it look uncorrelated to the previous one.

The application of a cryptographic hash function to a message permits to calculate its "digest", which is then appended to the message to demonstrate that it has not been modified in the transit: we have already seen an application of this type of function in figure 6.3.

Grover's algorithm can be applied also to break cryptographic hash functions, making the computational complexity of the pre-image attack the square root of the one needed for the basic brute-force attack. This attack consists of finding a message of n bits that has a specific hash value, so we would need to try randomly nearly 2^n messages before finding one, if n is the length of the digest, for a good cryptographic function.

However, the bit security of this algorithm is already half of the output bit length due to the birthday paradox attack[2] (the pollard ρ algorithm finds collisions even faster than Grover's one), so this algorithm is needed just to reduce the complexity of the pre-image attack and hence does not reduce the cryptographic strength of the hash functions.

[2] see https://en.wikipedia.org/wiki/Birthday_attack

One of the most used cryptographic hash functions is SHA3-256, so it is of interest to talk about the resource estimation to build Grover's circuit to find a message with a specific hash value computed with that algorithm.

The NIST specifications [78] for the SHA3-256 algorithm require an input register of 1600 bits that passes through 24 rounds of the Keccak Permutation, that is composed of 5 different transformation (θ, π, ρ, χ, ι).

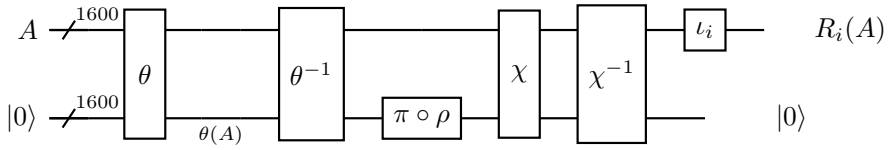


Figure 7.3. Implementation of the SHA3 oracle. Source: [79]

This implementation, requires 3200 qubits for the SHA oracle, plus one qubit for Grover's algorithm, while the depth is estimated to be $2 \cdot 10^{41}$ [79].

Other implementations [80] have been able to combine birthday paradox attacks and Grover's algorithm: creating a table of $\sqrt[3]{N}$ and using Grover's algorithm to find a collision in it, it is possible to reduce the computational complexity of the attack. In this way, a hash function with an output of n bit will have a $\frac{n}{3}$-bit security level.

However, as in the case of symmetric cryptography, it is possible thanks to the speed of the hash functions to use a bigger number of qubits, so the SHA2 and SHA3 families still remain quantum-resistant.

7.3 Quantum-resistant cryptography

The algorithm that really put our cryptographic world at risk was the one made by Shor in 1996 [7] to break the asymmetric cryptosystems. This algorithm, together with recent progress in quantum hardware, forced NIST to make a call for proposals [5] for public-key quantum-resistant algorithms in 2016. 23 signature schemes and 59 encryption/key encapsulation schemes were submitted, but just 7 of them (3 for signature and 4 for encryption/key encapsulation) advanced to the third round, which is ongoing.

The selected cryptosystems consist of implementations of three types of cryptography, listed in table 7.1

Lattice-based cryptography [82] includes all the algorithms that involve lattices in their construction or their security proof. A lattice is the set of all the integer linear combinations of a set of basis vectors b_i. It is important to note that given a lattice, the set of basis vectors is not unique. One of the finalist algorithms that is based on this type of cryptography is NTRU, which relies on the hardness of the Shortest Vector Path (SVP) problem, which is not quantum breakable. In this problem, we have as input a lattice (i.e. its basis) and we have to find the shortest non-zero vector in it. This type of cryptosystem has several advantages (low memory usage, efficient encryption, and decryption both in hardware and software implementations, fast key generation), so it seems to be the most promising one.

Code-based cryptosystems [81] are the ones where the underlying one-way function uses an error-correcting code, that is the addition of an error to a word or the computation of the syndrome of a parity check matrix. The first proposed was the McEliece algorithm, which still

Algorithm	Type	PKE/KE	DS
CRYSTALS-KYBER	Lattice-based	x	
NTRU	Lattice-based	x	
SABER	Lattice-based	x	
CRYSTALS-DILITHIUM	Lattice-based		x
FALCON	Lattice-based		x
Classic McEliece	Code-based	x	
Rainbow	Multivariate		x

Table 7.1. Selected cryptosystems for Round 3 of NIST call for proposals. PKE/KE stands for public key encryption and key encapsulation, while DS stands for digital signature.

remains unbroken even if it was presented in 1978. The private key in this algorithm is a certain binary irreducible Goppa code [81, sec. 2.1], a type of code for which an efficient error-correcting algorithm is known. The problem of this type of algorithm is the length of the keys: in McEliece, the public key is 1MB, while the private key is 11.5 KB

The Multivariate algorithms [83] are the ones where the one-way function is a multivariate quadratic polynomial map over a finite field: the public key is given by a set of quadratic polynomials with coefficients modulo q and the evaluation of these polynomials at any given value corresponds to either the encryption or the signature verification. These algorithms rely on the hardness of the inversion of these functions (MQ problem). However, since these functions must have a trapdoor in order to be used for public-key cryptography, we can't use random ones, but selected and studied ones, because attacks may exist for any of them (the security level is not based only on the MQ problem, but is different for any function).

7.4 Quantum cryptography

In quantum-resistant cryptography, we rely on the fact that there are no known quantum algorithms that can break certain types of mathematical problems, but since quantum computing is a relatively new field, we have no assurance that these algorithms will not be found, so we may think to other ways to ensure the wanted cryptographic functions. The quantum physics properties can be exploited also to perform cryptographic tasks, and researchers have been developing quantum algorithms suitable for different objectives in this field. The impossibility to observe a quantum state without changing it in some way, in fact, gives us a property that is not possible to achieve classically, that can be extremely important for the confidentiality of a transmission [84]. The most well-known example[3] is Quantum Key Distribution (QKD), but there are also other applications, like Quantum Random Number Generation (QRNG) and Quantum Digital Signature (QDS).

In the context of QKD, we have two different types of protocols: the ones that use a "prepare and measure" approach and the ones that rely on the entanglement property of quantum states, which we have seen in section 2.3.2.

In a "prepare and measure" protocol, the first step that the sender takes is to encode a classical variable α into non-orthogonal quantum states sent over a quantum channel, so that the

[3] see https://en.wikipedia.org/wiki/Quantum_key_distribution

receiver can retrieve and measure them obtaining a, probably different, classical variable β. By doing these steps more times, the two parties will have a sequence of data, that using a classical post-processing can be refined to obtain a shared key. In addition, part of the data is used to obtain information about the quantum channel, like the quantum noise present, to evaluate the quantity of post-processing needed and to perform an error correction phase that permits to reduce the possibility of an attacker to retrieve information eavesdropping the quantum channel to a negligible amount. The first example of a protocol of this type is the so-called BB84 [85], where the sender uses the Z- and X-basis to encode the classical variables: he prepares a random sequence of these two bases, and encodes the binary values using them, obtaining four possible states ($|0\rangle$, $|1\rangle$, $|+\rangle$ and $|-\rangle$). The states prepared are sent in the quantum channel, so that the receiver can retrieve and measure them using another random sequence of the Z- and the X-basis: in the cases where the basis used to prepare and to measure the state is the same, the bit will be equal, while in the cases where the basis is different, the measured bit will be random. Now the two parties exchange the sequence of the bases used, and they discard all the bits corresponding to the cases where different bases were used so that they will have the same sequence of bits.

The possibility to use the properties of the entangled states to perform QKD emerged in 1991 thanks to Artur Ekert [86], that presented the E91 protocol: here a source, that can be also an independent third party, creates pairs of entangled qubits and sends the first of each pair to one party and the second to the other one. The two parties measure the qubits using different bases, in such a way that they can calculate the probability of the presence of an eavesdropper using a Bell test[4]. These protocols based on entanglement are the key to the independence from the underlying device, providing another layer of security [6].

Nowadays, standards for QKD protocols have already emerged[5], and global networks to make the theoretical solutions possible are in development, since it is possible to use the optical fiber cables for that scope, using photons as qubits.

[4] see https://en.wikipedia.org/wiki/Bell_test#A_typical_CHSH_(two-channel)_experiment

[5] see https://www.etsi.org/committee/1430-qkd

Chapter 8

In-depth analysis of Shor's algorithm

We have seen in section 6.3.4 that there are various methods to break the factorization problem, such as the GNFS and the Fermat's method, but none of them runs in polynomial time, so they become unfeasible if the number of bits used to represent the modulus grows.

Another algorithm proposes the reduction of the factorization problem to period finding [88], opening the door to a new kind of attack: if we find an algorithm for period finding in polynomial time, we can solve also the factorization problem.

In this chapter, we will present the basic implementation for Shor's algorithm together with some optimizations that are useful to reduce the depth and the width of the circuit, performing an in-depth analysis of the gates that are needed.

8.1 From Integer Factorization to Period Finding

To find the factors p and q of a bi-primal number N, the first step to take is to choose a random integer $x < N$, that is used as a basis to find (with a period finding algorithm) its order r, i.e. the number such that $x^r \equiv 1 \pmod{N}$. If the order r is even, we can compute using the Extended Euclidean Algorithm seen in section 6.3.3 the factors p and q as:

$$p = \gcd(x^{\frac{r}{2}} - 1, N) \qquad 1 = \gcd(x^{\frac{r}{2}} + 1, N) \tag{8.1}$$

Since

$$(x^{\frac{r}{2}} - 1)(x^{\frac{r}{2}} + 1) = x^r - 1 \equiv 0 \pmod{N} \tag{8.2}$$

p is a non-trivial factor of N. This algorithm fails, and needs to be re-executed, only if r is odd or $x^{\frac{r}{2}} \equiv -1 \pmod{N}$.

There are particular cases where the first condition can be not considered: in fact, we need that only to divide r by 2 and compute the greatest common divisor, but when a is square we can perform the operation [87] even with an odd order. In this way is possible to do not discard useful values of r, even if square numbers are difficult to be obtained randomly when the module N is big so the efficiency is not greatly increased.

Applying this algorithm to a random number x, we can find a factor of N with probability $1 - 2^{1-k}$, where k is the number of distinct prime factors of N, so if N is bi-primal such as in RSA the probability of success will be the 50%.

The next ingredient for obtaining an algorithm able to break prime factorization in polynomial time is the period finding one. This problem classically is considered NP-hard, but the Shor's circuit, together with classical post-processing that runs in $O(\log_2 N)$, permits to break it polynomially.

8.2 Shor's proposal

The basic idea of Shor for the order finding problem was to apply QPE, which we have studied in section 3.5, to the unitary transformation in equation 8.3. The subsequent applications of this operators, represents the operation of equation 8.4, as every exponentiation can be written as in equation 8.5.

$$U|x\rangle = |ax \pmod N\rangle \tag{8.3}$$

$$|x\rangle|0\rangle \Rightarrow |x\rangle|a^x \pmod N\rangle \tag{8.4}$$

$$a^x = a^{2^{2n-1}x_{2n-1}+2^{2n-2}x_{2n-2}+\ldots 2^0 x_0} = a^{2^{2n-1}x_{2n-1}} \cdot a^{2^{2n-2}x_{2n-2}} \cdot \ldots \cdot a^{2^0 x_0} \pmod N \tag{8.5}$$

We have discussed in section 3.6 the high-level quantum circuit to find the period of the function $a^r \pmod N$. This is reported in circuit 8.1, where $n = \lceil \log_2 N \rceil$, while in figure 8.2 we have the decomposition of the oracle in unitary gates.

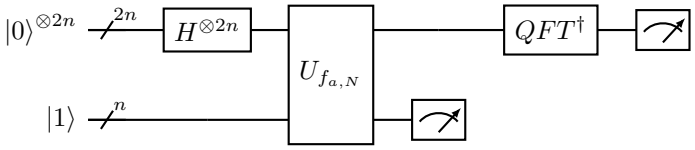


Figure 8.1. High-level representation of the circuit for the Shor's algorithm. Source: [90]

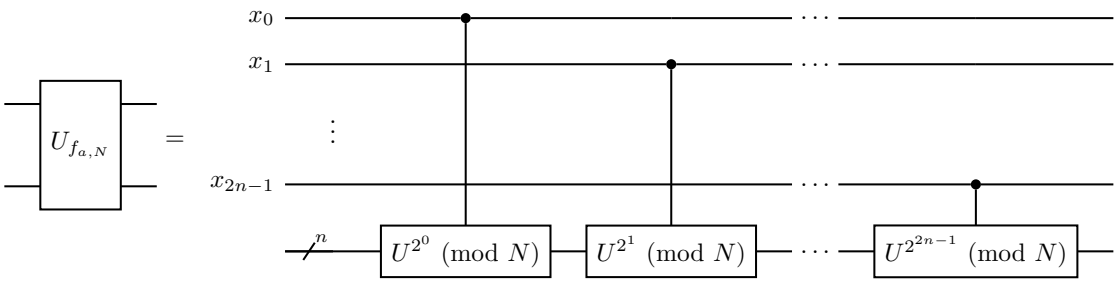


Figure 8.2. Implementation of the oracle for the period finding function. Source: [90]

As we can see from the implementation of the oracle, the complexity of this procedure is polynomial since there are as many U gates as the number of bits of the modulus times 2.

The circuit starts with the creation of the superposition of all the possible states in the top register, creating the state in the equation below.

$$|\psi\rangle = \frac{1}{2^n} \sum_{x=0}^{2^{2n}} |x\rangle |1\rangle \tag{8.6}$$

Now the oracle is applied, which leaves the system in the state described in the following equation.

$$|\psi\rangle = \frac{1}{\sqrt{q}} \sum_{x=0}^{q-1} |x\rangle |a^x \pmod N\rangle \tag{8.7}$$

where q is the power of 2 such that $n^2 \leq q < 2n^2$. Now we will apply the $QFT^\dagger$ gate on the first register, that will leave the system as in equation 8.8.

$$|\psi\rangle = \frac{1}{q} \sum_{x=0}^{q-1} \sum_{c=0}^{q-1} e^{\frac{2\pi i x c}{q}} |c\rangle |a^x \pmod N\rangle \tag{8.8}$$

Now it is possible to perform the measurements, so we can compute the probability P that our machine ends in a particular state $|c\rangle |a^k \pmod N\rangle$ where $0 \leq k < r$ summing all the possible ways to reach it, using the Born rule described in section 2.2.1.

$$P(|c\rangle |a^k \pmod N\rangle) = \left| \frac{1}{q} \sum_{x : a^x \equiv a^k} e^{\frac{2\pi i x c}{q}} \right|^2 \tag{8.9}$$

In this equation, the sum is computed over all $x < q$ such that $a^x \equiv a^k$ modulo N, but since r is the order of a, we can also say that this sum is over all x such that $x \equiv k$ modulo r. In this way, writing x as $br + k$, where $b \in \mathbb{N}$, the probability in 8.9 can be expressed as in equation 8.10, since the term $e^{\frac{2\pi i k c}{q}}$ has magnitude 1.

$$P(|c\rangle |a^k \pmod N\rangle) = \left| \frac{1}{q} \sum_{b}^{\lfloor \frac{(q-k-1)}{r} \rfloor} e^{\frac{2\pi i (br+k) c}{q}} \right|^2 = \left| \frac{1}{q} \sum_{b}^{\lfloor \frac{(q-k-1)}{r} \rfloor} e^{\frac{2\pi i b r c}{q}} \right|^2 \tag{8.10}$$

It can be proven, as shown in [7], that if the value rc is close to a multiple of q, the probability to obtain the correspondent state is larger: in particular, the probability of measuring a given state $|c\rangle |a^k \pmod N\rangle$ will be at least $\frac{1}{3r^2}$ if there is a $d \in \mathbb{N}$ such that

$$\left| \frac{c}{q} - \frac{d}{r} \right| \leq \frac{1}{2q} \tag{8.11}$$

Since $q \geq n^2$, we know that there is at most one fraction $\frac{d}{r}$ that satisfies the inequality, so we can retrieve it rounding $\frac{c}{q}$ to the closest fraction with a denominator smaller than n using the continued fraction expansion[89].

As stated in [7], there are $\phi(r) \cdot r$ different states that permit us to obtain a particular r, where $\phi(x)$ is the Euler's totient function[1], so the probability to obtain a good value as output of the quantum circuit is $\frac{\phi(r)}{3r}$.

[1]The Euler's totient function $\phi(x)$ calculates the number of numbers lower to x and coprime to x

8.3 Oracle design

The next step to take for the construction of the circuit is to understand the components that are used to build the oracles in circuit 8.2. To do this, we will follow a bottom-up approach, starting from the most basic gate up to the most complex one, taking as reference the circuit described in [90].

8.3.1 The adder gate

The first gate that we discuss is the adder gate, where we sum a classical number to a number encoded in a quantum register. This classical number will be the a of section 8.2, that we already know when we want to construct a circuit, so it is not necessary to have quantum-controlled gates: for this reason, in our implementation, we will use classical if instructions. However, as the addition takes place in the Fourier space, we have to transform a in such a way that the correct phase is applied to each qubit of the quantum register. We can see the circuit in figure 8.3.

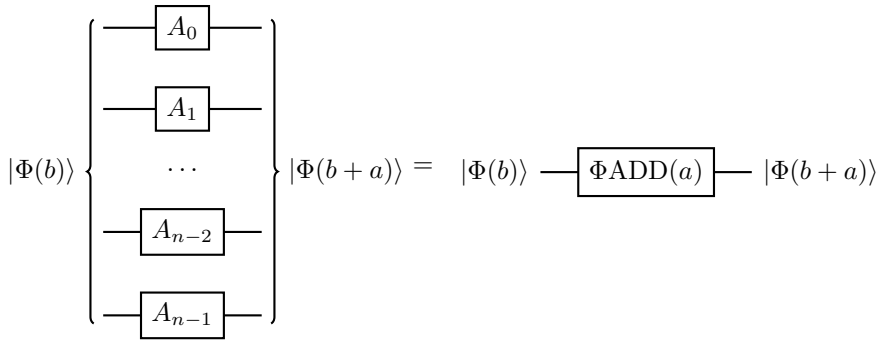


Figure 8.3. Addition of a classical value a to a quantum register encoding b in the Fourier space. The values A_i are computed classically. Source: [90]

The phases A_i are calculated as

$$A_i = \pi \sum_{j=0}^{i} a_j 2^{-(j-i)} \tag{8.12}$$

This gate, in order to prevent overflow, should be applied on a register of n+1 register, so $\Phi(b)$ is a register containing the QFT of an $n+1$ bit number where the most significant bit is always '0'. Being a reversible gate, we can apply the inverse of it, but in this case we will have two possibilities: the result will be $\Phi(b-a)$ if $b \geq a$, while it will be $\Phi(2^{n+1}-(a-b))$ if $b < a$. This inverted gate is useful to construct the modular version of the adder gate.

From figure 8.3, we can easily see that the adder gate has depth 1 and needs a register of $n+1$ qubits. For the next gates, we will also need the controlled and the doubly controlled versions of this, which can be obtained by simply controlling each of the gates: however, since the control qubits will be the same, the depth increases to n and to $n \cdot T$, where T represent the depth needed for a Toffoli gate, in the case it is doubly controlled.

8.3.2 The modular adder gate

For the circuit of the modular adder gate, we exploit the fact that performing $b - a$ when b is a number of n bits encoded on a register of $n + 1$ qubits (as in the adder gate case) will bring the most significant bit to '1' only if $a > b$. The circuit is described in its doubly controlled version in figure 8.4, as we will need this for the oracle.

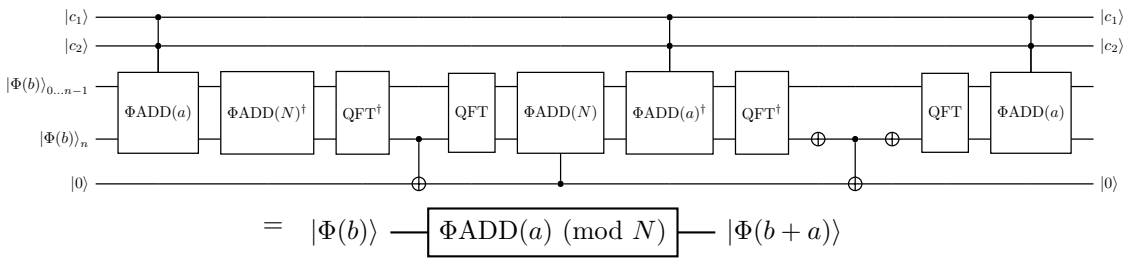


Figure 8.4. Doubly controlled modular addition of a classical value a to a quantum register encoding b in the Fourier space modulo N. The result will be $|\Phi(a+b) \pmod N\rangle$ if $c_1 = c_2 = |1\rangle$, $|\Phi(b)\rangle$ otherwise. Source: [90]

After the application of the first inverse QFT, the qubit used as control will be in state $|1\rangle$ only if $a + b < N$, so in this case we must sum the value N again: to do this, we have to pass from the fourier basis to the $\{|0\rangle, |1\rangle\}$ one in order to control the value on the most significant qubit. After the ΦADD(N) gate, the only purpose of the circuit is to recreate the state $|0\rangle$ in the bottom qubit.

In this case, the resource estimation will be affected by the type of QFT that will be used. If we use the exact QFT seen in section 3.4, we would require $O(n^2)$ gates for n qubits, but approximate versions of it exist that make it possible to reduce the number of gates used. However, the depth remains $O(n)$, si we will consider the depth of the exact QFT ($2n$) for our estimation. We have 4 single-qubit gates, 4 QFT, 1 adder gate, and 4 controlled adder gates, but two of them can be parallelized since subsequent and with different controllers, so we can consider only the doubly-controlled one for the depth calculus:

$$\text{depth} = 4 + 4 \cdot (2n + 1) + 1 + 3 \cdot (n \cdot T) = 3n \cdot T + 8n + 9 \tag{8.13}$$

The number of qubits is $n + 4$, with 2 of them used just as controllers.

8.3.3 The controlled modular multiplier gate

The next gate that we need is a modular multiplier that takes as input a controller qubit, a register of n qubits that encodes the value x, and a register of n qubits that encodes the value b, using this last one to output the value $b + (a \cdot x)$ if the controller qubit is $|1\rangle$.

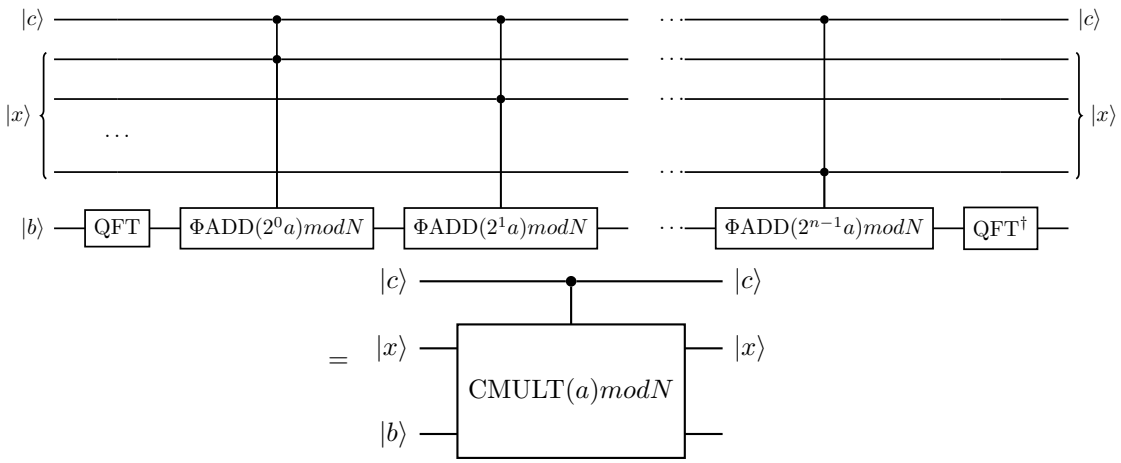


Figure 8.5. Controlled modular multiplication of a classical value a times a quantum register encoding x modulo N, summing the result to a third register encoding the value b. The result will be $|b + a \cdot x \pmod N\rangle$ if $c = |1\rangle$, $|b\rangle$ otherwise. Source: [90]

In order to do the multiplication, we divide it into a series of additions using the following equation:

$$ax \pmod N = (((2^0 a x_0 \pmod N) + 2^1 a x_1 \pmod N) + ... + 2^{n-1} a x_{n-1} \pmod N \tag{8.14}$$

In this way, the multiplier gate can be constructed using only modular adder ones.

This gate has 2 QFT over n qubits and n modular adder gates, so the resulting depth will be:

$$\text{depth} = 2 \cdot 2n + n \cdot (3n \cdot T + 8n + 9) = 3n^2 \cdot T + 8n^2 + 13n \tag{8.15}$$

The number of qubits will be $2n + 3$, as $n + 2$ are used by the modular adder gates, n for the register containing the state $|x\rangle$ and 1 for the controller.

8.3.4 The controlled U_a gate

The Shor's algorithm oracle performs the transformation $|x\rangle |0\rangle \Rightarrow |x\rangle |a^x \pmod N\rangle$, so we need to perform a reversible exponentiation with the exponent encoded in a quantum register of $2n$ qubits to create it. This exponentiation can be divided into a series of multiplications following equation 8.5, so the last step is to save the result of the multiplication in the same register used for the input so that we can save in it the intermediate result. The controlled U_a gate will therefore perform the transformation $|c\rangle |x\rangle \Rightarrow |c\rangle |a \cdot x \pmod N\rangle$ if $|c\rangle = |1\rangle$ and we can construct it in the following way:

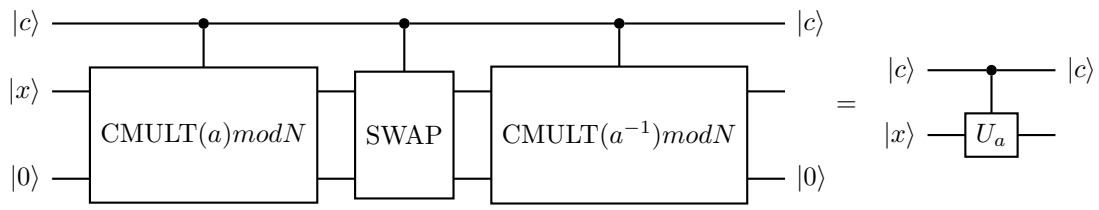


Figure 8.6. Controlled U_a gate. The result will be $|a \cdot x \pmod N\rangle$ if $c = |1\rangle$, $|x\rangle$ otherwise. Source: [90]

The controlled version of the swap gate can be implemented just using a Toffoli instead of a CNOT gate in the circuit in section 2.4.2, so it has a depth of $n \cdot T + 2$. The resulting depth of the U_a gate will therefore be

$$\text{depth} = 2 \cdot (3n^2 \cdot T + 8n^2 + 13n) + (n \cdot T + 2) = 6n^2 \cdot T + 16n^2 + 26n + n \cdot T + 2 \tag{8.16}$$

while the number of qubits needed is the same of the controlled modular multiplier gate $(2n + 3)$.

8.3.5 Discussion and resource estimation

In order to apply the multiplication times a^{2^i} we do not have to apply the U_a gate 2^i times, but we can perform classically a^{2^i} and apply the gate $U_{a^{2^i}}$, thanks to the fact that we can rewrite it as in the following equation.

$$a^i x \pmod N = (((a \cdot x \pmod N) \cdot x \pmod N)...) \cdot x \pmod N \tag{8.17}$$

In this way, following circuit 8.2, the oracle will need just $2n$ sequential controlled U_a gates, each controlled by a different qubit, so the total width will be $4n + 2$ and the total depth the sum between the oracle, the Hadamard gate and the inverse QFT applied on $2n$ qubits.

$$\text{depth} = 2n \cdot (6n^2 \cdot T + 16n^2 + 26n + n \cdot T + 2) + 1 + 2(2n) \simeq 12n^3 \cdot T + 32n^3 \tag{8.18}$$

If we want to factorize a number of 2048 bits, that is needed in order to break RSA 2048, we need in this way 8194 logical qubits that, given the high depth of the circuit, must be created using a big distance code. In addition, the described circuit would need a highly connected topology between these qubits.

A quantum device with these capabilities will probably be not available until the next decade looking at the current roadmaps of manufacturers. However, various improvements can be done to this basic version of the circuit, which we will see in the next sections.

8.4 Shor's algorithm enhancements

We will now propose different optimization methods from the literature. In our implementation, we will propose a version of the first two, but the third is an important case of study as a very recent estimation of the resources needed to factor RSA 2048 is provided for it. In table 10.1, we can see the resources needed to implement them.

8.4.1 Sequential QFT

The number of qubits of our estimation can be reduced using the sequential version of the inverse QFT [91], permitting us to use just one qubit for the register that will be measured: this reduces the width of the circuit from $4n + 2$ to $2n + 3$, without any increase in the depth. However, a classically-controlled quantum gate is needed, that is not currently available on IBM hardware.

Optimization	Logical qubits	Depth	Toffoli count
[90]	$2n+3$	$144n^3 \lg(n) + O(n^2 \lg(n))$	$576n^3 \lg^2(n) + O(n^3 \lg(n))$
[93]	$2n+2$	$52n^3 + O(n^2)$	$64n^3 \lg(n) + O(n^3)$
[95]	$3n + 0.002n \lg(n)$	$500n^2 + n^2 \lg(n)$	$0.3n^3 + 0.0005n^3 \lg(n))$

Table 8.1. Comparison between the different optimizations studied of the resuorce estimations. The estimation for the depth of [90] has an additional factor of $\lg(n)$ to account for the need to approximate arbitrary phase rotations using T states. Source: [95]

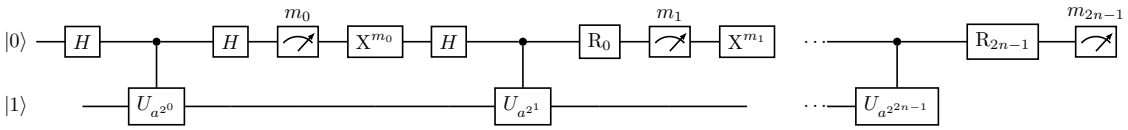


Figure 8.7. Circuit for the Shor's algorithm with the sequential QFT. The X^{m_0} are CNOT gates classically controlled using the result of each measurement (if the output of measurement m_i is '1', then the gate X^{m_i} is applied). The R gates depend on the results of all the previous measurements and are the ones that perform the QFT rotations. Source: [90]

In our implementation, we present a version of the circuit using this method, even if it is possible to only simulate that. This version permits us to test the implementation factorizing bigger numbers from the normal one since our tests are limited by the number of qubits needed.

8.4.2 In-place addition

Another possible optimization permits to save one qubit using a dirty ancilla qubit instead of a clear one in the modular addition [92]: since we have many dirty qubits available when we perform a modular addition, as the $|x\rangle$ register in figure 8.6, we will be able to run the circuit using just $2n+2$ qubits.

To perform the modular addition between a quantum register $|b\rangle$ and a classical value a, we compare the value of $|b\rangle$ with $N-a$: if $b > N-a$, we subtract $N-a$, else we add a. Then we compare the result with a, in order to clear the ancilla qubit.

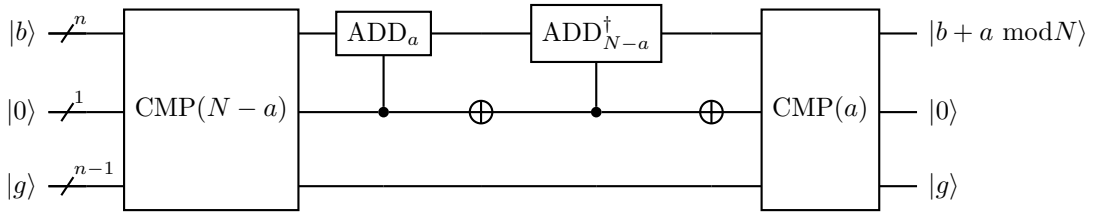


Figure 8.8. Modular adder using $n-1$ dirty ancillae of register $|g\rangle$ and 1 clear qubit. Source: [93]

To build this comparator, as suggested in [93], we need a gate that calculates the carry of the sum between a number encoded in a quantum register and a classical constant, which can be done using $n-1$ dirty ancillae qubits, available at each iteration of the controlled modular adder from the x register in figure 8.6.

To calculate the final carry of the addition between the number encoded in a register $|b\rangle$ and the classical value a, we would need to propagate the carry from each qubit to the last one so, as described in [93, sec. 2], we toggle the qubit g_i of the auxiliary register if the operation needs a carry from bit i to bit $i+1$. So g_i must toggle if $b_i = a_i = 1$, $g_{i-1} = b_i = 1$, or $g_{i-1} = a_i = 1$. It is also possible to save the qubit g_0, since it depends only on the condition $b_0 = a_0 = 1$: we will use a_0 instead of it to build the conditions for g_1.

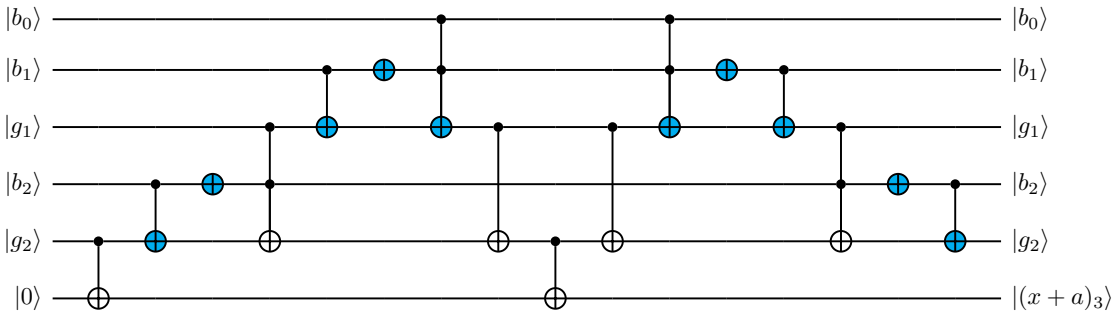


Figure 8.9. Carry gate for $n = 3$. The blue gates are present if $a_i = 1$. In order to build the circuit for n qubits, we must copy the same three gates of the $g_1 a_2 g_2$ triple. Source: [93]

Another possible optimization is based on not performing the addition in the Fourier space, so we will not need all the QFTs of the modular addition circuit, but it is based on the use of Toffoli gates. This adder gate will be important also in chapter 10, since we will use it for the implementation of Shor's algorithm for the Discrete Logarithm Problem. In order to construct the constant adder, we need a gate that calculates the carry of the sum between a number encoded in a quantum register and a classical constant using $n-1$ dirty ancillae qubits, that we already described, and a controlled incrementer.

For the controlled incrementer gate, it is possible to obtain an incrementer starting from a gate that performs the addition between two numbers encoded in two quantum registers of n qubits, where one is composed of dirty ancilla qubits:

$$|x\rangle|g\rangle \to |x-g\rangle|g\rangle \to |x-g\rangle|g'-1\rangle \to |x-g-g'-1\rangle|g'-1\rangle \to |x+1\rangle|g\rangle \tag{8.19}$$

In this equation, $g'-1$ is the bit-wise complement of g, so it is easy to obtain using n X gates. The controlled version of this gate can be made controlling every singe gate, as well as using the controller as the least significant qubit of the register to increment. The circuit to perform the addition between two quantum register is described in [94, sec. 2.2]: it does not use any ancilla qubits, except for the one to store the carry of the operation, as described in figure 8.10

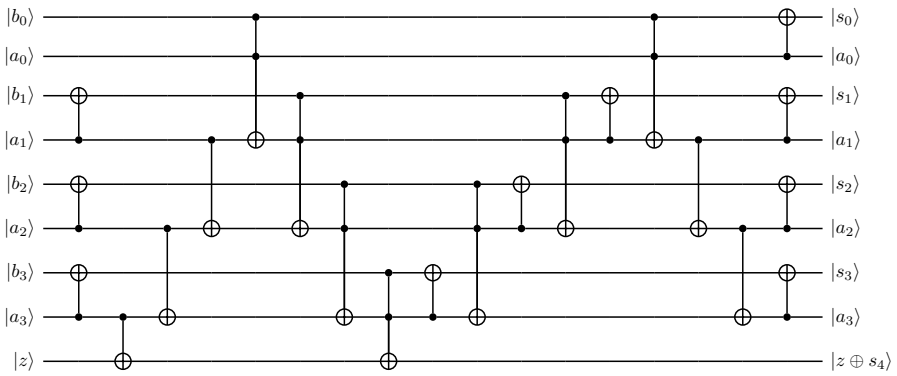


Figure 8.10. Circuit to perform the addition between two quantum registers of 4 qubits, where $s = a + b$ is stored on the register that contained b before its application. Source: [94]

The constant addition starts dividing the quantum register and the constant into two parts, the lower half (x_L and a_L) and the higher half (x_H and a_H), and calculating the carry bit of $x_L + a_L$ using the $|x_H\rangle$ register as dirty ancilla qubits: depending on this, the higher part of the $|x\rangle$ register is incremented or not. However, since we have many dirty qubits available in Shor's algorithm, we can use one of them just inserting another incrementer and inverting the x_H register if the dirty qubit was $|1\rangle$, as described in 8.11. This circuit is then repeated recursively both for the lower and the higher part.

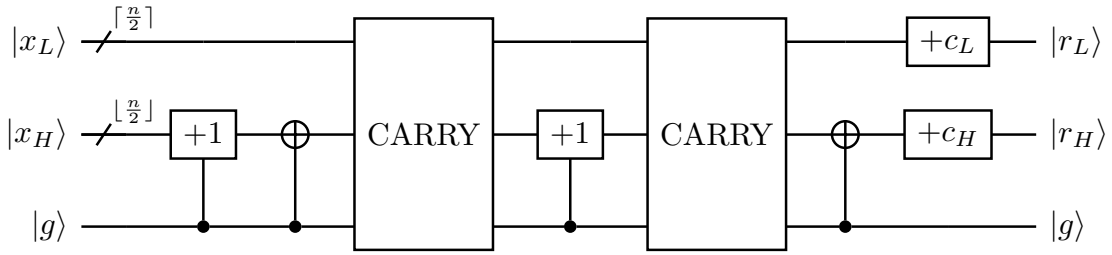


Figure 8.11. Circuit to perform the addition of a constant (a) to a quantum register. The CARRY gate computes the carry of $x_L + a_L$ into the qubit with initial state $|0\rangle$, borrowing the x_H qubits as dirty ancillae. Source: [93]

We can now use this gate to create the modular adder, which will have a Toffoli count of $8n\log_2 n + O(n)$. Following the procedure used in section 8.3, the overall number of Toffoli for the Shor's circuit is $64n^3\log_2 n + O(n^3)$.

8.4.3 Ekerå and Håstad's Algorithm

A recent study [95] used various optimization estimating a total time of 8 hours using 20 million physical qubits to complete the factorization of a 2048-bit number. In this method, derivatives of Shor's algorithm are used to reduce the number of multiplications, applying also windowed arithmetic to fuse several of them together [96]. In particular, the circuit used performs the DLP of the value y in equation 8.20, where $g \in 1, ..., N-1$ is a random number.

$$y = g^{N+1} \tag{8.20}$$

The discrete logarithm $d = \log_g y$, that is quantumly computed, will be in fact equal to $p + q \pmod r$, as shown in equation 8.21. This can be understood recalling that r divides $\phi(N) = (p-1)(q-1)$, which is the order of $\mathbb{Z}_N^*$ for the Euler's totient theorem[2], and that $pq + 1 = p + q \pmod{\phi(N)}$.

$$d = N + 1 = pq + 1 = p + q \pmod r \tag{8.21}$$

The value d can then be used to retrieve classically the roots of the following equation.

$$p^2 - dp + N = 0 \tag{8.22}$$

The quantum circuit performs period finding on the function $f(e_1, e_2) = g^{e_1} y^{-e_2}$, where e_1 has a bit length of $n + O(1)$ and e_2 of $\frac{n}{2} + O(1)$. More details about how Shor's algorithm for DLP works will be given in section 10.1. In this way, the total exponent has an asymptotical length of $1{,}5n$, which is lower with respect to the one used in the basic circuit of $2n$.

It is important to point out that their estimation is not based on the heavy hexagon code, but on a different quantum error correction code, that requires a lower number of physical qubits every logical one: using the same distance code ($d = 27$), if the heavy hexagon code is used the physical qubits count increases to 25.5 million qubits instead of 23 million (this number is a simplified estimation done in their study that will be lowered with some optimizations down to 20 million).

[2]see https://en.wikipedia.org/wiki/Euler's_totient_function

Chapter 9

Implementation of Shor's algorithm for factorization

In this chapter, we will see the implementation of Shor's algorithm, both of the basic version of $4n+2$ qubits and the sequential QFT version of $2n+3$ qubits. We created a library that permits to factor a number N selecting the wanted version of the Shor's algorithm as input for the called function, to facilitate its usability.

Qiskit contains a version of the basic algorithm adapted from the one present at the repository https://github.com/ttlion/ShorAlgQiskit, so we used it as a reference for our library as it is simpler to understand. The same repository contains also an implementation of Shor's algorithm with the sequential QFT, so we adapted it to build our parametric function. Another circuit can be simply constructed using the modular adder described in section 8.4.2, which we implemented from scratch following the theoretical paper: this can be used with both the described versions to save one qubit and to reduce the depth. In addition, an improvement in the efficiency of the previous implementations is provided considering the possibility of using odd orders as described in [87].

In the first section, we will talk about the classical computation needed, that is common to all the versions, while we will describe the code for the quantum part in the next ones.

9.1 Classical computation

To use the Shor function, we pass as input a number a and the number that has to be factored N. The first steps are to check if the modulus N can be easily factorized since in this case, it would be useless to use the quantum algorithm, so the following conditions have to be met to continue with the program:

- N is not even;
- N is not a power of an integer number, that can be checked in $O(n^3)$ time; and
- The gcd between a and N is 1 since in the other case it will be a factor of N.

Now we can start to build up the circuit, which will be done by selecting the wanted version of the QFT (normal or sequential) and of the adder (in the Fourier basis or in the normal one).

After the measurements, each retrieved result c will be used as input for the "get_factors" function, together with the modulus N and the value a.

In order to retrieve the two factors, we perform a cycle in which each iteration corresponds to putting one more term in the calculation of the continued fraction. To start this algorithm, we calculate $x = \frac{c}{2^{2n}}$ and we initialize a vector of integers b and a vector of floating numbers t: the first will contain coefficients of the continued fraction, while the second the values needed to calculate them.

```
b = array.array('i')
t = array.array('f')
b.append(math.floor(x))
t.append(x - b[i])
while(1):
    [...]
    b.append( math.floor( 1 / (t[i-1]) ) )
    t.append( ( 1 / (t[i-1]) ) - b[i] )
```

Listing 9.1. Python code to calculate the values of the cotinued fraction expansion

The denominator of the resulting fraction is retrieved at each iteration, which will be a candidate for r, so that we can calculate $a^{\frac{r}{2}}$. However, not every value of r can be used for this purpose, as it must be even if a is not a square. Now the factors are calculated as in equation 8.1 and returned if they are different from 1 and N.

If the function "get_factors" returns two values different from 0, these are the factors of the modulus N.

9.2 Basic version

In this version, we will have a register ("up_reg") of $2n$ qubits that are the ones to which the final QFT will be applied, a register ("down_reg") of n qubits that contains the results of the multiplications, and a register ("aux_reg") of $n+2$ qubits that contains the auxiliary qubits used in addition and multiplication. In addition, we have a classical register that will be used to store the result of the last measurement.

Now we apply the Hadamard gate on up_reg and a CNOT on the least significant qubit of down_reg. The next gates to apply are the $2n$ U_a, controlled each one by a different qubit of up_reg.

The U_a gate is created using the Qiskit QFT, imported from "qiskit.circuit.library", n doubly controlled modular adder multipliers, that are constructed using the function "ccphiADDmodN", and the inverse version of the QFT. A controlled swap gate between "down_reg" and the first n qubits of "aux_reg" is applied using the function "cswap" of the circuit library, so the two QFTs and the modular adders are reapplied as described in section 8.3.4.

The modular adders are made up of single-qubit gates, QFTs and inverse QFTs, and normal or controlled versions of adders. The non-trivial part of this gate is how to calculate the correct angles that are applied to the qubits in order to encode perform the sum: this is done using the function "getAngles", for which we provide here a detailed description.

```
def getAngles(a,N):
    s=bin(int(a))[2:].zfill(N)
    angles=np.zeros([N])
    for i in range(0, N):
        for j in range(i,N):
            if s[j]=='1':
                angles[N-i-1]+=math.pow(2, -(j-i))
        angles[N-i-1]*=np.pi
    return angles
```

Listing 9.2. Python code to calculate the angles that must be applied to a quantum register to perform the addition in the Fourier basis.

The number a, which must be summed to a quantum register, is transformed in its binary representation of N bits, and a new vector of N elements is created. Every angle to sum is calculated starting from the last one and adding a decreasing factor for each preceding qubit to the corresponding cell of the vector, finally multiplying it for π. These angles are then applied as gates to the quantum register using the "p" function of the circuit library. For the inverse version, it is just needed to apply the negative value of the angle retrieved with "getAngles".

After the series of U_a gates, the inverse QFT is applied to up_reg, and the same register is measured.

9.3 Sequential QFT

Using the sequential version of the QFT, as described in 8.4.1, permits to nearly halve the number of qubits used, but nowadays we can not execute this type of QFT in IBM quantum devices due to the lack of a classically controlled quantum gate. The circuit for the construction of the U_a gate is the same as in the basic version, but up_reg is here of only one qubit and the final inverse QFT is done in the cycle of the U_a gates using the function "getAngle", that calculates the rotation to apply at the qubit of up_reg after each measurement. In addition, the measurement is copied in a classical bit, so that in the next iteration this can be used to control a CNOT gate.

9.4 In-place adder

In order to use this type of adder, we eliminate the normal and inverse QFT performed at the beginning and at the end of every series of doubly controlled modular adders, since this time the additions will not be performed in the Fourier space. As described in figure 8.8, to create the modular adder gate we need a comparator, that is constructed using a gate that calculates the carry of the sum between the number encoded in a register and the number that we want to compare, and a controlled constant multiplier.

The first step is the creation of the comparator, which can be done as in listing 9.3.

```
bitwise_complement(circuit,b,n)
carry_gate(circuit,b,a,g,c,n)
bitwise_complement(circuit,b,n)
```

Listing 9.3. Circuit to create the comparator gate.

In this example, we use a clear qubit $|a\rangle$ to save the result of the comparison between the number encoded in register $|b\rangle$ and c and a register of $n-1$ dirty ancillae: in case $b < N-a$, $|a\rangle$ is toggled. To obtain this result, we consider the bitwise complement of $|b\rangle$ and we save the carry of its sum with $N-a$ in $|a\rangle$. The $n-1$ ancilla qubits are needed for the carry gate, as described in figure 8.9.

The instructions for constant adder follow figure 8.11, so we created a recursive function that terminates when the registers are 1-qubit long.

A non-trivial addition that we did to the circuit is about how the doubly-controlled version needed for the Shor's circuit is done: instead of simply controlling each gate, it is possible to control only the gates that affect the qubit used to control the constant additions, that are the two carry gates.

Chapter 10

Shor's Algorithm against Discrete Logarithm Problem

The Discrete Logarithm Problem, presented in section 6.2.2, is the second cryptography-related problem that can be transformed into an order finding one and resolved in polynomial time using the Shor's Algorithm.

In this chapter, we first describe the high-level circuit to solve this problem for multiplicative groups, like the one for DH, but then we focalize on the circuit for the elliptic curves providing a gate-by-gate explanation of it, as it is the most dangerous threat in terms of resources needed: we will see that, at an equal security level, Elliptic Curve Cryptography needs fewer qubits to be broken with respect to RSA and DH.

10.1 High-level circuit

In this section, we will follow for semplicity the description provided by Proos and Zalka in [97], to avoid the more mathematical one given in the original Shor paper. In order to find the value d of $x \equiv g^d \pmod{p}$, we consider the following function:

$$f(a,b) = g^a x^b \pmod{p} \tag{10.1}$$

This function has two different periods in $\mathbb{Z}^2$, that are

$$f(a+q,b) = f(a,b) \qquad \text{and} \qquad f(a+d,b-1) = f(a,b)$$

where q is the order of g. In this way, we can apply the ideas of the Shor's algorithm to solve this problem, but with three registers instead of two and using, after the first Hadamard gates that create the superposition of the two top registers, an oracle that performs the following transformation:

$$\frac{1}{q}\sum_{a=0}^{q-1}\sum_{b=0}^{q-1}|a,b,1\rangle \Rightarrow \frac{1}{q}\sum_{a=0}^{q-1}\sum_{b=0}^{q-1}|a,b,g^a x^b\rangle \tag{10.2}$$

Since $x \equiv g^d \pmod{p}$, the third register will be left in the state $|g^{a+db}\rangle = |g^{a_0}\rangle$, so for each b we will have only one solution and it is possible to rewrite the two top register as:

$$\frac{1}{\sqrt{q}}\sum_{b=0}^{q-1}|a_0 - db, b\rangle \tag{10.3}$$

Now we apply the inverse QFT on these two registers, obtaining the following state:

$$\frac{1}{q\sqrt{q}} \sum_{a',b'=0}^{q-1} \sum_{b=0}^{q-1} e^{\frac{2\pi i}{q}(a_0 - db)a'} e^{\frac{2\pi i}{q} bb'} |a', b'\rangle \tag{10.4}$$

The inner sum will be mantained only if $b' \equiv da' \pmod{q}$, since we are interested only in this cases.

$$\sum_{b=0}^{q-1} e^{\frac{2\pi i}{q}(a_0 - db)a'} e^{\frac{2\pi i}{q} bda'} = \sum_{b=0}^{q-1} e^{\frac{2\pi i}{q} a_0 a'} = q e^{\frac{2\pi i}{q} a_0 a'}$$

so we can rewrite the state as:

$$\frac{1}{\sqrt{q}} \sum_{a'=0}^{q-1} e^{\frac{2\pi i}{q} a_0 a'} |a', b'\rangle \tag{10.5}$$

with $b' = da' \pmod{q}$, so measuring the two registers we can obtain d by doing:

$$d = b'(a')^{-1} \pmod{q} \tag{10.6}$$

This circuit is able to extract a good pair (a', b') with probability $\frac{p}{240 \cdot q'}$, where q' represent the power of 2 such that $p < q' < 2p$, as stated in [7]. Looking at this probability, we can note that if the modulus of the algorithm is close to the value q', that is the next power of 2, the number of shots of the circuit needed will be lower. However, we have the possibility to lower-bound this probability, since $p < 2q'$, obtaining a constant minimum probability of $\frac{1}{480}$.

This high-level circuit can be applied to solve the DLP in the case of Elliptic Curves, just changing the operation performed by the oracle. It is important to note that many elliptic curves nowadays used for cryptographic purposes have moduli that are close to the next power of two: for example, Curve25519 is defined over $2^{55} - 19$, so the probability to obtaining a good value from the quantum circuit is really close to $\frac{1}{240}$.

If we want to break the ECDSA algorithm (section 6.5.2) finding the secret key d knowing the parameters of the curve, the generator G, and the point $B = d \cdot G$. In order to do so, we have to apply the following oracle transformation:

$$\frac{1}{2^n} \sum_{a=0}^{2^n-1} \sum_{b=0}^{2^n-1} |a, b, 1\rangle \Rightarrow \frac{1}{2^n} \sum_{a=0}^{2^n-1} \sum_{b=0}^{2^n-1} |a, b, aG + bB\rangle \tag{10.7}$$

Since the operation needed is a multiplication, we can follow the double and add technique as done for the factoring circuit, but the addition is based on the elliptic curve arithmetic, so we must find a reversible way to perform it.

To compute the new coordinates (x_3, y_3) of equation 6.25, we can use the algorithm described in [98] and reported in 1. In order to complete it, we need to build gates for the following operations:

- Modular addition of a constant to a number encoded in a quantum register, as well as its inverse for tu subtraction and their controlled versions;
- Controlled modular addition and subtraction between two numbers encoded in quantum registers;
- Modular multiplication performed out-of-place between two numbers encoded in quantum registers; and

- Modular squaring and inversion performed out-of-place of a number encoded in a quantum register.

Algorithm 1 Algorithm for reversible point addition between a point encoded in two quantum registers (x_1, y_1) and a point assumed to be a precomputed classical constant (x_2, y_2). In the case $ctrl = 1$, the result of $P_1 + P_2$ is stored in the register holding P_1. All the operations are performed in mudular arithmetic.

1: $x_1 \leftarrow x_1 - x_2$
2: **if** $ctrl = 1$ **then**
3: $\quad y_1 \leftarrow y_1 - y_2$
4: **end if**
5: $t_0 \leftarrow x_1^{-1}$
6: $\lambda \leftarrow y_1 \cdot t_0$
7: $y_1 \leftarrow y_1 \oplus \lambda \cdot x_1 \quad (= 0)$
8: $t_0 \leftarrow t_0 \oplus x_1^{-1} \quad (= 0)$
9: $t_0 \leftarrow \lambda^2$
10: **if** $ctrl = 1$ **then**
11: $\quad x_1 \leftarrow x_1 - t_0$
12: $\quad x_1 \leftarrow x_1 + 3x_2 \quad (= x_2 - x_3)$
13: **end if**
14: $t_0 \leftarrow t_0 \oplus \lambda^2 \quad (= 0)$
15: $y_1 \leftarrow \lambda \cdot x_1$
16: $t_0 \leftarrow x_1^{-1}$
17: $\lambda \leftarrow \lambda \oplus t_0 \cdot y_1 \quad (= 0)$
18: $t_0 \leftarrow t_0 \oplus x_1^{-1} \quad (= 0)$
19: **if** $ctrl = 1$ **then**
20: $\quad x_1 \leftarrow$ modular negation of x_1
21: $\quad y_1 \leftarrow y_1 - y_2 \quad (= y_3)$
22: **end if**
23: $x_1 \leftarrow x_1 + x_2 \quad (= x_3)$

10.2 Oracle design

We will now describe the various part of the oracle following a bottom-up approach as done in chapter 8, to study the resources needed to develop algorithm 1. As for the factorization circuit, the first basic block needed is a modular constant adder, that can be constructed as described in section 8.4.2 using also a comparator and a gate that performs the addition of two quantum registers. Starting from these, we can build the circuits of the next sections, which are described in [98].

10.2.1 Modular addition and doubling

The circuit to compute the addition modulo p is described in figure 10.2: we can construct it starting from the gates described in section 8.4.2 and a gate that performs the comparison between two quantum registers, using only one clear ancilla qubit and a dirty one. The strict comparison of two quantum register $|x\rangle$ and $|y\rangle$ is simple to build since it is made of just a subtraction and an addition without calculating the carry: in the first subtraction, the qubit that stores the result is toggled if $y < x$, then the addition without the carry restores the value $|y\rangle$ in the second register.

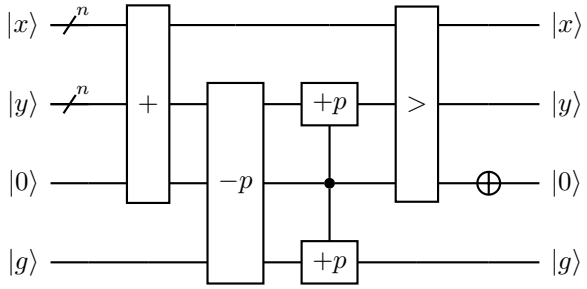


Figure 10.1. Circuit for the modular addition between two numbers encoded in quantum registers. The first gate represents the addition of two quantum registers using $|y\rangle$ to store the result and the clear ancilla qubit to store the carry bit of the operation, while the comparator toggles the ancilla qubit to restore its value if the value encoded in $|y\rangle$ is lower than the one in $|x\rangle$. Source: [98]

An important feature of this circuit regards its controlled version, which is used in algorithm 1: as done for previous circuits, we don't need to control every gate, but just the first (the adder) and the comparator.

The circuit for the modular doubling, which is useful for the modular multiplication, follows the same principles but requires one less register and a simple gate that performs the bit shift of a quantum register, built using only qubit swaps.

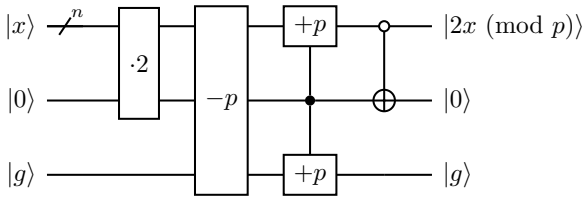


Figure 10.2. Circuit for the modular doubling of a number encoded in a quantum register. The first gate represents the doubling of a register using the clear ancilla qubit to store the carry bit of the operation. The last operation is controlled only by the least significant qubit of the $|x\rangle$ register, since the clear ancilla qubit is toggled only if $2x(\pmod p)$ is even. Source: [98]

For the controlled operation, the same reasoning done for the addition can be done for the doubling, since it is possible to create it by controlling only the binary shift and the last CNOT and adding a CNOT gate controlled by the control qubit swapped to be sure to restore the clear ancilla qubit.

10.2.2 Modular multiplication

The modular multiplication circuit that is needed for the point addition stores the result of the operation of two numbers encoded in two quantum registers in a third quantum register, so it needs at least $3n$ qubits in order to be run. It can be done in two ways: the first is following equation 8.14, while the second one is using the Montgomery arithmetic [99] since numbers in Montgomery representation will be used also in the modular inversion obtaining a most efficient algorithm.

The circuit using the double and add technique can be simply obtained as in figure 10.3 iterating the two gates seen in section 10.2.1. This circuit can be adapted to a squaring one by controlling the additions using the same register involved in it and a clear ancilla qubit, as in figure 10.4.

The first circuit needs $3n + 2$ qubits and has a Toffoli gate count of $32n^2 \log_2(n) - 59.4n^2$, while the second one $n2 + 3$ qubits and the same amount of Toffoli gates.

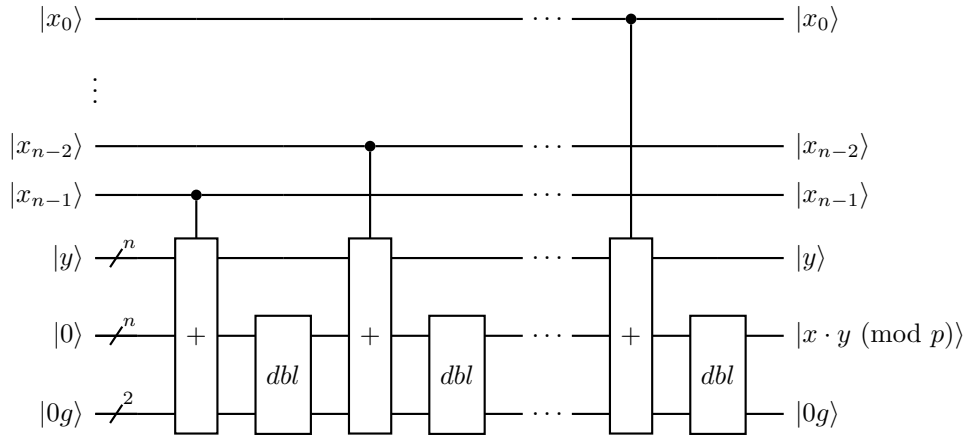


Figure 10.3. Circuit for the modular multiplication between two numbers encoded in quantum registers. Source: [98]

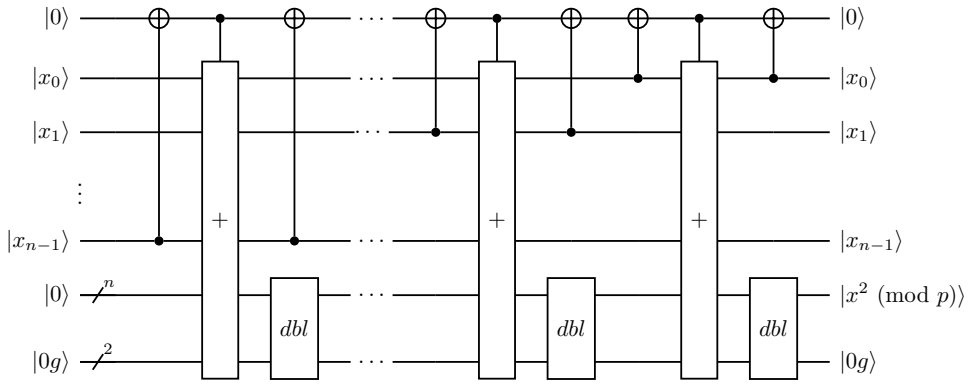


Figure 10.4. Circuit for the modular squaring of a number encoded in quantum register. Source: [98]

The Montgomery multiplication permits to obtain a trade-off between depth and width with respect to the normal modular multiplication. In order to understand it, we must recall that a number in Montgomery form is a number that is expressed as $aR \pmod p$ with R coprime and greater to p and that the Montgomery reduction algorithm computes $cR^{-1} \pmod p$ for an integer c given as input. In this way, if we perform the multiplication of two integers in Montgomery form $aR \pmod p$ and $bR \pmod p$ and we apply the reduction algorithm to the result, we obtain $abR \pmod p$, that is the Montgomery form of the result of the product between a and b. This can be simplified by selecting $R = 2^n$ so that the reduction can be performed with a series of binary shifts.

If the modular multiplication with the double and add approach needs two modular operations per round, the circuit described in figure 10.5 needs only one of them, as the modular doubling gate is not present: this however comes at the cost of an increase in the number of qubits because it is necessary to store the information regarding if the intermediate result is even or odd.

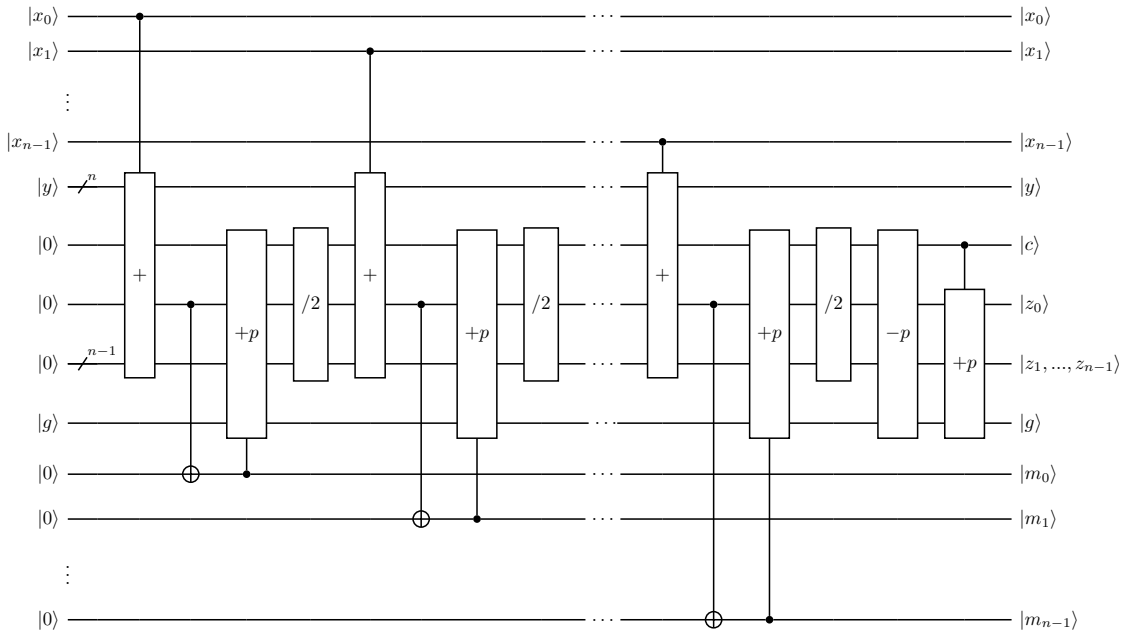


Figure 10.5. Circuit for the Montgomery modular multiplication between two numbers in Montgomery representation encoded in quantum registers. Source: [98]

After the application of the circuit in figure 10.5, the result must be copied in another register, so that its inverse can be executed to reset the clear ancillae. However, since this second execution does not affect the register where the result is stored, it can be run in parallel to other parts of the point addition algorithm, so we will consider just the first one for the depth count.

Considering also the register where the result will be stored, the Montgomery modular multiplication uses $5n + 4$ qubits and has a number of Toffoli gates of $16n^2 \log_2(n) - 26.3n^2$.

The same adaptation done with the double and add technique for the modular squaring circuit can be done for the Montgomery arithmetic: in this case, the number of needed qubits is $4n + 5$, as we have one ancilla qubit instead of a register of n qubits.

10.2.3 Modular inversion

The modular inversion is the most costly operation to perform to execute the point addition algorithm. For this task, we will report the circuit that implements Kaliski's algorithm [100] for inverting a number given in Montgomery representation with $R = 2^n$: if $x2^n \pmod p$ is given in input, the output of the circuit will be $x^{-1}2^n \pmod p$.

Kaliski's algorithm needs 5 input registers to calculate the GCD between two numbers x and p (4 of n bits to store the intermediate results and 1 of $l = \lceil \log_2 2n \rceil$ that is needed as a counter),

is divided into two phases as described in 2. In the first phase, the value $x^{-1}2^k$ is calculated, so we need to save the value of k in order to execute phase 2. The circuit for phase 1 is constructed considering the instruction inside the while loop as a basic block, and executing it conditionally on the value of a qubit that starts as state $|1\rangle$ and is toggled when v becomes 0. In figure 10.6 we can see the construction of the block inside the while loop, while in figure 10.7 we have a high-level description of the circuit for the execution of phase 1.

Algorithm 2 Kaliski's algorithm to calculate the pseudo inverse of x modulo p.

Phase 1
$u \leftarrow p$, $v \leftarrow x$, $r \leftarrow 0$, $s \leftarrow 1$, $k \leftarrow 0$
while $v > 0$ **do**
  if u is even **then**
    $u \leftarrow \frac{u}{2}$, $s \leftarrow 2s$
  else if v is even **then**
    $v \leftarrow \frac{v}{2}$, $r \leftarrow 2r$
  else if $u > v$ **then**
    $u \leftarrow \frac{u-v}{2}$, $r \leftarrow r + s$, $s = 2s$
  else
    $v \leftarrow \frac{v-u}{2}$, $s \leftarrow r + s$, $r = 2r$
  end if
  $k = k + 1$
end while
if $r \geq p$ **then**
  $r \leftarrow r - p$
end if
return $r \leftarrow p - r$, k
Phase 2
for i=1 to k-n **do**
  if r is even **then**
    $r \leftarrow \frac{r}{2}$
  else
    $r \leftarrow \frac{r+p}{2}$
  end if
end for
return $x \leftarrow r$

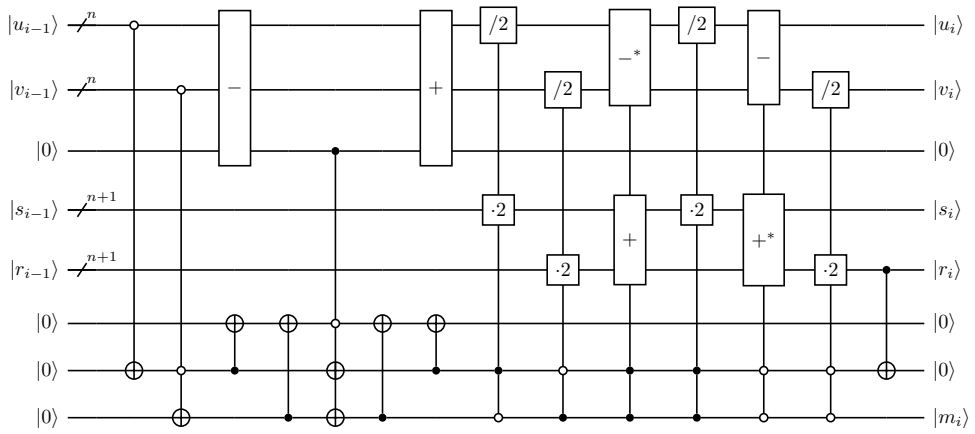


Figure 10.6. Block for the construction of the modular inverse circuit. The multi-wire gates store the output the register at the bottom, except for the cases with "*". Source: [98]

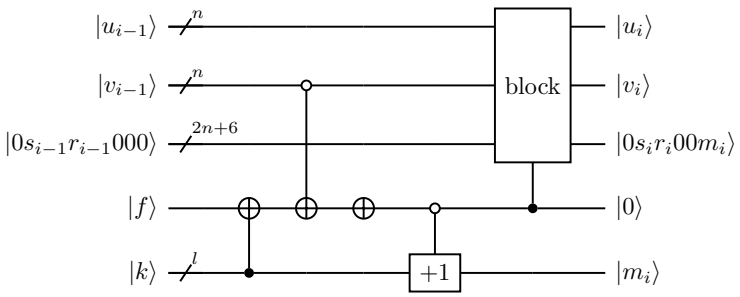


Figure 10.7. Round of the modular inversion circuit implementing Kaliski's algorithm. In each of the $2n$ rounds, a different m_i qubit is used. The block gate is executed until v reaches 0, then k starts to be incremented. Source: [98]

At the end of the $2n$ rounds, the register r stores the result, so we copy it in another register and execute the inverse of the circuit to restore the original values of the registers. However, we need to copy also the $|k\rangle$ register, since we need its value to know how many times we have to double the result. In this way, the total amount of qubits needed for phase 1 becomes $7n+2l+7$, while the number of Toffoli gates is $32n^2 \log_2 n$.

10.3 Resorce estimation

Now that we have all the gates needed for algorithm 1, we can provide an estimation of the total number of qubits needed. This is limited by the one for the modular inversion, but we have to add the qubit to which the sequential QFT is performed and 2 registers of n qubits that store intermediate results during each inversion. Regarding the depth, in [98] it is stated that the total number of Toffoli gates used scales as $224n^2 \log_2 n + 2045n^2$ for each point addition, so as $(448 \log_2 n + 4090)n^3$ for the whole Shor's circuit since we have $2n$ point additions. These numbers come from the fact that we have many ancillae qubits available for the modular multiplications, so it is possible to use the Montgomery version.

An important reminder is that the gates that we have described work for the most types of point additions, but do not for three exceptional cases: the addition of a point to itself, to its negative, and to the point at infinity. It is stated in [97] that these cases occur in total $4\frac{n}{p}$ times, so we have to add this fidelity loss to the probability to obtain a good pair $\frac{1}{480}$. In addition, we must change the input of the oracle, since it can not start the additions from the point at infinity: this problem is not difficult to solve because we can actually start from any point of the curve without affecting the resulting phase.

Now we can compare the resources needed to break the ECDLP and the ones for the factorization algorithm, looking at table 10.1. It is clear that ECC is more at risk than RSA nowadays. Elliptic curves using moduli of 256 bits are considered strong algorithms but breaking them requires less powerful quantum computers than the ones needed for RSA 1024.

ECDLP for curves modulo p			RSA with modulus N		
$\lceil \log_2 p \rceil$	#qubits	#Toffoli gates	$\lceil \log_2 N \rceil$	#qubits	#Toffoli gates
110	1014	$9{,}44 \cdot 10^9$	512	1026	$6{,}41 \cdot 10^{10}$
160	1466	$2{,}97 \cdot 10^{10}$	1024	2050	$5{,}81 \cdot 10^{11}$
224	2042	$8{,}43 \cdot 10^{10}$	2048	4098	$5{,}20 \cdot 10^{12}$
256	2330	$1{,}26 \cdot 10^{11}$	3072	6146	$1{,}86 \cdot 10^{13}$
384	3484	$4{,}52 \cdot 10^{11}$	7680	15362	$3{,}30 \cdot 10^{14}$

Table 10.1. Comparison between qubits needed to break ECDLP and RSA at similar security levels. The values for the RSA columns are derived by the version using the in-place adder. Source: [98]

10.3.1 Proposed improvements

In recent years some improvements have been done to the described circuit, permitting to solve the ECDLP with even fewer resources. In [101] the circuit is modified using pebbling, windowed arithmetic, and other peculiar optimizations, and a series of functions in Q# is provided. They have been able to reduce the number of qubits needed for the ECDLP on a curve of 256 bits from 2338 to 2124, the number of Toffoli gates by a factor of 119 (they estimate that it scales as $436n^3$), and the depth by a factor of 54. They provide also different optimized versions based on trying to reduce the depth so that it is possible to obtain different trade-offs of the resources.

Pebbling techniques permit to lower the depth of the circuits using the auxiliary qubits of an operation as input for the next one. In our circuit, for example, these can be used for step 6 of algorithm 1: here the modular square of λ is saved in t_0, which is then used to calculate $x_1 - t_0$ and restored to $|0\rangle$ we another modular squaring. Using the Montgomery squaring circuit as a black-box, the basic circuit that we described in section 10.2 used a not necessary modular squaring, since it is possible to create a more efficient version as done in figure 10.8.

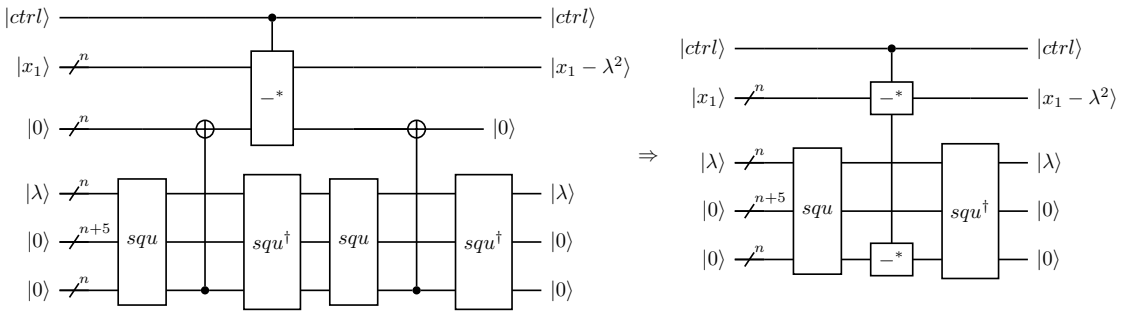


Figure 10.8. Modification on the circuit for squaring-then-add using pebbling. The multi-wire gates store the output the register at the bottom, except for the cases with "*". The register with $n + 5$ qubits collects all the other needed ancillae used. Source: [101]

The same idea can be used also for the modular division in steps 5,6, and 8 of algorithm 2 and needs two inversions and a multiplication: in this case, pebbling techniques permit to save the execution of a whole inversion circuit.

One of the improvements provided is a modification to the block of Kaliski's algorithm in figure 10.6: instead of using a series of doublings and halvings, they used swaps, as described in algorithm 3.

Algorithm 3 Modification provided in [101] to the operations inside the while loop of Kaliski's algorithm 2.

$b_{swap} \leftarrow$ false
if u is even and v odd, or u and v odd and $u > v$ **then**
    swap u and v
    swap r and s
    $b_{swap} \leftarrow$true
end if
if u and v odd **then**
    $v \leftarrow v - u$
    $s \leftarrow r + s$
end if
$v \leftarrow \frac{v}{2}$
$r \leftarrow 2r$
if b_{swap} is true **then**
    swap u and v
    swap r and s
end if

Since the swap has nearly the same cost as a binary shift, this algorithm requires one modular addition less, without adding any extra qubit.

Another improvement regards phase 2 of algorithm 2: instead of correcting the pseudo-inverse using a separate circuit, a doubling operation is added to each round of the uncomputation controlled by the same controller of the block operation toggled, so we either perform the block or double the output. This permits to reduce the depth and also to save the l qubits needed to copy the k register.

Providing these improvements, together with the use of windowed arithmetic for Montgomery multiplication, they estimated an asymptotic number of $8n+10.2\lfloor \log_2 n \rfloor - 1$ logical qubits needed for the low-width version, with a depth of $506n^3 - 1.84 \cdot 2^{27}$.

10.4 Implementation in Qiskit

In this work, we provide a list of functions written using the Qiskit API to construct the circuit described in this chapter. We also report tests performed with the ibmq_qasm_simuator in an external GitHub repository to prove the correctness of all the functions up to the one that creates the block for the Kaliski's algorithm (see figure 10.6), since we have not been able due to resources limitations to testing forward. In this section, we will explain the implementation starting from the code for the addition modulo p, until the inversion algorithm.

The addition and the doubling modulo p, together with their controlled versions, follow step-by-step the circuits in section 10.2.1. The comparator is in this case built using a subtraction and an addition, without actually computing the carry.

```
def comparator(circuit,x,y,c,n):
    subtraction(circuit,y,x,c,n)
```

```
    addition_without_carry(circuit,y,x,n)
```

Listing 10.1. Circuit to create the comparator gate using an addition and a subtraction.

In this way, $|c\rangle$ will be toggled if $x < y$, and the value on register $|x\rangle$ is restored without affecting $|c\rangle$.

The code for the modular multiplication and squaring with the double-and-add technique does not need any important consideration to be understood. Their Montgomery counterparts are built creating a subcircuit that represents figure 10.5, that is appended normally the first time and inversely the second time, after having copied the result in another register.

```
...
circuit.append(subcircuit,x[0:n]+y[0:n]+c[0:1]+z[0:n]+g[0:1]+m[0:n])
for i in range (0,n):
    circuit.cx(z[i],r[i])
circuit.append(subcircuit.inverse(),x[0:n]+y[0:n]+c[0:1]+z[0:n]+g[0:1]+m[0:n])
```

Listing 10.2. The Montgomery multiplication is performed appending the gate (`subcircuit`) represented in figure 10.5, copying out the result, and appending its inverse.

In the block function, useful to recreate the circuit in figure 10.6, the doubly controlled addition and subtraction are created without considering the carry bit in the operation, since these operations will never toggle it. We implemented also the improved version with swaps since the controlled version can be implemented controlling only the first 2 gates (a CNOT and a Toffoli), the doubling, and the halving. In this, we performed a correction on the circuit described in [101]: before controlling the addition and the subtraction, the $|a\rangle$ qubit (called cc in our code) shall be toggled, controlling this xoring gate in the controlled version.

To build the round function, we first need a function that toggles a qubit if all the qubits in a register are in state $|0\rangle$ or $|1\rangle$: this is called register_cnot.

```
def register_cnot(circuit,x,t,type,n_ctrl):
    if type=='0':
        for i in range(0,n_ctrl):
            circuit.x(x[i])
        circuit.mcx(x,t)
        for i in range(0,n_ctrl):
            circuit.x(x[i])
    else:
        circuit.mcx(x,t)
```

Listing 10.3. Python code that performs using Qiskit a Toffoli gate where the controllers are all the qubits of a register. If the type is '0', then the X-gate is applied if all the qubit of the register are in state $|0\rangle$, while if the type is '1' the X-gate is applied if all the qubit of the register are in state $|1\rangle$

The "type" variable represents the state that the controller must have to change the target qubit. The block presented in 10.6 is reported, but the controlled version is built using the version with the swaps of algorithm 3 since the only gates needed to be controlled are the first two gates (a CNOT and a Toffoli), the cyclic doubling and the cyclic halving. Here we also report an error in the circuit presented in [101]: the ancilla qubit that controls the addition and the subtraction needs to be toggled before performing the operations, and this X-gate is controlled in the controlled version of the circuit. This block is used in the round function and performed $2n$ times to calculate the wanted value, that is the pseudo-inverse of the input, so we have to perform the modular doublings needed, that are encoded in the k register and are up to n.

```
for i in range(0,n+1):
    constant_subtraction(circuit,k[0:l]+f[0:1],ca,i+1,l+1)
    controlled_doubling_modp(circuit,f,r,ca,cb,p,n)
    constant_addition(circuit,k[0:l]+f[0:1],ca,i+1,l+1)
bitwise_complement(circuit,r,n)
```

Listing 10.4. Part of code to create the circuit to calculate the inverse starting from the pseudo-inverse calculated with the Kaliski's allgorthm.

In this snippet of code, we subtract iteratively a growing classical value i to the register containing k, so that the ancilla qubit f is toggled only if $i > k$ and we perform the doubling depending on the value of f. After this step, the modular inversion circuit is run backward to restore the ancillae.

Chapter 11

Test and validation

In this chapter, we describe the test results regarding our solution. We adopted different metrics to conduct them:

- The time of execution is the amount of time that the simulator needed to calculate the output of the circuits;
- The depth, that we have described section 4.2, of the circuits where every gate has been decomposed in gates understandable by the IBM devices; and
- The quality of the solutions, that is the percentage of times an execution of a circuit gives as output a value that permits to retrieve the two factors.

To conduct the tests, we took into consideration the *ibmq_qasm_simulator*, which we described in section 4.2.1. We have not been able to use real quantum hardware because of user restrictions of IBM, which permits to use only 5-qubit devices to free users. Having full access to the IBM quantum devices would permit us to have real quantum computers of up to 127 qubits, but it is important to note, as we will see in section 11.3, that the quantum noise that is currently present in those devices would not permit us to obtain good results from long circuits like ours. The QV, a metric that measures the capabilities of a quantum device that we have described in section 4.2.2, of the most powerful device is 64, so it would be necessary to use a Quantum Error Correction Code to find good results.

Another option that we had was to use a local simulator, but we found the one given by IBM to be more powerful. The *ibmq_qasm_simulator* is a 32-qubit general-context simulator that can be programmed to execute every wanted circuit both ideally and subject to noise modeling.

In this section we will present the following tests:

- Study of the time of execution, the depth and the quality of the solutions of our implementations of the circuit for factorization;
- Demonstration of the obtained decrease in depth using our optimizations for the controlled gates for the circuit for ECDLP and estimation of the total depth of the circuit;
- Study of the behavior of our circuit for factorization simulating real noise; and
- Demonstration of the portability of our circuit simulating it on a Rigetti quantum simulator.

11.1 Tests on the implementations of the circuit for factorization

In this section, we describe the results obtained testing the four implementations for the factorization problem present in our library. These are:

- The aqua implementation ($4n + 2$ qubits);
- The Toffoli based implementation ($4n + 1$ qubits);
- The aqua implementation with the sequential QFT ($2n + 3$ qubits); and
- The Toffoli based implementation with the sequential QFT ($2n + 2$ qubits).

We will analyze for each of these implementations how the retrieved values differ from the expected ones with respect to the depth and the quality of solutions, recalling that the probability of obtaining a good value from the quantum circuit is at least $\frac{\phi(r)}{3r}$. The semiprimes that we use to test the circuits are: 15, 21, 33, 35, 39, 51, 55, 65, 77, 85, 91, 119, and 143. For resource and time limitations, we have been able to test the circuits with the normal QFT up until moduli with 6 bits (up until 55), using only 100 shots for simulation. This number may be low to retrieve good results with certainty, so more tests should be conducted in future works. We have been able to perform simulations with 2000 shots for the sequential versions until 55.

The first thing we studied is the time consumed by each of these simulations with respect to the method used. We can see from figure 11.1 how much the number of qubits used by the circuit affects the time needed by the simulations: the sequential QFT permits to significantly reduce it. Another important aspect that we can understand from figure 11.1 is the unexpected more rapid growth of the Toffoli-based method with respect to the basic one. In fact, if we look at the data in table 10.1, we expect to see that the basic method requires more time, but that does not happen.

The distance between our data from the ones in table 10.1 is present because of the decomposition we do of the circuit: in their estimation, a single Toffoli gate has count 1, while in our estimation it has an increasing count depending on the number of controller qubits.

In figure 11.2 we can see how the depth of the Toffoli-based implementation grows significantly faster than the one of the basic implementation in the decomposed circuit. Many factors can cause this, but it is to note that in the decomposed circuit the Toffoli gates are divided into a series of single-controlled gates, causing a notable increase in the depth and there are more Toffoli gates in the Toffoli-based implementation than in the basic one.

We have also important variations between the circuits with the normal QFT and the sequential QFT. This happens because in the first case it is possible to parallelize many operations, starting a U_a gate when the preceding one is not finished, since the control qubit is different, in opposition to the second case.

It is to note that the circuits using the Toffoli-based implementation with the same modulus have small variations in the depth concerning the value a, but these are negligible in the case of figure 11.2. These variations are present due to the construction of the carry gate, which inserts gates depending on the Hamming weight of the input: this gate is not present in the basic implementation, so there are no variations in it. The situation is described in figure 11.3 in the case of $N = 33$.

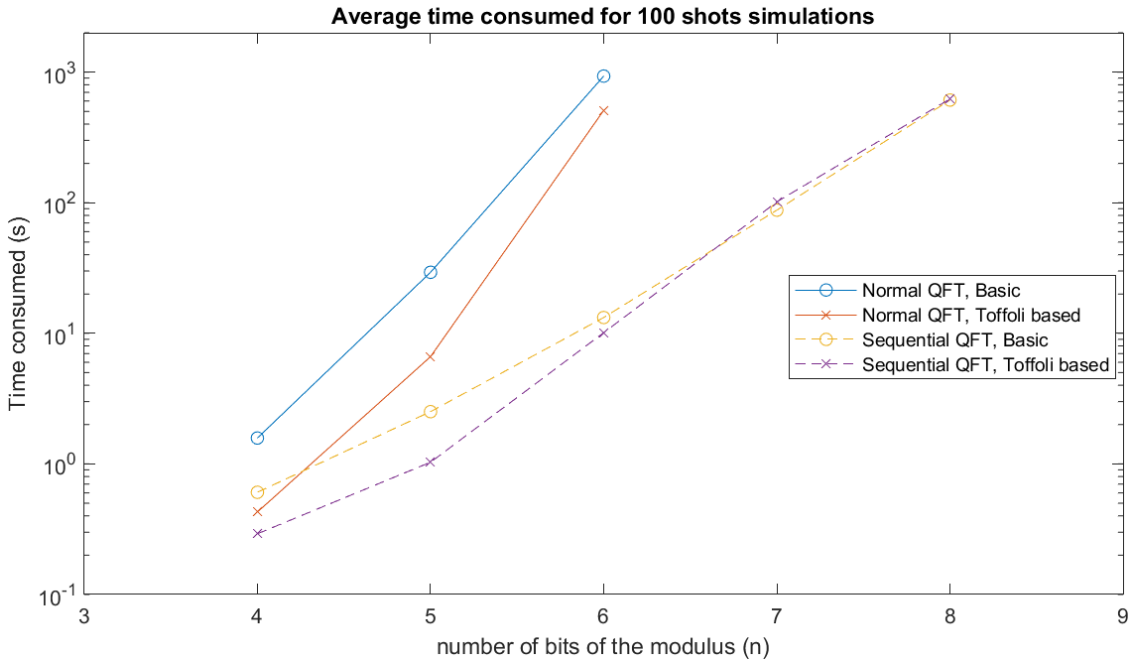


Figure 11.1. Average time consumed for each method performing simulations with 100 shots at the growth of the number of bits of the modulus n. The test was conducted using the *ibmq_qasm_simulator* without the presence of noise. The circuits using the normal QFT are not represented for $n > 6$ for resources limitations.

We can now concentrate on the quality of the solution given by the different circuits, that is the percentage of times we have been able to obtain a value that can be transformed into the period of the function. We also compare it to the minimum theoretical expected percentage of success, which is calculated as in section 8.2 as $\frac{\phi(r)}{3r}$.

In figures 11.4, 11.5, and 11.6 we see the different values that we obtain with the different implementations when the number of bits of the modulus and the value a changes. An important result is that whenever we have situations where the algorithm fails, each of the 4 implementations fails. As we can see, we have differences in the quality of the solution for the Basic and the Toffoli-based method, but for $n = 4$ and $n = 5$, the use of the sequential QFT does not seem to affect it (small variations are normal since we performed only 100 shots for the Normal QFT circuits).

However, if we take a look at the cases where $n = 6$, that are the bar graphs with $N = 33$ and $N = 55$, we can see that the quality of the solution of the circuit with the normal QFT and the Basic method collapses, also with respect to the circuit with the same method but with the sequential QFT. In this case, the circuit used 26 qubits, so if the only limitation of the simulator is the number of quits, it should be simulated without any problem and obtain similar results to the sequential QFT version. This tells us that the *ibmq_qasm_simulator* have other resource limitations other than only the number of qubits, that do not permit to maintain the same performances on the quality of the solution for every circuit with less than 32 qubits.

Another important aspect that can be retrieved by this figure, is that the obtained quality of the solution is bigger than expected: this can be seen also for greater moduli in figure 11.7. From this figure, we can also see that the quality of the solution of the Basic method is better than the one of the Toffoli-based: the only cases where this does not happen are when $n = 6$, where the

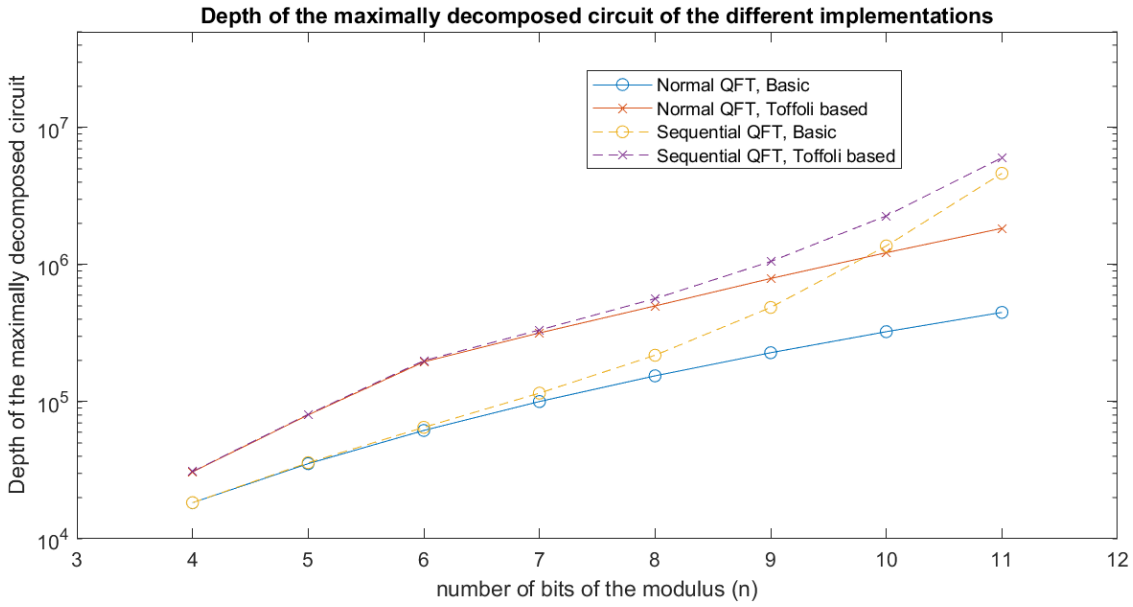


Figure 11.2. Retrieved depth of the circuits with different modulus bit length and $a = 2$. The data refers to maximally decomposed circuits, that were created using the qiskit function `decompose`. The Toffoli-based implementation has a depth increase that is higher than the one of the Basic implementation, in contrast with the theoretical data, because of the decomposition of the Toffoli gates. The sequential QFT versions have a higher depth than the normal one, because of the parallelization that is possible in the circuits with the normal QFT.

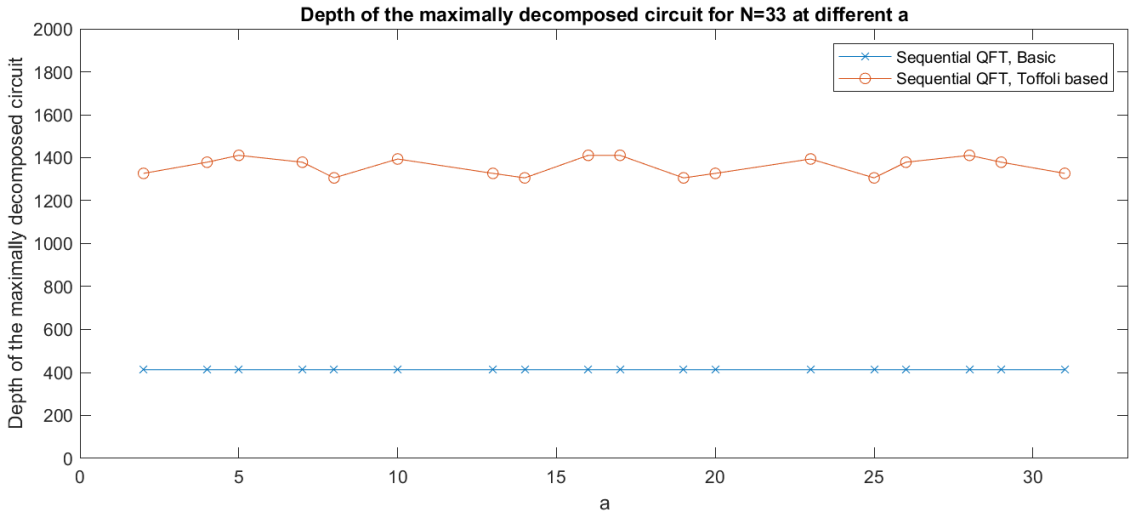


Figure 11.3. Retrieved depth of the circuits for the doubly controlled modular multiplication with $N = 33$ at different value of a. The data refers to maximally decomposed circuits, that were created using the qiskit function `decompose`. The Toffoli-based implementation has small variations in the depth, that are not present in the Basic implementation. This is caused by the use of the carry gate in the Toffoli-based circuit.

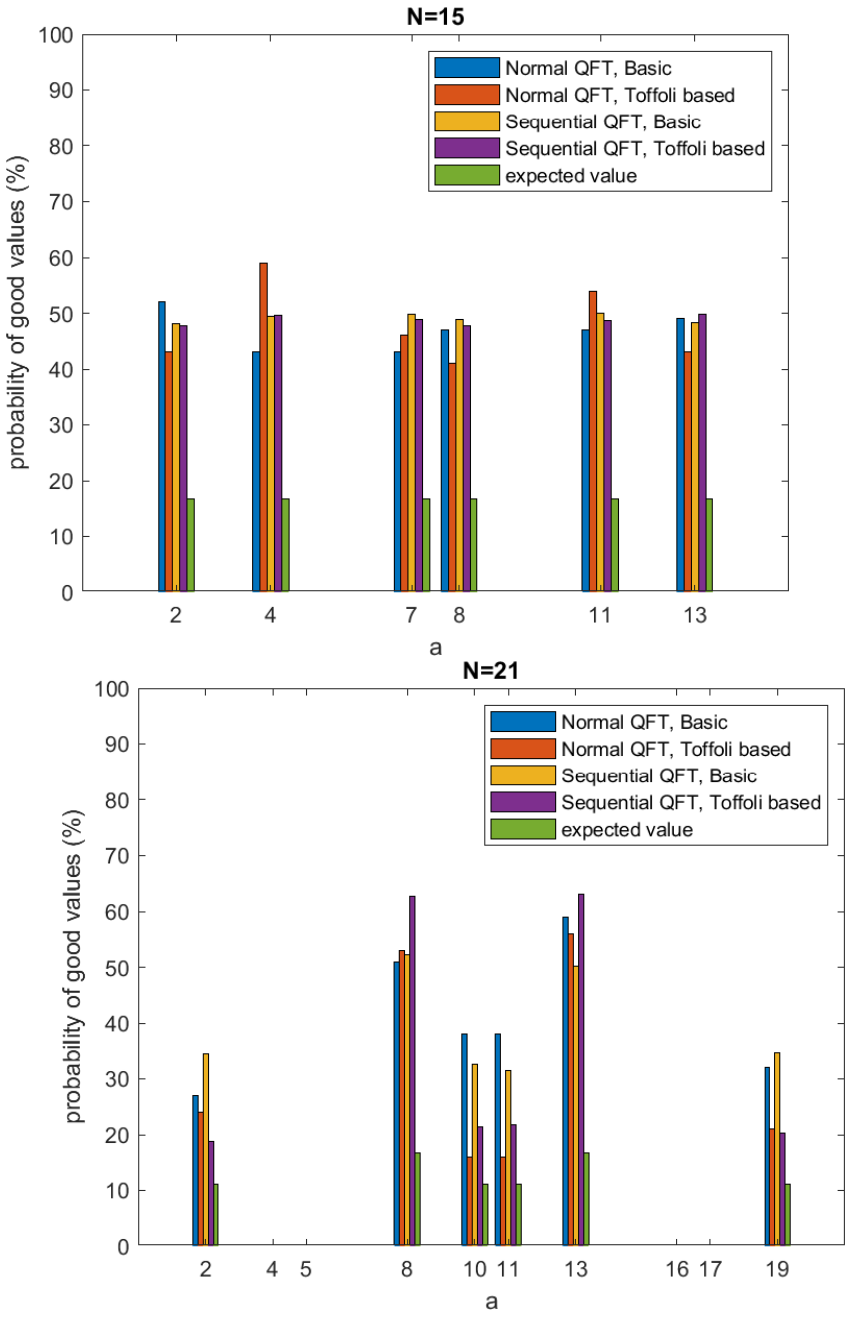


Figure 11.4. Retrieved probability of success of the different implementations against the minimum expected value for different a for N=15 (top) and N=21 (bottom). The tests were conducted using the *ibmq_qasm_simulator* without the presence of noise. These tests shows show a probability of success that strongly depends on the value a and on the method used, and that is greater the minimum expected value.

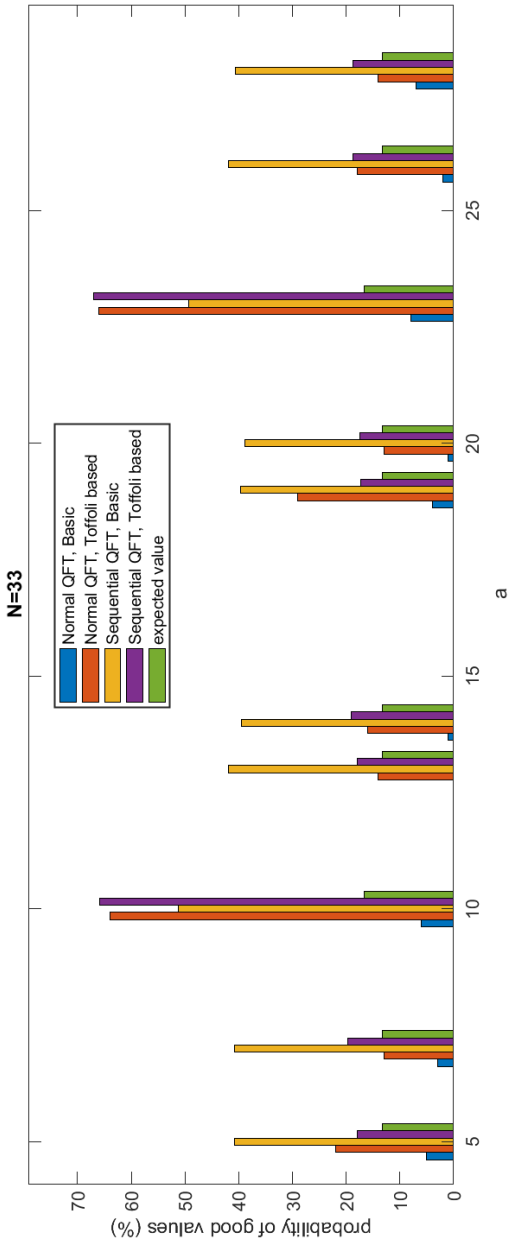


Figure 11.5. Retrieved probability of success of the different implementations against the minimum expected value for different a for N=33. The test was conducted using the *ibmq_qasm_simulator* without the presence of noise. This test shows how the probability of success of the circuit using the normal QFT and the Basic method is lower than the one of the circuit using the same method and the sequential QFT.

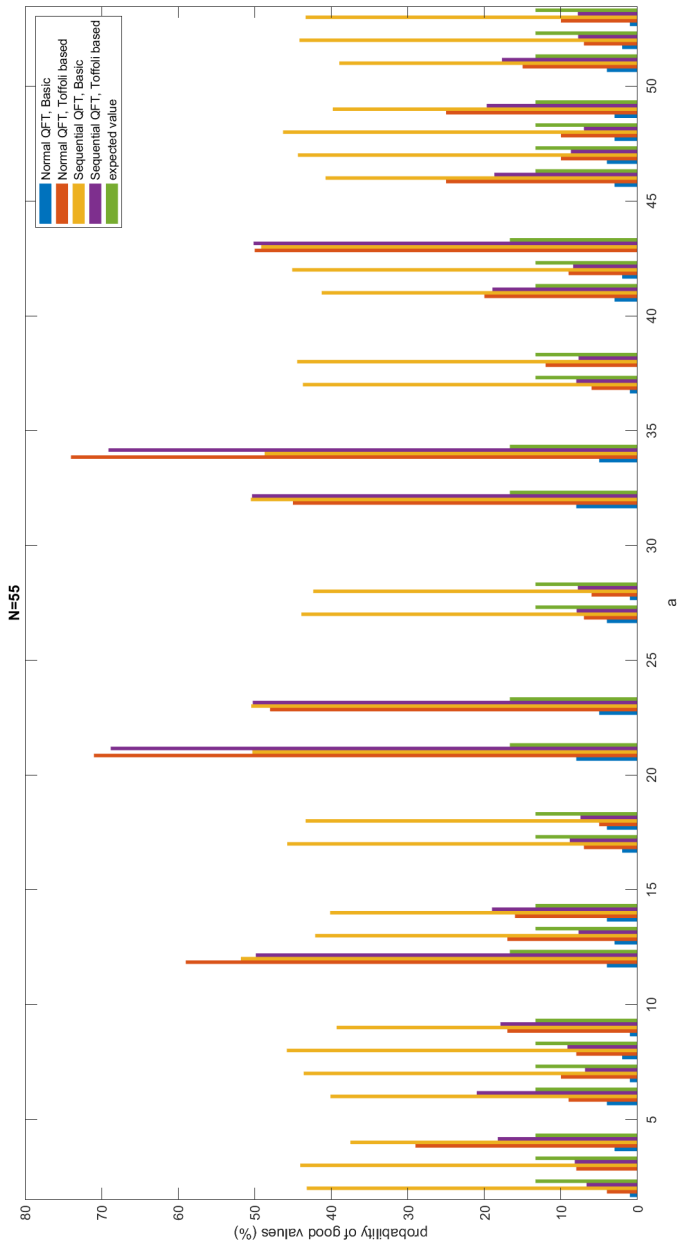


Figure 11.6. Retrieved probability of success of the different implementations against the minimum expected value for different a for N=55. The test was conducted using the *ibmq_qasm_simulator* without the presence of noise. This test shows how the probability of success of the circuit using the normal QFT and the Basic method is lower than the one of the circuit using the same method and the sequential QFT.

Basic method probability is affected by the limitations of the *ibmq_qasm_simulator*.

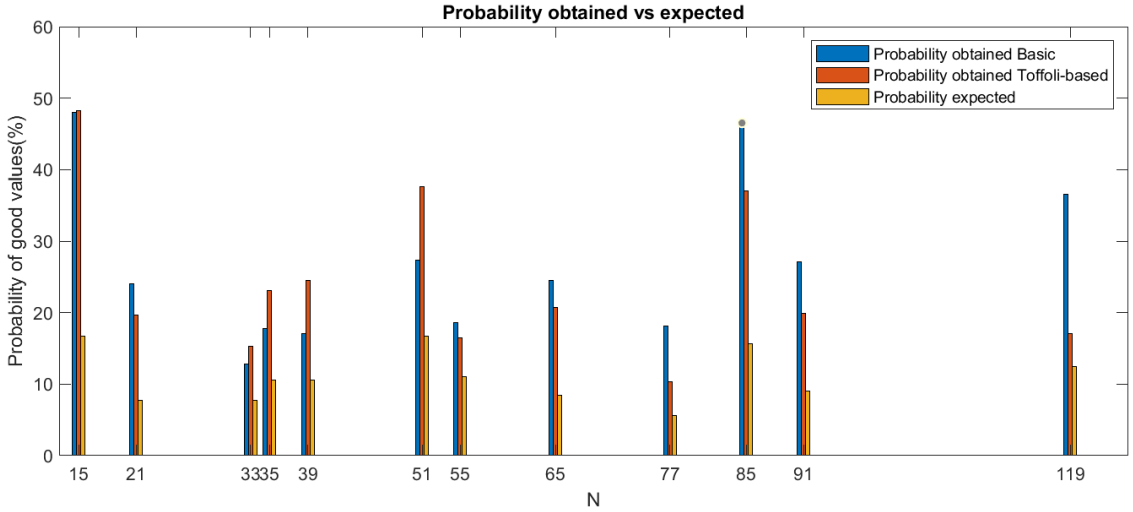


Figure 11.7. Average quality of the solution at different values of N compared with the expected minimum. The test was conducted using the *ibmq_qasm_simulator* without the presence of noise. The graph shows how (with the exception of the circuits where $n = 6$), the Basic method has a better probability of success than the Toffoli-based method, even if the latter remains higher than the minimum expected value.

It is to note that the expected probability is asymptotical, so the important differences between these two values can be understood recalling that we consider only small moduli.

We used our library to build a toy example of an RSA breaker. This program starts with the creation of an asymmetric RSA key pair from the inputs p and q, that is used to encrypt a string of bytes, using the code in Appendix A.2. The modulus and a random a are passed as input to the library, which creates the circuit and executes it with 100 shots in the *ibmq_qasm_simulator*. If no good values are returned, a different a is chosen and the circuit is recreated until the factors are found. With these two factors, it is now possible to recreate the secret key and decrypt the message.

We run this program with different values of p and q both with the Basic and the Toffoli-based method. We can discuss the values retrieved testing the program with $p = 23$ and $q = 17$, which give $N = 391$ (and $n = 9$): we tried these inputs using the sequential QFT and the two different methods. It is to note that a correct implementation would try each output of the circuit every time one is given, but for compatibility with our library and to avoid the problem of the queue time of IBM simulators, we tested each random a using 100 shots of the circuit. The Basic circuit has been able to try the factors using the first random a with probability 51%, spending 2881.573 seconds for the whole computation: this means that, given a good a, in average, it is possible to find a good solution in 56.50 seconds[1]. For the Toffoli-based method, we had to try two different values of a and the second computation succeded with a probability of 2%: it is possible that this number is affected by the limitations of the *ibmq_qasm_simulator* that we have discussed. The time needed to perform the computations was also higher than the one for the Basic method: 3116.345 seconds with the first a, and 4566.803 seconds with the second.

[1] (number of seconds per shot)/(probability of success)

We considered the importance to build an optimized controlled version of certain gates: we built different circuits to perform the controlled constant addition using an optimized version of the controlled carry gate in figure 8.9 (where the only controlled gates are the two CNOTs acting on the carry qubit) and similar circuits where the controlled carry gate was done simply controlling each gate of the same figure, using different constants with a bit-length of 9. We then measured the maximally decomposed circuit depth and the time of execution in the case of a simulation with only 1 shot. We obtained in this way the following results:

- For the optimized version, an average time of 2.8 ms and an average depth of 958; and
- For the basic version, an average time of 29.4 ms and an average depth of 8781;

This demonstrates the important speed-up that is obtained for the whole circuit for Shor's algorithm since it contains $8n^2$ doubly controlled carry gates, in addition to the ones needed for the controlled constant addition.

11.2 Tests on the implementations of the circuit for ECDLP

For our ECDLP circuit implementation, we provide all the functional tests conducted, that demonstrate the correctness of our functions. These are listed in Appendix A.3, and we have been able to simulate them up to the block for the modular inversion.

An important test, similar to the one done in section 11.1 for the controlled constant addition, has been conducted for the optimized controlled version of the block, that follows algorithm 3, instead of just controlling each gate of algorithm 2. To demonstrate also the correctness of the implementations, we used as input the values of figure 8 in [98]: using modulo 11, the pseudoinverse of 7 is 8, and the one of 8 is 6. We obtained, performing the needed iterations of the block, changing at each of them the input encoded in the registers with the output of the preceding iteration, the same evolution of the values as the theoretical one.

We found that using our block, we have been able to reduce the maximally decomposed circuit depth from 344041 to 2122 both for the inputs 7 and 8, going from an average time of execution of 3.553 seconds to 0.336 seconds in the first case using only one shot, and from 6.090 seconds to 0.332 seconds using as input 8.

Even though we could not simulate the circuit, we can still provide an estimation of the depth for it, summing the ones needed for the different parts of the algorithm. However, this estimation will not take into account the possibility of parallelization between the gates, which can be an important factor as we have described in the last section. The whole algorithm consists of 4 inversions, 2 squares, 4 multipliers, and 2 controlled constant additions. Important optimizations are described in section 10.3.1, permitting to use 2 inversions, 1 square, 4 multipliers, and 2 controlled constant additions.

The multiplications can be performed using the Montgomery modular multiplication as we have many ancilla qubits present, given that this gate has a lower depth with respect to the normal one, as we can see from figure 11.8.

Having these data, we calculated the depth of the maximally decomposed circuits of the inversion, of the squaring, and of the controlled constant adder, summing them following the construction of the basic and the optimized circuits. The results are shown in figure 11.9, both for the single U_a gate and for the whole circuit.

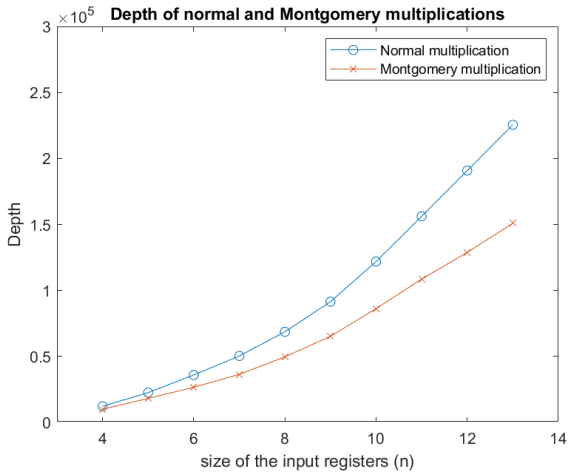


Figure 11.8. Depth of the normal multiplication and of the Montgomery multiplication circuits (without the copy of the result and the decomputation of the qubits) for input registers with a qubit-size from 4 to 13. It is clear how the Montgomery multiplication has a lower depth.

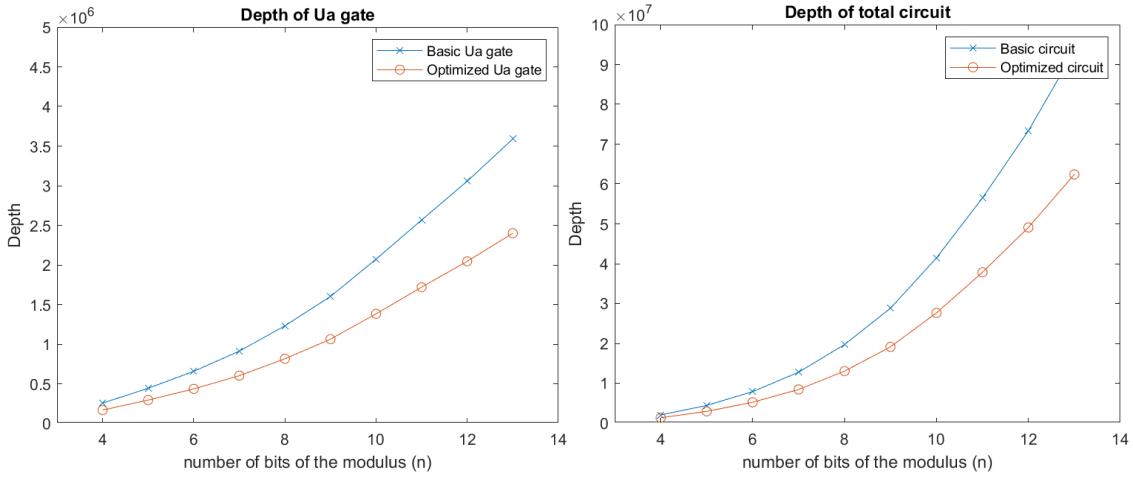


Figure 11.9. Estimation of the depth for the single U_a gate (left) and for the whole circuit (right) using the basic and the optimized circuits, for moduli that have bit-length from 4 to 13.

It is important to note that these plots derive from a simple sum of the depth of the single gates. As we have seen in figure 11.2, the parallelization of the gates permits to reduce the depth in a significant way. Important tests can be conducted creating the whole circuit and measuring its depth.

11.3 Tests in presence of noise

As we have said, we have not been able to execute our circuits in real quantum devices. However, it is possible to perform simulations where the qubits and the gates are affected by simulated real errors. It is in fact possible to manually set all the noise configurations or to use default ones that

simulate both the errors and the coupling maps of real quantum devices of IBM. For this purpose, we used the *qiskit.test.mock* library, trying to see the results obtained from the execution of the whole circuit for factorization in an environment that simulates the *ibmq_montreal* device. This quantum device has 27 qubits, a QV of 128, and a topology described in figure 5.6. In figure 11.10, we see the effect of the noise in a simple circuit that performs two cyclic binary halvings on a register that was initialized encoding in it the value 1 in the computational basis, that is part of the ECDLP library. This circuit consists of just n swaps, where n is the size of the register.

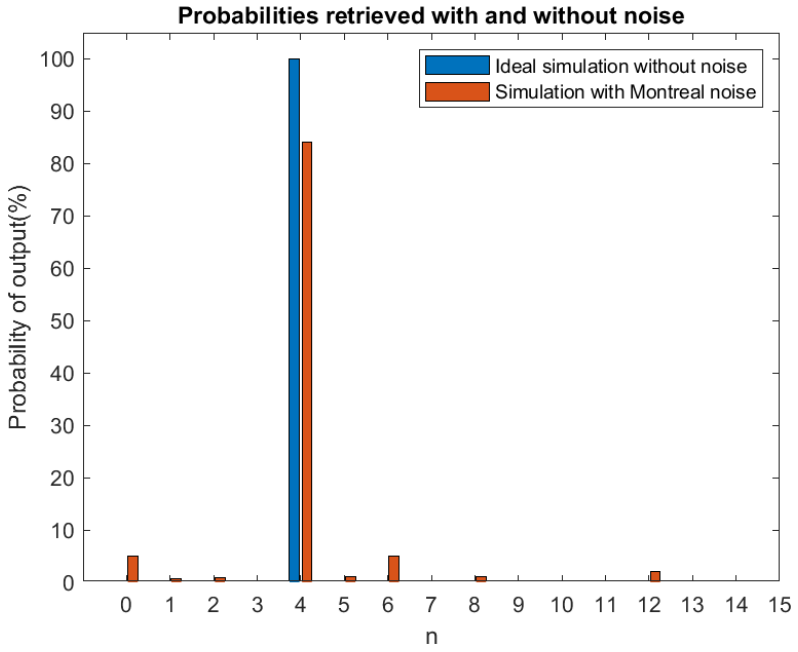


Figure 11.10. Results of the simulation of a circuit created using two binary halving gates created with only swap gates in the *ibmq_qasm_simulator* without the noise and with the *ibmq_montreal* noise and coupling map using 1000 shots.

In this figure, we can see how a simple circuit is already greatly affected by the presence of noise. In fact, when we try to perform the circuit to factorize the number 15 using the Sequential QFT and the Toffoli-based method performing 100 shots, no good solutions are highlighted, as shown in figure 11.11

This test demonstrates the need for an Error Correction Code to execute our circuits, as otherwise, it would be impossible to have higher probabilities of obtaining good values.

11.4 Rigetti simulator

To simulate the circuit for factorization using a different simulator, we used the Quantastica web application to transform our circuit written in qiskit language in *pyquil*, which is understandable by the Rigetti simulator, as described in appendix B.3.

This test demonstrates the exportability of our circuits, providing a simulation that does not rely on the *ibmq_qasm_simulator*. For this purpose, we used the Rigetti PURE-STATE-QVM, an implementation of a quantum virtual machine that evolves pure state quantum systems, where

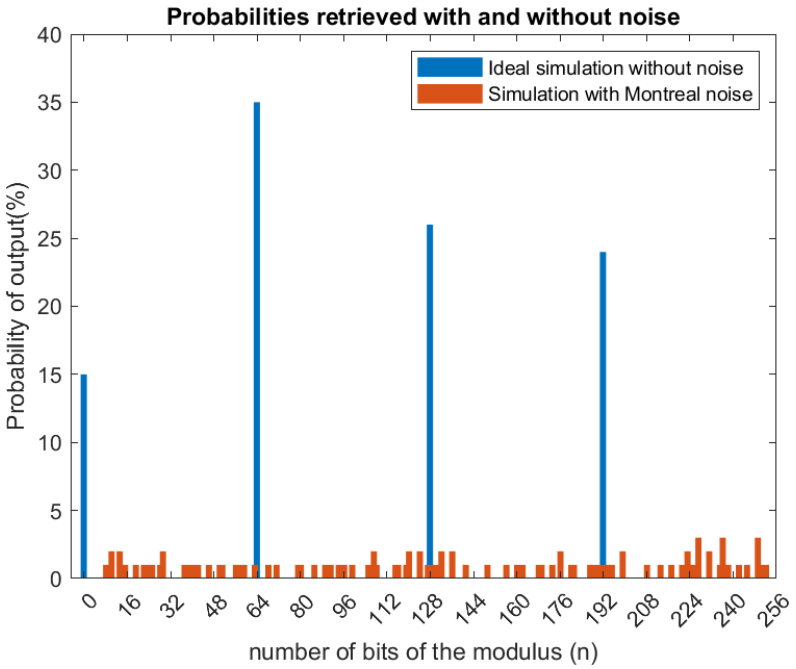


Figure 11.11. Results of the simulation of Shor's circuit with Sequential QFT and the Toffoli-based method in the *ibmq_qasm_simulator* without noise and with the simulation of the *ibmq_montreal* noise and coupling map using 100 shots.

the wavefunction of the quantum system is represented by a 2^n qubit vector of amplitudes. In this situation, the number of qubits is limited by the memory that can be allocated to it. In addition, being a local simulation, the time of the computation is greatly affected both by the performances of the local machine and by the width of the circuit.

We tested our circuit on this simulator trying to factorize $N = 15$ using $a = 2$ with the 4 different implementations, and $N = 21$ using $a = 2$ with the sequential QFT versions. The comparison between the results obtained with this quantum simulator and the *ibmq_qasm_simulator* are reported in figure 11.12.

From this, we can see how it is possible to find good solutions even if we use a different type of simulator. We have not been able to go further with the number of qubits of the circuit for resources limitations, so it would be interesting as future work to try the circuit with bigger moduli and also in other architectures.

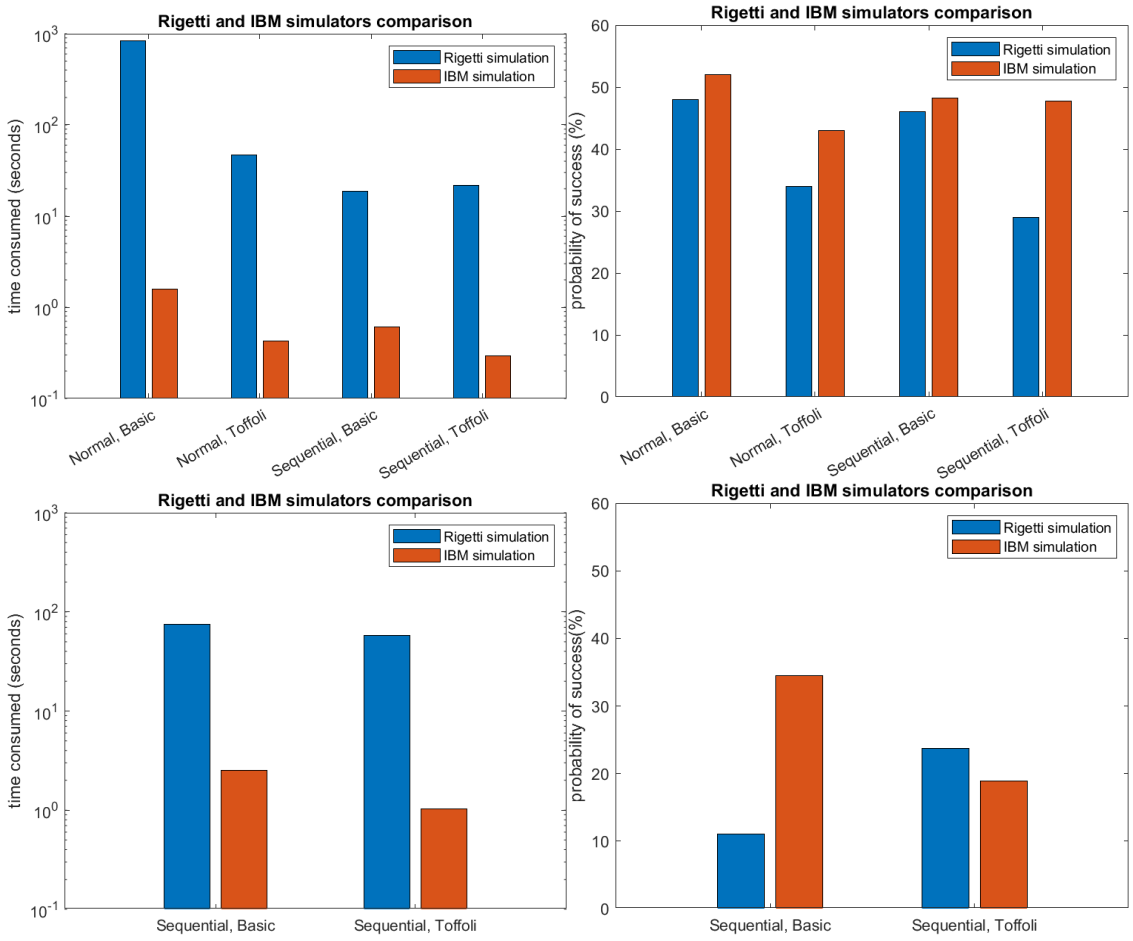


Figure 11.12. Comparison between the results obtained with the *ibmq_qasm_simulator* and the Rigetti QVM for $N = 15$ (top) and $N = 20$ (bottom) using 100 shots. The time needed is orders of magnitude higher for the Rigetti QVM, being it a local simulator, but the probability of success are similar in the two cases.

Chapter 12

Conclusions

The objective of this thesis work has been to analyze Shor's algorithm, in particular how it can be used as an attack vector for RSA and elliptic curve cryptography. We tried to understand why these cryptosystems are vulnerable and how distant we are from the possibility to build actual attacks against them.

For the factorization circuit, we have seen how the quantum part is able to find with good probability a value that can be used to retrieve the factors, and that the circuit runs in polynomial time. We have studied two different implementations found in the literature, understanding their construction and their resource estimations. The current Qiskit implementation is based on one of these methods, so we implemented the needed gates for the other one. In this way, we have been able to improve the current Qiskit implementation of Shor's algorithm, providing different gates that were not present in the SDK. We also modified some checks and operations performed during the classical part of the algorithm, in order to avoid losing good solutions.

In section 11.1 we have proved that the probability to find good solutions using our library is high as predicted, even if the resources limitations did not permit us to try to factorize numbers that are representable with more than 9 bits.

Putting everything together, we created a parametrized library to facilitate the usage of these circuits, providing the possibility to directly retrieve the wanted factors without knowledge of the quantum part of the algorithm.

For the ECDLP circuit, we have understood why it represents a more urgent threat than the first one as well as its challenges. Starting from a basic construction of the oracle, we analyzed each gate, studying the resources needed for it. In this case, we implemented using Qiskit all the different gates from scratch, providing some optimization to the basic circuit like optimized controlled gates, and tested them, creating a starting point for the construction of the whole circuit and gates that can be used also for other purposes.

We proved the portability of the implementations in section 11.4, performing the simulations in a local Rigetti simulator, but it would be interesting to try other architectures, and evaluate different other optimizations for Shor's algorithm. The most important work to do in the future is to test and use the library with real quantum devices, that are not available at the moment.

It is to note that Quantum Error Correction, as we have reported various times during this thesis work, represents the most important player in the development of fault-tolerant quantum devices, so future implementations and enhancements of Shor's algorithm must be done according to the future directions of this field.

Bibliography

[1] E. Yndurain, S. Woerner, and D.J. Egger, "Exploring quantum computing use cases for financial services", September 2019, https://www.ibm.com/thought-leadership/institute-business-value/report/exploring-quantum-financial

[2] IBM Quantum Project, https://quantum-computing.ibm.com/

[3] F. Arute, K. Arya, R. Babbush, et al. "Quantum supremacy using a programmable superconducting processor", Nature 574, 505-510, October 2019, DOI 10.1038/s41586-019-1666-5

[4] J. Gambetta, "IBM's roadmap for scaling quantum technology", September 2020, https://research.ibm.com/blog/ibm-quantum-roadmap

[5] NIST. Post-Quantum Cryptography Standardization, January 2017. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[6] U. Vazirani, and T. Vidick, "Fully Device-Independent Quantum Key Distribution", "Physical Review Letters", Vol. 113, No. 14, June 2014, DOI 10.1103/PhysRevLett.113.140501

[7] P.W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", SIAM Journal on Computing, Vol. 26, No. 5, January 1996, pp. 1484-1509, DOI 10.1137/S0097539795293172

[8] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3" RFC-8446, August 2018, DOI 10.17487/RFC5246

[9] B. Marshall, "How does ECDSA work in Bitcoin", February 2018, https://medium.com/@blairlmarshall/how-does-ecdsa-work-in-bitcoin-7819d201a3ec

[10] Qiskit project, https://qiskit.org/

[11] M. Santelia, and V. Saling, "What is quantum computing?", IBM, May 2020, https://www.ibm.com/quantum-computing/what-is-quantum-computing/

[12] S.P. Jordan, "Quantum Computation Beyond the Circuit Model", Massachusetts Institute of Technology, May 2008

[13] De Voorhoede, "Superposition and entanglement", https://www.quantum-inspire.com/kbase/superposition-and-entanglement/

[14] F. Giacosa, "On unitary evolution and collapse in Quantum Mechanics", Quanta 2014, Vol. 3, November 2014, DOI 10.12743/quanta.v3i1.26

[15] N.P. Landsman, "Born Rule and its Interpretation", "Compendium of Quantum Physics" (D. Greenberger, K. Hentschel, and F. Weinert), Springer, 2009, pp. 64-70, DOI 10.1007/978-3-540-70626-7_20

[16] E. Andersson, Lecture notes for WAQCT Summer School, August 2019, Heriot-Watt University

[17] A. Ketterer, "Modular variables in quantum information", October 2016

[18] IBM. Quantum composer user guide, Visualizations, https://quantum-computing.ibm.com/composer/docs/iqx/visualizations

[19] P. Krammer, "Quantum Entanglement: Detection, Classification, and Quantification", October 2005, Universitat Wien

[20] J. Pieper and M.E. Lladser, Quantum Computation, Scholarpedia, 2018, DOI 10.4249/scholarpedia.52499, http://www.scholarpedia.org/article/Quantum_Computation
[21] M. Schlosshauer, "Quantum Decoherence", Physics Reports, Vol. 831, pp. 1-57, November 2019, DOI 10.1016/j.physrep.2019.10.001
[22] C.P. Williams, "Explorations in Quantum Computing", Springer, 1997, ISBN 978-1-84628-887-6
[23] C.M. Lee and J.H. Selby, "Generalised phase kick-back: the structure of computational algorithms from physical principles", "New Journal of Physics", Vol. 18, No. 3, October 2015, DOI 10.1088/1367-2630/18/3/033023
[24] Yu.I. Bogdanov, N.A. Bogdanova, D.V. Fastovets, and V.F. Lukichev, "Representation of Boolean functions in terms of quantum computation", International Conference on Micro- and Nano-Electronics 2018, Zvenigorod (Russian Federation), October 1-5, 2018, DOI 10.1117/12.2522053
[25] P.W. Shor, "Scheme for reducing decoherence in quantum computer memory", "Physical Review A", Vol. 57, No. 2, October 1995, DOI 10.1103/PhysRevA.52.R2493
[26] L. Egan, D.M. Debroy, C. Noel, A. Risinger, D. Zhu, D. Biswas, M. Newman, M. Li, K.R. Brown, M. Cetina, and C. Monroe, "Fault-Tolerant Operation of a Quantum Error-Correction Code", September 2020
[27] A.G. Fowler, M. Mariantoni, J.M. Martinis, and A.N. Cleland, "Surface codes: Towards practical large-scale quantum computation", "Physical Review A", Vol. 86, No. 3, October 2012, DOI 10.1103/PhysRevA.86.032324
[28] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation", "Mathematical and Physical Sciences", Vol. 439, issue 1907, December 1992, pp. 553-558, DOI 10.1098/rspa.1992.0167
[29] E. Bernstein and U. Vazirani, "Quantum Complexity Theory", SIAM Journal on Computing, Vol. 26, No. 5, 1997, pp. 1411-1473, DOI 10.1137/S0097539796300921
[30] D.R. Simon, "On the Power of Quantum Computation", SIAM Journal on Computing, Vol. 26, No. 5, 1997, pp. 1474-1483, DOI 10.1137/S0097539796298637
[31] M.A. Nielsen and I.L. Chuang, "Quantum Computation and Quantum Information", Cambridge Univ. Press, Cambridge, August 2011, ISBN: 9781107002173, DOI 10.1080/00107514.2011.587535
[32] G. Brassard, P. Hoyer, and A. Tapp, "Quantum Counting", ICALP: International Colloquium on Automata, Languages, and Programming, Aalborg (Denmark), July 13-17, 1998, pp. 820-831, DOI 10.1007/BFb0055105
[33] J. Preskill, "Quantum computing 40 years later", June 2021
[34] X.Fu, L.Riesebos, L.Lao, C.G.Almudever, F.Sebastiano, R.Versluis, E.Charbon, and K.Bertels, "A Heterogeneous Quantum Computer Architecture", CF'16: Computing Frontiers Conference, Como (Italy), May 16-19, 2016, pp. 323-330, DOI 10.1145/2903150.2906827
[35] K. Bertels, R. Nane, I.Ashraf, and X. Fu, "Quantum Computer Architecture: Towards Full-Stack Quantum Accelerators", 2020 Design, Automation & Test in Europe Conference & Exhibition, March 09-13, 2020, pp. 1-6, DOI 10.23919/DATE48585.2020.9116502
[36] H. Bolhasani, and A. Rahmani, "An Introduction to Quantum Computers Architecture", January 2019
[37] C. Adami, and N.J. Cerf, "Quantum computation with linear optics", First NASA International Conference, QCQC'98 Palm Springs (CA), February 17-20, 1998, Quantum Computing and Quantum Communications, pp. 391-401, DOI 10.1007/3-540-49208-9_36
[38] C.D. Bruzewicz, J. Chiaverini, R. McConnell, J.M. Sage, "Trapped-Ion Quantum Computing: Progress and Challenges", "Applied Physics Reviews", Vol. 6, Issue 2, April 2019, id 021314, DOI 10.1063/1.5088164

[39] M. Kjaergaard, M.E. Schwartz, J. BraumÃ¼ller, P. Krantz, J. I-Jan Wang, S. Gustavsson, and W.D. Oliver, "Superconducting Qubits: Current State of Play", March 2020. Annual Review of Condensed Matter Physics, Vol. 11, No. 1, DOI 10.1146/annurev-conmatphys-031119-050605

[40] J. Preskill, "Quantum Computing in the NISQ era and beyond", "Quantum", Vol. 2, August 2018, DOI 10.22331/q-2018-08-06-79

[41] A.W. Cross, L.S. Bishop, S. Sheldon, P.D. Nation, and J.M. Gambetta, "Validating quantum computers using randomized model circuits", "Physical Review A", Vol. 100, No. 3, September 2019, DOI 10.1103/PhysRevA.100.032328

[42] A. Wack, H. Paik, A. Javadi-Abhari, P. Jurcevic, I. Faro, J.M. Gambetta, and B.R. Johnson, "Scale, Quality, and Speed: three key attributes to measure the performance of near-term quantum computers", IBM Quantum, October 2021

[43] T. Grurl, R. Kueng, J. Fuß, and R. Wille, "Stochastic Quantum Circuit Simulation Using Decision Diagrams", 2021 Design, Automation and Test in Europe, DATE 2021, Virtual Conference, February 1-5, 2021, pp. 194-199 DOI 10.23919/DATE51398.2021.9474135

[44] IonQ, "IonQ Opens Door to Dramatically More Powerful Quantum Computers, Debuts Industry First: Reconfigurable Multicore Quantum Architecture", August 2021, https://ionq.com/news/august-25-2021-reconfigurable-multicore-quantum-architecture

[45] Rigetti, "Scalable quantum systems built from the chip up to power practical applications", https://www.rigetti.com/what-we-build

[46] Honeywell, Honeywell System Model H1, https://www.honeywell.com/us/en/company/quantum/quantum-computer

[47] W. Wootters, and W. Zurek, "A single quantum cannot be cloned", Nature, Vol. 299, 1982, pp. 802-803, DOI 10.1038/299802a0

[48] C. Chamberland, G. Zhu, T.J. Yoder, J.B. Hertzberg, and A.W. Cross, "Topological and subsystem codes on low-degree graphs with flag qubits", "Physical Review X", Vol. 100, No. 1, December 2019, DOI 10.1103/PhysRevX.10.011022

[49] D. Bacon, and A. Casaccino, "Quantum Error Correcting Subsystem Codes From Two Classical Linear Codes", 44th Annual Allerton Conference on Communication, Control, and Computing 2006, September 27-29, 2006, pp. 520-527

[50] S. Sheldon, E. Magesan, J.M. Chow, and J.M. Gambetta, "Procedure for systematically tuning up cross-talk in the cross-resonance gate", "Physical Review A", Vol. 93, June 2016, id 060302, DOI 10.1103/PhysRevA.93.060302

[51] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, "On the qubit routing problem", 14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019), June 03-05, 2019, DOI 10.4230/LIPIcs.TQC.2019.5

[52] A.W. Cross, L. Bishop, J. Smolin, and J. Gambetta. "Open quantum assembly language", January 2017

[53] A.W. Cross, A. Javadi-Abhari, T. Alexander, et al., "OpenQASM 3: A broader and deeper quantum assembly language", April 2021.

[54] Oracle Documentation, https://docs.oracle.com/cd/E19575-01/820-2765/6nebir7eb/index.html

[55] I.F. Blake and T. Garefalakis, "On the complexity of the discrete logarithm and Diffie-Hellman problems", "Journal of Complexity", Vol. 20, issues 2-3, June 2004, pp. 148-170, DOI 10.1016/j.jco.2004.01.002

[56] B.D. Allen, "Implementing several attacks on plain ElGamal encryption", Iowa State University, January 2008, DOI 10.31274/etd-180810-1079

[57] L. Valenta, S. Cohney, A. Liao, J. Fried, S. Bodduluri, and N. Heninger, "Factoring as a Service", Financial Cryptography and Data Security: 20th International Conference, FC

2016, Christ Church (Barbados), February 22-26, 2016, pp. 321-338, DOI 10.1007/978-3-662-54970-4_19

[58] M. Nemec, M. Sys, and P. Svenda, "The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli", October 2017, CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas (TX), October 30 - November 3, 2017, pp. 1621-1648, DOI 10.1145/3133956.3133969

[59] NIST. Special Publication 800-57 Part 3 Revision 1, January 2015, DOI 10.6028/NIST.SP.800-57pt3r1

[60] M.E. Briggs, "An Introduction to the General No. Field Sieve", Virginia Polytechnic Institute and State University, April 1998

[61] D.W. Joseph, "Chinese Mathematics", "The Mathematics of Egypt, Mesopotamia, China, India and Islam: A Sourcebook", V.J. Katz; A. Imhausen, Princeton University Press, 2007, pp. 187-384.

[62] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1", Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara (CA), USA, August 23-27, 1998, pp. 1-12, DOI 10.1007/BFb0055716

[63] NIST, SP 800-57 Part 1 Rev. 5, "Recommendation for Key Management: Part 1 - General", May 2020

[64] M. Aydar, S. Cetin, S. Ayvaz, and B. Aygun, "Private key encryption and recovery in blockchain", July 2019

[65] Y. El Housni, "Introduction to the Mathematical Foundations of Elliptic Curve Cryptography", December 2018

[66] D.J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records", 9th International Conference on Theory and Practice in Public-Key Cryptography, New York (NY), April 24-26, 2006, pp. 207-228, DOI 10.1007/11745853_14

[67] K. Yokoyama, M. Yasuda, Y. Takahashi, and J. Kogure, "Complexity bounds on Semaev's naive index calculus method for ECDLP", "Journal of Mathematical Cryptology", Vol. 14, August 2020, pp. 460-485, DOI 10.1515/jmc-2019-0029

[68] D.J. Bernstein, S. Engels, T. Lange, R. Niederhagen, C. Paar, P. Schwabe, and R. Zimmermann, "Faster elliptic-curve discrete logarithms on FPGAs", August 2016

[69] J. Preskill, "Quantum computing and the entanglement frontier", 25th Solvay Conference on Physics ("The Theory of the Quantum World"), October 19-22, 2011

[70] E. Pednault, J. Gunnels, D. Maslov, and J. Gambetta, "On Quantum supremacy", IBM, October 2019, https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/

[71] C.S. Hamilton, R. Kruse, L. Sansoni, S. Barkhofen, C. Silberhorn, and I. Jex, "Gaussian Boson Sampling", "Physical Review Letters", Vol. 119, No. 17, April 2017, DOI 10.1103/physrevlett.119.170501

[72] C.-Y. Lu, J.-W. Pan et al. "Quantum computational advantage using photons", "Science", Vol. 370, No. 6523, December 2020, DOI 10.1126/science.abe8770

[73] Y. Wu, W.-S. Bao et al. "Strong quantum computational advantage using a superconducting quantum processor", "Physical Review Letters", Vol. 127, No. 18, June 2021, DOI 10.1103/PhysRevLett.127.180501

[74] M.-Z. Mina, and E. Simion, "Threats to Modern Cryptography: Grover's Algorithm", September 2020

[75] M.Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's algorithm to AES: quantum resource estimates", 7th International Workshop, PQCrypto 2016, Fukuoka (Japan), February 24-26, 2016, pp. 29-43, DOI 10.1007/978-3-319-29360-8_3

[76] S.Jaques, M. Naehrig, M. Roetteler, and F. Virdia, "Implementing Grover oracles for quantum key search on AES and LowMC", EUROCRYPT 2020, Virtual Conference, May 11-15,

2020, pp. 280-310, i10.1007/978-3-030-45724-2_10
[77] NIST FIPS Publication 197, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)", November 2016
[78] NIST FIPS Publication 202, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, DOI 10.6028/NIST.FIPS.202
[79] M. Amy, O. Di Matteo, V. Gheorghiu, M. Mosca, A. Parent, and J. Schanck, "Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3", 23rd International Conference, St. John's (NL, Canada), August 10-12, 2016, "Selected Areas in Cryptography – SAC 2016", pp. 317-337, DOI 10.1007/978-3-319-69453-5_18
[80] G. Brassard, P. Hoyer, and A. Tapp, "Quantum cryptanalysis of hash and claw-free functions", SIGACT News, Vol. 28, issue 2, January 1997, pp. 14-19, DOI 10.1145/261342.261346
[81] R. Overbeck, and N. Sendrier, "Code-based cryptography", "Post-Quantum Cryptography", D.J. Bernstein, J. Buchmann, and E. Dahmen, Springer, 2009, pp. 95-145, DOI 10.1007/978-3-540-88702-7_4
[82] D. Micciancio, and O. Regev, "Lattice-based cryptography", "Post-Quantum Cryptography", D.J. Bernstein, J. Buchmann, and E. Dahmen, Springer, 2009, pp. 147-191,DOI 10.1007/978-3-540-88702-7_5
[83] J. Ding, and B.-Y. Yang, "Multivariate Public Key Cryptography", "Post-Quantum Cryptography", D.J. Bernstein, J. Buchmann, and E. Dahmen, Springer, 2009, pp. 193-241,DOI 10.1007/978-3-540-88702-7_6
[84] S. Pirandola, U.L. Andersen, L. Banchi, M. Berta et al. "Advances in Quantum Cryptography", "Advances in Optics and Photonics", Vol. 12, No. 4, June 2019, DOI 10.1364/AOP.361502
[85] C.H. Bennett, and G. Brassard, "Quantum cryptography: Public key distribution and coin tossing", International Conference on Computers, Systems & Signal Processing, Bangalore (India), December 9-12, 1984, p. 175
[86] A.K. Ekert, "Quantum cryptography based on Bell's theorem", "Physical Review Letters", Vol. 67, issue 6, August 1991, DOI 10.1103/PHYSREVLETT.67.661
[87] T. Lawson, "Odd orders in Shor's factoring algorithm", "Quantum Information Processing", Vol. 14, No. 3, January 2015, pp. 831-838, DOI 10.1007/s11128-014-0910-z
[88] G.L. Miller, "Riemann's Hypothesis and Tests for Primality", "Journal of Computer and System Sciences", Vol. 13, Issue 3, January 1976, pp. 300-317, DOI 10.1016/S0022-0000(76)80043-8
[89] G.H. Hardy, and E.M. Wright, "An introduction to the theory of numbers", Oxford University Press, 1979, ,ISBN 978-0-19-853171-5
[90] S. Beauregard, "Circuit for Shor's algorithm using 2n+3 qubits", "Quantum information & computation", Vol. 3, No. 2, March 2003, pp. 175-185, DOI 10.26421/QIC3.2-8
[91] R.B. Griffiths and C. Niu, "Semiclassical Fourier Transform for Quantum Computation", "Physical Review Letters, Vol. 76, No. 17, November 1995, pp. 3228-3231, DOI 10.1103/PhysRevLett.76.3228
[92] Y. Takahashi and N. Kunihiro, "A quantum circuit for Shor's factoring algorithm using 2n+2 qubits", "Quantum information & computation", Vol. 6 No. 2, March 2006, pp.184-192, DOI 10.26421/QIC6.2-4
[93] T. Haner, M. Roetteler, and K.M. Svore, "Factoring using 2n+2 qubits with Toffoli based modular multiplication", "Quantum information & computation", Vol. 18, No. 7-8, June 2017, pp. 673-684, DOI 10.5555/3179553.3179560
[94] Y. Takahashi, S. Tani, and N. Kunihiro, "Quantum Addition Circuits and Unbounded Fan-Out", "Quantum information & computation", Vol. 10, No. 9-10, September 2010, pp. 872-890, DOI 10.26421/QIC10.9-10-12

[95] C. Gidney and M. Ekerå, "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits", "Quantum", Vol. 5, April 2021, DOI 10.22331/q-2021-04-15-433

[96] C. Gidney, "Windowed quantum arithmetic", May 2019.

[97] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves", "Quantum Information & Computation", Vol. 3, No. 4, July 2003, pp. 317-344, DOI 10.26421/QIC3.4-3

[98] M. Roetteler, M. Naehrig, K.M. Svore, and K. Lauter, "Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms", Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong (China), December 3-7, 2017, Proceedings, Part II, pp. 241-270, DOI 10.1007/978-3-319-70697-9_9

[99] P.L. Montgomery, "Modular multiplication without trial division", "Mathematics of Computation", Vol. 44, No. 44, April 1985, pp. 519-521.

[100] B.S. Kaliski, "The Montgomery inverse and its applications", "IEEE Transactions on Computers", Vol. 44, No. 8, August 1995, pp. 1064-1065, DOI 10.1109/12.403725

[101] T. Haner, S. Jaques, M. Naehrig, M. Roetteler, and M. Soeken, "Improved quantum circuits for elliptic curve discrete logarithms", The Eleventh International Conference on Post-Quantum Cryptography, Paris (France), September 21-23, 2020, pp. 425-444

[102] C.H. Bennett, "Time/Space Trade-Offs for Reversible Computation", "SIAM Journal on Computing", Vol. 18, No. 4, August 1989, pp. 766-776, DOI 10.1137/0218053

Appendix A

User's manual

This appendix describes how to configure the environment used to implement the libraries and the tests, and how the libraries' functions can be used.

A.1 Using Qiskit with IBMQ devices in Jupyter

The first step to take to use the Qiskit framework is to install it, which can be done running the following command:

```
pip install qiskit
```

To execute this command, Python 3.6 or later must be installed in the environment.

It is recommended to install Anaconda, a cross-platform Python distribution for scientific computing. Jupyter, included in Anaconda, will be used for interacting with Qiskit.

In Jupyter, python notebooks can be written and executed step-by-step, with the possibility to add explanatory text sections. To run it, we open the Anaconda prompt, and execute the following command:

```
jupyter notebook [folder]
```

The folder chosen will be opened in the predefined browser of the environment, so that we can create, manage or delete new python notebooks (.ipynb extension) or python files (.py extension).

When a new python notebook is created, we use the first cell of the file to import the necessary libraries. If we want to create a quantum circuit, the library to import from qiskit is QuantumCircuit, which permits us to initialize the qubits and put every quantum gate that is available in Qiskit.

```
from qiskit import QuantumCircuit
# Create a circuit with a register of three qubits
circ = QuantumCircuit(3,3)
# H gate on qubit 0, putting this qubit in a superposition of |0> + |1>.
circ.h(0)
# A CX (CNOT) gate on control qubit 0 and target qubit 1.
circ.cx(0, 1)
```

```
# CX (CNOT) gate on control qubit 0 and target qubit 2.
circ.cx(0, 2)
# Measure the qubits
for i in range(0,3):
       circ.measure(i,i)
```

The circuits can be simulated locally, or using the IBMQ hardware. In the first case, we import the Aer library and execute the following code:

```
from qiskit import Aer, BasicAer, execute
backend = BasicAer.get_backend('qasm_simulator')
n_shots = 100
simulation = execute(circuit, backend=backend, shots=n_shots)
sim_result = simulation.result()
```

It is important to select a big enough number of shots to have a good view of the probability of each output. In case we want to simulate our circuit using the iqmb_qasm_simulator, we create an IBMQ account, and use the token they provide us to access their systems.

```
from qiskit import IBMQ, execute
IBMQ.load_account();
provider = IBMQ.get_provider('ibm-q')
backend = provider.get_backend('ibmq_qasm_simulator')
n_shots = 100
simulation = execute(circuit, backend=backend ,shots=n_shots)
sim_result = simulation.result()
```

The load_account function authenticates the user with the credentials stored in the environment.

The next step is to see the results of the circuit. The clearest way is to use the `plot_histogram` function, which lets the user visualize a histogram with all the different outputs and the number of times they happened.

```
from qiskit.visualization import plot_histogram
plot_histogram(sim_result.get_counts())
```

Using Qiskit we have also the possibility to execute our algorithms on real quantum devices of IBMQ. To do it, we select the wanted quantum device and pass it as the backend of the function `execute`.

```
from qiskit import IBMQ, execute
IBMQ.load_account();
provider = IBMQ.get_provider('ibm-q')
quantum_machine = 'ibmq_manila'
number_shots=1000
qcomp = provider.get_backend(quantum_machine)
job = execute(qc, backend=qcomp, shots=number_shots)
result = job.result()
```

IBM lets us have access to a restricted number of quantum devices with a free account, which have a maximum of 5 qubits at the moment. Often there are also long queues, so even the simplest experiment can be time-consuming. All the information, like errors and coupling maps, about each quantum device, can be accessed from the IBMQ web application.

This also gives us much information on each circuit that was executed in one of their systems, like its measurement outcomes, its graphical representation, and its status timeline (how much it stayed in the queue, how much it took for the computation, and so on): these are accessible from the jobs overview.

An important auxiliary instrument to create quantum circuits are the functions to create quantum and classical registers, that can be used in this way:

```
from qiskit import ClassicalRegister, QuantumCircuit, QuantumRegister
n=5
x = QuantumRegister(n)
y = QuantumRegister(n)
c = QuantumRegister(1)
clas = ClassicalRegister(n)
clas_c = ClassicalRegister(1)
circuit = QuantumCircuit(x,y,c,clas,clas_c)
[...]
for i in range(0, n):
       circuit.measure(y[i],clas[i])
circuit.measure(c,clas_c)
```

A.2 Library for factorization

The first library that we implemented contains all the needed functions to factorize a biprimal number. The interface function is called `Shor`, and is a parametrized function that, depending on the input, provides different ways and methods to execute the algorithm. The first thing that the function does is to check if the value N, that is the number to factorize, is even or of the form x^y, since in these two cases we can find easily the factors. It can not be checked the biprimal feature of it, so it must be ensured by the user (a public key of RSA is with certainty a biprimal number). Then it checks if the value a, passed as input or taken randomly, has a common factor with N, as in this case we would have been lucky and this factor can be retrieved classically with the extended Euclidean Algorithm.

Then the quantum circuit is executed and all the different values measured are given as input for the function that performs the continued fraction expansion method. At the end of it, if non-trivial factors are found from even one of all the different outputs, they are returned by the function, otherwise, the pair (0,0) is given to indicate that the algorithm failed.

The only value that must be given to the function is the number to factorize, while it is possible to use default values for the other inputs. These inputs are:

- N, the number to factorize.
- a, the random number needed by Shor's algorithm. If not present, the function uses a random integer between 2 and N-1.
- number_shots, the number of times the circuit is executed or simulated. The default value is `100`.
- sequential, a binary value that tells if the sequential QFT method is used or not. The default value is `True`.

- simulator, a binary value that tells if the circuit must be simulated or executed in a real quantum device. The default value is `True`.
- ibm, a binary value useful if `simulator='True'` that indicates if the IBM qasm simulator must be used (`True`) or the circuit must be executed locally (`False`). The default value is `True`.
- quantum_machine, the name of the quantum machine that must be used in case the circuit must be executed on a real quantum device. This value must be expressed if `simulator='False'`.
- method, the name of the method used to build the modular additions, which is `Basic` if the method that performs the additions in the Fourier space is used, `Inplace` if the method that uses Toffoli based additions is used. The default value is `Basic`.
- runtime, a binary value that is `True` if the runtime of the circuit is wanted in the outputs. The default value is `False`.
- prob, a binary value that is `True` if the probability of success is wanted in the outputs. The default value is `False`.

The outputs will be the two factors p and q, or two zeros if the algorithm failed. The user can perform a while instruction, as in the example below to be sure that the program finds a solution.

```
(p_extracted,q_extracted) = Shor_library.Shor(N)
while (p_extracted == 0):
       (p_extracted,q_extracted) = Shor_library.Shor(N)
```

A way to avoid reperforming the circuit with the same value for a can be taken into consideration.

```
used_a=[]
a = random.randint(2,N-1)
used_a.append(a)
(p_extracted,q_extracted) = Shor_library.Shor(N,a)
while (p_extracted == 0):
       a = random.randint(2,N-1)
       if a not in vec:
              (p_extracted,q_extracted) = Shor_library.Shor(N,a)
```

Optional outputs, that will be given with respect to the inputs are the needed time to execute the circuit and the probability of success of the circuit.

```
# Option with only the two factors as outputs
(p,q) = Shor_library.Shor(N)
# Option with the two factors and the time as outputs
(p,q,T) = Shor_library.Shor(N,runtime='True')
# Option with the two factors and the probability as outputs
(p,q,P) = Shor_library.Shor(N,prob='True')
# Option with the two factors, the time and the probability as outputs
(p,q,T,P) = Shor_library.Shor(N,runtime='True',prob='True')
```

We created a program to cipher using RSA and decipher with the values retrieved by the library a message. The first part of this program create the RSA key pair and encrypts a message, then the library is used to find the factors, and the last part recreates the secret key from the retrieved factors and decrypt the message.

```
N = p*q
phi = (p-1)*(q-1)
e = random.randint(3,phi-1)
while(gcd(e,phi)!=1):
                e = random.randint(3,phi-1)
d = modinv(e,phi)
ciphertext = []
for m in message:
                c = pow(m,e,N)
                ciphertext.append(c)

IBMQ.load_account();
a = random.randint(2,N-1)
while(gcd(a,N)!=1):
       a = random.randint(2,N-1)
(p_extracted,q_extracted,T,P) = Shor_library.Shor(N,a,100)
while (p_extracted == 0):
       a = random.randint(2,N-1)
       if(gcd(a,N)!=1):
                a = random.randint(2,N-1)
                (p_extracted,q_extracted,T,P) = Shor_library.Shor(N,a,100)

phi_extracted = (p_extracted-1)*(q_extracted-1)
d_extracted = modinv(e,phi_extracted)
message_extracted = []
n = len(ciphertext)
for i in range (0,n):
                m_extracted = pow(ciphertext[i],d_extracted,N)
                message_extracted.append(m_extracted)
```

In this code is not represented the needed division of every character of the message into two nibbles in the case $N < 2^8$.

A.3 Library for ECDLP

The second library provided is a starting point for the construction of a circuit to break the ECDLP. It includes all the needed gates to implement a reversible elliptic curve point addition algorithm, as described in Algorithm 1. Parts of this library can be used also for different purposes. Each one of these gates up to the Montgomery modular squaring has been functionally tested, and optimized controlled versions have been created for some of them.

Addition

This gate can be appended to the circuit using the function `addition`. It calculates the sum of two numbers encoded in two quantum registers, saving also the carry bit of the operation. The inverse of this gate performs the subtraction, and the function to use it is `subtraction`. The inputs of the two functions are:

1. the circuit to which the gate is appended;

2. a quantum register of n qubits that encodes one of the addends;
3. a quantum register of n qubits that encodes one of the addends and that will be used to store the output;
4. a clear qubit that is used to store the carry bit of the operation; and
5. n, the number of qubits of the two registers.

Constant addition

A function (`constant_addition`) to append a gate that performs a reversible implementation of the constant addition is provided, with its inverse implementation (`constant_subtraction`). The inputs of the function are:

1. the circuit to which the gate is appended;
2. the quantum register of n qubits that encodes one of the addends;
3. a dirty ancilla qubit;
4. the classical addend represented in binary form (of n bits); and
5. n, the number of qubits of the quantum register.

Optimized controlled versions have been created that can be called using the functions `controlled_constant_addition` and `controlled_constant_subtraction`, that require another input (the control qubit) after the circuit.

Comparator

This gate flips a qubit with respect to the value encoded in two different quantum registers. The inputs of the function are:

1. the circuit to which the gate is appended;
2. a quantum register of n qubits that encodes one of the two values to compare;
3. a quantum register of n qubits that encodes the other one of the two values to compare;
4. a qubit that is flipped is the value encoded in the second quantum register is $\leq$ than the value encoded in the first one; and
5. n, the number of qubits of the quantum registers.

Modular addition

The function to create the modular addition circuit is called `addition_modp`. The inputs of the function are:

1. the circuit to which the gate is appended;

2. a quantum register of n qubits that encodes one of the addends;
3. a quantum register of n qubits that encodes one of the addends and that will be used to store the output;
4. a clear ancilla qubit;
5. a dirty ancilla qubit;
6. the modulus of the operation represented in binary form (of n bits); and
7. n, the number of qubits of the quantum registers.

Optimized controlled versions have been created that can be called using the functions `controlled_addition_modp` and `doubly_controlled_addition_modp`, that require one or two additional inputs (the control qubits) after the circuit.

Modular doubling

The modular doubling gate can be appended to the circuit with the function `doubling_modp`. The inputs of the function are:

1. the circuit to which the gate is appended;
2. a quantum register of n qubits that encodes the number to double and that will be used to store the output;
3. a clear ancilla qubit;
4. a dirty ancilla qubit;
5. the modulus of the operation represented in binary form (of n bits); and
6. n, the number of qubits of the quantum register.

An optimized controlled version has been created that can be called using the functions `controlled_doubling_modp`, which requires one additional input (the control qubit) after the circuit.

Modular multiplication

A modular multiplication can be applied to the circuit using the function `multiplication_modp`. The inputs of the function are:

1. the circuit to which the gate is appended;
2. a quantum register of n qubits that encodes one of the factors;
3. a quantum register of n qubits that encodes the second factor;
4. a quantum register of n qubits that is used to store the output;
5. a clear ancilla qubit;
6. a dirty ancilla qubit;
7. the modulus of the operation represented in binary form (of n bits); and
8. n, the number of qubits of the quantum registers.

Modular squaring

A modular squaring can be applied to the circuit using the function `square_modp`. The inputs of the function are:

1. the circuit to which the gate is appended;
2. a clear ancilla qubit;
3. a quantum register of n qubits that encodes the number which is squared;
4. a quantum register of n qubits that is used to store the output;
5. a clear ancilla qubit;
6. a dirty ancilla qubit;
7. the modulus of the operation represented in binary form (of n bits); and
8. n, the number of qubits of the quantum registers.

Montgomery modular multiplication

A Montgomery modular multiplication, as described in section 10.2.2 can be applied to the circuit using the function `montgomery_multiplication_modp`. The inputs of the function are:

1. the circuit to which the gate is appended;
2. a quantum register of n qubits that encodes one of the factors;
3. a quantum register of n qubits that encodes the second factor;
4. a clear ancilla qubit;
5. a clear ancilla register of n qubits;
6. a dirty ancilla qubit;
7. a clear ancilla register of n qubits;
8. a clear ancilla register of n qubits that is used to store the output;
9. the modulus of the operation represented in binary form (of n bits); and
10. n, the number of qubits of the quantum registers.

Montgomery modular squaring

A Montgomery modular squaring can be applied to the circuit using the `montgomery_squaring_modp` function. The inputs of the function are:

1. the circuit to which the gate is appended;
2. a clear ancilla qubit;

3. a quantum register of n qubits that encodes the number which is squared;
4. a clear ancilla qubit;
5. a clear ancilla register of n qubits;
6. a dirty ancilla qubit;
7. a clear ancilla register of n qubits;
8. a clear ancilla register of n qubits that is used to store the output;
9. the modulus of the operation represented in binary form (of n bits); and
10. n, the number of qubits of the quantum registers.

Modular inversion

The last gate implemented applies a modular inversion to the value encoded in a register as described in section 10.2.3. This gate follows the construction of the controlled block described in algorithm 3. The inputs of the function `mod_inv` are:

1. the circuit to which the gate is appended;
2. a clear ancilla register of n qubits;
3. a quantum register of n qubits that contains the value x encoded;
4. a clear ancilla qubit;
5. a clear ancilla register of n+1 qubits;
6. a clear ancilla register of n+1 qubits;
7. a clear ancilla qubit;
8. a clear ancilla qubit;
9. a clear ancilla register of 2n qubits;
10. a clear ancilla qubit;
11. a clear ancilla register of $l = \lceil \log_2 n \rceil$ qubits;
12. a clear ancilla register of n qubits that will store the output of the function;
13. a clear ancilla register of l qubits;
14. the modulus of the operation represented in binary form (of n bits);
15. l, the number of qubit of two quantum registers; and
16. n, the number of qubits of most of the quantum registers.

We have not been able to test the whole gate due to resources limitations.

Appendix B

Developer's manual

This appendix describes the details of our Qiskit implementations contained in the two libraries and the code we used to make the tests of chapter 11.

B.1 Libraries

The classical part of the library to create the circuit for the factorization problem is intended to be used for small moduli, as certain instructions perform controls on the variables that must not be considered when dealing with real-life moduli. These are intended to avoid wasting time in unnecessary computations when the values are small. In future applications, the function *get_factors* must be modified, doing the following actions:

- improve the maximum number of iteration of the continued fraction expansion method, now set to 15;
- use an optimized algorithm for the modular exponentiation ($a^{\frac{r}{2}}$);
- erase the check on the value $a^{\frac{r}{2}}$.

The library containing the gates for the ECDLP is intended to be a starting point for the creation of the circuit. After having created the whole circuit just putting together the gates as described in chapter 10, the only classical part needed to find the secret key of the cryptosystem is to find the modular inverse of a number and a modular multiplication (fast python implementations of these functions are widely known). To complete the whole circuit, the first step to take is to initialize a qubit which will be the one where the sequential QFT is applied with the same method used for the factorization problem. This qubit will control the $2n$ different oracles, each one performing a reversible point addition as in algorithm 1. An important action to take is to initialize the registers of input of the first oracle to some coordinates (x, y) of the curve that are different from the ones of the point at infinity.

B.2 Code for the test

The tests conducted in chapter 11 have been conducted using the Jupyter notebooks to create the circuits and retrieve the results and using MATLAB to plot the figures shown. In this section, we

will explain the methods used to retrieve the depth of the circuits and the time spent in quantum working.

For what regards the depth, we used the Qiskit function `depth()` of the `QuantumCircuit` class, which permits to retrieve the length of the critical path of the circuit. However, this function does not resolve the compound gates that we created and used in our libraries, giving wrong results if used in the wrong way. In order to solve the problematic situations, we had to use the qiskit function `decompose`, which performs a 1-level decomposition of the circuit. An example of code is here listed.

```
from qiskit import QuantumCircuit
N=15
a=2
circuit = circuit_builder(N,a,'True','Basic')
print(circuit.depth())
dec1 = circuit.decompose()
print(dec1.depth())
```

The `circuit_builder` function creates our circuit for factorization based on the inputs so that we can consider its depth. Using this code, the first depth that we have retrieved is 98, while after the decomposition is 273. There are also cases where just one decomposition is not enough, so we have to use several times this function until the values of the depth of two consecutive applications are the same so that we can be sure to have obtained the depth of the maximally decomposed circuit. Using the following code, for example, we have as output the string "*dec0: 51 dec1: 387 dec2: 487 dec3: 487*", so we are sure that the real depth of the circuit is 487.

```
from qiskit import QuantumCircuit
N=15
a=2
circuit = circuit_builder(N,a,'True','Inplace')
dec1 = circuit.decompose()
dec2 = dec1.decompose()
dec3 = dec2.decompose()
print('dec0:',circuit.depth(),'dec1:',dec1.depth()
        ,'dec2:',dec2.depth(),'dec3:',dec3.depth())
```

The `decompose` functions actually divide every gate that can not be directly passed to the IBM simulator into a series of other gates. For example, a single Toffoli gate is decomposed into a series of CNOT and P gates, that can be passed to the simulator (or to the quantum machine). If we have created, using the qiskit function `to_gate()`, a compound gate, with the first iteration it will be decomposed in the circuit that it was prior to the call of this function, that can contain Toffoli gates for example, so we have to reuse the `decompose()` on it.

Regarding the time consumed in quantum working, that in reality is the time in the simulator system, it is a parameter of the job that is created with the function `execute`. For example, in our library, we retrieve it from the field `time_taken` of the job result, as described in this code.

```
job = execute(circuit)
result = job.result()
T = result.time_taken
```

The result retrieved by the job created by `execute` contains in fact, along the time taken for the execution or simulation, different interesting parameters, such as the number of qubits, the method of simulation, the noise model used, and the memory reserved.

B.3 Rigetti simulator

To use the simulator necessary to perform the tests in section 11.4 we follow the instruction present in https://github.com/quantastica/qps-api. The first thing to do is to install the necessary modules using anaconda, which can be done by running the following command and installing the Rigetti Forest SDK from https://qcs.rigetti.com/sdk-downloads.

```
$ pip install quantastica-qps-api
```

The next step is to login to Quantum Programming Studio (QPS) and copy the API token from the Profile section. This token is passed to the function `QPS.save_account`, which will permit us to use the QPS API.

To use the Rigetti quantum virtual machine, we have to open an anaconda prompt and run the following command.

```
qvm -S -c
```

This will create a virtual machine with a server running on it, to which we will perform HTTP requests with the program to run. To create and simulate the circuits in it, we can create a Jupyter notebook with the following code.

```
from pyquil import Program, get_qc
from pyquil.gates import X, CPHASE, CNOT, CSWAP, H, SWAP, MEASURE, PHASE
from functools import reduce
import numpy as np

from quantastica.qps_api import QPS

qc = circuit_builder(N,a,'True','Basic')
QASM_circ = qc.qasm()
output_program = QPS.converter.convert(QASM_circ, "qasm", "pyquil")
exec(output_program[0:len(output_program)-281])
shots = 100
p.wrap_in_numshots_loop(shots)
qc = get_qc('12q-qvm',execution_timeout=2000)
results_list = qc.run(p).readout_data.get("ro")
results = list(map(lambda arr: reduce(lambda x, y: str(x) + str(y),
    arr[::-1], ""), results_list))
counts = dict(zip(results,[results.count(i) for i in results]))
```

The output of the converter selects by default 1024 shots, so if we want to use a different number we must execute the output program with the exception of the last 281 characters so that we can select the number of shots wanted. Another important parameter is the `execution_timeout`, that is the seconds waited by the client for the server response: in case of long circuits, this must be changed to a correct value. The qvm selected (`'12q-qvm'` in the code) must be changed with regard of the number of qubits of the circuit.

From the `counts` dictionary now we can retrieve the probability of success of the circuit, while the time consumed is shown in the terminal where the server runs.