Classes of Quantum Complexity.

Clases de complejidad cuántica.

Bachelor's thesis in Computer Science
Double major in Computer Science and Mathematics
Facultad de Informatica
Universidad Complutense de Madrid

Year 2022-2023

Javier Lobillo Olmedo
Supervised by Alberto Antonio del Barrio García and Angelo Lucia

Contents

Acknowledgements

To my mother, and to those who are not here either, for taking care of me everyday. To my tutors, Alberto and Ángelo, for their patience and guidance. To Sergio, my friend and partner in many adventures throughout these years in university. To my family and friends, for supporting me and loving me.

Abstract

This bachelor's thesis is about quantum complexity classes, which are the quantum analogues of classical complexity classes. We start by introducing the basic mathematical concepts that support quantum computing, such as the bra-ket notation, the state space, and the quantum gates.

Then, in the second chapter, we study some classical complexity theory, such as classical complexity classes and the P vs NP problem. Afterwards, we introduce the quantum complexity classes, and we study the relationship between them and the classical complexity ones, specially the relationship between BPP, BQP and PSPACE. We see that one is contained in the next one.

In the third chapter we study some quantum algorithms, such as Deutsch's algorithm, Simon's algorithm, and Shor's algorithm. We show apparent differences between the classical and the quantum world, such as the fact that quantum computers can solve the factoring problem in polynomial time, while classical computers are not known to be able to do so.

Finally, in the fourth chapter, we implement Shor's algorithm in several quantum simulators, and we study how it works in practice, comparing the different backends. Additionally, we implement the quantum Fourier transform and run it in different real quantum backends.

Keywords: Quantum Computing, Quantum Algorithms, Complexity Theory, BPP, BQP, QFT, Shor's Algorithm.

Resumen

Este trabajo de fin de grado trata sobre las clases de complejidad cuántica, que son el análogo cuántico de las clases de complejidad clásica. Empezamos en el primer capítulo introduciendo los conceptos matemáticos básicos que sustentan la computación cuántica, como la notación bra-ket, el espacio de estados y las puertas cuánticas.

Después, en el segundo capítulo, estudiamos algo de teoría de complejidad clásica, como las clases de complejidad clásicas y el problema P vs NP. Después introducimos las clases de complejidad cuánticas, y estudiamos la relación entre ellas y las clases de complejidad clásicas, especialmente la relación entre BPP, BQP y PSPACE, viendo que una está contenida en la siguiente.

En el tercer capítulo estudiamos algunos algoritmos cuánticos, como el algoritmo de Deutsch, el algoritmo de Simon y el algoritmo de Shor. Mostramos diferencias aparentes entre el mundo clásico y el mundo cuántico, como el hecho de que los ordenadores cuánticos pueden resolver el problema de factorización de un numero natural en tiempo polinomial, mientras que los ordenadores clásicos no se sabe que puedan hacerlo.

Finalmente, en el cuarto capítulo, logramos implementar el algoritmo de Shor en varios simuladores cuánticos, y estudiamos cómo funciona en la práctica, comparando los diferentes backends. Además, implementamos la transformada cuántica de Fourier y la ejecutamos en varios ordenadores cuánticos reales.

Palabras clave: Computación Cuántica, Algoritmos Cuánticos, BPP, BQP, Teoría de la Complejidad, QFT, Algoritmo de Shor.

Introduction

Motivation

Classical computers have been one of the most efficient and useful tools that mankind has ever created. They have accelerated and eased tasks that were previously hard or inimaginable to do. Even so, some problems emerged which either are or appear to be too hard to solve efficiently with classical computers. In the study of complexity theory, this lead to the creation of many classifications of problems depending on their difficulty, and the creation of the P vs NP conjecture, which is maybe the most important problem in computer science. This lead scientists of different backgrounds to try to produce a new tool which could be even more potent than classical computers.

For most of their existance, the disciplines of quantum mechanics and computer science have been most unrelated. However, since the 1980s, the two fields have been increasingly intertwined as physicists became aware of the potential to store information in quantum systems and to manipulate that information with quantum operations. This led to the merge of the two fields, and the birth of quantum computing.

In 1980, Paul Benioff introduced the first model for a quantum Turing machine. Later, research led to the discovery of quantum algorithms which could handle information more efficiently than classical algorithms could. Some examples where Deutsch's algorithm, Deutsch-Jozsa algorithm and Simon's algorithm, which are formally described in this work. However, the most famous algorithm was Shor's algorithm, introduced by Peter SHor in 1994, which could factorize a number in polynomial time, theoretically solving the factoring problem, which is a problem of great interest in both complexity theory and cryptography.

Although the P vs NP problem is still open, and therefore we do not know if the separation we have found between classical and quantum complexity classes is as big as we think, quantum computing has proven to be a very useful tool so far, and it is expected to be even more useful in the future as we learn to produce more powerful and accurate quantum computers.

Objectives

Altjough the discipline of quantum computing is very wide and complex, in this work we will focus on the study of quantum complexity classes, which are the quantum analogues of classical complexity classes. We will study the relationship between them and the classical complexity ones.

In order to study quantum complexity classes, we need a model to be based on. This model will consist of circuits for both classical and quantum computers. We will study how they are constructed, and how they work. Once we have done so, we will construct some complexity classes upon them in a separate way for both classical and quantum circuits. Only after, we will merge the two models and study the relationship between the classes we have found in order to quantify the difference between classical and quantum computing. Although we will not be able to prove that the we can efficiently solve a set of problems with quantum computers that is strictly larger than the set of problems we can efficiently solve with classical computers, we will be able to prove that the first set appears to be larger than the second one. This separation, however, is not known to be actually true, and remains an open problem which is not expected to be solved in the near future.

Work plan

This work will be divided in three chapters.

- In the first chapter we will introduce the basic mathematical concepts that support quantum computing, such as the bra-ket notation, the state space, and the quantum gates. This will be done only to understand that there is a formal basis for all we will do in the next chapters.

- In the second chapter we will study some classical complexity theory, such as classical complexity classes and the P vs NP problem. Afterwards we will introduce the quantum complexity classes, and we will study the relationship between them and the classical complexity ones, specially the relationship between BPP, BQP and PSPACE. We will see that one is contained in the next one.

- In the third chapter we will study some quantum algorithms, such as Deutsch's algorithm, Simon's algorithm, and Shor's algorithm. We will show apparent differences between the classical and the quantum world, such as the fact that quantum computers can solve the factoring problem in polynomial time, while classical computers are not know to be able to do so.

- The fourth chapter will contain the implementation of Shor's algorithm in several quantum simulators, and we will study how it works in practice. We will compare the difference in performance between the different backends, measuring the time it takes to run the algorithm in each of them. Additionally, we will implement the quantum Fourier transform and we will run it in several quantum backends. We will take a statistical approach in order to study the differences between backends.

In order to be able to effectively carry out this work, we will first just study the subject following the bibliography, and only then we carry out the implementations and write the work. As this is a Bachelor's thesis, the focus will be on the ability to independently study and understand a new subject which has not been studied in the degree, and to be able to explain it in a clear and concise way. Of course, the work supervisors will serve as guides and will help to orient the flow of the work.

Chapter I

A model for Quantum Mechanics

I.1 Bra-Ket and state space

This mathematically introductory chapter is based on the first two chapters of [5].

Definition I.1.1 (Bra-Ket). Given a vector space V, we say a *ket* is of the form $|v\rangle$, which denotes a vector $v \in V$. Also, we say a *bra* is of the form $\langle f|$, which denotes a linear map $f : V \to \mathbb{C}$. Therefore, letting a linear map $\langle f|$ act on a vector $|v\rangle$ is written as $\langle f|v\rangle \in \mathbb{C}$.

It is important to note, although we will not be very formal about it, that we are working on a complex Hilbert space.

Definition I.1.2. A *(complex) Hilbert space* is a pair $(H, \langle\cdot|\cdot\rangle : H \times H \to \mathbb{C})$ such that:

1. H is a complex vector space.
2. The product $\langle\cdot|\cdot\rangle$ is:
    - *Conjugate symmetric*: $\langle x|y\rangle = \overline{\langle y|x\rangle}$ for all $|x\rangle, |y\rangle \in H$.
    - *Linear*: if $|x\rangle, |y_1\rangle, |y_2\rangle \in H$, $a, b \in \mathbb{C}$ then $\langle x| (a|y_1\rangle + b|y_2\rangle) = a\langle x|y_1\rangle + b\langle x|y_2\rangle$.
    - *Positive definite*: if $|x\rangle \in H$, $\langle x|x\rangle \geq 0$ and $\langle x|x\rangle = 0 \Leftrightarrow |x\rangle = |0\rangle$.
3. H is complete with the metric induced by $\|x\| = \sqrt{\langle x|x\rangle}$.

Also, we need to talk about basis and orthonormal basis in a Hilbert space.

Definition I.1.3. An *orthonormal basis* for a Hilbert space H is a basis $\{u_i\}_{i\in I}$ of H as a vector space such that $\langle u_i|u_j\rangle = \delta_{ij}$, for all $i, j \in I$.

When studying models for quantum computing, all the considered Hilbert spaces will be finite dimensional. We will consider the canonical finite Hilbert space as $\mathbb{C}^n$, where $n \in \mathbb{N}$. Every finite dimensional Hilbert space is isomorphic to $\mathbb{C}^n$ for some $n \in \mathbb{N}$, therefore we will focus on this space. From now on, all the Hilbert spaces that we will consider will be finite dimensional.

Definition I.1.4. A *ray* is an equivalence class of vectors defined by the next equivalence relation:

$$|x\rangle \sim |y\rangle \text{ if and only if } |x\rangle = \alpha\,|y\rangle \text{ for some } \alpha \in \mathbb{C} \text{ with } \alpha \neq 0.$$

We will choose a representative of the equivalence class, x, such that $\langle x|x\rangle = 1$.

Now, the state space that we will consider as a model for quantum computation will be the set of all rays in $\mathbb{C}^n$, or projective Hilbert space.

Definition I.1.5. A *state* is a normalized ray in a Hilbert space.

It is important to note that the overall phase of a state is irrelevant: $|x\rangle$ and $e^{i\theta}\,|x\rangle$ describe the same state, where $\theta \in \mathbb{R}$ and $|e^{i\theta}| = 1$.

Definition I.1.6. A *qubit* is an isolated quantum system with a two-dimensional state space.

We wil usually fix an orthonormal basis $\{|0\rangle, |1\rangle\}$ for the state space of a qubit. Therefore, a state of a qubit is of the form

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \text{ with } \alpha, \beta \in \mathbb{C} \text{ and } |\alpha|^2 + |\beta|^2 = 1.$$

Now that we have defined an n-dimensional Hilbert space, and we have our small elements, called qubits, from which we will build our larger Hilbert spaces, we need to provide a way of constructing this larger systems from our smaller bricks.

Proposición I.1.7. *Let H_1, H_2 be two Hilbert spaces with orthonormal bases $\{|u_i\rangle\}_{i\in I}$ and $\{|v_j\rangle\}_{j\in J}$ respectively. Then, $H_1 \otimes H_2$ is a Hilbert space with orthonormal basis $\{|u_i\rangle \otimes |v_j\rangle\}_{(i,j)\in I\times J}$.*

Once we have found ourselves in a projective Hilbert space, we will need ways to operate inside it. This is done through unitary operators, which will act as the gates of our quantum circuits.

Definition I.1.8. A *unitary operator U* on a Hilbert space H is a linear operator or matrix $U : H \to H$ such that $U^\dagger U = UU^\dagger I$, where $U^\dagger$ is the adjoint of U, and I is the identity operator.

The reason we use unitary operators is that they preserve the norm of the vectors, and therefore they take representatives of the rays (of norm 1) to representatives of rays.

Proposición I.1.9. *Let H_1, H_2 be two Hilbert spaces with orthonormal bases $\{|u_i\rangle\}_{i\in I}$ and $\{|v_j\rangle\}_{j\in J}$ respectively, and let U_1 and U_2 be unitary operators on H_1 and H_2 respectively. Then, $U_1 \otimes U_2$ is a unitary operator on $H_1 \otimes H_2$ given by*

$$U_1 \otimes U_2(|u_i\rangle \otimes |v_j\rangle) = U_1\,|u_i\rangle \otimes U_2\,|v_j\rangle\,.$$

In addition, if we write the matrices like $U_1 = (u_{ij})$, then the matrix representation of $U_1 \otimes U_2$ is given by

$$U_1 \otimes U_2 = (u_{ij}U_2).$$

Lastly, we need to define what a measurement is. This is the way we will extract information from our quantum system.

Definition I.1.10. We say that the computational measurement of a quantum state system of dimension $N = 2^n$ with state

$$|\psi\rangle = \sum_{i=0}^{N-1} \alpha_i |i\rangle ,$$

is a discrete random variable such that

$$P(X = |i\rangle) = |\alpha_i|^2.$$

Now we should mention some unitary operators that will be useful for us in the future.

Definition I.1.11. The *Hadamard gate* is the unitary operator on a qubit given by

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Definition I.1.12. The *controlled-not gate* is the unitary operator on two qubits given by

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Definition I.1.13. The *SWAP gate* is the unitary operator on two qubits given by

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Definition I.1.14. The *Pauli-X gate* is the unitary operator on a qubit given by

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The *Pauli-Y gate* is the unitary operator on a qubit given by

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}.$$

The *Pauli-Z gate* is the unitary operator on a qubit given by

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Chapter II

Classical and quantum circuits

In this chapter we will precise our model of quantum computing, we will point out some basic properties for the model, and afterwards we will investigate the power of the model. In order to do so in the second part of the chapter, we have to begin, in the first chapter, with the study of what a classical computer does.

This chapter is based on the fifth chapter of [5].

II.1 Classical Circuits

We start with the definition of a classical computer focusing on what *it does*. Later in the subsection this will translate this to what *it is* in terms of circuits.

Universal gates

Definition II.1.1. A (deterministic) classical computer evaluates a function

$$f : \{0,1\}^n \to \{0,1\}^m,$$

which takes n-bits of input and produces m-bits of output.

A function with an m-bit output can be seen as a function with m functions with a 1-bit output. Therefore, we can assume that $m = 1$. This produces the following definition:

Definition II.1.2. A Boolean function is a function

$$f : \{0,1\}^n \to \{0,1\},$$

which takes n-bits of input and produces 1-bit of output.

We may think of one Boolean functions as a binary string of length 2^n, where each bit is the output $f(x)$ of the function for one of the 2^n possible inputs x. It is clear that there are 2^{2^n} different Boolean functions of n variables, which is a very large number. For example, for $n = 5$ there are $2^{2^5} = 2^{32} = 4,294,967,296$ different Boolean functions.

Definition II.1.3. We say that a Boolean function f *accepts* an n-bit string x if $f(x) = 1$, and *rejects* x if $f(x) = 0$.

Therefore we may regard a Boolean function as the subset Σ containing all the accepted strings.

The evaluation of a Boolean function f can be reduced to a sequence of simple logical operations. To see how, we denote the n-bit strings accepted by f as $\Sigma = x^{(1)}, x^{(2)}, x^{(3)}, \ldots$, and note that for a given $x^{(a)}$ we can define a function

$$f^{(a)} : \{0,1\}^n \to \{0,1\}, \quad x \mapsto \begin{cases} 1 & \text{if } x = x^{(a)} \\ 0 & \text{otherwise,} \end{cases}$$

which accepts only $x^{(a)}$. Then we can write f as

$$f(x) = f^{(1)}(x) \vee f^{(2)}(x) \vee f^{(3)}(x) \vee \ldots,$$

where $\vee$ denotes the logical OR operation. In binary arithmetic, the OR operation may be represented

$$x \vee y = x + y - x \cdot y,$$

which has the value 0 if x and y are both 0, and 1 otherwise.

Now, we consider the evaluation of $f^{(a)}$. We may express the n-bit string $x^{(a)}$ as

$$x^{(a)} = x^{(a)}_{n-1} x^{(a)}_{n-2} \ldots x^{(a)}_1 x^{(a)}_0.$$

If $x^{(a)} = 11\ldots1$, we may simply write

$$f^{(a)}(x) = x_{n-1} \wedge x_{n-2} \wedge \cdots \wedge x_1 \wedge x_0,$$

where $\wedge$ denotes the logical AND operation. In binary arithmetic, the AND operation is the product

$$x \wedge y = x \cdot y.$$

If $x^{(a)}$ has a 0 in the i-th position, we may apply the NOT ($\neg$) operation to the i-th bit of x, and then write $f^{(a)}$ as the same expression as above. In binary arithmetic, the NOT operation is the subtraction

$$\neg x = 1 - x.$$

For example

$$f^{(a)}(x) = \ldots (\neg x_3) \wedge x_2 \wedge (\neg x_1) \wedge x_0,$$

if $x^{(a)} = \ldots 0101$.

We have now constructed the function f form three elementary logical operations: NOT, AND and OR. The expression we obtained is denoted the *disjunctive normal form* of f. Of course, we have implicitly used another operation INPUT(x_i) which inputs the i-th bit of x.

Now we are ready to transform the definition of a function to that of a circuit model for classical computation. A computation is a finite sequence of elementary operations, a *circuit*, applied to a specify input. Each operation is called a *gate*. The result of the computation is the final value of the remaining bits after the last gate has been applied. For a Boolean function, there is only one output bit. A circuit can be seen as a directed acyclic graph, where each node is a gate and the directed edges represent the flow of bits throughout the circuit, with the direction specifying the order in which the gates are applied.

Finally, we say that the gate set {NOT, AND, OR, INPUT} is *universal* for classical computation, which means that any function may be computed by building a circuit from only these gates. It is a remarkable fact that any arbitrary computation can be performed using such simple building blocks.

Most functions require large circuits

Our previous DNF construction shows that any Boolean function can be evaluated with no more that 2^n OR gates, $n2^n$ AND gates, $n2^n$ NOT gates and $n2^n$ INPUT gates, with a total of $(3n+1)2^n$ gates. Although some Boolean functions can be computed with a smaller number of gates, it is clear that *most* functions require an exponentially large (in n) number of gates.

How many circuits are there with G gates acting on a n-bit input? Considering the gate set from which we constructed the DNF, we will also allow imputing of a constant bit (either 0 or 1) as an input. Therefore there are $n+5$ different gates:

AND, OR, NOT, INPUT(0), INPUT(1) and INPUT(x_i) for $i = 0, 1, \ldots, n-1$.

Each two-bit gate acts on a pair of bits which are outputs of previous gates, and there are G such gates, so this pair can be chosen in fewer that G^2 ways. Therefore, the total number of size-G circuits is at most

$$N_{circuit}(G) \leq ((n+5)G^2)^G.$$

If we take $G = c\frac{2^n}{2n}$ where c is a constant independent of n, then

$$\log_2 N_{circuit}(G) \leq G(\log_2(n+5) + 2\log_2 G) = c2^n\left(1 + \frac{1}{2n}\log_2\left(\frac{c^2(n+5)}{4n^2}\right)\right) \leq c2^n,$$

where the second inequality holds for n sufficiently large. Comparing with the number of Boolean functions, $N_{function}(n) = 2^{2^n}$, we see that

$$\log_2\left(\frac{N_{circuit}(G)}{N_{function}(n)}\right) \leq c2^n - 2^n = (c-1)2^n,$$

for n sufficiently large. Therefore, for any $c < 1$, the number of circuits is smaller than the number of functions by a huge factor. Although we did this analysis for one particular universal gate set, the result would have not been very different for any other universal gate set.

We conclude that for any $\varepsilon > 0$ then most Boolean functions require circuits of size at least $(1-\varepsilon)\frac{2^n}{2n}$ gates. The circuit size is so large because most functions have no structure that can be exploited to construct a more compact circuit. We basically cannot do much better than consulting a "look-up table" of the function values for all possible inputs, which is essentially what the DNF construction does.

Circuit complexity

So far we have only considered a computation that acts on a fixed input (consisting of n bits), but we may also consider *families* of circuits that act on inputs of different lengths. Circuit families provide a useful scheme for analyzing and classifying the *complexity* of computations, a scheme that will have a natural generalization to quantum computation.

Boolean functions arise naturally in the study of computational complexity. A Boolean function f is said to encode the solution to a "decision problem" - the function examines the input and returns 1 if the answer is "yes" and 0 if the answer is "no". Often what might not be stated colloquially as a decision problem can be reformulated as one. For example, the function that defines the FACTORING problem is

$$f(x,y) = \begin{cases} 1 & \text{if integer } x \text{ has a factor } z \text{ such that } 1 < z < y, \\ 0 & \text{otherwise.} \end{cases}$$

Knowing $f(x,y)$ for all x and y is equivalent to knowing the complete factorization of x.

Another example of a decision problem is the HAMILTONIAN PATH problem: let the input be an l-vertex graph, represented by its $l \times l$ adjacency matrix (a 1 in the i,j entry means that there is an edge between vertices i and j). The function is

$$f(x) = \begin{cases} 1 & \text{if the graph has a Hamiltonian path,} \\ 0 & \text{otherwise.} \end{cases}$$

A Hamiltonian path is a path that visits each vertex exactly once.

For the FACTORING problem the size of the input is the number of bits needed to represent the integers x and y, while for the HAMILTONIAN PATH problem the size of the input is the number of bits needed to represent the adjacency matrix. Therefore each problem really defines a family of Boolean functions with variable input size. We denote such a function family as

$$f : \{0,1\}^* \to \{0,1\},$$

where $\{0,1\}^*$ is the set of all finite bit strings. When x is an n-bit string, by writing $f(x)$ we mean the Boolean function in the family that corresponds to the input size n. The set L of strings accepted by a function family f

$$L = \{x \in \{0,1\}^* : f(x) = 1\},$$

is called a *language*.

We can quantify the hardness of a problem by stating how the computational resources required to solve it scale with the size of the input n. In the circuit model of computation, it is natural to use the number of gates as a measure of the computational resources required. Alternatively we could be interested in how much *time* it takes to do the computation if many gates can be executed in parallel; the *depth* of the circuit is the number of time steps required, assuming that gates acting on distinct bits can be executed in parallel (that is, the depth is the maximum length of a directed path from the input to the output of the circuit). The *width* of the circuit is the maximum number of gates that in any one time step, and quantifies the storage space required to do the computation.

We want to divide the decision problems in two classes: easy and hard. However it is not possible to give a precise definition of what easy and hard should be. We will consider decision problems with variable input size and we will examine how the size of the circuit that solves the problem scales with the input size.

If we use the scaling behavior of a circuit family to characterize the complexity of a problem, there is a subtlety that we must address. It would be cheating to hide the difficulty of the problem in the *design* of the circuit. Therefore we should restrict attention to circuit families that have acceptable "uniformity" properties - it must be "easy" to construct the circuit with $n+1$ bits of input once we have constructed the circuit with an n-bit input.

Associated with a family of functions $\{f_n\}$ (where f_n has n-bit input) is a family of circuits $\{C_n\}$ where C_n is a circuit that computes f_n.

Definition II.1.4. We say that a circuit family $\{C_n\}$ is *polynomial-size uniform* if the size $|C_n|$ of the circuit C_n grows with n no faster than a power of n,

$$|C_n| \leq \mathrm{poly}(n),$$

where poly denotes a polynomial.

Then we naturally define:

Definition II.1.5. P is the set of decision problems that can be solved by a polynomial-size uniform circuit family.

Now, the natural choice is to define problems in P as "easy" and problems not in P as "hard". Notice that C_n computes $f_n(x)$ for every possible n-bit input x, and therefore, if a decision problem is in P we can find the answer to the problem even for the "worst-case" input using a circuit of size no greater than $\mathrm{poly}(n)$. As noted before, we implicitly assume that the circuit family is "uniform" so that the *design* of the circuit can itself be solved by a polynomial-time algorithm. Under this assumption, solvability in polynomial time by a circuit family is equivalent to solvability in polynomial time by a universal Turing machine.

Of course, to determine the size of a circuit that computes f_n, we must know what the elementary components of the circuit are. Fortunately, though, it turns out that the precise choice of elementary components does not matter. We can use any universal set of gates to efficiently simulate any other universal set of gates, as long as the gate set is finite and each gate act on a constant number of bits.

The way of distinguishing between easy and hard problems might seem rather arbitrary. If $|C_n| \sim n^{1000}$ we might consider the problem to be intractable in practice, even though it is in P. On the other hand, if $|C_n| \sim n^{\log\log\log n}$ we might consider the problem to be easy in practice, although the scaling is "superpolynomial". Furthermore, even if $|C_n|$ scales like a modest power of n, the constant factor in front of the power might be so large that the circuit is too big to be practical. Nevertheless, such pathological cases are rare, and the distinction between easy and hard problems is useful in practice.

Of particular interest are decision problems that can be answered by providing an example of a solution that is easy to check. For example, given x and $y < x$, it is hard (in the worst case) to determine if x has a factor less than y, but if we are given a factor $z < y$ of x, it is easy to check that z is indeed a factor of x. Similarly it is hard to find a Hamiltonian path in a graph, but if we are given a Hamiltonian path, it is easy to check that it is indeed a Hamiltonian path.

The concept of a problem that is hard to solve but a given solution can be easily checked is formalized as follows:

Definition II.1.6. A language L is in the class NP if and only if there is a polynomial-size verifier $V(x, y)$ such that:

(1) If $x \in L$, then there exists y such that $V(x, y) = 1$ (*completeness*).

(2) If $x \notin L$, then for all y, $V(x, y) = 0$ (*soundness*).

The verifier V is the uniform circuit family that decides L.

Completeness means that if x is in the language, then there is a "witness" y such that the verifier accepts the input if the witness is provided. Soundness means that if x is not in the language, then no matter what witness is provided, the verifier rejects the input. It is immplicit that the witness is of polynomial length since the verifier has a polynomial number of gates, including input gates. NP stands for "nondeterministic polynomial time", name which is used for historical reasons. The name comes from the fact that the verifier V can be thought of as a nondeterministic Turing machine that guesses the witness y and then checks that it is correct in polynomial time.

It is easy to see that $P \subseteq NP$. If L is in P, then there is a polynomial-size circuit C that decides L without any witness whatsoever. Nevertheless, there are problems in NP that are not known to be in P. Naturally arises the conjecture upon which the much of the field of complexity theory is based:

$$\text{Conjecture: } P \neq NP.$$

If this conjecture was false, that is if $P = NP$, that would mean that we could easily find a solution for any problem whose solutions are easy to verify. For example, we could discover all mathematical theorems which have short proofs. If $P \neq NP$, then our computers would never achieve such power and the mere existence of a succint proof for a statement would not ensure that we could find it in any reasonable amount of time.

An important example of a Boolean problem that is in NP but not known to be in P is CIRCUIT-SAT. In this problem, the input is a Boolean circuit C and the question is whether there is an input x such that $C(x) = 1$. The problem is in NP because if the answer is yes, then we can provide the input x as a witness which can be easily checked as the circuit C is of polynomial size. The problem is not known to be in P because we do not know how to find the input x in polynomial time. In this case, the function to be evaluated is

$$f(C) = \begin{cases} 1 & \text{if there exists } x \text{ such that } C(x) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

The problem CIRCUIT-SAT is particularly important because if we have a machine that solves CIRCUIT-SAT in polynomial time, then we can use it to solve any problem in NP in polynomial time. We say that every problem in NP us (efficiently) *reducible* to CIRCUIT-SAT. More generally, we say that problem B reduces to problem A if a machine that solves A can be used to solve B as well.

Definition II.1.7. If A and B are Boolean function families, then B *reduces* to A if there is a function family R, computed by polynomial-size circuits, such that $B(x) = A(R(x))$ for all x.

What this definition says, in other words, is that B accepts x if and only if A accepts $R(x)$. In particular, then, if we have a polynomial-size circuit family that solves A, we can hook A up with R to obtain a polynomial-size circuit family that solves B.

Thus, any problem B can be effectively reduced to CIRCUIT-SAT. This reduction is possible because problem B possesses a polynomial-size verifier $V(x, y)$ which accepts x if and only if a valid witness y exists, satisfying $V(x, y) = 1$. To clarify, for each fixed x, the question of whether such a witness y exists becomes an instance of CIRCUIT-SAT.

Consequently, the existence of a poly-size circuit family capable of solving CIRCUIT-SAT would also imply the ability to solve problem B by utilizing the same circuit family.

Definition II.1.8. We say a problem $A \in NP$ is *NP-complete* if every problem in NP reduces to A.

With the above definition, we can say that CIRCUIT-SAT is NP-complete. The NP-complete problems are the "hardest" problems in NP in the sense that if we can solve any one of them in polynomial time, then we can solve every NP problem in polynomial time. Furthermore, to show that any problem is NP-complete, it suffices to show that it is in NP and that it reduces to any other NP-complete problem. In fact, if C is any problem in NP and B is NP-complete, then there is a polynomial-size reduction R such that $C(x) = B(R(x))$, and if B is reductible to A then there is another polynomial-size reduction R' such that $B(y) = A(R'(y))$. Hence $C(x) = A(R'(R(x)))$ and, since the composition of two polynomial-size circuits is also polynomial-size, we see that an arbitrary problem C in NP reduces to A, and therefore A is NP-complete. NP-completeness is a useful concept because hundreds of "natural" computational problems turn out to be NP-complete. For example, there is a polynomial reduction of CIRCUIT-SAT to the HAMILTONIAN PATH problem, and it follows that HAMILTONIAN PATH is also NP-complete.

Another noteworthy complexity class is called co-NP. While NP is the class of problems whose solutions are easy to verify by giving an *example* of a solution, co-NP is the class of problems whose solutions are easy to verify by giving a *counterexample* if the answer is NO. More formally:

Definition II.1.9. A language L is in co-NP if and only if there is a polynomial-size verifier $\bar{V}(x, y)$ such that

(1) If $x \notin L$ then there exists a witness y such that $\bar{V}(x, y) = 1$,

(2) if $x \in L$ then for all y, $\bar{V}(x, y) = 0$.

For NP the witness y testifies that x is in the language while for co-NP the witness y testifies that x is *not* in the language. Therefore if a language L is in NP, then its complement $\bar{L}$ is in co-NP and vice-versa. We see that whether we consider a problem to be in NP or co-NP depends on how we define the problem. For example, the problem "Is there a Hamiltonian path in the graph G?" is in NP while the question "Is there no Hamiltonian path in the graph G?" is in co-NP.

While the distinction between NP and co-NP may appear arbitrary, it raises intriguing questions about problems that belong to both classes. When a problem is in both NP and co-NP, we can efficiently verify the answer, given a suitable witness, whether it is a "YES" or "NO" instance.

An open and unsolved conjecture in complexity theory is whether NP is distinct from co-NP. For instance, demonstrating the existence of a Hamiltonian path in a graph can be achieved by providing an example, but proving the non-existence of a Hamiltonian path in a graph is considerably more challenging.

If we assume that $P \neq NP$, it is known that there exist problems in NP of intermediate difficulty, the class NPI, which are neither in P nor NP-complete. Also, assuming that $NP \neq co-NP$, it is known that no $co-NP$ problems are NP-complete. Thus, problems in the intersection of NP and co-NP, if not in P, are good candidates for problems in NPI.

In fact, a problem in $NP \cap co-NP$ believed not to be in P is the FACTORING problem. As we have seen, FACTORING is in NP because if we provide a factor of x as a witness, we can easily verify that it is indeed a factor. But it is also in co-NP because if we are given a prime number, we can efficiently verify its primality by using the AKS primality test, for example. Thus, if someone gives us the prime factors of x, we can efficiently check that the prime factorization is right, and can exclude that any integer less than y is a divisor of x. Therefore it seems that FACTORING is a problem in NPI.

We are led to a conjectured and crude picture of the structure of $NP \cup co - NP$. NP and co-NP are not equal, but they are not disjoint either. P lies in $NP \cap co - NP$, and $NP \cap co - NP$ also contains problems that are not in P (like FACTORING). No NP-complete or co-NP-complete problems lie in $NP \cap co - NP$.

Randomized Computation

It is sometimes useful to consider *probabilistic* circuits that have access to a random number generator. For example, a gate in a probabilistic circuit might act in either one of two ways, depending on the outcome of a coin toss. Such a circuit, for a single input, can sample many possible computational paths. An algorithm performed by a probabilistic circuit is said to be "randomized".

When running a randomized computation multiple times on the same input, we obtain a probability distribution over the set of outputs. The usefulness of this computation lies in the high probability of obtaining the correct answer. For a decision problem, we desire a randomized computation to accept an input x in the language L with a probability of at least $1/2+\delta$, and to accept an input x not in the language L with a probability of at most $1/2-\delta$, where δ is a constant greater than zero and independent of the input size.

In such cases, we can *amplify* the probability of success by repeating the computation numerous times and taking a majority vote on the outcomes. For $x \in L$, when the computation is repeated N times, the probability of rejecting in more than half the iterations is no more than $e^{-\delta^2 N/2}$ (known as the the *Chernoff bound*), which exponentially diminishes as N increases. Similarly, for $x \notin L$, the probability of accepting in more than half the iterations is no more than $e^{-\delta^2 N/2}$.

To prove this, we have to see that there are 2^N possible sequences of outcomes in the N trials, and the probability of any particular sequence with N_w wrong answers is

$$\left(\frac{1}{2}-\delta\right)^{N_w}\left(\frac{1}{2}+\delta\right)^{N-N_w}.$$

The majority is wrong only if $N_w > N/2$, so the probability of any sequence with an incorrect majority is at most

$$\left(\frac{1}{2}-\delta\right)^{N/2}\left(\frac{1}{2}+\delta\right)^{N/2}=\frac{1}{2^N}(1-4\delta^2)^{N/2}.$$

Using $1-x \le e^{-x}$ and multiplying by 2^N, the total number of sequences, we get the Chernoff bound:

$$\text{Probability of incorrect majority} \le e^{-\delta^2 N/2}.$$

If we are willing to accept a probability of error no larger than a given ε, then, it suffices to run the computation a number of times N that satisfies

$$N \ge \frac{1}{2\delta^2}\ln\left(\frac{1}{\varepsilon}\right).$$

Because we make the probability of error exponentially small in N, the value of the constant δ is not important as long as it is greater than zero. The standard convention is to take $\delta = 1/6$, so that $x \in L$ is accepted with probability at least $2/3$ and $x \notin L$ is accepted with probability at most $1/3$. This criterion defines the class BPP (bounded-error probabilistic polynomial time).

Definition II.1.10. The class BPP is the set containing decision problems solved by polynomial-size randomized uniform circuit families.

It is clear that $P \subset BPP$, since a deterministic algorithm is a special case of a randomized algorithm. It is widely believed that $BPP = P$, that randomness does not enhance our computational power, but this is not known. It is not even known whether BPP is contained in NP.

We can define a randomized class analogous to NP called MA (Merlin-Arthur), which consists of languages that can be efficiently checked when a randomized verifier is provided with a suitable witness:

Definition II.1.11. A language L is in MA if and only if there is a polynomial-size randomized verifier $V(x, y)$ such that

1. if $x \in L$, then there is a witness y such that $Pr[V(x, y) = 1] \geq 2/3$, and
2. if $x \notin L$, then for all y, $Pr[V(x, y) = 1] \leq 1/3$.

The colorful name evokes a scenario in which the all-powerful Merlin, who knows the answer to the decision problem, uses his magic to conjure the witness y and gives it to Arthur, who then uses his probabilistic algorithm to check that y is a valid witness. Obviously $BPP \subset MA$, but we expect $BPP \neq MA$, just as we expect $P \neq NP$.

II.2 Quantum Circuits

Now we are ready to formulate a mathematical model of a quantum computer. We will generalize the circuit model of classical computation to the quantum circuit model of quantum computation.

A classical computer processes bits and it is equipped with a finite set of gates that act on sets of bits. A quantum computer processes qubits. We will assume that it too is equipped with a discrete ser of fundamental components, called *quantum gates*. Each quantum gate is a unitary transformation that acts on a fixed number of qubits. In a quantum computation, a finite number n of qubits are initialized to the state $|0\ldots0\rangle$. A circuit constructed from a finite number of quantum gates is then applied to these qubits. Finally, a computational measurement of a subset of the qubits (proper or not) is performed, projecting each measured qubit onto the basis $\{|0\rangle, |1\rangle\}$. The outcome of this measurement is the result of the computation.

Several features of this model are worth noting:

(1) **Preferred decomposition into subsystems.** It is implicit but important that the Hilbert space of the device has a preferred decomposition into a tensor product of subsystems of lower dimension, in this case, the qubits. We assume that there is a natural decomposition into subsystems that is respected by the quantum gates, i.e. the gates act only on a limited number of qubits. Mathematically, this feature of the gates is crucial for

establishing a clearly defined notion of quantum complexity. Physically it is also important, but that exceeds the content of this work.

(2) **Finite instruction set**. Since unitary transformations form a continuum, it may seem unnecessarily restrictive to postulate that the machine can execute only those quantum gates chosen from a finite set. We nevertheless make this assumption, because we do not want to invent a new physical implementation each time we are faced with a new computation to perform. We will see later that only a discrete set of quantum gates can be well protected from error and we will be gald to only have assumed a finite gate set in our formulation of the quantum circuit model.

(3) **Unitary gates and orthogonal measurements**. This can be proven to be a sufficient model although this demonstration is beyond the scope of this work.

(4) **Simple preparations**. Choosing the initial state of the n input qubits to be $|0\dots0\rangle$ is merely a convention. We might want the input to be a nontrvial classical bit string instead, and in that case we would just include the NOT gates in the first computational step of the circuit to flip the input bits from 1 to 0. What is important is that the initial state be simple, i.e. easy to prepare. If we allowed the input state to be a complicated entangled state of the n qubits, then we might be hiding the difficulty of the computation in the preparation of the input state.

(5) **Simple measurements**. We might allow the final measurement to be a collective measurement, or a projection onto a different basis. But any such measurement can be implemented by performing a suitable unitary transformation followed by a projection onto the standard basis $\{|0\rangle, |1\rangle\}^n$. We have that complicated collective measurements can be transformed into measurements in the standard basis only with some difficulty, and it is appropiate to take into account this difficulty while considering the complexity of the computation.

(6) **Measurements delayed until the end.** While we might allow intermediate measurements, we must consider that subsequent choices of quantum gates depend on the outcomes of these measurements. However, it is important to note that we can always achieve the same result using a quantum circuit with all measurements delayed until the end.

A quantum gate, being a unitary transformation, is reversible. In fact a classical reversible computer is a special case of a quantum computer. A classical reversible gate,

$$x \mapsto y = f(x),$$

implementing a permutation of k-bit strings can be regarded as a unitary transformation U acting on k qubits, which maps the computational basis of product state $\{|x_i\rangle, i = 0, 1, \dots, 2^k - 1\}$ to another basis of product states $\{|y_i\rangle, i = 0, 1, \dots, 2^k - 1\}$ according to

$$U|x_i\rangle = |y_i\rangle.$$

Since U maps one orthonormal basis to another it is manifestly unitary. A quantum computation constructed from such gates takes $|0\dots0\rangle$ to one of the computational basis states, so that the outcome of the final measurement in the $|0\rangle, |1\rangle$ basis is deterministic.

There are four main issues concerning our model that we would like to address in this Chapter. The first issue is *universality*. The most general unitary transformation that can be performed on n qubits is a $U(2^n)$ transformation. Our model would seem incomplete if there were transformations in $U(2^n)$ that we were not able to reach. In fact, we will show that there are many ways to choose a discrete set of universal quantum gates. Using a universal gate set we can construct circuits that compute a unitary transformation coming as close as we like to any desired unitary transformation. This will be used explicitly at the end of this work when we implement Shor's algorithm.

Due to universality, there is also a machine-independent notion of *quantum complexity*. We can define a new complexity class BQP (bounded error quantum polynomial time) as the class of languages that can be decided by polynomial-size uniform quantum circuit families with bounded error probability. As one universal quantum computer can efficiently simulate another, this class does not depend on the details of our hardware implementation, i.e., on the choice of the universal gate set.

We should note that a quantum computer can simulate a probabilistic classical computer efficiently. This is because it can prepare a superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and then project to $|0\rangle$ or $|1\rangle$ with equal probability, thus generating a random bit. Therefore BQP certainly contains BPP. As we discussed earlier, it seems reasonable to conjecture that BPP is a proper subset of BQP, because a probabilistic classical computer cannot efficiently simulate a quantum computer. The fundamental difficulty is that the Hilbert space of n qubits is 2^n-dimensional, and hence exponentially large in n, so the mathematical description of a typical vector in the space is exceedingly complex.

Our second concern is to characterize more precisely the resources required to simulate a quantum computer on a classical one. We will demonstrate that despite the vastness of Hilbert space, a classical computer can effectively simulate an n-qubit quantum computer even when constrained to polynomial memory in n. This implies that the BQP complexity class, which encompasses problems solvable by polynomial-size uniform quantum circuit families with bounded error probability, is contained within the *PSPACE* complexity class. *PSPACE* consists of problems that a classical computer can solve using a polynomial amount of memory, though they may require exponential time.

Furthermore, we already know that NP is included in *PSPACE*. This follows from the ability to determine whether a verifier $V(x, y)$ accepts a string x for any witness y by running the verifier on all possible witnesses y and checking for acceptance. Although the number of possible witnesses is exponential, we only need to store one at a time, ensuring the memory requirement remains polynomial (even if the time complexity remains exponential).

The third important issue we should address is *accuracy*. The class BQP is defined formally under the assumption that quantum gates can be constructed and executed with perfect precision. Clearly we should relax this assumption in any real world implementation of quantum computation. A polynomial size quantum circuit family that solves a hard problem would obviously not be of much interest if the quantum gates in the circuit were required accuracy that consumes an exponential amount of resources. In fact, we will show that this is not the case. An idealized T-gate quantum circuit can be simulated with acceptable accuracy by noisy gates, provided that the error probability per gate scales like $1/T$.

The fourth important issue is *coverage*. As we have seen in the previous section, polynomial-size classical circuits can reach only a tiny fraction of all Boolean functions, because there are many more functions than circuits. A similar issue arises for unitary transformations - the unitary group acting on n-qubits is vast, and there are not nearly enough polynomial-size quantum circuits to explore it thoroughly. Most quantum states of n qubits can never be realized in Nature, as they cannot be prepared using only reasonable resources.

Despite this limited reach of polynomial-size quantum circuits, quantum computers nevertheless pose a serious challenge to the strong Church-Turing thesis, which contends that any physically reasonable model of computation can be simulated by probabilistic classical circuits with, at most, a polynomial slowdown. We have good reason to believe that classical computers are unable in general to simulate quantum computers efficiently. In other, more precise terms

$$BPP \subsetneq BQP,$$

yet this remains as another unproven conjecture. Proving that $BPP \neq BQP$ is a great challenge and no proof should be expected soon. Indeed, a corollary would be

$$BPP \subsetneq PSPACE,$$

which would settle a long-standing and pivotal open question in classical complexity theory.

It might be less unrealistic to hope for a proof that $BPP \neq BQP$ follows from another standard conjecture of complexity theory such as $P \neq NP$, though no such proof has been found so far. The most persuasive evidence we have suggesting that $BPP \neq BQP$ is that there are some problems which *seem* to be hard for classical circuits yet can be solved efficiently by quantum circuits.

Therefore we can expect that the classification of complexity will differ depending on whether we use a classical or quantum computer to solve problems.

Accuracy

Now let's discuss the issue of accuracy and let's imagine that we want to implement a computation in which the quantum gates $\mathbf{U}_1, \mathbf{U}_2, \ldots \mathbf{U}_t$ are applied sequentially to the initial state $|\varphi_0\rangle$. The state prepared by our ideal quantum circuit is

$$|\varphi_t\rangle = \mathbf{U}_t \mathbf{U}_{t-1} \ldots \mathbf{U}_2 \mathbf{U}_1 |\varphi_0\rangle .$$

In a realistic implementation, the gates $\mathbf{U}_i$ will not be of perfect accuracy. When we try to apply the unitary gate $\mathbf{U}_i$ we really are applying a "nearby" unitary transformation $\hat{\mathbf{U}}_i$. Physically we can understand this as a consequence of imperfections in the control of the environment around the quantum circuit, so we may regard $\hat{\mathbf{U}}_i$ as a transformation acting jointly on the quantum circuit and its environment, where the ideal unitary is a product $\mathbf{U}_i \times \mathbf{V}_i$, with $\mathbf{U}_i$ acting on the computer and $\mathbf{V}_i$ acting on the environment.

The errors cause the actual state of the quantum computer to wander away from the ideal state. How far does it wander? After one step, the ideal state would be

$$|\varphi_1\rangle = \mathbf{U}_1 |\varphi_0\rangle ,$$

But if the actual transformation $\hat{\mathbf{U}}_1$ where applied instead, the state would be

$$\hat{\mathbf{U}}_1 |\varphi_0\rangle = \mathbf{U}_1 |\varphi_0\rangle + |E\rangle_1 = |\varphi_1\rangle + |E_1\rangle ,$$

where

$$|E_1\rangle = (\hat{\mathbf{U}}_1 - \mathbf{U}_1) |\varphi_0\rangle ,$$

is an unnormalized vector. Although we could also suppose that the initial state deviates from $|\varphi_0\rangle$ by some small error vector which contributes an aditional error to the computation that does not depend on the circuit size, we will ignore this possibility for we are trying to understand how the error scales with the circuit size.

Now if $\hat{\mathbf{U}}_t$ denotes the actual gate applied at step t, $|\hat{\varphi}_t\rangle$ the actual state of the computer after step t, and $|\varphi_t\rangle$ the ideal state, then we have

$$\begin{aligned}|\hat{\varphi}_t\rangle &= \mathbf{U}_t |\varphi_{t-1}\rangle + (\hat{\mathbf{U}}_t - \mathbf{U}_t) |\varphi_{t-1}\rangle + \hat{\mathbf{U}}_t(|\hat{\varphi}_{t-1}\rangle - |\varphi_{t-1}\rangle) \\ &= |\varphi_t\rangle + |E_t\rangle + \hat{\mathbf{U}}_t(|\hat{\varphi}_{t-1}\rangle - |\varphi_{t-1}\rangle),\end{aligned}$$

where $|E_t\rangle = (\hat{\mathbf{U}}_t - \mathbf{U}_t) |\varphi_{t-1}\rangle$. Hence we have

$$\begin{aligned}|\hat{\varphi}_2\rangle = \hat{\mathbf{U}}_2 |\hat{\varphi}_1\rangle &= |\varphi_2\rangle + |E_2\rangle + \hat{\mathbf{U}}_2 |E_1\rangle , \\ |\hat{\varphi}_3\rangle = \hat{\mathbf{U}}_3 |\hat{\varphi}_2\rangle &= |\varphi_3\rangle + |E_3\rangle + \hat{\mathbf{U}}_3 |E_2\rangle + \hat{\mathbf{U}}_3\hat{\mathbf{U}}_2 |E_1\rangle ,\end{aligned}$$

and so on. We can see that after T steps we obtain

$$\begin{aligned}|\hat{\varphi}_T\rangle = |\varphi_T\rangle + |E_T\rangle + \hat{\mathbf{U}}_T |E_{T-1}\rangle + \hat{\mathbf{U}}_T\hat{\mathbf{U}}_{T-1} |E_{T-2}\rangle \\ + \cdots + \hat{\mathbf{U}}_T\hat{\mathbf{U}}_{T-1} \ldots \hat{\mathbf{U}}_2 |E_1\rangle .\end{aligned}$$

Therefore we have expressed the difference between $|\hat{\varphi}_T\rangle$ and $|\varphi_T\rangle$ as a sum of T remainder terms. The worst case error is obtained when the remainder terms are all lined up in the same direction, so that the errors add up. Therefore we conclude that

$$\| \, |\hat{\varphi}_T\rangle - |\varphi_T\rangle \, \| \leq \| \, |E_T\rangle \, \| + \| \, |E_{T-1}\rangle \, \| + \cdots + \| \, |E_1\rangle \, \|,$$

where the norm of than unitary matrix is 1.

Let $\|\mathbf{A}\|_{\sup}$ denote the sup norm of the matrix $\mathbf{A}$, that is, the maximum of the eigenvalues of $\sqrt{\mathbf{A}^\dagger \mathbf{A}}$. We then have

$$\| \, |E_t\rangle \, \| = \|(\hat{\mathbf{U}}_t - \mathbf{U}_t) |\varphi_{t-1}\rangle \, \| \leq \|\hat{\mathbf{U}}_t - \mathbf{U}_t\|_{\sup},$$

since $\| \, |\varphi_{t-1}\rangle \, \| = 1$. Now suppose that, for each value of t, the error in our quantum gate is bounded by

$$\|\hat{\mathbf{U}}_t - \mathbf{U}_t\|_{\sup} \leq \epsilon; \tag{II.1}$$

then, after T steps, the error in the final state is bounded by

$$\| \, |\hat{\varphi}_T\rangle - |\varphi_T\rangle \, \| \le T\epsilon. \tag{II.2}$$

In this sense, the accumulated error in the state grows linearly with the length of the computation.

The distance bounded in (II.1) can equivalently be expressed as $\|\mathbf{W}_t - I\|_{\text{sup}}$, where $W_t = \mathbf{U}_t^\dagger \hat{\mathbf{U}}_t$. Since $\mathbf{W}_t$ is unitary, its eigenvalues have unit modulus, and each one of them is a phase $e^{i\theta_t}$, and the corresponding eigenvalue of $\mathbf{W}_t - I$ has modulus

$$|e^{i\theta_t} - 1| = (2 - 2\cos\theta)^{1/2},$$

so that (II.1) is the requirement that each eigenvalue satisfies

$$\cos\theta_t > 1 - \frac{\epsilon^2}{2}.$$

or $|\theta_t| < \epsilon$ for $\epsilon \ll 1$. The origin of equation (II.2) is clear. In each time step, $|\hat{\varphi}_t\rangle$ is rotated by an angle of order ϵ at worst, and the distance between te vector increases by at most of order ϵ.

How much accuracy is good enough? In the final step of our computation, we perform an orthogonal measurement, and the probability of obtaining outcome a, in the ideal case, is

$$p_a = |\langle a|\varphi_T\rangle|^2.$$

Because of the error, the actual probability is

$$\hat{p}_a = |\langle a|\hat{\varphi}_T\rangle|^2.$$

It can be proven, although we will not do so here, that the L^1 distance between the ideal and actual probabilities is bounded by

$$\frac{1}{2}\|\hat{p} - p\|_1 = \frac{1}{2}\sum_a |\hat{p}_a - p_a| \le \| \, |\hat{\varphi}_T\rangle - |\varphi_T\rangle \, \| \le T\epsilon. \tag{II.3}$$

Therefore, if we keep $T\epsilon$ fixed and small as T increases, the error in the probability of the outcome will be fixed and small.

If we use a quantum computer to solve a decision problem, we want the actual quantum circuit to get the right answer with probability $\frac{1}{2} + \hat{\delta}$, where $\hat{\delta} > 0$. If the ideal quantum circuit has T gates and success probability $\frac{1}{2} + \delta$ where $\delta > 0$, then eq. (II.3) shows that $\hat{\delta}$ is also positive provided $\epsilon < \delta/T$. We should be able to solve hard problems using quantum computers as long as we can improve the accuracy of the gates linearly with the circuit size. This is still a demanding requirement, since performing very accurate quantum gates is a daunting challenge for the hardware builder. Fortunately, we will be able to show, using the theory of quantum fault tolerance, that *physical* gates with constant accuracy (independent of T) suffice to achieve *logical* gates acting on encoded quantum states with accuracy improving like $1/T$, as is required for truly scalable quantum computing.

BQP $\subset$ PSPACE

A randomized classical computer can simulate any quantum computer if we grant the classical computer enough time and storage size. But how much memory does the classical computer require? Naively, since the simulation of an n-qubit circuit involves manipulating matrices of size 2^n, it may seem that we need an amount of memory space exponential in n. However, we will now show that the classical simulation of a quantum computer can be done to acceptable accuracy (although very slowly indeed) in polynomial space. This means that the quantum class BQP is contained in the class PSPACE of problems that can be solved with polynomial space on a classical computer.

The primary objective of the randomized classical simulation is to generate samples from a probability distribution that closely approximates the distribution of measurement outcomes for the given quantum circuit. Remarkably, our classical simulation goes beyond mere sampling, as it tackles an even more challenging task – estimating the probability $p(a)$ for each potential outcome a of the final measurement, which can be expressed

$$p(a) = |\langle a|\mathbf{U}|0\rangle|^2,$$

where

$$\mathbf{U} = \mathbf{U}_T\mathbf{U}_{T-1}\ldots\mathbf{U}_1,$$

is a product of T quantum gates. Each $\mathbf{U}_i$, acting on the n qubits, can be represented by a $2^n \times 2^n$ unitary matrix, characterized by the complex matrix elements

$$\langle y|\mathbf{U}_t|x\rangle\,,$$

where $x, y \in \{0, 1, \ldots, 2^n - 1\}$. Writing out the matrix multiplication explicitly we have

$$\langle a|\mathbf{U}|0\rangle = \sum_{\{x_t:t=1,\ldots,T-1\}} \langle a|\mathbf{U}_T|x_{T-1}\rangle\,\langle x_{T-1}|\mathbf{U}_{T-1}|x_{T-2}\rangle\ldots\langle x_1|\mathbf{U}_1|0\rangle\,. \tag{II.4}$$

Eq. (II.4) is a sort of "path integral" representation of the quantum computation - the probability amplitude for the final outcome a is expressed as a coherent sum of amplitudes for each of a vast number $2^{n(T-1)}$ of possible computational paths that begin at 0 and terminate at a after T steps.

Our classical simulator is to add up the $2^{n(T-1)}$ complex numbers in eq. (II.4) to compute $\langle a|\mathbf{U}|0\rangle$. The first problem we face is that finite size classical circuits do integer arithmetic, while the matrix elements $\langle y|\mathbf{U}_T|x\rangle$ need not be rational numbers. The classical simulator must therefore settle for an approximate calculation to reasonable accuracy. Each term in the sum is a product of T complex factors, and there are $2^{n(T-1)}$ terms in the sum. The accumulated error in the sum is therefore small if we express the matrix elements to an accuracy of m bits, where m is a large number compared to $nT\log T$. Therefore we can replace each complex matrix element by pairs of signed integers - the binary expansions, each of length m, of the real and imaginary parts.

Our simulator will need to compute each term in the sum (II.4) and accumulate a total of all the terms. However, each addition requires only a modest amount of scratch space, and furthermore, since we only need to storage the accumulated sum to an accuracy of m bits, not much space is needed to sum all the terms, even though there are exponentially many of them.

Therefore it only remains to consider the evaluation of a typical term in the sum, a product of T matrix elements. In order to do this, we will need a classical circuit that computes

$$\langle y|\mathbf{U}_t|x\rangle\,;$$

this circuit receives the $2n$-bit input (x,y), and outputs the $2m$-bit approximation to the complex matrix element. Given a circuit that performs this function, it will be easy to build a circuit that multiplies the complex numbers together without using much space.

This task would be difficult if $\mathbf{U}_t$ were an arbitrary $2^n \times 2^n$ unitary matrix. However, we can exploit the properties that we demanded of our quantum gate set - the gates from a discrete set, and each gate acts on a bounded number of qubits. Because there are a fixed finite number of gates, there are only a fixed number of gate subroutines that our simulator needs to be able to call. And because each gate acts on a few qubits, nearly all of its matrix elements vanish (when n is large), and the value $\langle y|\mathbf{U}_t|x\rangle$ can be computed to the required accuracy by a simple circuit requiring only a small amount of memory.

For example, in the case of a single qubit acting on the first qubit, we have

$$\langle y_{n-1}\ldots y_1 y_0|\mathbf{U}_t|x_{n-1}\ldots x_1 x_0\rangle = 0 \text{ if } y_{n-1}\ldots y_1 \neq x_{n-1}\ldots x_1.$$

A simple circuit can compare x_1 with y_1, x_2 with y_2, and so on, and if any of the bits disagree, the circuit outputs 0. In the event of an equality, the circuit outputs one of the four complex numbers

$$\langle y_0|\mathbf{U}_t|x_0\rangle\,,$$

to m bits of accuracy. A simple classical circuit can encode the $8m$ bits of this 2×2 complex-valued matrix. Similarly, a simple circuit, requiring only space polynomial in m, can evaluate the matrix elements of any gate of fixed size.

We conclude then, that a classical computer with memory space scaling like $nT\log T$ can simulate a quantum circuit with T gates acting on n qubits. If we wished to consider quantum circuits with superpolynomial size T, we would need a lot of memory, but for quantum circuit families with size polynomial in n, a polynomial amount of memory suffices. We have therefore proved that BQP is contained in PSPACE.

But it is also evident that the simulation is not efficient, as it requires exponential time, because we need to evaluate the sum of $2^{n(T-1)}$ complex terms (where each term in the sum is a product of T complex numbers). Though most of these terms are zero, we still need to evaluate an exponential number of nonvanishing terms.

Most unitary transformations require large quantum circuits

We saw that any given Boolean function can be computed by an exponential size classical circuit, and also that exponential size circuits are required to compute most functions. What are the corresponding statements about unitary transformations and quantum circuits?

We will not discuss how large a quantum circuit *suffices* to compute a given unitary transformation, focusing instead on showing that quantum circuits with exponential size are *necessary* to compute most unitary transformations.

The question about quantum circuits is different than the corresponding question about classical circuits, because there is a finite set of Boolean functions action on n bits, but there is a continuum of unitary transformations acting on n qubits. Since the quantum circuits are countable (if the quantum computer's gate set is finite), and the unitary transformations are not, we cannot reach arbitrary unitaries with finite-size circuits. We will be satisfied to accurately *approximate* an arbitrary unitary.

As noted in our discussion of quantum circuit accuracy, to ensure that we have a good approximation in the L^1 norm to the probability distribution for any measurement performed after applying a unitary transformation, it suffices for the actual unitary $\tilde{\mathbf{U}}$ to be close to the ideal unitary $\mathbf{U}$ in the sup norm. Therefore we will say that $\tilde{\mathbf{U}}$ is δ-close to $\mathbf{U}$ if $\|\tilde{\mathbf{U}} - \mathbf{U}\|_{\sup} \leq \delta$. How large should the circuit size T be if we want to approximate any arbitrary n-qubit unitary to accuracy δ?

If we consider balls of radius δ (in the sup norm) centered at each unitary achieved by some circuit with T gates, then the balls should cover the unitary group $U(N)$, where $N = 2^n$. The number N_{balls} of balls needed satisfies

$$N_{balls} \geq \frac{\mathrm{Vol}(U(N))}{\mathrm{Vol}(\delta - ball)},$$

where $\mathrm{Vol}(U(N))$ is the volume of the unitary group $U(N)$ with respect to the sup norm and $\mathrm{Vol}(\delta - ball)$ is the volume of a ball of radius δ. The geometry of $U(N)$ is actually curved, but we may safely approximate it by a flat space and ignore that subtlety - all we need to know is that $U(N)$ contains a ball centered at the identity with a small but constant redius C (independent of N). Ignoring the curvature, because $U(N)$ has real dimension N^2, the volume of this ball (a lower bound on the volume of $U(N)$) is $\Omega_{N^2} C^{N^2}$, where Ω_{N^2} denotes the volume of a unit ball in flat space; likewise, the volume of a δ-ball is $\Omega_{N^2}\delta^{N^2}$. Therefore we have that

$$N_{balls} \geq \left(\frac{C}{\delta}\right)^{N^2}.$$

On the other hand, if our universal gate set contains a constant number G of quantum gates (independent of n), and each gate act on no more than k qubits, where k is a constant, then the number of ways to choose the quantum gate at step t is at most $G\binom{n}{k} = \mathrm{poly}(n)$. Therefore the number N_T of quantum circuits with T gates acting on n-qubits is at most $(\mathrm{poly}(n))^T$.

We conclude that if we want to reach every element of $U(N)$ to accuracy δ with circuits of size T, hence $N_T \geq N_{balls}$, then we must have

$$T \geq 2^{2n} \frac{\log(C/\delta)}{\log(\mathrm{poly}(n))},$$

i.e. the circuit size must be exponential in n. With polynomial-size quantum circuits, we can achieve a good approximation to unitaries that occupy only an exponentially small fraction of the volume of $U(N)$.

Reaching any desired quantum state by applying a suitable quantum circuit to a fixed initial (e.g., product) state is easier than reaching any desired unitary, but still hard, because the volume of the 2^n-dimensional n-qubit Hilbert space is exponential in n. Therefore, circuits with size exponential are required. Future quantum engineers will know the joy of exploring Hilbert space, but no matter how powerful their technology, most quantum states will remain forever out of reach.

Chapter III

Shor's Algorithm

The main goal of this chapter is to introduce and explain the algorithm Peter Shor presented to the world in [9]. This chapter is based on the sixth chapter of [5].

III.1 Some Quantum Algorithms

Although we are not able to prove that BPP $\subsetneq$ BQP, there are three approaches that we can take to study the differences between the capabilities of classical and quantum computers:

(1) **Nonexponential speedup.** We can find quantum algorithms that are faster than the best classical algorithm, but not *exponentially* faster. These algorithms shed no light on the conventional classification of complexity. However they do demonstrate a type of separation between tasks that classical and quantum computers can perform. For example, we can mention Grover's quantum speedup of the search of an unsorted data base, as can be studied in [4].

(2) **"Relativized" exponential speedup.** We can consider the problem of analyzing the contents of a "quantum black box". This is a device that performs an *a priori* unknown unitary transformation on an input state. We can prepare said input for the box, and we can measure the output, but we cannot see inside the box. Our task is to find what the box does. It is possible to prove that quantum black boxes, or oracles, exist with this property. By feeding quantum superpositions to the box, one can learn what is inside with an *exponential* speedup, relative to how long it would take if we were only allowed classical inputs. We could say BQP $\supsetneq$ BPP relative to the oracle. For example, we can mention Simon's exponential quantum speedup for finding the period of a 2-to-1 function.

(3) **Exponential speedup for "aparently" hard problems.** We can exhibit a quantum algorithm that solves a problem in polynomial time, where the problem appears to be hard for classical computers, so that it is strongly suspected (though not proven) that the problem is not in BPP. For example, we can mention Shor's exponential quantum speedup for factoring integers.

Deutsch's problem

We will discuss examples for each of the three approaches. But first, we will introduce a simple problem that will be useful for illustrating the power of quantum computation. Deutsch's algorithm for distinguishing between constant and balanced functions $f : \{0,1\} \to \{0,1\}$. We are presented with a quantum black box that computes $f(x)$; that is, it enacts the two-qubit unitary transformation

$$U_f : |x\rangle\,|y\rangle \to |x\rangle\,|y \oplus f(x)\rangle\,,$$

which flips the second qubit if and only if $f(x) = 1$. Our mission is to determine whether $f(0) = f(1)$ or not. If we are restricted to the "classical" inputs $|0\rangle$ and $|1\rangle$, we need to access the box twice ($x = 0$ and $x = 1$) to get the answer. However if we are allowed to input a coherent superposition of these "classical" states, then one is enough.

The quantum circuit that solves the problem is:

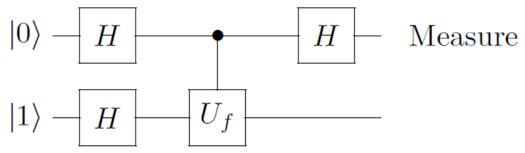

Figure III.1: Circuit for Deutsch's problem

Here H denotes the Hadamard transform

$$\mathbf{H} : |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_y (-1)^{xy}\,|y\rangle\,,$$

or

$$\begin{aligned} \mathbf{H} : |0\rangle &\mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \\ |1\rangle &\mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle), \end{aligned}$$

that is, $\mathbf{H}$ is the 2×2 matrix

$$\mathbf{H} : \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}.$$

The circuit takes the input $|0\rangle\,|1\rangle$ to

$$\begin{aligned} |0\rangle\,|1\rangle &\mapsto \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle) \\ &\mapsto \frac{1}{2}\left((-1)^{f(0)}\,|0\rangle + (-1)^{f(1)}\,|1\rangle\right)(|0\rangle - |1\rangle) \\ &\mapsto \frac{1}{2}\left[\left((-1)^{f(0)} + (-1)^{f(1)}\right)|0\rangle\right. \\ &\quad \left. + \left((-1)^{f(0)} - (-1)^{f(1)}\right)|1\rangle\right] \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \end{aligned}$$

Then when we measure the first qubit, we find the outcome $|0\rangle$ with probability one if $f(0) = f(1)$ (constant function) and the outcome $|1\rangle$ the probability one if $f(0) \neq f(1)$ (balanced function).

A quantum computer enjoys an advantage over a classical computer because it can use *quantum parallelism*. Because we input a superposition of $|0\rangle$ and $|1\rangle$, the output is sensitive to both the values of $f(0)$ and $f(1)$, even though we ran the box just once.

Deutsch-Jozsa problem

Now we will consider some generalizations of Deutsch's problem as was exposed in [3]. We will continue to assume that we are to analyze a quantum oracle. However, in the hope of learning something about complexity, we will imagine that we have a family of oracles, with variable input size. We are interested in how the time needed to find out what is inside the box scales with the size of the input (where "time" is measured by how many times we query the box).

In the *Deutsch-Jozsa problem* we are presented with a quantum oracle that computes a function taking n bits to 1,

$$f : \{0,1\}^n \to \{0,1\},$$

and we have in good authority that f is either constant ($f(x) = c\,\forall x$) or balanced ($f(x) = 0$ for exactly half of the inputs). Our mission is to determine which of the two cases holds.

In fact we can solve this problem also with a single query to the back box, using the same circuit as for Deutch's problem, but with n input qubits instead of just one. We note that if we apply n Hadamard gates in parallel to n qubits,

$$\mathbf{H}^{(n)} = \mathbf{H} \otimes \mathbf{H} \otimes \cdots \otimes \mathbf{H},$$

then the n-qubit state transforms as

$$\mathbf{H}^{(n)} : |x\rangle \mapsto \prod_{i=1}^{n} \left(\frac{1}{\sqrt{2}} \sum_{y_i=\{0,1\}} (-1)^{x_i y_i} |y_i\rangle \right) \equiv \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} (-1)^{x\cdot y} |y\rangle ,$$

where x, y represent n-bit strings and $x \cdot y$ is the bitwise inner product modulo 2 or AND

$$x \cdot y = (x_1 \wedge y_1) \oplus (x_2 \wedge y_2) \oplus \cdots \oplus (x_n \wedge y_n).$$

Acting on the input $|0\rangle^n |1\rangle$, the action of the circuit is

$$\begin{aligned} |0\rangle^n |1\rangle &\mapsto \left(\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \right) \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &\mapsto \left(\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle \right) \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &\mapsto \left(\frac{1}{2^n} \sum_{x=0}^{2^n-1} \sum_{y=0}^{2^n-1} (-1)^{f(x)} (-1)^{x\cdot y} |y\rangle \right) \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \end{aligned} \tag{III.1}$$

Now, let's evaluate the sum

$$\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}.$$

If f is constant, the sum is

$$(-1)^{f(x)}\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{x\cdot y}=(-1)^{f(x)}\delta_{y,0};$$

it vanishes unless $y=0$. Therefore, when we measure the n-bit register, we obtain the result $|y=0\rangle \equiv |0\rangle^n$ with probability one. On the other hand, if f is balanced, the sum is

$$\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}=0,$$

because half of the terms are $+1$ and half are -1. Therefore, the probability of measuring $|y=0\rangle$ is zero.

We conclude that one query of the quantum oracle suffices to determine whether the function f is constant or balanced with all certainty. The measurement $y=0$ is the only possible outcome if f is constant, and it is impossible if f is balanced.

So quantum computation solves this problem neatly, but is the problem really hard classically? If we are restricted to classical input states $|x\rangle$, we can query the oracle repeatedly, choosing the input x randomly (without replacement) each time. Once we obtain distinct outputs for two different inputs, we know that the function is balanced. But if the function is constant, we will not be *certain* that the function is constant until we have tried $2^{n-1}+1$ inputs and have gotten the same output each time. In contrast, the quantum algorithm gives us a definite answer after just one query. So in this sense (demanding absolute certainty), the classical calculation requires a number of steps that grows exponentially with n, while the quantum calculation requires only a constant number of steps. Thus we may claim that the quantum algorithm is exponentially faster than the classical algorithm.

Perhaps it is not reasonable to expect absolute certainty from classical computations, especially since real quantum computers are subject to errors that prevent them from providing absolute certainty as well. Therefore, we may adopt a more practical approach where we are content with making guesses about whether a given input is balanced or constant, with a probability of error no greater than ε. Hence,

$$P(\text{success})=1-\varepsilon.$$

If the function is actually balanced and we make k queries, the probability of getting the same response every time is $p=2^{-(k-1)}$. If after receiving the same response k times we guess that the function is constant, then a quick Bayesian analysis shows that our guess is wrong with probability $\frac{1}{2^{k-1}+1}$. So if we guess after k queries, the probability of error is

$$1-P(\text{success})=\frac{1}{2^{k-1}(2^{k-1}+1)}.$$

Therefore we can achieve success probability $1-\varepsilon$ for $\varepsilon^{-1} = 2^{k-1}(2^{k-1}+1)$, or $k \simeq \frac{1}{2}\log(\frac{1}{\varepsilon})$. Since we can reach an exponentially good success probability with a number of queries that grows only polynomially with n, it is not fair to say that the problem is hard classically.

Bernstein-Vazirani problem

Exactly the same circuit can be used to solve another variation on the Deutsch-Jozsa problem as was introduced in [2]. Let's suppose that our quantum oracle computes one of the functions f_a where

$$f_a(x) = a \cdot x,$$

and a is an n-bit string. Our job is to determine the value of a.

The quantum algorithm can solve this problem with certainty, given just one n-qubit quantum query. For this particular function, the quantum state in eq. III.1 becomes

$$\frac{1}{2^n}\sum_{x=0}^{2^n-1}\sum_{y=0}^{2^n-1}(-1)^{a\cdot x}(-1)^{x\cdot y}\,|y\rangle\,.$$

However, in fact

$$\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{a\cdot x}(-1)^{x\cdot y} = \delta_{a,y},$$

so this state is just $|a\rangle$. We can execute the circuit once and measure the n-qubit register, finding the n-bit string a with probability one.

If only classical queries are allowed, we acquire only one bit of information per query and it takes n queries to determine the value of a. Therefore, we have a clear separation between the classical and quantum algorithms. Even so, this example des not probe the relation between BPP and BQP, because the problem is not hard classically.

Simon's problem

Bernstein and Vazirani managed to formulate a version on the previous problem that *is* hard classically and so establish for the first time a "relativized" separation between classical and quantum complexity. We will find it more instructive to consider a simpler example proposed later by Daniel Simon in [10].

Once again we are presented with a quantum black box, and this time we are assured that the oracle computes a function

$$f : \{0,1\}^n \to \{0,1\}^n,$$

that is two-to-one. Furthermore, the function has a period given by the n-bit string a, so that

$$f(x) = f(y) \iff y = x \oplus a,$$

where $\oplus$ denotes addition modulo 2 or XOR. So a is the period if we regard x as taking values in $\mathbb{Z}_2^n$ rather than $\mathbb{Z}_{2^n}$. This is all our information about f. Our job is to determine the value of the period a.

Classically the problem is, in fact, known to be hard. We need to query the oracle an exponentially large number of times to determine the period with reasonable probability. We do not learn anything until we are fortunate enough to choose two queries x and y that give the same response that happen to sattisfy $y \oplus x = a$. Suppose, for example, that we choose $2^{n/4}$ queries. The number of pairs of queries is less than $(2^{n/4})^2 = 2^{n/2}$, and for each pair $\{x, y\}$ the probability of getting the same response is 2^{-n}. So the probability of success is less than

$$2^{n/2}2^{-n} = 2^{-n/2}.$$

Even with exponentially many queries, the probability of success is exponentially small.

If we wish, we can frame the question as a decision problem: Either f is a 1-to-1 function, or it is a 2-to-1 function with period a, each occurring with probability $\frac{1}{2}$. We are asked to determine which is the case. Then, after $2^{n/4}$ classical queries, our probability of making a correct guess is

$$P(\text{success}) < \frac{1}{2} + \frac{1}{2^{n/2}},$$

which does not remain bounded away from $\frac{1}{2}$ as n grows.

But with quantum queries the problem is easy. The circuit we use is essentially the same as above but now *both* registers are expanded to n qubits. We prepare the equally weighted superposition of all n-bit strings (by acting on $|0\rangle$ with $\mathbf{H}^{(n)}$), and then we query the oracle:

$$U_f : \left(\sum_{x=0}^{2^n-1} |x\rangle \right) |0\rangle \mapsto \sum_{x=0}^{2^n-1} |x\rangle \, |f(x)\rangle \, .$$

Now we measure the second register. (This step is not actually necessary, but it makes the argument easier to follow.) The measurement outcome is selected at random from the 2^{n-1} possible values of $f(x)$, each occurring equiprobably. Suppose that the outcome is $f(x_0)$. Then because both x_0 and $x_0 \oplus a$ are mapped by f to $f(x_0)$, we have prepared the state

$$\frac{1}{\sqrt{2}} \left(|x_0\rangle + |x_0 \oplus a\rangle \right),$$

in the first register.

Now we want to extract some information about a. Clearly it would do us no good to measure the register in the computational basis at this point. We would obtain either the outcome x_0 or $x_0 \oplus a$, each with probability $\frac{1}{2}$, but we would not learn anything about a.

Suppose we apply the Hadamard transform $\mathbf{H}^{(n)}$ to the register before measuring it.

$$\begin{aligned}\mathbf{H}^{(n)} :&\frac{1}{\sqrt{2}}\left(|x_0\rangle + |x_0 \oplus a\rangle\right)\\ &\mapsto \frac{1}{2^{(n+1)/2}}\sum_{y=0}^{2^n-1}\left[(-1)^{x_0\cdot y} + (-1)^{(x_0\oplus a)\cdot y}\right]|y\rangle\\ &= \frac{1}{2^{(n-1)/2}}\sum_{a\cdot y=0}(-1)^{x_0\cdot y}\,|y\rangle\,.\end{aligned}$$

If $a \cdot y = 1$, then the terms in the coefficient of $|y\rangle$ cancel. Hence only states with $a \cdot y = 0$ remain. The measurement outcome, then, is selected at random from all possible values of y such that $a \cdot y = 0$, each occurring with probability $\frac{1}{2^{n-1}}$.

We run this algorithm repeatedly, each time obtaining another value of y such that $a \cdot y = 0$. After we have found n such linearly independent values $\{y_1, \ldots, y_n\}$, that is, linearly independent over $(\mathbb{Z}_2)^n$, we can solve for a by solving the system of equations

$$\begin{aligned}a \cdot y_1 &= 0\\ a \cdot y_2 &= 0\\ &\vdots\\ a \cdot y_n &= 0.\end{aligned}$$

Once we have determined this unique value for a, our problem is solved. It is easy to see that with $O(n)$ repetitions, we can obtain a success probability that is exponentially close to 1.

So we finally have found a case where quantum computation is exponentially more efficient than classical computation, i.e. there exists an oracle relative to which $\mathbf{BQP} \neq \mathbf{BPP}$.

Note that whenever we compare classical and quantum complexity relative to an oracle, we are considering a quantum oracle, where queries and replies are represented as states in Hilbert space, but with a preferred orthonormal basis. If we submit a classical query (an element of the preferred basis), we always receive a classical response (another basis element).

III.2 Periodicity

So far, the one problem for which we have found an exponential separation between the speed of a quantum algorithm and the speed of the best known corresponding classical algorithm is the case of Simon's problem. Simon's algorithm exploits quantum parallelism to speed up the search for the period of a function. Its success encourages us to seek other quantum algorithms that exploit period finding.

Simon studied periodic functions taking values in $(\mathbb{Z}_2)^n$. For that purpose the n-bit Hadamard transform was a powerful tool. But we can also consider periodic functions taking values in $\mathbb{Z}_{2^n}$, and for that purpose the discrete Fourier transform will be a tool of comparable power.

The moral of Simon's power is that, while finding needles in a haystack may be difficult, finding *periodically* spaced needles in a haystack can be far easier.

Imagine a quantum oracle that computes a function

$$f : \{0,1\}^n \to \{0,1\}^m,$$

that has an unknown period r, where r is a positive integer satisfying

$$1 \ll r \ll 2^n.$$

That is,

$$f(x) = f(x + mr),$$

for all integers m such that $x, x + mr \in \{0, 1, \dots 2^n - 1\}$. We are to find the period r. Classically this is a hard problem. If r is, say, of order $2^{n/2}$, we will need to query the oracle of order $2^{n/4}$ times before we are likely to find two values of x that are mapped to the same value by f, and hence learn something about r. However we will see that there is a quantum algorithm that can find r with high probability using only in time $\mathrm{poly}(n)$.

Even if we know how to compute efficiently the function $f(x)$, it may be hard to find its period. Our quantum algorithm can be applied to finding, in polynomial time, the period of any function f that can be computed in $\mathrm{poly}(n)$ time. Efficient period finding allows us to efficiently solve a variety of (apparently) hard problems such as factoring an integer or evaluating a discrete logarithm. The main purpose of this work is to show how the first can be done.

The key idea underlying quantum period finding is that the Fourier transform can be evaluated by an efficient quantum circuit, as was discovered by Peter Shor. The *Quantum Fourier Transform* (QFT) exploits the power of quantum parallelism to achieve an exponential speedup over the best known classical algorithm for computing the Fast Fourier Transform (FFT). It is natural to expect that, as the FFT has such a wide variety of applications, the QFT will see widespread use as well.

Finding the period

Now we are entering the heart of the matter. Shor's algorithm is composed of two parts: the quantum period finding algorithm and the classical part. We start with the quantum part in this section, and continue with the classical part in the next one.

We may start by defining the Quantum Fourier Transform as the unitary transformation that acts on the computational basis according to

$$QFT : |x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{2\pi i x y / N} \, |y\rangle \, , \qquad \text{(III.2)}$$

where $N = 2^n$. For now let's suppose we can perform the QFT efficiently (and we will see later that this is indeed the case), and let's see how it enables us to extract the period of the periodic function f.

Emulating Simon's algorithm, we first query the oracle with the input $\frac{1}{\sqrt{N}} \sum_x |x\rangle$ (easily prepared by applying the Hadamard transform to $|0\rangle$). This gives us the state

$$\frac{1}{\sqrt{N}} \sum_{x}^{N-1} |x\rangle |f(x)\rangle .$$

Then we measure the output register, obtaining the result $|f(x_0)\rangle$ for some $0 \leq x_0 < r$. This measurement prepares the input register for the cohererent superposition of the A values that are mapped to $f(x_0)$:

$$\frac{1}{\sqrt{A}} \sum_{j=0}^{A-1} |x_0 + jr\rangle , \tag{III.3}$$

where

$$N - r \leq x_0 + (A-1)r < N,$$

or, equivalently,

$$A - 1 \leq \frac{N}{r} < A + 1.$$

Actually the measurement of the output register is unnecessary. If it is omitted, the state of the input register is an incoherent superposition (summed over $x_0 \in \{0, 1, \ldots, r-1\}$) of states of the form eq. III.3. The rest of the algorithm works just as well acting on the initial state.

Now our job is to extract the value of r from the state eq. III.3 that we have prepared. Were we to measure the input register by projecting onto the computational basis at this point, we would learn nothing about r. Instead, just like in Simon's algorithm, we should Fourier transform first and, only then, measure.

By applying the QFT to the state eq. III.3, we obtain

$$\frac{1}{\sqrt{NA}} \sum_{y=0}^{N-1} e^{2\pi i x_0 y/N} \sum_{j=0}^{A-1} e^{2\pi i j r y/N} |y\rangle .$$

If we now measure in the computational basis, the probability of obtaining the outcome y is

$$P(y) = \frac{A}{N} \left| \frac{1}{A} \sum_{j=0}^{A-1} e^{2\pi i j r y/N} \right|^2 .$$

This probability distribution strongly favors values of y such that yr/N is close to an integer. For example, if N/r happened to be an integer (and therefore equal to A), we would have

$$P(y) = \frac{1}{r} \left| \frac{1}{A} \sum_{j=0}^{A-1} e^{2\pi i j y/A} \right| = \begin{cases} 1/r & \text{if } y \text{ is a multiple of } A \\ 0 & \text{otherwise.} \end{cases}$$

More generally we may sum the geometric series

$$\sum_{j=0}^{A-1} e^{i\theta j} = \frac{e^{iA\theta}-1}{e^{i\theta}-1}, \tag{III.4}$$

using

$$\theta_y = \frac{2\pi yr \mod N}{N}.$$

There are precisely r values of y in $\{0, 1, \ldots, N-1\}$ such that

$$-\frac{r}{2} \leq yr \mod N \leq \frac{r}{2}. \tag{III.5}$$

To see this, imagine marking the multiples of r and N on a number line ranging from 0 to $rN-1$. For each multiple of N, there is a multiple of r no more than distance $r/2$ away. For each of these values, the corresponding θ_y satisfies

$$-\pi\frac{r}{n} \leq \theta_y \leq \pi\frac{r}{n}.$$

Now, since $A-1 < \frac{N}{r}$, for these values of θ_y all of the terms in the sum over j in eq. (III.4) lie in the same half-plane of the complex plane, so that the terms interfere constructively and the sum is substantial.

We know that

$$|1-e^{iA\theta}| \leq |\theta|,$$

because the straight-line distance from the origin is less than the arc length along the circle, and for $A|\theta| \leq \pi$ we have

$$|1-e^{iA\theta}| \geq \frac{2A|\theta|}{\pi},$$

because we can see that this distance is a convex function. We actually have $A < \frac{N}{r}+1$, and hence $A\theta_y < \pi(1+\frac{r}{N})$ but by applying the above bound to

$$\left|\frac{e^{i(A-1)\theta}-1}{e^{i\theta}-1} + e^{i(A-1)\theta}\right| \geq \left|\frac{e^{i(A-1)\theta}-1}{e^{i\theta}-1}\right| - 1,$$

we can still conclude that

$$\left|\frac{e^{i(A-1)\theta}-1}{e^{i\theta}-1}\right| \geq \frac{2(A-1)|\theta|}{\pi|\theta|} - 1 = \frac{2A}{\pi} - (1+\frac{2}{\pi}).$$

Ignoring a possible correction of order $2/A$ then, we find that

$$P(y) \geq \left(\frac{4}{\pi^2}\right)\frac{1}{r},$$

for each of the r values that satisfy eq. (III.5). Thus, with a probability of, at least, $4/\pi^2$, the measured value of y will satisfy

$$k\frac{N}{r} - \frac{1}{2} \leq y \leq k\frac{N}{r} + \frac{1}{2},$$

or

$$\frac{k}{r} - \frac{1}{2N} \leq \frac{y}{N} \leq \frac{k}{r} + \frac{1}{2N},$$

where $k \in \{0, 1, \ldots, r-1\}$. The output of the computation is reasonably likely to be within distance $1/2$ of an integer multiple of N/r.

Suppose that we know that $r < M \ll N$. Thus, we know that N/r is a rational number with a denominator smaller than M. Two distinct rational numbers with denominators less than M cannot be closer than $1/M^2$. Thus, if we measure y to be within distance $1/2$ from an integer multiple of N/r, then there is a unique value of k/r (with r < M) determined by y/N, provided that $N \geq M^2$. This value of k/r can be efficiently extracted from the measured y/N by continued fractions.

Now, with probability exceeding $4/\pi^2$, we have found a value of k/r where K is selected from $\{0, 1, \ldots, r-1\}$. It is reasonably likely that k and r are relatively prime, so that we have succeeded in finding r. With a query of the oracle, we may check whether $f(x) = f(x+r)$. But if $\gcd(k, r) \neq 1$, we have found only a factor r_1 of r.

In the case where we do not initially succeed in finding r using the quantum algorithm, we have various strategies to improve the chances of success. For instance, we can test nearby values of y since the measured value might be close to the range $-r/2 \leq yr \mod N \leq r/2$, but not precisely within it. Additionally, trying a few multiples of r (the value of $\gcd(k, r)$, if not 1, is probably not large) can also help refine our result.

If the above attempts do not yield the desired factor r, we can resort to repeating the quantum circuit to obtain a new value k'/r with a probability exceeding $4/\pi^2$. At this point, k' might share a common factor with r, leading us to determine another factor r_2 of r. However, it is reasonably likely that $\gcd(k, k') = 1$, in which case we can compute $r = \operatorname{lcm}(r_1, r_2)$.

We can estimate the probability that randomly selected k and k' are relatively prime. Since a prime number p divides a fraction $1/p$ of all integers, the probability that p divides both k and k' is $1/p^2$. Conversely, k and k' are relatively prime if no prime divides both k and k'. Thus, the probability that k and k' are relatively prime is given by:

$$P(k, k' \text{ coprime}) = \prod_{p \text{ prime}} \left(1 - \frac{1}{p^2}\right) = \frac{1}{\zeta(2)} = \frac{6}{\pi^2} \approx 0.61,$$

where ζ is the Riemann zeta function.

Hence, after some constant number of repetitions of the algorithm, the probability of success in finding r becomes significantly high, thanks to the repeated trials and the likelihood of k and k' being relatively prime. This makes the quantum algorithm a promising approach for factoring large numbers efficiently.

From FFT to QFT

Now let's consider the implementation of the quantum Fourier transform. The Fourier transform

$$\sum_x f(x) |x\rangle \mapsto \sum_y \left(\frac{1}{\sqrt{N}} \sum_x e^{2\pi i xy/N} f(x) \right) |y\rangle ,$$

is a multiplication by an $N \times N$ unitary matrix. In said matrix, the (x, y) matrix element is $(e^{2\pi i/N})^{xy}$. Naively, this transforms requires $O(N^2)$ elementary operations. However there is a well-known and very useful classical procedure that reduces the number of operations to $O(N \log N)$. Assuming $N = 2^n$, we start by expressing x and y in binary as

$$x = x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + \cdots + x_0, y = y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \cdots + y_0.$$

In the product of x and y we may discard any terms containing n or more powers of 2, as there is no contribution to $e^{2\pi i xy/N}$ from such terms. Thus, we may write

$$\frac{xy}{2^n} \equiv y_{n-1}(.x_0) + y_{n-2}(.x_1 x_0) + \cdots + y_0(.x_{n-1} \ldots x_0), \qquad \text{(III.6)}$$

where the factors in parentheses are binary expressions; e.g.

$$.x_2 x_1 x_0 = x_2 \cdot 2^{-1} + x_1 \cdot 2^{-2} + x_0 \cdot 2^{-3}.$$

We can now evaluate

$$\tilde{f}(x) = \frac{1}{\sqrt{N}} \sum_y e^{2\pi i xy/N} f(y),$$

for each of the 2^n values of x. However the sum over y factors into n sums over $y_k = 0, 1$, which can be done sequentially in a time of order n.

With quantum parallelism, we can do far better. From eq. (III.6) we obtain

$$\begin{aligned} QFT : |x\rangle &\mapsto \frac{1}{\sqrt{N}} \sum_y e^{2\pi i xy/N} |y\rangle \\ &\mapsto \frac{1}{\sqrt{2^n}} (|0\rangle + e^{2\pi i(.x_0)} |1\rangle)(|0\rangle + e^{2\pi i(.x_1 x_0)} |1\rangle) \\ &\ldots (|0\rangle + e^{2\pi i(.x_{n-1} \ldots x_0)} |1\rangle). \end{aligned}$$

The QFT takes each computational basis state to an *unentangled* state of n qubits; thus it is reasonable to anticipate that it can be efficiently implemented. Indeed, let's consider the case $n = 3$. We can readily see that the circuit

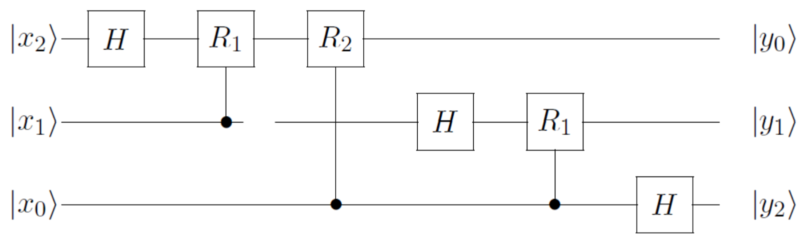

Figure III.2: Circuit for the QFT.

does the job. Note that the order of the bits has been reversed in the output. Each Hadamard gate acts as

$$\mathbf{H} : |x_k\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(.x_k)}|1\rangle).$$

The other contributions to the relative phase of $|0\rangle$ and $|1\rangle$ in the k-th qubit are provided by the two-qubit conditional rotations, where

$$\mathbf{R}_d = \begin{pmatrix} 1 & 0 \\ 0 & e^{\pi i/2^d} \end{pmatrix},$$

and $d = (k - j)$ is the "distance" between the qubits.

In the case $n = 3$, the QFT is constructed from three $\mathbf{H}$ gates and three controlled-$\mathbf{R}$ gates. In general, the QFT is constructed from n $\mathbf{H}$ gates and $\binom{n}{2} = n(n-1)/2$ controlled $\mathbf{R}$ gates. A two-qubit gate is applied to each pair of qubits, again with controlled relative phase $\pi/2^d$, where d is the distance between the qubits. Thus the circuit family that implements the QFT has a size of order $\log(N)^2$.

We can further reduce the circuit complexity to linear in $\log(N)$ by accepting an implementation with fixed accuracy. The two-qubit gates acting on distant qubits contribute only exponentially small phases, allowing us to drop gates acting on pairs of qubits separated by more than m qubits. Consequently, each term in equation (III.6) is replaced by an approximation with m bits of accuracy.

The resulting error in $xy/2^n$ is no worse than $n2^{-m}$, enabling us to achieve an accuracy of ε in $xy/2^n$ with $m \geq \log(n/\varepsilon)$. By retaining only the gates acting on qubit pairs with distances smaller than m, the circuit size becomes of order $mn \sim m\log(n/\varepsilon)$.

In fact, if we plan to measure in the computational basis immediately after the Quantum Fourier Transform (QFT), there are further simplifications we can exploit. Notably, there is no need to apply any two-qubit gates at all. The controlled-$\mathbf{R}_d$ gate exhibits a symmetric action on the two qubits: it acts trivially on states $|00\rangle$, $|01\rangle$, and $|10\rangle$, while modifying the phase of state $|11\rangle$ by $e^{i\theta_d}$. Consequently, we can interchange the control and target bits without affecting the gate's action. With this change, our circuit for the 3-qubit QFT becomes:

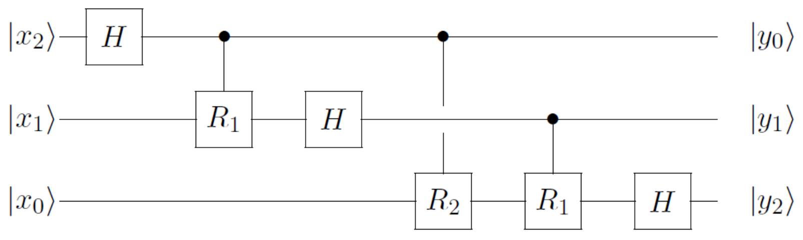

Figure III.3: Circuit for the QFT, with interchange.

Once we have measured $|y_0\rangle$, we know the value of the controlled bit in the controlled-$\mathbf{R}_1$ gate that acted on the first two qubits. Therefore, we will obtain the same probability distribution of measurement outcomes if, instead of applying the controlled-$\mathbf{R}_1$ and then measuring, we instead measure y_0 first and only then apply $(\mathbf{R}_1)^{y_0}$ to the next qubit, conditioned on the outcome of the measurement of the first qubit. Similarly we can replace the controlled-$\mathbf{R}_1$ and controlled-$\mathbf{R}_2$ acting on the third qubit by the single qubit rotation

$$(\mathbf{R}_2)^{y_0}(\mathbf{R}_1)^{y_1},$$

that is, a rotation with relative phase $\pi(.y_1y_0)$, after the values of y_0 and y_1 have been measured.

Altogether then, if we are going to measure after performing the QFT, only n Hadamard gates and $n-1$ single qubit rotations are required. The circuit size for the QFT is remarkably simple.

However, we should note that, however interesting these results are, such simplifications are currently very far of being implemented. Such rotation gates are constructed from the primitive ones that the corresponding backend has. And this is not expected to change in a long time.

III.3 Factoring as period finding

What does the factoring problem have to do with periodicity? There is a well-known randomized reduction of factoring to determining the period of a function. Although this reduction is not directly related to quantum computing, we will discuss it here for the completeness of this work, and because the prospect of using a quantum computer to factor large numbers has generated so much excitement.

Suppose that we want to find a factor of the n-bit number N. Select pseudo-randomly $a < N$ and compute the Euclidean algorithm, which has cost $O(\log(N)^3)$ in order to get $\gcd(a, N)$. If $\gcd(a, N) \neq 1$, then we have found a factor of N.

Let's study the case in which $\gcd(a, N) = 1$. We know that the numbers which are coprime to N form a group under multiplication modulo N. Therefore, a has an order r, which is the smallest positive integer such that

$$a^r \equiv 1 \mod N.$$

The order r is the period of the function

$$f_{N,a}(x) = a^x \mod N.$$

As we have seen, the period of a function can be found efficiently using the QFT. Thus, if we can compute $f_{N,a}(x)$ efficiently, we can find the order r of a efficiently.

Computing $f_{N,a}$ may look difficult at first, since the exponent x can be very large. However, we can use the binary representation of x to compute $f_{N,a}$ efficiently. We can write x as

$$x = x_{m-1} \cdot 2^{m-1} + x_{m-2} \cdot 2^{m-2} + \cdots + x_1 \cdot 2 + x_0,$$

and we have

$$a^x \mod N = (a^{2^{m-1}})^{x_{m-1}} \cdot (a^{2^{m-2}})^{x_{m-2}} \cdots (a^2)^{x_1} \cdot a^{x_0} \mod N.$$

Each a^{2^j} can be computed efficiently by a classical computer using repeated squaring:

$$a^{2^j} \mod N = \begin{cases} a & \text{if } j = 0 \\ (a^{2^{j-1}})^2 \mod N & \text{if } j > 0. \end{cases}$$

So only $m-1$ classical multiplications $\mod N$ are required to assemble a table of all a^{2^j}'s.

The computation of $a^x \mod N$ is accomplished by multiplying together the relevant entries from the table. This process requires a maximum of $m-1 \mod N$ multiplications, with each multiplication necessitating at most an order of $(\log(N))^2$ operations. As $r < N$, selecting $m \sim 2\log N$ provides a reasonable chance of finding r. Consequently, the total cost of computing $f_{N,a}$ is approximately $(\log(N))^3$. This analysis highlights the efficiency of the algorithm and its polynomial complexity in terms of $\log(N)$. Schematically, the circuit has the form:

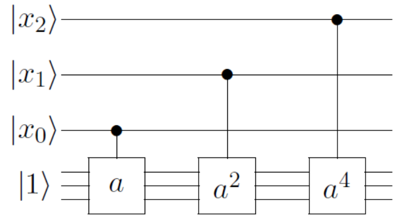

Figure III.4: Circuit for calculating $f_{N,a}(x)$.

Multiplication by a^{2^j} is performed if the control qubit x_j is 1.

Let's suppose now that we have found the period r of $f_{N,a}$. If r is even, then we have

$$N \mid a^r - 1 = (a^{r/2} - 1)(a^{r/2} + 1).$$

We know that N does not divide $a^r - 1$, since if it did, the order of a would be smaller than $r/2$. Thus, if it is also the case that N does not divide $a^{r/2} + 1$, or

$$a^{r/2} \neq -1 \mod N,$$

then N must have a nontrivial common factor with each of $a^{r/2} \pm 1$. Therefore, we $\gcd(N, a^{r/2} + 1) \neq 1$ is a nontrivial factor of N and we are done.

We see that, once we have found r, we succeed in factoring N unless either r is odd or r is even and $a^{r/2} \equiv -1 \mod N$. How likely is success?

Let's suppose that N is a product of two prime factors $p \neq q$, which is actually the least favorable case.

$$N = pq.$$

For each $a < pq$, there exist $a_1 < p$ and $a_2 < q$ such that

$$\begin{aligned} a &\equiv a_1 \mod p, \\ a &\equiv a_2 \mod q. \end{aligned}$$

Choosing a random $a < N$ is, therefore, equivalent to choosing two random numbers $a_1 < p$ and $a_2 < q$.

Now let r_1 and r_2 be the orders of a_1 and a_2 modulo p and q, respectively. The Chinese Remainder theorem tells us that solving $a^r \equiv 1 \mod N$ is equivalent to

$$\begin{aligned} a^{r_1} &\equiv 1 \mod p, \\ a^{r_2} &\equiv 1 \mod q. \end{aligned}$$

Therefore, $r = \operatorname{lcm}(r_1, r_2)$. If r_1 and r_2 are both odd, then so is r and we lose.

However, if either r_1 or r_2 is even, then r is even and we have a chance of success. If

$$\begin{aligned} a^{r_1/2} &\equiv -1 \mod p, \\ a^{r_2/2} &\equiv -1 \mod q, \end{aligned} \tag{III.7}$$

then we have $a^{r/2} \equiv -1 \mod N$ and we lose. Otherwise, if either

$$\begin{aligned} a^{r/2} &\equiv -1 \mod p, \\ a^{r/2} &\equiv 1 \mod q, \end{aligned} \tag{III.8}$$

or

$$\begin{aligned} a^{r/2} &\equiv 1 \mod p \\ a^{r/2} &\equiv -1 \mod q, \end{aligned} \tag{III.9}$$

then $a^{r/2} \not\equiv -1 \mod N$ and we succeed.

Suppose that

$$\begin{aligned} r_1 &= 2^{s_1} r_1', \\ r_2 &= 2^{s_2} r_2', \end{aligned}$$

where r_1' and r_2' are odd and $s_1 > s_2$. Then $r = \operatorname{lcm}(r_1, r_2) = 2 r_2 \cdot$integer, so that $a^{r/2} \equiv 1 \mod q$ and eq. (III.8) is satisfied and we win. Analogously, $s_2 > s_1$ implies that eq. (III.9) is satisfied and we win as well. But if $s_1 = s_2$, then $r = r_1 \cdot \text{odd} = r_2 \cdot \text{odd}'$ so that eq. (III.7) is satisfied and we lose.

Therefore it all comes down to: for $s_1 = s_2$ we lose and for $s_1 \neq s_2$ we win. What is the probability of $s_1 \neq s_2$? We know that the multiplicative group $\mathbb{Z}_p^*$ is cyclic of order $p-1$ and has a generator element. Suppose that $p - 1 = 2^k \cdot c$, where c is odd, and consider the orders of all the elements of the group. For brevity, we will discuss only the case $k = 1$, which is the least favourable case for us. Then if b is a generator of $\mathbb{Z}_p^*$, the even powers of b have odd order, and the odd powers of b have order $2 \cdot \text{odd}$. In this case, then, $r = 2^t \cdot \text{odd}$

where $t \in \{0, 1\}$, each occurring with equal probability. Therefore, if p, q are both of this unfavorable form, and a_1 and a_2 are chosen at random, then the probability of $s_1 \neq s_2$ is $1/2$. Hence, once we found r, the probability of finding a factor is, at least $1/2$, if N is a product of two primes. If N has more than two prime factors, the probability of success is even higher. The method fails if N is a prime power, but prime powers can be efficiently factored by other methods.

Chapter IV

Real World Implementations

This last chapter will be divided in two sections. In the first section, we will discuss the current state of the art of quantum computing, implementing Shor's algorithm to factorize a number and running it in three different simulators. In this part we will study how long it takes right now to factor a number using Shor's algorithm and the size of the numbers we will be able to factorize. In the second section, we will implement the Quantum Fourier Transform in a real quantum computer, and we will discuss the results obtained. We will study the noise of the current open-source IBM quantum computers.

IV.1 Shor's Algorithm implementation

As we have previously proven, there is a time-efficient way to factorize a number using a quantum computer. On the other hand, we do not know any polynomial time algorithm to factorize a number using a classical computer. In this chapter, we will implement Shor's algorithm and run it in three quantum computer simulators made by IBM Quantum using the Qiskit environment. A good and useful textbook and tutorial can be found in [6] and [7]. We have based our code in the implementation by Qiskit at [8]. This guide implements the algorithm using $n+4$ qubits. This means that, in order to implement the algorithm to factor the number 15 as the guide does, we would need 8 qubits. We have chosen a simulator instead of a real quantum computer because there is not any decent size open-source quantum computer available yet. In fact, none of them offer more than 7 qubits. A lot of progress need to be made in this matter. However, this will do the job as a proof of concept.

Simulators

Our goal will be to observe the different time performances of the three simulators and try to understand why they are the way they are.

The three simulator we will use are:

- **simulator_statevector**: This simulator computes the wavefunction of the qubit's statevector by simulating the quantum circuit, applying gates and instructions as they are encountered. It also supports general noise modeling, allowing for more realistic quantum simulations.

- **ibmq_qasm_simulator**: This versatile simulator designed was with the intention of building a general-purpose quantum circuit simulator. It can simulate quantum circuits both ideally and with noise modeling. The simulation method is automatically selected based on the input circuits and provided parameters, making it adaptable to different simulation scenarios.

- **simulator_mps**: This simulator works by employing a Matrix Product State (MPS) representation for the state, making it particularly efficient for states with weak entanglement. Tensor-network techniques are used for simulations, allowing for faster computations in scenarios where entanglement is limited.

Code and results

In this section we will show the code used to implement Shor's algorithm and the results obtained. The code is written in Python and follows the code given in [5]. It is also based on the implementation provided in [8].

Code

The final version of the code can be found in the following link: `https://github.com/JaviLobillo/TFG_Quantum_Complexity/blob/main/Shor.py`. The license for the whole repository is the MIT License.

As mentioned before, the code uses the Qiskit environment. Following the code given in the Qiskit documentation, we have implemented Shor's algorithm as well as auxiliary functions to run the algorithm in the different simulators for a given number of times n. Moreover, we have carefully used the $time$ library to measure the time it takes to run the algorithm in each simulator.

We have measured time in two different ways. First, we have measured the "real" time it takes for the algorithm to run. This time, called *TIME* in the code, is the time that the computer takes to run the classical part of the algorithm, plus the time that the simulator takes to run the quantum part of the algorithm. Second, we have measured what we called the *grossTIME* of the algorithm. This time is the time it takes for the algorithm to run once, which includes *TIME* plus the time it takes for the circuit to get to the simulator, the time the circuit is on hold, and the time the circuit takes to get back to the computer.

One may think that the only one of these times that is relevant is the *TIME* of the algorithm, as it is the time that the algorithm takes to run, purely speaking. However, we have decided to measure the *grossTIME* as well because it is the time that the user of the simulator will experience. This is important because, as we will see, the time that the circuit is on hold is not negligible. This is due to the fact that the simulators are shared by all the users, and hence the circuit has to wait in a queue until it is its turn to be run.

This is, in fact, the current state of the art of quantum computing: the computers are not individual computers, but shared computers, on the contrary of what happens with classical computers. We are used to have our own classical computer - in fact many of them - but we currently do not have our own quantum computer and, more importantly, we are not expected to have one in the near future.

Results

We have chosen to run the algorithm to factorize 15 for a number of times $n = 100$. We have chosen this number because it is big enough to get a good idea of the time performance of the simulators, but small enough to not take too long to run. The results obtained are shown in the table IV.1.

	TIME	grossTIME
simulator_statevector	9.021681609153747	9.820292329788208
simulator_mps	10.217290041446686	10.602933163642883
ibm_qasm_simulator	10.479080560207366	11.271872143745423

Table IV.1: Time performance of the simulators.

All the results are in seconds. As well as in this table, the code is prepared as to print the results in several $.txt$ files, which can be found in the same Github repository as the code.

We have conducted an in-depth analysis of the code and the obtained results, aiming to identify the time spent in each part of the program. To achieve this, we extensively utilized the $time$ library, printing the execution time for each segment of the code. As expected, we observed variations in the runtime for different simulators, depending on their construction and underlying algorithms.

Notably, we noticed that a significant amount of time is dedicated to processing the job results received from the backend, which constitutes part of the classical code. On average, this processing time hovers around 8 seconds. This observation indicates that there may be room for optimization in the data structures used in this part of the code. Future improvements in data structure design and implementation have the potential to substantially enhance the algorithm's time performance.

As observed, the different simulators do not exhibit the same time performance. The simulator_statevector proves to be the fastest, followed by the simulator_mps and then the ibm_qasm_simulator. Notably, this discrepancy in performance holds consistently across various backends, as evident from the "pure" time taken by the algorithm.

This variation in time efficiency is independent of backend load, indicating that the simulators themselves possess inherent differences in their computational capabilities and efficiency. The simulator_statevector's high speed can be attributed to its ability to compute the wavefunction of the qubit's statevector, making it particularly efficient for certain quantum circuit simulations.

On the other hand, the simulator_mps leverages a Matrix Product State representation, which provides advantages for states with weak entanglement, resulting in a competitive performance. Meanwhile, the ibm_qasm_simulator offers general-purpose simulation, but its computational speed may be influenced by the complexity and type of circuits being processed.

By understanding these differences in time performance, we can choose the most suitable simulator for specific quantum computation tasks and optimize the algorithm's overall efficiency for various scenarios.

In order to understand the results, we have to take into account the nature of Shor's algorithm itself. As we have studied before, Shor's algorithm is a probabilistic algorithm. This means that the algorithm will not always succeed in finding the factors of the number we want to factorize. However, the probability of success is high enough to make the algorithm useful. During the execution of the algorithm, the iterations not always succeeded in the first attempt. We can see in the table IV.2 how many attempts were needed to factorize 15 in each simulator.

Attempts	1	2	3	4	5
simulator_statevector	69	21	9	1	0
simulator_mps	71	25	4	0	0
ibm_qasm_simulator	68	25	6	0	1

Table IV.2: Number of attempts needed to factorize 15 in each simulator.

This is important because, whenever we measure the time an iteration took to factorize 15, we are counting the time each attempt took. Otherwise, if we only measured the "good" attempt, we would be, in a certain way, cheating. In this way we can see that around 70% of the iterations succeeded in the first attempt, and around 90% of the iterations succeeded in the first two attempts. This means that the algorithm is, not only theoretically, but also practically, a probabilistic algorithm with a very high probability of success. We can see that the number of attempts needed to factorize 15 is similar in all the simulators, which is a good sign of the reliability of the algorithm and backend implementations. Therefore, when we take into account the average time that each iteration took, we should take into account that around 30% of the iterations took more than one attempt.

By identifying and addressing the time-intensive parts of the code, we can strive for better overall performance and efficiency in quantum simulations. This analysis provides valuable insights into the computational bottlenecks and areas for potential optimization in the quantum simulator, leading to further advancements in quantum computing technology.

IV.2 QFT implementation

In this last section we show our implementation of the Quantum Fourier Transform. We have used the Qiskit environment as well. We have chosen to implement the QFT as a proof of concept of the quantum computing capabilities of the Qiskit environment because we do not need as many qubits as in Shor's algorithm. This allows us to use real quantum computers instead of simulators, which is a great advantage, specially as it allows us to study the noise in the quantum computers.

In order to study correctly our implementation of the QFT, we have decided to apply it to a quantum state in which all the qubits have had a Hadamard gate applied to them so we get a uniform superposition of the qubits. We have done so because we know that the result of applying the QFT to this state and then measuring it should be equal to 0. As noise will change the probability of measuring 0, we can study our implementation in both a quantum simulator with no noise and a real quantum computer so we can compare the obtained results.

For this, we have decided to execute both quantum circuits in five different backends. The first backend is the *simulator_statevector*, which we used before for Shor's algorithm. The four following backends are *ibm_lagos*, *ibm_nairobi*, *ibm_perth* and *ibmq_jakarta* which real quantum computer with 7 qubits. These are all the open-source available quantum computers with 7 qubits made by IBM. We have chosen to use 7 qubits so we can have as many examples as possible in order to study noise. There were not any larger quantum computers by IBM that we could access in the moment of writing this work.

Code and circuit design

All code can be found in the same Github repository as the code for Shor's algorithm. The code, which is completely original, shows two implementations for the QFT, as studied in the theoretical background in the subsection in page 36. A scheme for the implementations is actually the figure III.2. Both implementations can be adjusted to a desired number of qubits, n. The first implementation is the most generic one, following step by step the first definition of the QFT, as previously explained in this work. One example for five qubits is shown in figure IV.1.

The second implementation is a more optimized version, which takes advantage of the fact that the QFT does not really make use of all the smaller rotations. For this reason, this second implementation only uses rotations up to a given number m of desired rotations. An example for 5 qubits and 2 rotations at most is shown in figure IV.2.

Both implementations save the results as $.png$ files, which can be found in the same Github repository as the code.

One can see that both implementations use some SWAP gates at the end. This is because the resulting qubits are in reversed order (as can be seen in figure III.2), so we need to reverse the order back. Once we apply the circuit without the SWAP gates, we can see the resulting qubits as the factors which appear in the equation (III.2). However, as one can see throughout the subsection in page 36, and more clearly in the figure III.2, these factors are in reversed order, so we need to reverse the order back.

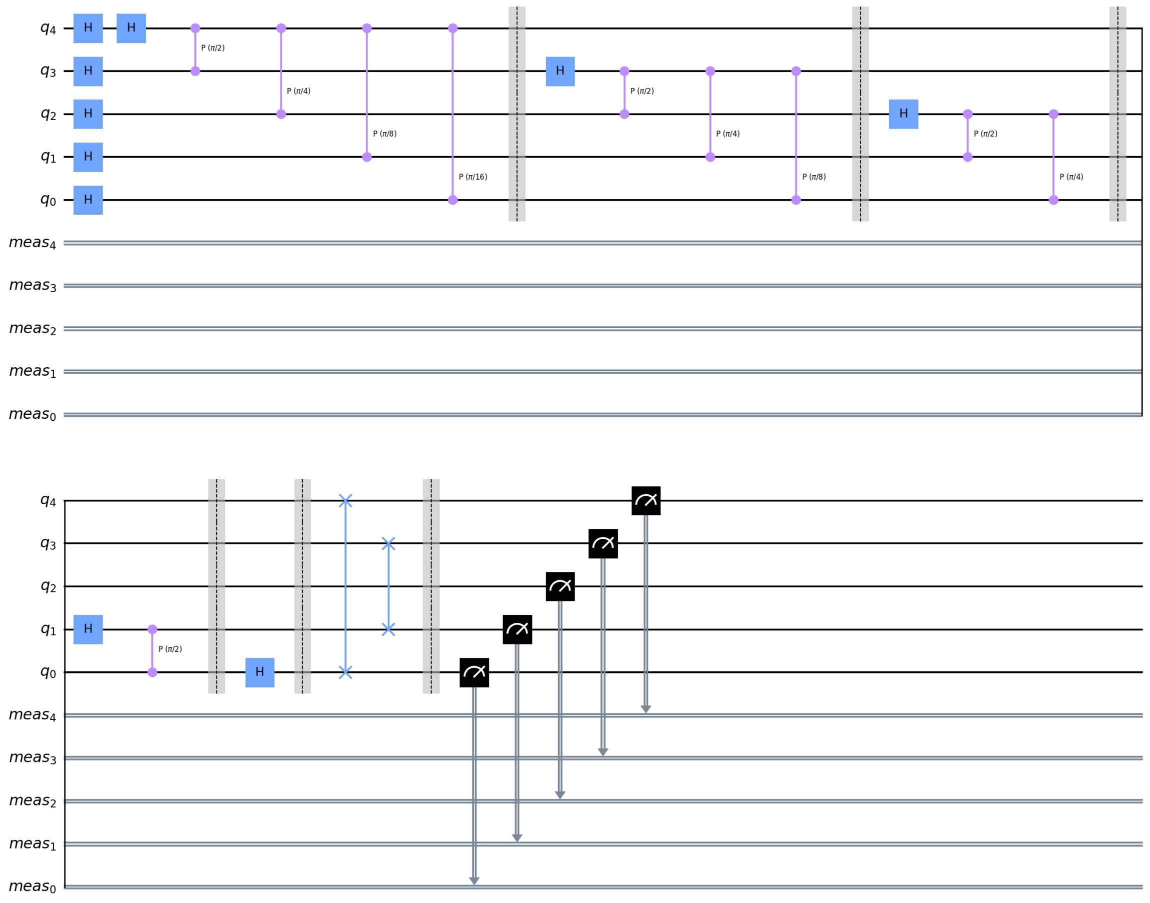

Figure IV.1: Generic QFT for 5 qubits.

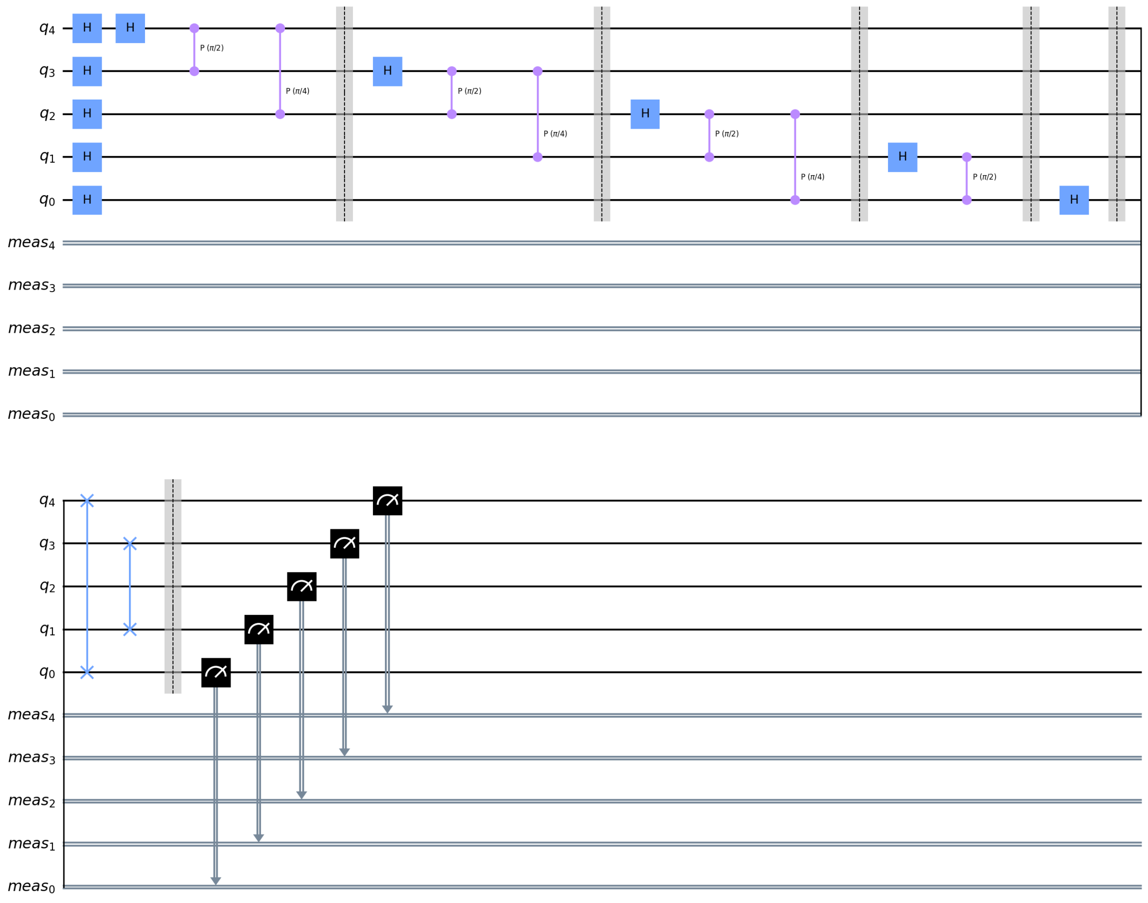

Figure IV.2: Bounded QFT for 5 qubits and bound 2.

Executions and results

Once we have the code, we need to decide the number of qubits we want to use, as well as the bound for the rotations in the second implementation. However, our aim is to study how the noise affects the QFT, so we have decided to take a statistical approach in order to study this phenomenon. We have decided to execute the QFT for n qubits, with n ranging from 3 to 7, both included. Also, we have chosen the bound for the rotations in the second implementation to be $1 \leq m \leq n-2$. This is because if the bound is $n-1$ or higher, the bounded circuit will end up being the generic circuit. Each execution was repeated 8000 times, and the results were stored in different diagrams, as mentioned before.

We can observe, as expected, that the execution on the simulator is perfect in the sense that it shows no error at all. All the runs in this backend provide the result 0, which is the expected one. Because of this, we will only show the results obtained for the generic QFT for 3 qubits in the simulator, as they are exactly the same as the results obtained for the other number of qubits. This is shown in figure IV.3.

However, when we work with the real quantum computer, we start to see how the noise affects the results. Two examples of histograms of this type are the figures IV.4 and IV.5.

In order to show how noise affects in all studied cases, we present the following table with the results obtained for each possible QFT implementation and each quantum backend. In the columns we present what percentage of times the result 0 was obtained and what position does the result 0 occupy in the probability distribution of measurement results. In the rows we present the different backends used, as well as the number of qubits used and the bound for the bounded QFT.

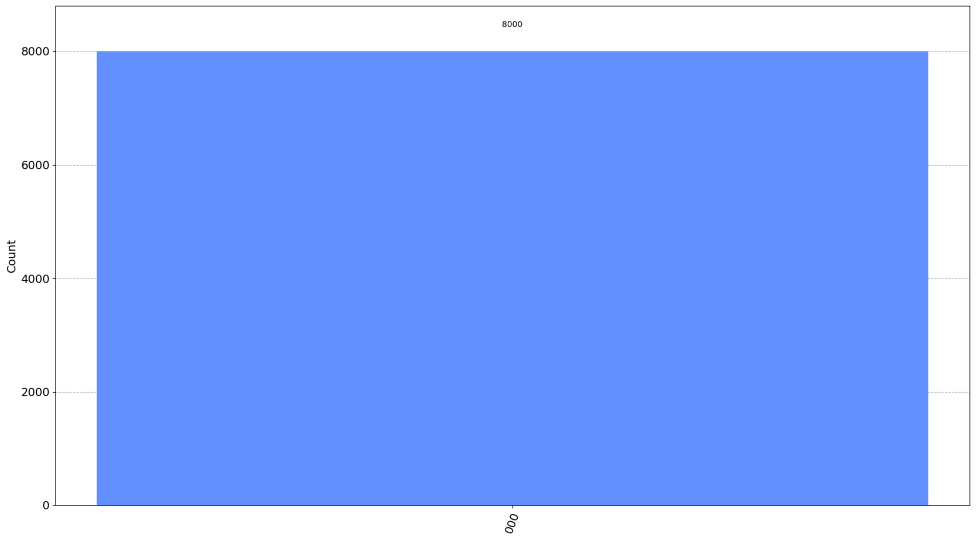

Figure IV.3: **Simulator_Statevector:** generic QFT for 3 qubits in the simulator.

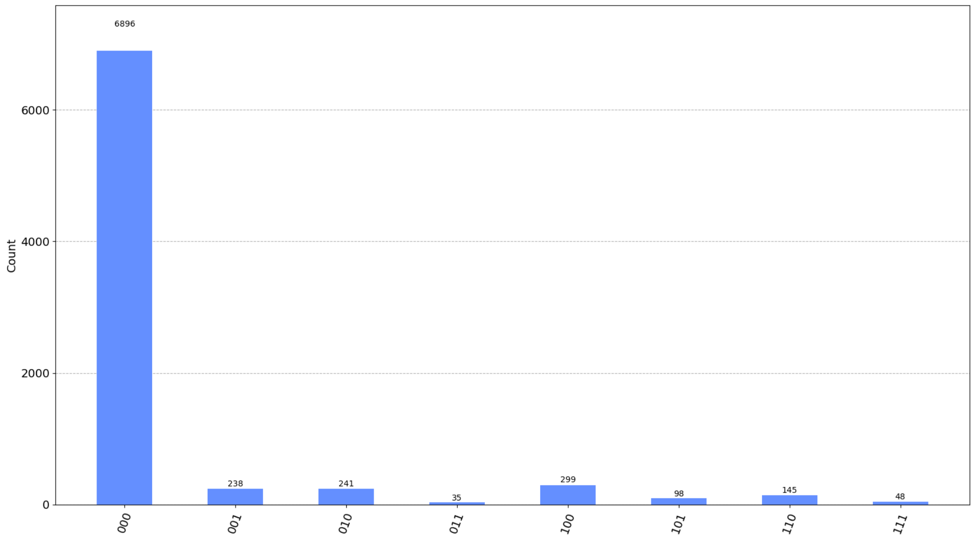

Figure IV.4: **IBMQ_Jakarta:** generic QFT for 3 qubits in IBMQ_Jakarta.

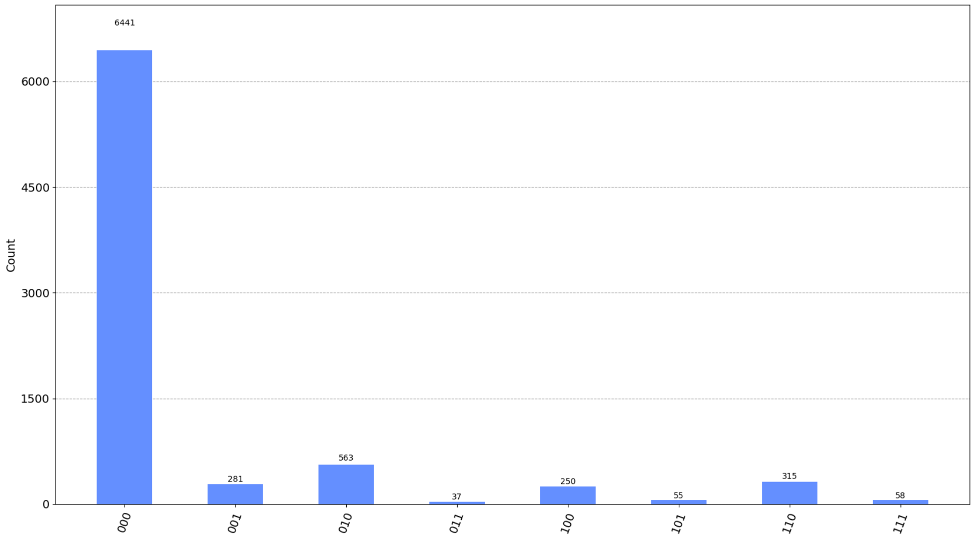

Figure IV.5: **IBMQ_Nairobi:** generic QFT for 3 qubits in IBMQ_Nairobi.

Backend	Percentage	Position
$Q = 3$		
IBM_Lagos	86.58%	1
IBM_Nairobi	80.51%	1
IBM_Perth	79.21%	1
IBMQ_Jakarta	86.2%	1
$Q = 3, B = 1$		
IBM_Lagos	90.25%	1
IBM_Nairobi	85.96%	1
IBM_Perth	84.39%	1
IBMQ_Jakarta	90.74%	1
$Q = 4$		
IBM_Lagos	48.21%	1
IBM_Nairobi	48.59%	1
IBM_Perth	45.01%	1
IBMQ_Jakarta	58.18%	1
$Q = 4, B = 1$		
IBM_Lagos	81.84%	1
IBM_Nairobi	64.15%	1
IBM_Perth	66.36%	1
IBMQ_Jakarta	83.04%	1
$Q = 4, B = 2$		
IBM_Lagos	71.46%	1
IBM_Nairobi	39.49%	1
IBM_Perth	52.96%	1
IBMQ_Jakarta	67.65%	1
$Q = 5$		
IBM_Lagos	11.35%	3
IBM_Nairobi	8.55%	4
IBM_Perth	21.6%	2
IBMQ_Jakarta	32.05%	1
$Q = 5, B = 1$		
IBM_Lagos	61.79%	1
IBM_Nairobi	69.5%	1
IBM_Perth	47.56%	1
IBMQ_Jakarta	68.0%	1
$Q = 5, B = 2$		
IBM_Lagos	25.28%	1
IBM_Nairobi	34.56%	1
IBM_Perth	24.68%	1
IBMQ_Jakarta	41.1%	1

Backend	Percentage	Position
$Q = 5, B = 3$		
IBM_Lagos	6.2%	5
IBM_Nairobi	8.74%	4
IBM_Perth	8.66%	3
IBMQ_Jakarta	30.4%	1
$Q = 6$		
IBM_Lagos	2.8%	11
IBM_Nairobi	2.69%	14
IBM_Perth	2.33%	17
IBMQ_Jakarta	5.96%	2
$Q = 6, B = 1$		
IBM_Lagos	20.66%	2
IBM_Nairobi	42.91%	1
IBM_Perth	28.94%	1
IBMQ_Jakarta	51.65%	1
$Q = 6, B = 2$		
IBM_Lagos	10.73%	2
IBM_Nairobi	6.73%	4
IBM_Perth	6.53%	4
IBMQ_Jakarta	18.39%	1
$Q = 6, B = 3$		
IBM_Lagos	2.03%	25
IBM_Nairobi	9.65%	2
IBM_Perth	4.96%	2
IBMQ_Jakarta	7.24%	2
$Q = 6, B = 4$		
IBM_Lagos	2.23%	16
IBM_Nairobi	3.18%	10
IBM_Perth	2.91%	6
IBMQ_Jakarta	9.35%	2
$Q = 7$		
IBM_Lagos	1.24%	31
IBM_Nairobi	2.7%	1
IBM_Perth	1.74%	6
IBMQ_Jakarta	1.43%	27
$Q = 7, B = 1$		
IBM_Lagos	7.38%	4
IBM_Nairobi	22.51%	1
IBM_Perth	19.94%	1
IBMQ_Jakarta	40.19%	1

Backend	Percentage	Position
$Q = 7, B = 2$		
IBM_Lagos	3.25%	9
IBM_Nairobi	5.23%	4
IBM_Perth	2.21%	10
IBMQ_Jakarta	7.53%	2
$Q = 7, B = 3$		
IBM_Lagos	1.95%	15
IBM_Nairobi	7.29%	1
IBM_Perth	2.93%	7
IBMQ_Jakarta	4.3%	5
$Q = 7, B = 4$		
IBM_Lagos	1.1%	37
IBM_Nairobi	3.33%	2
IBM_Perth	2.46%	7
IBMQ_Jakarta	2.24%	7
$Q = 7, B = 5$		
IBM_Lagos	1.46%	18
IBM_Nairobi	2.34%	4
IBM_Perth	2.19%	2
IBMQ_Jakarta	1.89%	15

Table IV.3: Results for the different qubits, bound and backend choices.

All results can be seen in the following link: `https://github.com/JaviLobillo/TFG_Quantum_Complexity/tree/main/Results`

Nonetheless we can interpret the results obtained in a more general way. All main points in this regard are:

- First and most important, we can see that the noise is so high right now that it is not useful to work with real quantum computers yet. Many advancements in the building of quantum computing are coming and they will be welcome in order to change the current statu quo.

- IBMQ_Jakarta seems to be the most accurate backend for most of the cases, although it is still not even near to be accurate enough to be useful. However, the fact that it is the most accurate seems somehow counter-intuitive because, although all four quantum backends share the same Falcon r5.11H type architecture, IBMQ_Jakarta has the least quantum value with a value of 16, compared to the other's value of 32. These, and other parameters, can be seen in Table IV.2. This would mean that IBMQ_Jakarta should be less accurate, and we see it is not the case. However, when we work with such inaccurate machines, the fact that one has more or less accuracy can be difficult to attribute to a particular reason, beyond general lack of accuracy. Furthermore, these performance measures lose reliability when we consider, for example, the fact that the quantum computers need to be continuously calibrated, so the larger the time from last calibration induces larger accuracy loss. Also, not every circuit uses the resources in the same way, so accuracy parameters can vary from circuit to circuit (the QFT being just one of the possible ones).

- We can see that the more qubits we use, the more noise we get. This is because the more qubits we use, the more gates we need to apply, and hence the more noise we get. This problem becomes even more important as the right result is not even the most probable one for most of the 5 qubits (or more) implementations.
- Using the bounded QFT instead of the generic one reduces the noise significantly. This is because the bounded QFT uses less gates than the generic one, and hence the noise is reduced. This might be counter-intuitive, as theoretically this would be a less accurate implementation of the QFT. A bound value of 1 and 2 makes the 5 qubit QFT to accurate enough as to make the right result of 0 the most probable again for all backends.

Backend	IBM_Lagos	IBM_Perth	IBM_Nairobi	IBMQ_Jakarta
Processor type	Falcon r5.11H	Falcon r5.11H	Falcon r5.11H	Falcon r5.11H
Qubits	7	7	7	7
QV	32	32	32	16
CLOPS	2.7K	2.9K	2.6K	2.4K
Median T1	113.19 μs	174.07 μs	101.23 μs	124.11 μs
Median T2	60.61 μs	86.27 μs	62.52 μs	39.66 μs

Table IV.4: Parameters for the different backends.

The table contains the main physical parameters that differentiate the four quantum backends. QV means Quantum Value and CLOPS stands for Circuit Layer Operations Per Second. Median T1 and median T2 are also included. T1 is the *relaxation time*, the time it takes to decay and lose its amplitude value. T2 is the *dephasing time*, the time it takes to lose the phase coherence. As we mentioned before, one could think that IBMQ_Jakarta ought to be the worst performing backend, and experimental results tell us otherwise.

Results and conclusions

This Bachelor's thesis has presented itself as an opportunity to better understand the new world which has been presented to us in the last few years, that is, the world of quantum computing. The best way we can try to understand this world is by first having a background research on its theoretical bases and then try to work in a more practical way with what it can offer. And that is exactly how this work was conducted.

The first chapter was dedicated to the theoretical background of quantum computing. We have seen how quantum mechanics can be used to perform computations in a different way than the classical one. We have seen as well how the qubit is the quantum equivalent of the classical bit, and how the quantum gates are the quantum equivalent of the classical gates. Finally we have seen how the quantum gates can be represented by unitary matrices. This approach has provided a mental structure to understand the quantum circuits and algorithms that we would study later on. This mental structure was necessary as the beginner quantum computing student is not expected to think in terms of this structure as fluidly as he does in terms of classical computing.

In the second chapter we provided the basis for understanding how classical and quantum circuits work. This base is critical as all complexity theory that came in the third chapter is based on circuit design and size rather than time complexity. We got to categorize the complexity of a problem in terms of the complexity of the circuits that solve it, both in the classical and quantum case. We have related the complexity of a problem to the complexity of the circuits that solve it. Also we have seen how we can relate the depth of a circuit to the time complexity of the problem it solves. We got to explicitly differenciate easy problems from hard problems, which is crucial to understand the importance of quantum computing. Finally we got to relate quantum and classical complexity classes.

In the third chapter we have seen the main results of quantum computing. We have seen how we can use quantum circuits to solve problems in constant time that would take linear time to solve classically, although as both cases are considered to be easy, this was only to introduce Simon's and Shor's algorithms. Shor's algorithm, which uses the quantum Fourier transform, is the most important quantum algorithm to date, as it provides a polynomial time algorithm to factorize numbers, which is a problem that is believed to be hard classically. If the factoring problem is confirmed to be classically hard, this would mean that we have found a problem that can be solved in polynomial time using a quantum computer but not using a classical one.

We have seen that the whole structure of the work leads us to understand one of the reasons why quantum computing is so important, which is that it can solve problems that are believed to be hard classically in polynomial time. The whole thesis works constructively, giving all the bricks necessary, in order to build the understanding of the reader to the point where he can understand how important Shor's algorithm and quantum computing is.

Chapter four of the project focuses on two distinct proofs of concept. The first proof of concept involves the implementation of Shor's algorithm within a quantum simulator. While this approach offers the advantage of testing the main algorithm discussed in this work, it's worth noting that a simulation does not reflect the algorithm's true efficiency. The second proof of concept centers around the implementation of the quantum Fourier transform, a crucial component of Shor's algorithm. Unlike the first implementation, this one is executed on real quantum computers. Its primary purpose is to analyze how noise influences the results. Consequently, this second approach offers valuable insights into the current state of quantum computing technology.

Bibliography

[1] S. Aaronson. *Introduction to Quantum Information Science Lecture Notes*, 2018. Online PDF at `https://www.scottaaronson.com/qclec.pdf`.

[2] E. Bernstein and U. Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.

[3] D. Deutsch and R. Jozsa. Rapid solutions of problems by quantum computation. *Royal Society*, 439(1907), 1992.

[4] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[5] J. Preskill. *Lecture Notes for Physics 219: Quantum Computation*, 1997-2022. Online PDF at `http://theory.caltech.edu/~preskill/ph219/index.html#lecture`.

[6] Qiskit. *Qiskit Textbook*, 2023. `https://qiskit.org/learn`.

[7] Qiskit. *Qiskit Tutorials*, 2023. `https://qiskit.org/documentation/tutorials.html`.

[8] Qiskit. *Shor's Algorithm*, 2023. `https://learn.qiskit.org/course/ch-algorithms/shors-algorithm`.

[9] P. W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.*, 26:1484, 1997.

[10] D. R. Simon. On the power of quantum computation. *SIAM Journal on Computing*, 26(5):1474–1483, 1997.