Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

On the Hidden Subgroup Problem as a Pivot in Quantum Complexity Theory

Relatore:
Chiar.mo Prof.
UGO DAL LAGO

Presentata da:
ANDREA COLLEDAN

Sessione I
Anno Accademico 2017-2018

Contents

Introduction

It is in the mid-1990s that some of the most notorious quantum algorithms make their first appearance in the research community. In 1995, Peter Shor presents two game-changing quantum algorithms: one solves the integer factorization problem, the other the discrete logarithm problem. Both do so in polynomial time, i.e. exponentially faster than any of the best known classical algorithms [12]. One year later, Lov Kumar Grover presents a quantum algorithm for searching databases that is quadratically faster than any possible classical algorithm for the same purpose [6].

Shor's work, in particular, deals effectively with two well-known *intractable* computational problems and hints at the possibility of using quantum computers to tackle more of those problems that, although very valuable, cannot be reasonably solved by classical machines. However, because quantum algorithms are significantly harder than classical algorithms to design, breakthroughs such as Shor's are very rare to come by. This is where the idea of finding a single abstract problem that generalizes a larger class of computationally hard problems starts being of great interest and this is where the *hidden subgroup problem* comes into play.

It is not uncommon, in computer science, for a particular problem to gain importance and notoriety not thanks to its inherent practical value, but rather for being an exceptional representative of a larger class of attractive problems. This is the case with the hidden subgroup problem (also labeled HSP), an abstract problem of group theory that happens to be an excellent generalization of *exactly* those problems that quantum computers can solve exponentially faster than their classical counterparts.

In its purest form, the hidden subgroup problem consist in, given a group, determining which ones of its subgroups give birth to cosets according to a certain function f, which can be consulted like an oracle. The abstract nature of this problem allows it to generalize numerous aspects of, among others, *order finding* and *period finding* problems [11]. Furthermore, many cases of the HSP are already known to allow efficient quantum solutions, a fact that makes the HSP a perfect choice of representative for many of the intractable problems that we are interested in.

A Top-down Approach

Most of the literature on the hidden subgroup problem approaches the topic in a *bottom-up* fashion. That is, first the principles that rule quantum mechanics and quantum computing are introduced, then they are used to give solutions to a number of problems. Finally, these problems are generalized to the HSP.

In this thesis, we attempt to partially reverse this approach by giving a *top-down* perspective on the role of the HSP in quantum complexity theory. First we present the problem in its pure, mathematical form. Next, we give a computational definition of the same problem and we show how an algorithm that solves it can be used to solve other, more interesting problems. After that, we introduce those concepts of quantum computing that are strictly necessary to understand some of the quantum algorithms that efficiently solve the HSP on specific group families. Lastly, we discuss these algorithms.

More in detail, this thesis is structured as follows:

- In Chapter 1 we discuss the hidden subgroup problem *per se*. First, we introduce the essential group-theoretical concepts necessary to understand the problem at hand, of which we give a formal, mathematical definition. We then shift our attention to the computational details of the HSP, to define more rigorously what it means to *compute* a solution to it. It is in this chapter that we define the concept of *HSP functional*.
- In Chapter 2 we introduce some essential concepts of complexity theory and reducibility. We then use these concepts to discuss the pivotal role of the HSP in the context of quantum complexity theory. We also go out of our way to concretely show how two significant computational problems can be reduced to the HSP.
- Those readers who are only acquainted with classical computing may perceive the subject of *quantum computation* as equally fascinating and daunting. In preparation for the quantum algorithms that are shown in Chapter 4, Chapter 3 presents a brief introduction on the principles of quantum mechanics and their application to computer science.
- Lastly, in Chapter 4 we examine how the *quantum Fourier transform* can be used to implement quantum algorithms that efficiently solve the HSP on specific types of groups, namely Abelian groups. As a consequence, this algorithms allow us to efficiently solve the problems that we introduced in Chapter 2.

The idea behind this order of exposition is that of allowing the target reader, i.e. the computer scientist with little to no knowledge of things quantum, to get acquainted with the hidden subgroup problem and to be convinced of the importance of its role in complexity theory, without necessarily having to deal with the intricacies of quantum computing. However, we *do* hope that this thesis will stimulate the reader into further pursuing the study of this fascinating subject.

Chapter 1

The Hidden Subgroup Problem

1.1 A Mathematical Approach

The first step in our top-down approach is, of course, to provide a definition of the problem at hand and to get acquainted with it. To do so, it is first advisable to review a number of concepts of group theory, starting with the very definition of *group*.

Definition 1.1.1 (Group)**.** *Let* $\mathbb{G}$ *be a set and* $\circ$ *a binary operation on the elements of* $\mathbb{G}$*. The tuple* $(\mathbb{G}, \circ)$ *constitutes a* group *when:*

- $\mathbb{G}$ *is closed under* $\circ$*, i.e.* $\forall g_1, g_2 \in \mathbb{G} : g_1 \circ g_2 \in \mathbb{G}$*.*
- *There exists an* identity element $e \in \mathbb{G}$ *such that* $\forall g \in \mathbb{G} : g \circ e = g = e \circ g$*.*
- *For every element* $g \in \mathbb{G}$ *there exists an* inverse element*, i.e. an element* $g^{-1} \in \mathbb{G}$ *such that* $g \circ g^{-1} = e = g^{-1} \circ g$*.*

Example 1.1.1: A simple example of a group is $(\mathbb{Z}, +)$, the set of all integers under addition. This is a group because the sum of any two integers is itself an integer, there exists an identity element ($0 \in \mathbb{Z}$) and for all $n \in \mathbb{Z}$, there exists $(-n) \in \mathbb{Z}$ such that $n + (-n) = n - n = 0$.

Example 1.1.2: On the other hand, the tuple $(\mathbb{Z}, \times)$, although similar, does *not* constitute a group: $\mathbb{Z}$ is closed under multiplication and there exists an identity ($1 \in \mathbb{Z}$), but in general the inverse of an $n \in \mathbb{Z}$ (i.e. $1/n$) does not belong to $\mathbb{Z}$.

From now on we will often refer to a group of the form $(\mathbb{G}, \circ)$ as simply $\mathbb{G}$, specifying the underlying operation only when strictly necessary. Furthermore, we will work mostly with *additive* and *multiplicative* groups (groups in which the operation is addition or multiplication, respectively). When working with additive groups we will write $g_1 + g_2$

instead of $g_1 \circ g_2$ and $-g$ instead of g^{-1}. Similarly, when working with multiplicative groups, we will write $g_1 \cdot g_2$ (or simply $g_1 g_2$) and g^{-1}.

If a subset of the elements of $\mathbb{G}$ exhibits itself group structure under $\mathbb{G}$'s operation, then such a subset is called a *subgroup* of $\mathbb{G}$.

Definition 1.1.2 (Subgroup)**.** *Let $\mathbb{H}$ be a subset of the elements of a group $(\mathbb{G}, \circ)$. $\mathbb{H}$ is a* subgroup *of $\mathbb{G}$ (and we write $\mathbb{H} \leq \mathbb{G}$) when:*

- *$\mathbb{H}$ is closed under $\mathbb{G}$'s operation, i.e. $\forall h_1, h_2 \in \mathbb{H} : h_1 \circ h_2 \in \mathbb{H}$.*
- *The identity element of $\mathbb{G}$ is in $\mathbb{H}$.*
- *For every element $h \in \mathbb{H}$, the inverse element h^{-1} is also in $\mathbb{H}$.*

Example 1.1.3: Let us consider the group $\mathbb{G} = (\mathbb{Z}, +)$ from the previous example. The set $\mathbb{H} = \{2n | n \in \mathbb{Z}\}$ of even integers is a subgroup of $\mathbb{G}$, as it is closed under addition (the sum of any two even numbers is even), contains the identity element 0 and the inverse of an even number is even as well.

Example 1.1.4: On the other hand, the subset of odd integers $\mathbb{K} = \{2n+1 | n \in \mathbb{Z}\}$ does *not* form a group (because $0 \notin \mathbb{K}$) and neither does the subset $\mathbb{Z}^+$ of positive integers (no positive number has a positive inverse).

Both groups and subgroups can be described in terms of a subset of their elements, from which we can obtain all the other elements via the group operation. Such a subset is called a *generating set* for the group.

Definition 1.1.3 (Generating set of a group)**.** *Let $\mathbb{G}$ be a group and $S = \{s_1, s_2, \ldots, s_n\}$ a subset of its elements. We denote by $\langle S \rangle$ the smallest subgroup of $\mathbb{G}$ that contains all the elements of S. We say that S is a* generating set *for $\mathbb{G}$ when $\langle S \rangle = \mathbb{G}$.*

Example 1.1.5: Consider $\mathbb{G}$ and $\mathbb{H}$ from the previous example. We have that $\mathbb{G} = \langle 1 \rangle$, as we can easily express any integer n with $|n|$ sums of 1 or -1. It is also evident, by its very definition, that $\mathbb{H} = \langle 2 \rangle$.

By applying $\circ$ between elements of $\mathbb{H}$ and generic elements of $\mathbb{G}$ we obtain elements that do not necessarily belong to $\mathbb{H}$. In particular, if we apply $\circ$ between *all* the elements of $\mathbb{H}$ and *one* element of $\mathbb{G}$, we obtain what is called a *coset*.

Definition 1.1.4 (Coset)**.** *Let $\mathbb{G}$ be a group and let $\mathbb{H} \leq \mathbb{G}$. For every $g \in \mathbb{G}$ we can define:*

- $g\mathbb{H} = \{g \circ h | h \in \mathbb{H}\}$, *or the* left coset of $\mathbb{H}$ with respect to g.
- $\mathbb{H}g = \{h \circ g | h \in \mathbb{H}\}$, *or the* right coset of $\mathbb{H}$ with respect to g.

For $h \in \mathbb{H}$, we have $h\mathbb{H} = \mathbb{H} = \mathbb{H}h$, so $\mathbb{H}$ is one of its own cosets. In general, however, the left and right cosets are different. When the left and right cosets coincide for all $g \in \mathbb{G}$ then $\mathbb{H}$ is said to be a *normal subgroup*. Of course, this is always the case when $\mathbb{G}$'s operation is commutative. In this case, $\mathbb{G}$ is said to be *Abelian*.

Definition 1.1.5 (Abelian group)**.** *A group $\mathbb{G}$ is an* Abelian group *when $\forall g_1, g_2 \in \mathbb{G}$: $g_1 \circ g_2 = g_2 \circ g_1$. That is, when $\circ$ is commutative.*

Example 1.1.6: Consider again $\mathbb{G}$ and $\mathbb{H}$ from Example 1.1.3. First, we note that $\mathbb{G}$ is Abelian, as addition is commutative. Next, we see that $\mathbb{H}$ only has one coset, that is $1\mathbb{H} = \mathbb{H}1 = \{2n + 1 | n \in \mathbb{Z}\}$, the set of odd numbers.

We now have enough background to introduce the crucial concept of *coset-separating function*, which lies at the very heart of the hidden subgroup problem.

Definition 1.1.6 (Coset-separating function)**.** *Let $\mathbb{G}$ be a group and let $\mathbb{H} \leq \mathbb{G}$. Let $f : \mathbb{G} \to \mathbb{S}$ be a function from the elements of $\mathbb{G}$ to finite-length binary strings. We say that f* separates cosets *for $\mathbb{H}$ when:*

$$\forall g_1, g_2 \in \mathbb{G} : f(g_1) = f(g_2) \iff g_1\mathbb{H} = g_2\mathbb{H}.$$

In other words, f separates cosets when it is constant on the individual cosets of $\mathbb{H}$ and different between different cosets. Essentially, f *identifies* the cosets of $\mathbb{H}$ by labeling them with distinct strings of bits. We are now ready to introduce the hidden subgroup problem:

Definition 1.1.7 (Hidden Subgroup Problem)**.** *Let $\mathbb{G}$ be a group and $\mathbb{H} \leq \mathbb{G}$ an unknown subgroup. Let $f : \mathbb{G} \to \mathbb{S}$ be a coset-separating function for $\mathbb{H}$. The* Hidden Subgroup Problem *(HSP) consists in finding a generating set for $\mathbb{H}$ by using the information provided by f.*

In practice, we will only work with finite groups. Consider a family $\{\mathbb{G}_N\}_{N \in \mathbb{I}}$, where $\mathbb{I} \subseteq \mathbb{N}$ is a set of indices. The members of such a family are groups with similar operations, whose elements come from the same superset. What distinguishes them is their order. Namely, for all $\mathbb{G}_N \in \{\mathbb{G}_N\}_{N \in \mathbb{I}}$ we want $|\mathbb{G}_N|$ to be a function of N.

Example 1.1.7: All the groups inside $\{\mathbb{Z}_N\}_{N \in \mathbb{N}}$ have elements in the non-negative integers $\mathbb{Z}^*$. Each $\mathbb{Z}_N \in \{\mathbb{Z}_N\}_{N \in \mathbb{N}}$ has addition modulo N as group operation and has order $|\mathbb{Z}_N| = N$.

We therefore define a variant of the HSP, generalized to families of groups:

Definition 1.1.8 (Parametrized HSP)**.** *Let $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$ be a family of groups. The HSP on $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$ consists in, given N and a function f_N that separates cosets for some $\mathbb{H}_N \leq \mathbb{G}_N$, finding a generating set for $\mathbb{H}_N$ by only using information obtained from evaluations of f_N.*

1.2 A Computational Approach

Up to this point we have built a mathematical definition of the HSP. For our purposes, however, we need a more rigorous definition of what it means to *solve* the HSP on a certain input. Namely, we want to define a functional for a group family $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$, which takes a specific N and some representation of a coset-separating function f as inputs and outputs some representation of a generating set for the desired subgroup. Essentially, we want to give a computational definition of the parametrized HSP that we defined in the previous section. To do so, we first need a way to effectively encode groups and functions.

1.2.1 Group Encoding

Suppose we are working with a group family $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$. Consider the total disjoint union $\mathcal{G}$ of the elements of this family:

$$\mathcal{G} = \bigsqcup_{N\in\mathbb{I}} \mathbb{G}_N.$$

This new set $\mathcal{G}$ consists of ordered couples of the form (g, N), such that $g \in \mathbb{G}_N$. Consider now a function $\rho : \mathcal{G} \to \mathbb{S} \times \mathbb{N}$ such that ρ is an injection from $\mathcal{G}$ to the pairs of finite-length binary strings and integers. We call such ρ an *encoding function* for the group family $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$. We want $\rho(g, N) = (b_g, N)$, so that every element g of every possible group in the group family can be represented as a binary string b_g. From now on, we will often refer to such b_g as $\rho(g, N)$ or simply $\rho(g)$, ignoring N in the process.

Definition 1.2.1 (Polylogarithmic function)**.** *A function $f : \mathbb{N} \to \mathbb{N}$ is said to be* polylogarithmic *if it consists of a polynomial in the logarithm of the input. That is, f is of the form:*

$$f(x) = a_0 + a_1\lfloor\log x\rfloor + a_2\lfloor\log x\rfloor^2 + \cdots + a_k\lfloor\log x\rfloor^k.$$

Definition 1.2.2 (Length-predictable function)**.** *Let $\{A_N\}_{N\in\mathbb{I}}$ be a family of sets and let $\mathcal{A}$ be their total disjoint union. A function $\rho : \mathcal{A} \to \mathbb{S}$ is said to be* length-predictable *(LP) if there exists $f : \mathbb{N} \to \mathbb{N}$ such that, for all $(a, N) \in \mathcal{A}$, we have $|\rho(a, N)| = f(N)$. If f is a polylogarithmic function, then we say that ρ is* polylogarithmically length-predictable *(PLP).*

If an encoding ρ is LP for some $f : \mathbb{N} \to \mathbb{N}$, then we can identify the elements of a group $\mathbb{G}_N$ with strings of exactly $f(N)$ bits. We now attempt to provide encoding functions for some of the group families we will work with.

Groups of Binary Strings Consider the group family $\{(\mathbb{B}^N, \oplus)\}_{N\in\mathbb{N}}$, whose groups consist of the binary strings of length N under addition modulo 2.

Lemma 1.2.1. *For all N, the set $\mathbb{B}^N$ of binary strings of length N forms a group of order 2^N under addition modulo 2.*

Proof. The set of binary strings of length N has exactly 2^N elements. For all $b_1, b_2 \in \mathbb{B}^N$, $b_1 \oplus b_2$ is also a binary string of length N. Furthermore, there exists in $\mathbb{B}^N$ the identity element $e = 0^N$ such that for all $b \in \mathbb{B}^N, b \oplus 0^N = b$. Lastly, every element $b \in \mathbb{B}^N$ is its own inverse, as $b \oplus b = 0^N$. □

In this case our encoding function ρ is trivially the identity function, as $\mathbb{B}^N$ is already a subset of $\mathbb{S}$ (its elements are fixed-length binary strings). It is obvious that the output of ρ grows linearly with N. As such, ρ is LP for $f(N) = N$.

Cyclic Additive Groups Before we say anything about the encoding of cyclic additive groups, let us provide a definition:

Definition 1.2.3 (Cyclic group)**.** *A group $\mathbb{G}$ is said to be* cyclic *if there exists an element $g \in \mathbb{G}$ such that $\langle g\rangle = \mathbb{G}$. That is, $\mathbb{G}$ can be generated by a single element, known as the* generator *of the group.*

Consider now $\{\mathbb{Z}_N\}_{N\in\mathbb{N}}$, the family of cyclic additive groups of integers modulo N. In this case, and for all N, our ρ is the "identity" function $\rho(z, N) = (\mathrm{bin}_N(z), N)$, where $\mathrm{bin}_N(x)$ is the zero-padded binary representation of an integer $x \leq N$. Note that for all such x, $\mathrm{bin}_N(x)$ is always a string of $\lfloor \log_2 N \rfloor + 1$ bits. Being the elements of $\mathbb{Z}_N$ exactly the integers from 0 to $N - 1$, we can say that this ρ is PLP.

Finite Abelian Groups Consider $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$, a family of generic Abelian groups. Let us introduce the concept of *direct sum* of groups:

Definition 1.2.4 (Direct sum of groups)**.** *Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two groups. The* direct sum *of $\mathbb{G}_1$ and $\mathbb{G}_2$ is the group $\mathbb{G}$ of pairs (g_1, g_2), where $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. We write $\mathbb{G} = \mathbb{G}_1 \oplus \mathbb{G}_2$.*

Of course, the direct sum of any two Abelian groups is also Abelian. We have the following result [8]:

Theorem 1.2.1. *Every finite Abelian group $\mathbb{G}_N$ is isomorphic to a direct sum of cyclic additive groups. That is, there exist $N_1, N_2, \ldots, N_k$ such that:*

$$\mathbb{G}_N \cong \mathbb{Z}_{N_1} \oplus \mathbb{Z}_{N_2} \oplus \cdots \oplus \mathbb{Z}_{N_k}.$$

As a result, we have that every element $g \in \mathbb{G}_N$ can be represented by a k-tuple $(z_1, z_2, \ldots, z_k)$ such that $z_i \in \mathbb{Z}_{N_i}$ for every $i = 1 \ldots k$. At this point, we can define ρ' on $\mathbb{Z}_{N_1} \oplus \cdots \oplus \mathbb{Z}_{N_k}$ as the concatenation of the outputs of the cyclic additive ρ on the elements of the tuple:

$$\rho'((z_1, z_2, \ldots, z_k), N) = (\mathrm{bin}_{N_1}(z_1) \| \ldots \| \mathrm{bin}_{N_k}(z_k), N).$$

Of course, N_i is smaller than N for every $i = 1 \ldots k$. Furthermore, we have an upper bound on k as well:

Lemma 1.2.2. *Let $\{\mathbb{G}_N\}_{N \in \mathbb{I}}$ be a family of groups an let $g : \{\mathbb{G}_N\}_{N \in \mathbb{I}} \to \mathbb{N}$ be a function such that, for all $\mathbb{G}_N \in \{\mathbb{G}_N\}_{N \in \mathbb{I}}$, $g(N) = |\mathbb{G}_N|$. Then if $\mathbb{G}_N \cong \mathbb{Z}_{N_1} \oplus \mathbb{Z}_{N_2} \oplus \cdots \oplus \mathbb{Z}_{N_k}$ we have that*

$$k \leq \lceil \log_2 g(N) \rceil.$$

Proof. If $\mathbb{G}_N$ and $\mathbb{Z}_{N_1} \oplus \mathbb{Z}_{N_2} \oplus \cdots \oplus \mathbb{Z}_{N_k}$ are isomorphic, then necessarily $g(N) = |\mathbb{G}_N| = |\mathbb{Z}_{N_1} \oplus \mathbb{Z}_{N_2} \oplus \cdots \oplus \mathbb{Z}_{N_k}| = N_1 N_2 \cdots N_k$. Of course, k is greatest (we have the greatest number of factors) when $N_1, N_2, \ldots, N_k$ are the prime factors of $g(N)$. For every k, the smallest number with k prime factors is 2^k. Therefore, any number n cannot have more prime factors than the smallest $2^k \geq n$, i.e. $2^{\lceil \log_2 n \rceil}$, which has exactly $\lceil \log_2 n \rceil$ prime factors. It follows that $g(N)$ has at most $\lceil \log_2 g(N) \rceil$ prime factors. □

As a result, our encoding ρ' is surely LP for some $f \in O(\log N(\log g(N)))$. Moreover, if g is a polynomial then $f \in O(\log^2 N)$ and ρ' is PLP.

Properties of the encoding One last property that we expect from our encoding functions is that it must be possible to efficiently compute common group operations on the binary representations they produce. In the case of groups of integers, these operations are addition, multiplication and exponentiation, and we know that polytime algorithms exist that can compute them on the binary representations produced by ρ [7]. Furthermore, we require that ρ allow efficient inversion.

1.2.2 Function Encoding

Once we have ρ that encodes the elements of $\mathbb{G}_N$ as finite-length binary strings, we have all the means necessary to turn any coset-separating function as given in Definition 1.1.6 into an equivalent coset-separating function that operates on binary strings. To define such a function, we first need to introduce the concept of *boolean circuit*.

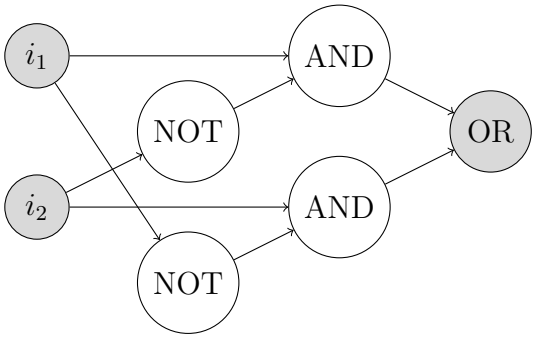


Figure 1.1: A simple example of a boolean circuit that computes addition modulo 2 (i.e. the *exclusive or* operation).

Definition 1.2.5 (Boolean circuit)**.** *A* boolean circuit C *of* n *inputs and* m *outputs is a directed acyclic graph in which:*

- *There are* n input nodes*, labeled* $i_1, i_2, \ldots, i_n$*, which have an in-degree of* 0*.*
- *The remaining nodes are labeled with either* AND*,* OR *or* NOT *and have in-degrees of* 2 *(*AND*,* OR*) or* 1 *(*NOT*).*
- *Among the non-input nodes, there are* m output nodes*, which have an out-degree of* 0*.*

Definition 1.2.6 (Function of a circuit)**.** *Let* C *be a boolean circuit of inputs* $i_1, i_2, \ldots, i_n$ *and outputs* $o_1, o_2, \ldots, o_m$*. We say that* C *computes a function* $f_C : \mathbb{B}^n \to \mathbb{B}^m$ *defined as*

$$f_C(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_m),$$

where $y_1, y_2, \ldots, y_m$ *are the boolean values produced by* $o_1, o_2, \ldots, o_n$ *when the input nodes are assigned values* $x_1, x_2, \ldots, x_n$*.*

Note that once we know how to represent a boolean circuit as a directed acyclic graph, it is also easy to encode it as a string. All we need to do is store every node, along with a label, and every edge as a couple of nodes. The size of such an encoding is fairly easy to estimate. Suppose we are given a boolean circuit as a graph $G_C = (V, E)$:

- **Nodes:** For each each node in V we need to store an identifier and a label. We can uniquely identify the nodes of G_C using the integers from 0 to $|V| - 1$. As a result, each identifier requires $O(\log |V|)$ bits to be stored. The label, on the other hand, requires constant space. The nodes can thus be stored using $|V| \cdot O(\log |V|) = O(|V| \log |V|)$ bits.

- **Edges:** For each of the $|E| = O(|V|^2)$ edges, we need to store the identifiers of the nodes involved. Each identifier requires $O(\log |V|)$ bits to be stored, so the total number of bits required is $O(|V|^2) \cdot O(\log |V|) = O(|V|^2 \log |V|)$.

Putting the two things together, we have that a a boolean circuit C with n gates can be represented by a string of $O(n^2 \log n) + O(n \log n) = O(n^2 \log n)$ bits. We will refer to such a string as $\lceil C \rceil$.

Now that we have the notion of a circuit that computes a function from binary strings to binary strings, we can bring in our ρ to define the notion of *coset-separating circuit*.

Definition 1.2.7 (Coset-separating circuit)**.** *Let $\mathbb{G}_N$ be a group in a group family $\{\mathbb{G}_N\}_{N \in \mathbb{I}}$ and let $\mathbb{H}_N \leq \mathbb{G}_N$. Also, let $\rho : \mathcal{G} \to \mathbb{S} \times \mathbb{N}$ be a LP encoding function for some $f : \mathbb{N} \to \mathbb{N}$. We say that a boolean circuit C* separates cosets *for $\mathbb{H}_N$ with respect to ρ if C computes a function $f_C : \mathbb{B}^{f(N)} \to \mathbb{B}^m$ (for some big enough m), such that:*

$$\forall g_1, g_2 \in \mathbb{G}_N : f_C(\rho(g_1, N)) = f_C(\rho(g_2, N)) \iff g_1\mathbb{H}_N = g_2\mathbb{H}_N.$$

We can then encode the graph of such coset-separating circuit C as a binary string $\lceil C \rceil$, which can serve as an effective way to pass a coset-separating function as the input of our functional.

1.2.3 The HSP Functional

We conclude this chapter by giving a formal definition of the *HSP functional*, which will serve as the foundation of our discussion on the computational importance of the HSP.

Definition 1.2.8 (HSP functional)**.** *Let $\{\mathbb{G}_N\}_{N \in \mathbb{I}}$ be a family of groups and let $\rho : \mathcal{G} \to \mathbb{S}$ be a suitable encoding function for such family, LP for some function $f : \mathbb{N} \to \mathbb{N}$. A function $F : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ is said to be a* HSP functional *for $\{\mathbb{G}_N\}_{N \in \mathbb{N}}$ and ρ if and only if on inputs $\text{bin}(N)$ and $\lceil C \rceil$, where C is a boolean circuit that separates cosets for some $\mathbb{H}_N \leq \mathbb{G}_N$ with respect to ρ, F returns $\rho(h_1, N) \| \rho(h_2, N) \| \ldots \| \rho(h_m, N)$, such that $\{h_1, h_2, \ldots, h_m\}$ is a generating set for $\mathbb{H}_N$.*

Let us try and decode this definition. In informal terms, what a HSP functional does is solve the HSP problem on all the members of a fixed group family with a valid encoding. For every input group $\mathbb{G}_N$ (represented by N) and coset-separating function f on it (represented by the description of circuit C), such a functional outputs a concatenation of the elements that generate the subgroup $\mathbb{H}_N \leq \mathbb{G}_N$ identified by f.

Chapter 2

The Importance of the HSP

In the previous chapter we presented the hidden subgroup problem in its purest form and we defined, through the HSP functional, what it means to compute a solution to one of its instances. It is now time to address *why* the HSP is such a significant problem from a computer science perspective.

2.1 Reducibility and Complexity

A fundamental concept in theoretical computer science is that of *reduction* between problems. A problem is said to be *reducible* to a second problem when an algorithm that solves the latter can be employed as a subroutine inside an algorithm that solves the former.

On a practical level, reductions are particularly useful when some properties about either one of the problems (usually the one that is reduced) are unknown. A typical example of such property is computability: if we know that a certain function f can be computed (i.e. an algorithm exists that computes it), then by reducing any other function g to f we prove that g can be computed as well, as in the process we must have shown that an algorithm for g can be built using the algorithm for f.

That said, the formal definition we provide is in some measure stricter than the informal description of the above paragraphs, but it is particularly well-suited for our purposes.

Definition 2.1.1 (Reducibility)**.** *Let f_A and f_B be two functions. We say that f_A is* reducible *to f_B (and we write $f_A \leq f_B$) if and only if there exist two computable functions* push_A *and* pull_A *such that:*

$$\mathrm{pull}_A(f_B(\mathrm{push}_A(x))) = f_A(x).$$

That is, if x_A is an input for f_A, push_A converts x_A into an input for f_B ("pushes" x_A into f_B). This new input is such that pull_A can convert f_B's output into the correct output for f_A (pull_A "pulls" the desired results out of f_B). Let us examine an informal (and imaginative) case of reducibility:

Example 2.1.1: Imagine we know how to sort a list of integers, by ascending or descending order. Imagine also, with a willing suspension of disbelief, that we want to find the smallest and greatest elements of a list, but we don't know how. Specifically, we know how to compute

$$sort(list, order),$$

in which *order* can be *asc* or *desc*. This function returns a new list in which the elements of *list* are sorted accordingly. We want to find a way to compute

$$extreme(list, which),$$

where *which* can be either *min* or *max*. This function needs to return the smallest/greatest integer in *list*, depending on *which*. We find a way to compute *extreme* by reducing it to *sort*. In particular, we define:

$$\text{push}_{extreme}(list, which) = \begin{cases} (list, asc) & \text{if } which = min, \\ (list, desc) & \text{if } which = max. \end{cases}$$

If we run $sort(\text{push}_{extreme}(list, which))$ we get a sorted list in which the first element is the smallest integer if $which = min$, or the greatest integer if $which = max$. That being said, we define:

$$\text{pull}_{extreme}(sortedList) = first(sortedList).$$

This way, the final output of $\text{pull}_{extreme}(sort(\text{push}_{extreme}(list, which)))$ is the smallest integer in *list* if $which = min$ and the greatest one if $which = max$, which is exactly the behavior we expected from the definition of *extreme*. By reducing *extreme* to *sort*, we proved that the former is computable.

Reductions in complexity theory Another field in which reductions play a significant role is that of complexity theory, as we can often provide insights on the complexity of a function f_A by reducing it to a second function f_B of known complexity (or vice-versa). In particular, if $f_A \leq f_B$ then we have proof that f_A cannot be harder than f_B to compute (or conversely, that f_B is at least as hard as f_A).

Of course, to say so we must first impose some constraints on the complexity of push_A and pull_A. Taken as it is, Definition 2.1.1 allows us to say, for example, that any function f_A is reducible to the identity function id, as by setting $\text{push}_A = f_A$ and $\text{pull}_A = \text{id}$ we have $\text{id}(\text{id}(f_A(x))) = f_A(x)$. Although correct, this reduction is devoid of any usefulness and it is obviously no proof that f_A is as easy as id to compute.

Therefore, for $f_A \leq f_B$ to be a sensible reduction we impose not only that $\text{push}_A \neq f_A \neq \text{pull}_A$, as that would defeat the purpose of a reduction entirely, but also that these functions be easier, or at least no harder than f_B to compute. That is, the computational complexity of push_A and pull_A must be *negligible* within the structure of a reduction.

Let us clarify what we mean by first providing a review of some basic concepts of complexity theory. Let M be a deterministic Turing machine. Assume we have these functions at our disposal:

- $\text{steps}_M(x)$: the execution time of M on input x, i.e. the number of computational steps required by M to accept (or reject) x.
- $\text{cells}_M(x)$: the space required by M on input x, i.e. the number of distinct cells visited by M's head during its computation on x.

We use them to define the following *time* and *space* functions:

Definition 2.1.2 (Time and space functions)**.** *Let M be a deterministic Turing machine. We define the following functions:*

- $\text{time}_M(n) = \max\{\text{steps}_M(x) \mid n = |x|\}$,
- $\text{space}_M(n) = \max\{\text{cells}_M(x) \mid n = |x|\}$.

Which measure the worst-case number of steps (and cells, respectively) required by M to compute on an input of length n.

If M computes a certain function f_M, then time_M and space_M provide us with information on the computational requirements (i.e. time and space resources needed) of f_M on inputs of a certain length.

With that in mind, we can now define precisely what we mean when we talk about the *complexity* of a function. More specifically, we can define *complexity classes*, for time and space both. If two different functions belong to the same complexity class, then we can expect their computational requirements to grow similarly with respect to the size of their inputs. In fact, what a complexity class tells us about its members is precisely *how* such requirements grow. More formally:

Definition 2.1.3 (FTIME and FSPACE classes)**.** *Let $g : \mathbb{N} \to \mathbb{N}$ be a function between natural numbers. We define:*

- *FTIME*$(g) = \{f : \mathbb{S} \to \mathbb{S} \mid \exists M : f = f_M, \text{time}_M \in O(g)\}$,
- *FSPACE*$(g) = \{f : \mathbb{S} \to \mathbb{S} \mid \exists M : f = f_M, \text{space}_M \in O(g)\}$.

Where M is a Turing machine. In other words, we say that f belongs to FTIME (FSPACE) of g *if a Turing machine M exists that computes f and whose* time_M *(*space_M*) function is $O(g)$.*

Therefore the complexity of a function is nothing more than the relationship between the size of its inputs and the computational resources needed to process them. For some specific choices of g, we obtain classes of particular interest. For example:

$$\text{FP} = \bigcup_{c \in \mathbb{N}} \text{FTIME}(n^c).$$

FP constitutes the class of functions which run in polynomial time (their worst-case running time grows like a polynomial in the size of the input). Another important complexity class is:

$$\text{FLOGSPACE} = \text{FSPACE}(\log).$$

FLOGSPACE is the class of functions whose space requirements grow logarithmically with the size of their input.

Let us return to reductions. Previously, we said that in the case of a reduction $f_A \leq f_B$ we wanted our push_A and pull_A functions to be of negligible complexity. What we meant was that if f_B belongs to FTIME(g) for some g, we want the composition $\text{pull}_A \circ f_B \circ \text{push}_A$ to *still* belong to FTIME(g).

In the case of $f_B \in \text{FP}$, this amounts to finding $\text{push}_A, \text{pull}_A \in \text{FP}$, as FP is closed under composition. As we will see in the following chapters, this is the case that most interests us. Because of that, we define the concept of *polynomial-time reductions.*

Definition 2.1.4 (Polynomial-time reducibility)**.** *Let f_A and f_B be two functions. We say that f_A is* reducible to f_B in polynomial time *(and we write $f_A \leq_p f_B$) if $f_A \leq f_B$ for some functions* $\text{push}_A, \text{pull}_A \in FP$.

We conclude this small digression on complexity by formally introducing the concept of *circuit families* that compute a function.

Definition 2.1.5 (Circuit family)**.** *Let $g : \mathbb{N} \to \mathbb{N}$ be a function. A* g-size circuit family *is a sequence $\{C_N\}_{N\in\mathbb{N}}$ of circuits in which C_N has N inputs and size $|C_N| \leq g(N)$. A function $f : \mathbb{S} \to \mathbb{S}$ is said to be in SIZE(g) if there exists a g-size circuit family $\{C_N\}_{N\in\mathbb{N}}$ such that, for all $x \in \{0,1\}^N$, we have $f(x) = f_{C_N}(x)$.*

Definition 2.1.6 (P-uniform circuit family)**.** *A circuit family $\{C_N\}_{N\in\mathbb{N}}$ is said to be* P-uniform *if there exists a polynomial-time Turing machine that on input 1^N outputs C_N.*

We prove that there exists a strong link between functions in FP and P-uniform circuit families. To do so, we need to introduce the concept of *oblivious Turing machine*, which is a Turing machine whose head movements are fixed. More formally:

Definition 2.1.7 (Oblivious Turing machine)**.** *A Turing machine M is said to be* oblivious *if the movements of its heads are fixed instead of being dictated by the machine's input. That is, M is oblivious if its transition function is of the form $\delta : Q\times\Gamma^k \to Q\times\Gamma^k$.*

Lemma 2.1.1. *Let M be a Turing machine that runs in time $O(g)$. An oblivious Turing machine M' exists that simulates M and runs in time $O(g^2)$[1].*

With these considerations in mind, we proceed to prove that every polynomial-time function can be computed by a P-uniform circuit family. Note that this is a modified version of a result given in [1] (Theorem 6.6):

Lemma 2.1.2. *Let $f : \mathbb{S} \to \mathbb{S}$ be a function between binary strings. If $f \in$ FP, then f is computable by a P-uniform circuit family.*

Proof. Let us show how, once we fix a generic n, we can build a circuit that computes f on inputs of size n. If $f \in$ FP, then there exists a polynomial-time Turing machine M that computes f. By Lemma 2.1.1, there exists an oblivious Turing machine N that simulates M with a quadratic slowdown. It follows that N also computes f in polynomial time. Now, let $x \in \{0,1\}^n$ be an input for N. Define the *transcript* of N on x to be the sequence $m_1, m_2, \ldots, m_{\text{time}_N(n)}$ of *snapshots* (current state and symbols read by each head) of N's computation. We can encode each snapshot m_i as a fixed-length binary string. Furthermore, we can compute such string from:

- The input x,
- The previous snapshot m_{i-1},

- The snapshots $m_{i_1}, \ldots, m_{i_k}$, where m_{i_j} denotes the last step in which N's jth head was in the same position as it is in the ith step. Note that since N is oblivious, these do not depend on the input.

Because these are a constant number of fixed-length strings, we can use a constant-sized circuit to compute m_i from the previous snapshots. By composing $\text{time}_N(n)$ such circuits, we obtain a bigger circuit C that computes from x to N's final state, i.e. C computes f. Also note that since N is polytime, C is of size polynomial in n. □

We now have enough material to actually start talking about the importance of the hidden subgroup problem. Although in its simplest form the HSP may appear to be nothing more than non-trivial, yet uninteresting problem of group theory, it is proven that a significant number of traditionally hard computational problems can be reduced efficiently to specific instances of the HSP. These include the simplest quantum problems (e.g. Deutsch-Jozsa and Simon's problem), as well as some of the hardest classical problems, such as the integer factorization and discrete logarithm problems, or the graph isomorphism problem.

It follows that finding an efficient solution to the HSP – and for some specific types of groups such a solution exists – would entail having an efficient way to solve problems that have always been deemed intractable. Even more specifically, the HSP models well those computational problems for which quantum algorithms exist that are exponentially faster than their classical counterparts. This is the reason why this problem plays such a valuable role in quantum algorithmics.

For a function f_A to be reducible to the HSP there must exist a group family $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$, an encoding function ρ and two functions push_A, pull_A such that:

- push_A takes the same inputs as f_A and returns an input of the form $(\text{bin}(N), \lceil C \rceil)$ for the HSP functional on $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$ and ρ.
- pull_A takes as input the output $\rho(x_1)\| \ldots \|\rho(x_m)$ of the HSP functional on $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$ and ρ and returns the output expected from f_A.

2.2 Interesting Problems

The main goal of this chapter is to show how some of these hard problems can be reduced to the HSP. Namely, we first reduce Simon's problem, a problem specifically devised to showcase the superiority of quantum algorithms. After that, we reduce a more complex and practical problem, i.e. the discrete logarithm problem.

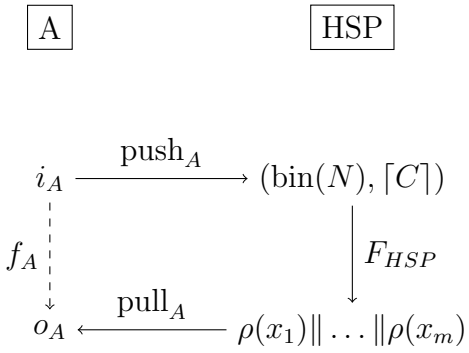


Figure 2.1: A schematic representation of the reduction of a generic problem A to the HSP, where F_{HSP} is the HSP functional as defined in the previous chapter.

2.2.1 Simon's Problem

Simon's problem was conceived in 1994 by computer scientist Daniel R. Simon [13]. It is a problem of little practical value, explicitly designed to be hard for a classical computer to solve, but easy for a quantum computer to tackle.

Definition 2.2.1 (Simon's Problem)**.** *Let $f : \mathbb{B}^n \to \mathbb{B}^n$ be a function such that there exists some $s \in \mathbb{B}^n$ for which the following property is satisfied:*

$$\forall x, y \in \mathbb{B}^n : f(x) = f(y) \iff (x = y \vee x \oplus s = y).$$

That is, f is the same when its arguments are the same or when they differ by summation modulo 2 with s. Simon's Problem *(SP) consists in, given f, finding s.*

In computational terms, we can assume that SP takes as inputs $\lceil C_{SP} \rceil$, the representation of a circuit computing a suitable function $f_{SP} : \mathbb{B}^N \to \mathbb{B}^N$, and $\text{bin}(N)$, and that it outputs s itself. The reduction of SP to the HSP is fairly straightforward [11].

Consider the family of binary string groups $\{(\mathbb{B}^N, \oplus)\}_{N \in \mathbb{N}}$. It is easy to prove that a function $f : \mathbb{B}^N \to \mathbb{B}^N$ that satisfies Simon's property is also a coset-separating function on $\mathbb{B}^N$.

Lemma 2.2.1. *Let $f : \mathbb{B}^N \to \mathbb{B}^N$ be a function from binary strings to binary strings. If f satisfies Simon's property for some $s \in \mathbb{B}^N$, then f is a coset separating function for the subgroup $\{0, s\} \leq \mathbb{B}^N$.*

Proof. By definition, $f(x) = f(y)$ if and only if $x = y$ or $x \oplus s = y$. For all $x \in \mathbb{B}^N$, f is the same on $\{x \oplus 0, x \oplus s\} = x\{0, s\}$, or the left coset of $\{0, s\}$ with respect to x, and distinct for different choices of x. That is, f separates cosets for $\{0, s\} \leq \mathbb{B}^N$. □

It is also worth noticing that $\{0, s\} = \langle s \rangle$. With these results in mind, we choose the identity function as both our push_{SP} and pull_{SP} and prove that we have in fact built a reduction.

Theorem 2.2.1. *Simon's problem is reducible to the hidden subgroup problem in polynomial time (SP $\leq_p$ HSP).*

Proof. Let $\text{bin}(N)$ and $\lceil C_{SP} \rceil$ be inputs for SP and let F_{HSP} be the HSP functional for the group family $\{(\mathbb{B}^N, \oplus)\}_{N \in \mathbb{N}}$ and $\rho = \text{id}$. By Lemma 2.2.1 we know that C_{SP} is a coset-separating circuit for $\{0, s\} \leq \mathbb{B}^N$, so $F_{HSP}(\text{bin}(N), \lceil C_{SP} \rceil)$ outputs the only element that generates $\{0, s\}$, which is s, Simon's string. □

2.2.2 The Discrete Logarithm Problem

Like the HSP, the discrete logarithm problem is a group theory problem. To understand it, we must first define what a discrete logarithm is. Let us begin by providing some necessary notation. Given a generic group $\mathbb{G}$ and a generic element $g \in \mathbb{G}$, we define

$$g^k = \underbrace{g \circ g \circ \cdots \circ g}_{k \text{ times}},$$

where k is a positive integer and $\circ$ is $\mathbb{G}$'s operation. Essentially, g^k is shorthand for applying $\circ$ k times on g.

Note that this definition does not depend on the particular nature of $\mathbb{G}$. Of course, for some specific choices of group, g^k can denote some well-known operations. On $(\mathbb{R}^+, \times)$, for example, g^k actually denotes exponentiation, while on $(\mathbb{Z}, +)$ we see that g^k is simply the product of g and k. This notation, however, can be used on any group, regardless of the underlying operation.

We are interested in the inverse operation of g^k. Namely, what we call the *discrete logarithm* of a group element.

Definition 2.2.2 (Discrete logarithm)**.** *Let $\mathbb{G}$ be a group and let $a, b \in \mathbb{G}$. The* discrete logarithm of a to the base b *is the least positive integer k for which $b^k = a$. We write* $\log_b a = k$.

Note that since we require k to be the *least* integer for which $b^k = a$, the discrete logarithm is well-defined on cyclic groups too. It is towards these groups that we turn our attention, as they provide an adequate context for the definition of the *Discrete Logarithm Problem*.

Definition 2.2.3 (Discrete Logarithm Problem)**.** *Let* $\{\mathbb{G}_N\}_{N\in\mathbb{I}}$ *be a family of cyclic groups. We assume each of its groups to be provided inside a tuple of the form* $(\mathbb{G}_N, o_N, g_N)$*, where* $|\mathbb{G}_N| = o_N$ *and* $\mathbb{G}_N = \langle g_N\rangle$*. The* Discrete Logarithm Problem *(DLP) consists in, given* N *and* $a \in \mathbb{G}_N$*, finding the least positive integer* k *such that* $g_N^k = a$*.*

Notice how the base of the discrete logarithm is not an input, but rather part of the problem. As usual, we need a computational definition. We can assume the DLP to take as inputs $\text{bin}(N)$ and $\text{bin}(a)$ such that $a \in \mathbb{G}_N$. We expect its output to be $\text{bin}(k)$, such that $g_N^k = a$. Let us try and reduce this problem to the HSP.

First, we choose to work with the HSP functional on $\{\mathbb{Z}_N \oplus \mathbb{Z}_N\}_{N\in\mathbb{N}}$. In Section 1.2.1 we saw that a direct sum of cyclic additive groups can be encoded through a concatenation of binary representations of integers. We hereby define our encoding function ρ as

$$\rho((z_1, z_2), N) = (\text{bin}_N(z_1) \| \,\text{bin}_N(z_2), N).$$

Note that in this case ρ is PLP, as its output is the concatenation of two strings of exactly $\lfloor \log_2 N \rfloor + 1$ bits each.

It is now time to find a suitable function on $\mathbb{Z}_N \oplus \mathbb{Z}_N$. Unlike Simon's problem, the DLP requires us to build a coset-separating function from scratch. It is therefore helpful to first provide a mathematical definition of this function (in doing this, we partially follow the work of McAdam [9]), only to later prove that a circuit can always be built that computes it on a fixed-length input. Let $a \in \mathbb{G}$ for some cyclic group $\mathbb{G}$ such that $\mathbb{G} = \langle g\rangle$ and $|\mathbb{G}| = N$. We define $f_a : \mathbb{Z}_N \oplus \mathbb{Z}_N \to \mathbb{G}$ as such:

$$f_a(u, v) = a^u g^v.$$

The operation of $\mathbb{G}$ is implicit between a^u and g^v. Note that f_a is a homomorphism between its domain and codomain groups. That is, f_a preserves operations in a way that for all $(u, v), (i, j) \in \mathbb{Z}_N \oplus \mathbb{Z}_N$, we have $f_a((u, v) + (i, j)) = f_a(u, v)f_a(i, j)$:

$$\begin{aligned} f_a((u, v) + (i, j)) &= f_a(u + i, v + j) \\ &= a^{u+i} g^{v+j} \\ &= a^u g^v a^i g^j = f_a(u, v) f_a(i, j). \end{aligned}$$

We can now check whether f_a is in fact a coset-separating function for some subgroup of $\mathbb{Z}_N \oplus \mathbb{Z}_N$. We find that this is exactly the case with f_a's own kernel.

Lemma 2.2.2. *Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of order N and let $a \in \mathbb{G}$. The homomorphism $f_a : \mathbb{Z}_N \oplus \mathbb{Z}_N \to \mathbb{G}$ defined as $f_a(u, v) = a^u g^v$ is a coset-separating function for the subgroup $\mathbb{H} = \{(u, v) \in \mathbb{Z}_N \oplus \mathbb{Z}_N \mid f_a(u, v) = 1\}$, i.e. for its own kernel.*

Proof. Let us start by proving that f_a is constant on any coset of $\mathbb{H}$. If two elements of $\mathbb{Z}_N \oplus \mathbb{Z}_N$ belong to the same coset $(z_1, z_2)\mathbb{H}$, then they can be written as $(z_1 + h_1, z_2 + h_2)$ and $(z_1 + h'_1, z_2 + h'_2)$ for some $(h_1, h_2), (h'_1, h'_2) \in \mathbb{H}$. We have

$$\begin{aligned} f_a(z_1 + h_1, z_2 + h_2) &= a^{z_1+h_1} g^{z_2+h_2} \\ &= a^{z_1} g^{z_2} a^{h_1} g^{h_2} \\ &= a^{z_1} g^{z_2} \\ &= a^{z_1} g^{z_2} a^{h'_1} g^{h'_2} \\ &= a^{z_1+h'_1} g^{z_2+h'_2} = f_a(z_1 + h'_1, z_2 + h'_2), \end{aligned}$$

as $a^{h_1} g^{h_2} = a^{h'_1} g^{h'_2} = 1$ by $\mathbb{H}$'s very definition. Let us now prove that if $f_a(z_1, z_2) = f_a(z'_1, z'_2)$ then $(z_1, z_2)\mathbb{H} = (z'_1, z'_2)\mathbb{H}$. We start by proving that $(z_1, z_2)\mathbb{H} \subseteq (z'_1, z'_2)\mathbb{H}$, i.e. that every element of the form $(z_1 + h_1, z_2 + h_2)$ can be written as $(z'_1 + h'_1, z'_2 + h'_2)$, where $(h_1, h_2), (h'_1, h'_2) \in \mathbb{H}$. Note that

$$(z_1 + h_1, z_2 + h_2) = (z'_1 + (z_1 - z'_1 + h_1), z'_2 + (z_2 - z'_2 + h_2)).$$

It is easy to show that $(z_1 - z'_1 + h_1, z_2 - z'_2 + h_2)$ is, in fact, an element of $\mathbb{H}$:

$$\begin{aligned} f_a(z_1 - z'_1 + h_1, z_2 - z'_2 + h_2) &= a^{z_1 - z'_1 + h_1} g^{z_2 - z'_2 + h_2} \\ &= a^{z_1} g^{z_2} a^{-z'_1} g^{-z'_2} a^{h_1} g^{h_2} \\ &= a^{z_1} g^{z_2} (a^{z'_1} g^{z'_2})^{-1} a^{h_1} g^{h_2} = 1. \end{aligned}$$

This is because $a^{z_1} g^{z_2} = a^{z'_1} g^{z'_2}$ by hypothesis and $a^{h_1} g^{h_2} = 1$ by definition. The opposite inclusion is proven in the exact same way. This shows that f_a is in fact a coset-separating function.

□

Once we have such a function, we can define an encoding $\xi : \mathbb{G} \to \mathbb{S}$ that extends ρ and such that $\xi \circ f_a : \mathbb{Z}_N \oplus \mathbb{Z}_N \to \mathbb{S}$ is a coset-separating function that complies with Definition 1.1.6. For the sake of simplicity, we still call this composition f_a.

Of course, we need this function to be computable by a circuit for any choice of group in $\{\mathbb{Z}_N \oplus \mathbb{Z}_N\}_{N\in\mathbb{N}}$. Thanks to Lemma 2.1.2, all we need to do is justify the existence of an algorithm which computes f_a and runs in time polynomial in the size of the representations produced by ρ and ξ. In Section 1.2.1 we explicitly required our encoding functions to allow efficient multiplication and exponentiation, so it is evident that such an algorithm exists. Because of that, there exists a P-uniform circuit family that efficiently computes f_a, for all a.

It is time to start building the reduction of the DLP to the HSP on $\{\mathbb{Z}_N \oplus \mathbb{Z}_N\}_{N\in\mathbb{N}}$. Assume that M is a Turing machine that, given 1^n and a representation of a, builds the circuit that computes f_a on inputs of size n. We define push_{DLP} as follows:

Algorithm 1: push_{DLP}

Input: $\text{bin}(N)$ such that N identifies a cyclic group $(\mathbb{G}_N, o_N, g_N)$ and $\xi(a)$ such that $a \in \mathbb{G}_N$.

Output: $\text{bin}(N')$ and $\lceil C \rceil$, inputs for the HSP functional on $\{\mathbb{Z}_N \oplus \mathbb{Z}_N\}_{N\in\mathbb{N}}$ and ρ.

$i \leftarrow 1^{2|\,\text{bin}(o_N)|}$;
$\lceil C \rceil \leftarrow$ Run M on i and $\xi(a)$ and extract the output circuit;
return $(\text{bin}(o_N), \lceil C \rceil)$;

It is clear that push_{DLP} runs in time polynomial in the size of the input (due to M). Of course, because it computes f_a on inputs of size $2(\lfloor \log_2 o_N \rfloor + 1)$, C is a coset-separating circuit on $\mathbb{Z}_{o_N} \oplus \mathbb{Z}_{o_N}$. Before we proceed with pull_{DLP}, let us consider what we can actually do once we have some elements of $\mathbb{H}$:

Lemma 2.2.3. *Let (u, v) be an element of $\mathbb{H} = \{(u, v) \in \mathbb{Z}_{o_N} \times \mathbb{Z}_{o_N} \mid f(u, v) = 1\}$, where $f(u, v) = a^u g_N^v$ and $g_N^k = a$. We have that $k \equiv -uv^{-1} \mod o_N$.*

Proof. By $\mathbb{H}$'s very definition, we have that $a^u g^v = 1$. Since $a = g^k$, we can write as $g^{ku+v} = 1$. It follows that $ku + v$ is the order of g_N and thus is a multiple of o_N. We have $ku + v \equiv 0 \mod o_N$ and therefore $k \equiv -vu^{-1} \mod o_N$. □

So once we have at least one element of $\mathbb{H}$, we can compute $k = -vu^{-1} \mod o_N$. We therefore define pull_{DLP} as such:

Algorithm 2: pull_{DLP}

Inputs : The output $\rho((u_1, v_1), o_N) \| \rho((u_2, v_2), o_N) \| \dots \| \rho((u_m, v_m), o_N)$ of the HSP functional on inputs $\text{bin}(o_N), \lceil C \rceil$.

Outputs: $\text{bin}(k)$ such that $g_N^k = a$.

$o_N \leftarrow$ Extract o_N from the input;
$(u, v) \leftarrow$ Extract (u_1, v_1) from the input;
$k \leftarrow -vu^{-1} \mod o_N$;
return $\text{bin}(k)$;

The extraction of o_N and (u_1, v_1) from the input is trivial and can be performed efficiently in linear time (in the size of the output of ρ, which is PLP). Furthermore, u^{-1} can be computed in time polynomial using Euclid's algorithm and modular arithmetic can be performed efficiently. We state our result:

Theorem 2.2.2. *The Discrete Logarithm Problem is reducible in polynomial time to the HSP (DLP $\leq_p$ HSP).*

Proof. Let $(\mathbb{G}_N, o_N, g_N)$ be a cyclic group in $\{\mathbb{G}_N\}_{N \in \mathbb{I}}$ and let $a \in \mathbb{G}_N$ be an element such that $g_N^k = a$, for some integer k. Let $\text{bin}(N)$ and $\xi(a)$ be inputs for the DLP on $\{\mathbb{G}_N\}_{N \in \mathbb{I}}$. Let F_{HSP} be the HSP functional on the group family $\{\mathbb{Z}_N \oplus \mathbb{Z}_N\}_{N \in \mathbb{N}}$. By lemmata 2.2.2 and 2.1.2, we know that for all a we can build a coset-separating circuit for $\mathbb{H} = \{(u, v) \in \mathbb{Z}_{o_N} \oplus \mathbb{Z}_{o_N} | a^u g_N^v = 1\} \leq \mathbb{Z}_{o_N} \oplus \mathbb{Z}_{o_N}$ in time polynomial. Let C be such a circuit. By running F_{HSP} on $\text{bin}(o_N)$ and $\lceil C \rceil$, we obtain a generating set for $\mathbb{H}$. By Lemma 2.2.3, we can use $(u, v) \in \mathbb{H}$ to efficiently compute $k = -vu^{-1} \mod o_N$. □

Chapter 3

An Introduction to Quantum Computing

At this point we have given a computational definition of the HSP (Chapter 1) and we have examined *why* a solution to this problem is desirable (Chapter 2). What we still have to do is actually *provide* such a solution, i.e. give an implementation of the HSP functional as defined in Section 1.2. That is exactly what we will do in the next chapter, at least for the HSP on families of Abelian groups.

Our implementation of the HSP functional, however, will not be classical. Rather, we will avail ourselves of the quantum circuit model to give an efficient quantum-computational solution to the problem at hand. Because of that, it is first advisable to introduce the principles that rule the quantum world, as well as some fundamentals of quantum computing. This is the purpose of this chapter.

Note that this chapter cannot (and therefore does not attempt to) present an exhaustive introduction to the complex and vast world of quantum mechanics and quantum computing. Rather, it covers little more than what is strictly necessary to understand the next chapter. For a more thorough introduction to the topic, refer to the excellent work of Yanofsky and Mannucci [15] or to Nielsen and Chuang [11].

3.1 Quantum Systems

To try and approach the quantum world through intuition is, to say the least, counterproductive. That is because the very principles that rule this world are strongly counterintuitive and do not reflect in any way our everyday perception of reality. On the other hand (and surprisingly enough), a formal, mathematical approach tends to explain these concepts more clearly and is furthermore better suited to our goals.

3.1.1 The State of a System

A fundamental concept (regardless of the chosen approach) is that of *system*. A system (quantum or classical) is any portion of reality that can exist in a number of distinct *basis states*. Each of these states corresponds to a different configuration in which the system can be found at any given moment. Conversely, every one of such configurations is associated to a basis state. In other words, the basis states represent *all* the possible configurations of a system.

Example 3.1.1: An ideal coin is a system which exhibits two basis states: "heads up" and "tails up".

The main difference between classical and quantum systems is that whilst classical systems exist in one and only one basis state at a time, quantum systems can exist in multiple basis states at once, with varying weight. More formally, quantum systems can exist in a *superposition* of basis states.

Here is where mathematics really come into play. The concept of superposition is easily captured through the *state vector* of a quantum system, defined as follows:

Definition 3.1.1 (State vector of a quantum system)**.** *Consider a system with n basis states, which we label $\beta_1, \beta_2, \ldots, \beta_n$. The* state vector *of the system is a column vector in $\mathbb{C}^n$ defined as*

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix},$$

where $0 \leq |c_j|^2 \leq 1$ for every $j = 1 \ldots n$ and where $\sum_{j=1}^{n} |c_j|^2 = 1$. Each $|c_j|^2$ corresponds to the probability of observing the system in its basis state β_j.

Essentially, each c_j tells us *how much* the system is in the basis state β_j. The use of the word "probability" in the definition should not mislead the reader into believing that the system is actually in a single, defined state and that we just happen not to know which one it is. A quantum system can *actually* exists in a superposition of two or more basis states at a time

Example 3.1.2: An ideal "quantum coin" is a quantum system with basis states $\beta_1 =$ "heads up" and $\beta_2 =$ "tails up". Such a coin can exist in a superposition of these states, for example:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}.$$

Because $|1/\sqrt{2}|^2 = 1/2$, this state vector corresponds to the coin being half "heads up", half "tails up".

Note that classical systems are also captured by this form of representation. A column vector in which all entries are zero except for the jth one (for which necessarily $|c_j|^2 = 1$) corresponds to a classical system in the basis state β_j. These state vectors are obviously orthogonal and can be treated as the basis of a vector space, which we call the *state space* of the system.

Example 3.1.3: An ideal "classical coin" is a classical system with the same basis states as the previous example and whose only allowed state vectors are:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

These state vectors correspond to the coin being "heads up" and "tails up", respectively.

Example 3.1.4: The state space of an ideal quantum coin is the subspace of $\mathbb{C}^2$ spanned by $[1, 0]^T$ and $[0, 1]^T$, which in this case coincides with $\mathbb{C}^2$.

Let us get back to the "probability" part of the last definition. When we observe, or rather *measure* a system, any potential superposition *collapses* and the system appears to be in a single basis state. The probability of a system with basis states $\beta_1, \beta_2, \ldots, \beta_k$ of collapsing, once measured, to any β_j depends entirely on its state vector's c_j entry. More specifically, this probability is equal to $|c_j|^2$.

It is obvious that classical systems are unaffected by measurements. A classical system in the β_j basis state will have $c_j = |c_j|^2 = 1$ and will appear to be in the β_j state with 100% probability. Quantum systems, on the other hand, are disrupted by measurements. In fact, once a superposition collapses, the system *persists* in the new collapsed state, even once the measurement is over. In other words, measuring a quantum system puts it into one of its possible classical states, depending on the magnitudes of his state vector.

Example 3.1.5: Consider the ideal quantum coin of Example 3.1.2. If we try and observe which side is up, we may find that it is "heads up" or "tails up", with equal probability. Note that any other subsequent measurement will have the same outcome, as the system will have collapsed to a classical state.

From now on, when writing about states and operations on them, we will always employ the standard *bra-ket* notation. In this notation, a generic column vector describing a state can be written $|\varphi\rangle$ (this is called a *ket*) and the corresponding row vector can

be written $\langle\varphi|$ (which is a *bra*). Note that φ is just a placeholder. With this notation, the basis states of a system can be written $|\beta_1\rangle, |\beta_2\rangle \dots$ and so on. Furthermore, $\langle\varphi|\psi\rangle$ denotes the *inner product* of vectors φ and ψ and $|\varphi\rangle\langle\psi|$ the *outer product*.

Example 3.1.6: Let us once again consider the ideal quantum coin from the previous examples. With the bra-ket notation we can write something like

$$|heads\rangle = \begin{bmatrix}1\\0\end{bmatrix}, \quad |tails\rangle = \begin{bmatrix}0\\1\end{bmatrix}.$$

Furthermore, we can now write the state from Example 3.1.2 as

$$\frac{1}{\sqrt{2}}|heads\rangle + \frac{1}{\sqrt{2}}|tails\rangle = \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix}.$$

3.1.2 The Evolution of a System

At this point we know that measuring a quantum systems alters it irreversibly. We are interested in other ways to modify the state of a quantum system. That is, we are interested in the *evolution* of quantum systems. Because states are represented as vectors, it is only natural to describe the evolution from one state to another through a matrix. However, to correctly represent a quantum transformation, a matrix must obey some constraints. Namely:

- A transformation U acting on a state vector must produce a valid state vector. That is, if $|\varphi\rangle$ is the state vector of a system and U describes a transformation of such system, then $U|\varphi\rangle = [c_1, c_2, \dots, c_k]^T$ is such that $0 \leq |c_j|^2 \leq 1$ for all j and $\sum_{j=1}^{k} |c_j|^2 = 1$.
- Quantum system evolve reversibly. It follows that every possible quantum transformation U must be reversible (i.e. its matrix must be invertible).

These requirements are fulfilled by *unitary matrices*. To give a definition of *unitary matrix*, however, we first need to define the concepts of *complex conjugate* and *conjugate transpose*.

Definition 3.1.2 (Complex conjugate)**.** *Let $c \in \mathbb{C}$ be a complex number such that $c = a + bi$. Its* complex conjugate *is written $\bar{c}$ and is defined as*

$$\bar{c} = a - bi.$$

Definition 3.1.3 (Conjugate transpose)**.** *Let A be a matrix in $\mathbb{C}^{m\times n}$. Its* conjugate transpose *$A^\dagger$ is the matrix in $\mathbb{C}^{n\times m}$ defined as*

$$A^\dagger_{i,j} = \overline{A_{j,i}}.$$

That is, the elements of $A^\dagger$ are the conjugated elements of A's transpose.

Definition 3.1.4 (Unitary matrix)**.** *A complex square matrix U is said to be* unitary *when its inverse exists and coincides with its conjugate transpose. That is, when*

$$UU^\dagger = U^\dagger U = I.$$

Unitary matrices preserve the validity of quantum states and therefore are perfect to describe the evolution of quantum systems. Consider a system with n basis states and consider a unitary matrix U, defined as follows:

$$U = \begin{array}{c} \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{array} \begin{array}{c} \begin{array}{cccc} \beta_1 & \beta_2 & \dots & \beta_n \end{array} \\ \begin{bmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,n} \\ u_{2,1} & u_{2,2} & \dots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n,1} & u_{n,2} & \dots & u_{n,n} \end{bmatrix} \end{array}.$$

If U acts on a state $|\varphi\rangle = [c_1, c_2, \dots, c_n]^T$ to produce a new state $U|\varphi\rangle = [c'_1, c'_2, \dots, c'_n]^T$, then each $u_{j,k}$ entry in U determines how much the magnitude c_k weighs in determining the resulting magnitude c'_j. This is just a direct consequence of matrix multiplication.

Example 3.1.7: As usual, we consider the ideal quantum coin of the previous examples. We can define the following F transformation on the system:

$$F = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

It is not hard to see that F simply *flips* the coin. This is evident when we consider F's action on the basis states:

$$F|heads\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |tails\rangle,$$
$$F|tails\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |heads\rangle.$$

Of course, since $FF^\dagger = F^2 = I$ we have that F is unitary and thus corresponds to a valid unitary transformation.

Example 3.1.8: We wonder whether it is also possible to define the *toss* transformation on a coin. We expect such operation to resemble the following matrix:

$$T = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

Intuitively, T does its job. In fact, we can see that it acts on the basis states just as intended:

$$T|heads\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix},$$
$$T|tails\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{bmatrix}.$$

In both cases we end up in a superposition that, once observed, has equal probability of collapsing to heads up or tails up (which is exactly what we expect from a coin toss). However, when we check if T is unitary, we find that

$$TT^\dagger = T^2 = \frac{1}{2}\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix} \neq I.$$

Therefore T is not unitary and cannot be a valid reversible transformation. This should come as no surprise, as it is clear that if we were to pick up a coin and toss it, then ask a third person to join us and observe the outcome, he or she would have no way to tell whether the coin was originally heads up or tails up. However, in the next section we will that there exists a way to circumvent this form of irreversibility.

At this point, we might wonder whether there exist transformations that are *inherently* impossible to perform on quantum systems. It turns out that this is in fact the case. For example, it is impossible to copy a quantum state exactly, a result known as the *no-cloning theorem*. We will discuss this result more precisely in the next section.

3.1.3 Composite Systems

We conclude this general introduction to quantum system and their evolution with the composition of systems.

Suppose we have two independent systems S and S', with basis states $\beta_1, \beta_2, \ldots, \beta_n$ and $\beta'_1, \beta'_2, \ldots, \beta'_m$, respectively. Previously, we said that to every quantum system is associated a state space. Consider thus $\mathbb{V} = \langle \beta_1, \beta_2, \ldots, \beta_n \rangle$ and $\mathbb{V}' = \langle \beta'_1, \beta'_2, \ldots, \beta'_m \rangle$, i.e. the state spaces of S and S', respectively. The composition of S and S' is the quantum system whose state space is the *tensor product* of $\mathbb{V}$ and $\mathbb{V}'$. Let us approach this concept gradually.

Definition 3.1.5 (Kronecker Product)**.** *Let A and B be two matrices of dimensions $n \times m$ and $p \times q$, respectively. The* Kronecker product *of A and B is written $A \otimes B$ and*

is defined as the following $np \times mq$ matrix:

$$A \otimes B = \begin{bmatrix} A_{1,1}B & A_{1,2}B & \dots & A_{1,m}B \\ A_{2,1}B & A_{2,2}B & \dots & A_{2,m}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1}B & A_{n,2}B & \dots & A_{n,m}B \end{bmatrix},$$

where each $A_{i,j}B$ is a submatrix of $A \otimes B$ obtained by multiplying B with the scalar $A_{i,j}$.

Example 3.1.9:

$$A = \begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix},$$

$$A \otimes B = \begin{bmatrix} 1\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} & 3\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \\ 5\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} & 7\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 2 & 4 & 6 & 12 \\ 6 & 8 & 18 & 24 \\ 10 & 20 & 14 & 28 \\ 30 & 40 & 42 & 56 \end{bmatrix}.$$

Intuitively, the Kronecker product of two matrices A and B is a larger matrix $A \otimes B$ such that for every entry in A *and* every entry in B there is a distinct entry in $A \otimes B$. We can also see clearly that $A \otimes B$ partially preserves the structure of A and B. The Kronecker product is a form of *tensor product* between matrices (more on that later), so its application is often referred to as *tensoring*. Of course, being defined on matrices of arbitrary size, the Kronecker product is also defined on column and row vectors.

Example 3.1.10:

$$u = \begin{bmatrix} 3 \\ 5 \end{bmatrix}, \quad v = \begin{bmatrix} 4 \\ 6 \end{bmatrix},$$

$$u \otimes v = \begin{bmatrix} 3\begin{bmatrix} 4 \\ 6 \end{bmatrix} \\ 5\begin{bmatrix} 4 \\ 6 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 12 \\ 18 \\ 20 \\ 30 \end{bmatrix}.$$

Naturally, since quantum states are nothing but complex column vectors, it is also possible to tensor quantum states. The result of such operation should be self-explanatory, as what we are performing is nothing but a multiplication of probabilities.

Example 3.1.11:

$$|\varphi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \quad |\psi\rangle = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{\sqrt{2}} \end{bmatrix},$$

$$|\varphi\rangle \otimes |\psi\rangle = |\varphi\rangle|\psi\rangle = |\varphi\psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \\ \\ \frac{1}{\sqrt{2}} \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} \\ \frac{1}{2} \end{bmatrix}.$$

It is obvious that if $|\varphi\rangle$ and $|\psi\rangle$ are valid quantum states, then so is $|\varphi\rangle \otimes |\psi\rangle$. Please pay attention to the notation employed in the last example. In particular, note that $|\varphi\rangle \otimes |\psi\rangle$, $|\varphi\rangle|\psi\rangle$ and $|\varphi\psi\rangle$ are equivalent and all denote the tensor product of the two vectors. That being said, we can now define the tensor product between vector spaces.

Definition 3.1.6 (Tensor product of vector spaces)**.** *Let $\mathbb{V}$ and $\mathbb{V}'$ be two vector spaces of dimension n and m, respectively. The* tensor product *of $\mathbb{V}$ and $\mathbb{V}'$ is written $\mathbb{V} \otimes \mathbb{V}'$ and is the vector space of dimension nm defined as follows:*

$$\mathbb{V} \otimes \mathbb{V}' = \{u \otimes v \mid u \in \mathbb{V}, v \in \mathbb{V}'\}.$$

That is, the space containing all those vectors obtained by tensoring a vector from $\mathbb{V}$ with one from $\mathbb{V}'$.

A brief note on $\otimes$: in this definition of tensor product between vector spaces we employed the Kronecker product. However, this was a choice made purely out of convenience, as the actual definition of tensor product between vector spaces does not involve the Kronecker product at all. In fact, the only constraints that are imposed on the $\otimes$ operation are those of bilinearity, that is:

- For all $u \in \mathbb{V}$ and $v \in \mathbb{V}'$ and for every scalar λ: $\lambda(u \otimes v) = (\lambda u) \otimes v = u \otimes (\lambda v)$.
- For all $u_1, u_2 \in \mathbb{V}$ and $v \in \mathbb{V}'$: $(u_1 + u_2) \otimes v = u_1 \otimes v + u_2 \otimes v$.
- For all $u \in \mathbb{V}$ and $v_1, v_2 \in \mathbb{V}'$: $u \otimes (v_1 + v_2) = u \otimes v_1 + u \otimes v_2$.

Of course, the Kronecker product exhibits all these properties, among others. In our approach, we chose to present a particular case of tensor product (i.e. the Kronecker product) *first*, in order to make the tensoring of vector spaces more tangible and easier to comprehend. For a more abstract, yet thorough discussion on tensor products and their applications to quantum computer science, refer to [11].

Let us go back to $\mathbb{V} \otimes \mathbb{V}'$, which we know contains all the vectors that can be produced by tensoring a vector from $\mathbb{V}$ with one from $\mathbb{V}'$. Suppose that $B = \{\beta_1, \beta_2, \ldots, \beta_n\}$ is a basis for $\mathbb{V}$ and $B' = \{\beta'_1, \beta'_2, \ldots, \beta'_m\}$ is a basis for $\mathbb{V}'$. Then two generic vectors $u \in \mathbb{V}$ and $v \in \mathbb{V}'$ can be written like

$$\begin{aligned} u &= c_1\beta_1 + c_2\beta_2 + \cdots + c_n\beta_n, \\ v &= c'_1\beta'_1 + c'_2\beta'_2 + \cdots + c'_m\beta'_m. \end{aligned}$$

Then, by the properties we saw before, their tensor product $u \otimes v$ can be written like:

$$\begin{aligned} u \otimes v &= (c_1\beta_1 + c_2\beta_2 + \cdots + c_n\beta_n) \otimes (c'_1\beta'_1 + c'_2\beta'_2 + \cdots + c'_m\beta'_m) \\ &= c_1\beta_1 \otimes c'_1\beta'_1 + c_1\beta_1 \otimes c'_2\beta'_2 + \cdots + c_2\beta_2 \otimes c'_1\beta'_1 + \cdots + c_n\beta_n \otimes c'_m\beta'_m \\ &= c_1c'_1(\beta_1 \otimes \beta'_1) + c_1c'_2(\beta_1 \otimes \beta'_2) + \cdots + c_2c'_1(\beta_2 \otimes \beta'_1) + \cdots + c_nc'_m(\beta_n \otimes \beta'_m). \end{aligned}$$

Because u and v are generic, we have that *any* vector in $\mathbb{V} \otimes \mathbb{V}'$ can be written as a linear combination of the tensor products of the elements of the two bases B and B'. That is, $B \times B'$ is a basis for $\mathbb{V} \otimes \mathbb{V}'$. This result gives a second characterization of the tensor product between vector spaces. Namely, it tells us that $\mathbb{V} \otimes \mathbb{V}'$ can be defined simply as the vector space spanned by the basis $B \times B'$, where B is a basis for $\mathbb{V}$ and B' is a basis for $\mathbb{V}'$.

At the beginning of our discussion on composite states, we said that two quantum systems can be assembled by tensoring their respective state spaces. The meaning of this statement should now be clearer. Nevertheless, we present an example that explains, perhaps more intuitively, the link between the two concepts.

Example 3.1.12: Consider two ideal quantum coins. On their own, they each have exactly two basis states: $|heads\rangle$ and $|tails\rangle$. Now, consider them as the two parts of a single, larger system. We have that for each of the basis states the first coin can be in, the second coin can be in either basis state, as the two subsystems are independent. Formally, we find that the resulting system has *four* basis states:

$$|heads\rangle|heads\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ 0 \begin{bmatrix} 1 \\ 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

$$|heads\rangle|tails\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, |tails\rangle|heads\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, |tails\rangle|tails\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

These new basis states constitute a basis for the tensor product of the state spaces of the two coins. In Example 3.1.4 we saw that the state space of a coin is $\mathbb{C}^2$. Here we can easily see that the state space of the composite system is $\mathbb{C}^4 = \mathbb{C}^2 \otimes \mathbb{C}^2$.

Of course, the two systems do not need to be identical. Also note that composite systems can be further tensored with one another to give birth to quantum systems of arbitrary complexity.

Tensoring of transformations The Kronecker product is defined on generic matrices. This hints at the possibility of tensoring quantum transformations. In fact, if we have a transformation U that acts on a system S with n basis states and a transformation U' that acts on a system S' with m basis states, we can define a new transformation $U \otimes U'$ that acts on the composition of S and S'. The starting transformations are represented respectively by a $n \times n$ matrix and a $m \times m$ matrix, which means that $U \otimes U'$ is the transformation defined by the following $nm \times nm$ matrix:

$$U \otimes U' = \begin{bmatrix} u_{1,1}U' & u_{1,2}U' & \dots & u_{1,n}U' \\ u_{2,1}U' & u_{2,2}U' & \dots & u_{2,n}U' \\ \vdots & \vdots & \ddots & \vdots \\ u_{n,1}U' & u_{n,2}U' & \dots & u_{n,n}U' \end{bmatrix}.$$

The action performed by $U \otimes U'$ on the assembled system is straightforward and consists in the application of U on the subsystem S and the application of U' on S'. In other words, given two generic states $|\varphi\rangle$ of S and $|\psi\rangle$ of S' we can write

$$(U \otimes U')(|\varphi\rangle \otimes |\psi\rangle) = (U|\varphi\rangle) \otimes (U'|\psi\rangle).$$

3.1.4 Entanglement

Now that we know how to assemble quantum states, we can discuss a very interesting quantum phenomenon called *entanglement*. In the classical world, the state of a composite system can always be described in terms of the states of its parts.

Example 3.1.13: If we have a classical system composed of two classical coins, we can determine its state by looking at the states of the individual coins. That is, we can take any of its states (and because the system is classical, these are only the four basis states) and decompose it into two single-coin states. If, for example, the first coin is heads up and the second one is tails up, we can perform the following decomposition:

$$heads\ and\ tails = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = heads \otimes tails.$$

This form of decomposition is not only intuitive, it is almost trivial. In fact, our everyday comprehension of reality relies on our innate ability to describe a complex

situation in terms of its simpler components, to the point where the two alternative ways of describing the two coins in the last example bear little to no difference before our perception.

Quantum systems are a generalization of classical systems. Therefore, the following question arises naturally: can this form of decomposition be carried out on quantum systems as well? The answer to this question is, to say the least, surprising, as we find that there exist quantum states that simply *cannot* be decomposed. Take, for demonstration, the two-coin example from above and consider the following state:

$$|\varepsilon\rangle = \frac{|head\rangle|head\rangle \otimes |tails\rangle|tails\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}.$$

This is of course a perfectly valid quantum state that describes a situation in which, upon observation, the two coins can be found either both heads up or both tails up, with equal probability. Our intuition tells us that this system must by describable in terms of the states of the individual coins. In other words, there must exist two single-coin states $|\varphi\rangle$ and $|\psi\rangle$ such that

$$|\varepsilon\rangle = |\varphi\rangle \otimes |\psi\rangle.$$

Let us try and find $|\varphi\rangle$ and $|\psi\rangle$. We keep the two states as generic as possible, writing them as

$$|\varphi\rangle = c_1|heads\rangle + c_2|tails\rangle,$$
$$|\psi\rangle = c_1'|heads\rangle + c_2'|tails\rangle.$$

We now rewrite their tensor product in terms of the basis states and attempt to calculate the desired magnitudes:

$$\begin{aligned} |\varphi\rangle \otimes |\psi\rangle &= (c_1|heads\rangle + c_2|tails\rangle) \otimes (c_1'|heads\rangle + c_2'|tails\rangle) \\ &= c_1c_1'|heads\rangle|heads\rangle + c_1c_2'|head\rangle|tails\rangle + c_2c_1'|tails\rangle|heads\rangle + c_2c_2'|tails\rangle|tails\rangle. \end{aligned}$$

In order for this last line to equate $|\varepsilon\rangle = \frac{1}{\sqrt{2}}|heads\rangle|heads\rangle \otimes \frac{1}{\sqrt{2}}|tails\rangle|tails\rangle$, the following conditions must hold:

$$\begin{cases} c_1c_1' = \frac{1}{\sqrt{2}} \\ c_1c_2' = 0 \\ c_2c_1' = 0 \\ c_2c_2' = \frac{1}{\sqrt{2}} \end{cases}$$

However, one can easily see that this systems has no solutions, which implies that such decomposition of $|\varepsilon\rangle$ into $|\varphi\rangle$ and $|\psi\rangle$ is not possible.

What does this mean? It means that once we take superpositions into account, we are bound to encounter systems whose states can only be described as a whole. These are called *entangled states*, to reflect the fact that their components cannot be taken apart and described individually. Quantum states that *can* be decomposed into smaller substates, on the other hand, are called *separable states*.

Although entanglement may appear to be nothing more than a curiosity, arising from the possibility of having superpositions of states, it actually lends itself to significant practical applications. Consider the same two-coin entangled state from before. If we measure the whole system, then we have equal chances of finding both coins heads up or tails up. What happens if we instead measure only one of the coins? Assume, without loss of generality, that we observe the first coin in the $|heads\rangle$ state. The global state collapses and only those basis states in which the first coin is $|heads\rangle$ are left. That is, we have

$$|\varepsilon\rangle = \frac{1}{\sqrt{2}}|heads\rangle|heads\rangle \otimes \frac{1}{\sqrt{2}}|tails\rangle|tails\rangle \xrightarrow{\text{measure 1st}} |heads\rangle|heads\rangle.$$

That is, even though we have not bothered the second coin at all, its probability of being observed tails up vanished with the measurement of the first coin. That is because the states of the two individual coins are intimately related (entangled, indeed), in a manner that makes it impossible to manipulate one of them without influencing the other. This property of entangled states will be particularly useful in the algorithm given in section 4.2.

The practical value of this mechanism is clear once we discover that the entanglement of two or more parts of a system persists even when said parts are separated by arbitrarily large distances. This crucial detail lies at the heart of *quantum teleportation*, a process through which the state of a quantum system with two basis states (as we will soon see, a *qubit*) can be transmitted instantaneously over long distances [15].

3.2 Quantum Computing

At the present time we know how to describe quantum systems in terms of their state, we know how to describe their evolution and how to assemble them, starting from their simplest components. We also examined the phenomenon of entanglement. It is time to apply this knowledge to define what quantum *computing* is.

As we know, the smallest possible unit of classical information is the *bit*. Through the concepts of system and state, we can give a formal definition of what a bit is.

Definition 3.2.1 (Bit)**.** *Any classical system that can exist in two distinct basis states is called a* bit*.*

This is a rather abstract definition of what a bit is. However, if we think about it, we realize that the concrete implementation of a bit can be ignored entirely once it complies with the "interface" given in this definition. The basis states of a bit are usually referred to as 0 and 1.

Example 3.2.1: With the definition above, we find that a lot of systems can implement a bit. Of course, a wire, in which current can be flowing or not, is a bit. Any form of switch is also a bit, with *off* and *on* basis states. A door can also be considered a bit, with *closed* and *open* basis states.

Of course, it goes without saying that a *qubit*, the smallest unit of quantum information, is nothing but the quantum counterpart of a classical system implementing a bit.

Definition 3.2.2 (Qubit)**.** *Any quantum system that can exist in a superposition of two distinct basis states is called a* qubit*.*

Just like quantum systems were a generalization of classical ones, qubits are a generalization of bits. The basis states of a qubit are usually referred to as $|0\rangle$ and $|1\rangle$.

Example 3.2.2: The ideal quantum coin that we saw in the previous section is in fact a qubit implementation, with basis states $|heads\rangle = |0\rangle$ and $|tails\rangle = |1\rangle$.

Of course, in our discussion we will not concern ourselves with the actual implementation of bits and qubits. Note that just like bits can be assembled together to form classical registers, qubits can be assembled to form quantum registers.

Example 3.2.3: Eight qubits can be combined to form a *qubyte*, which is a larger quantum system with $2^8 = 256$ basis states:

$$|00000000\rangle, |00000001\rangle, |00000010\rangle, \ldots, |11111110\rangle, |11111111\rangle.$$

3.2.1 Quantum Gates

In classical computers, bits can be acted on by *logic gates*, which represent the smallest units of computation. It is easy to show that any logic gate with n inputs and m outputs can be represented as a $2^m \times 2^n$ matrix. However, for the sake of simplicity, we only show the matrix representation of *some* of the most common logic gates. We label the rows and columns of the matrices in order to better convey their meaning:

$$\text{NOT} = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{array}.$$

Intuitively, this matrix "takes as input" a single-bit system and "outputs" a new single-bit system in the basis state 1, if the input system is in the basis state 0, or 0, if the input system is in the basis state 1. This is exactly what we expect from the NOT gate. Let us see some more examples:

$$\text{AND} = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array},$$

$$\text{OR} = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \end{array}.$$

These examples are slightly more complicated, but they should be easy to understand nonetheless. Take the AND gate. The AND matrix "takes as input" a system with two bits and "outputs" a single-bit system, which is in the basis state 1 if and only if the input bits are in a basis state that verifies the logical conjunction. Therefore, the given matrix performs the same operation as the AND logic gate.

Note that in the previous examples we assumed the input bits and the output bits to be two distinct systems. This is not at all necessary. We can easily examine the case of a system that includes the input and output bits (which might be disjoint or not), on which the previous operations can be defined as transformations. Consider again the AND gate. We incorporate its inputs and outputs in a single 3-bit system and we define a transformation as follows:

$$\text{AND}' = \begin{array}{c} \\ 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{array} \begin{array}{c} \begin{array}{cccccccc} 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{array} \\ \left[\begin{array}{cccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right] \end{array}$$

This transformation takes the first two bits of the state, performs the AND operation between them and "puts" the result in the third bit of the system.

Here is where we encounter the first great divide between classical and quantum computing. In the previous section, we required transformations on quantum systems to be reversible. However, most logic gates are *not* reversible. Namely, all gates that have fewer outputs than inputs are not injective and cannot therefore be reversible. Take for example the AND operation. If we only know that $x \wedge y = 0$, we have no way of finding out whether $x = 0$ and $y = 0$, or $x = 0$ and $y = 1$, or $x = 1$ and $y = 0$.

On the other hand, the NOT gate is reversible, as $\neg x = 0$ implies $x = 1$, and vice-versa. Incidentally, we have that AND$'$ is reversible as well. This is because we carry the (otherwise unknown) inputs into the output, and that allows us to reconstruct the original state. In fact, this technique is commonly employed to obtain reversible transformations from irreversible gates.

If, in the general case, classical gates are irreversible, then what are the basic building blocks of quantum computing? In other words, what are the reversible quantum gates that we need in order to perform quantum computations? We already know that the NOT gate is reversible. In quantum computing, the gate that performs negation is called the *Pauli-X* gate and is defined as

$$X = \begin{array}{c} \\ |0\rangle \\ |1\rangle \end{array} \begin{array}{c} \begin{array}{cc} |0\rangle & |1\rangle \end{array} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{array}.$$

Naturally, the X gate acts on a single qubit. From now on, we will omit the basis state labels when defining single-qubit gates. There are two more Pauli gates. Specifically, the *Pauli-Y* and *Pauli-Z* gates, defined as

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

The operations performed by these two gates are much less intuitive than the one performed by X. In particular, it is the first time that we encounter complex entries in a transformation. To better visualize the effects of Y and Z, we introduce a graphical representation of qubits called the *Bloch sphere*.

The Bloch sphere We know that a generic quantum state $|\psi\rangle$ can be written as a linear combination of its basis states. In particular, if $|\psi\rangle$ is the state of a qubit, it can be written as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where $\alpha, \beta \in \mathbb{C}$. Because they are the magnitudes of a quantum state vector, it must be that $|\alpha|^2 + |\beta|^2 = 1$. We can therefore write

$$|\psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle\right),$$

for some $\gamma, \theta, \varphi \in \mathbb{R}$. The factor of $e^{i\gamma}$ has no observable effects on the state [11] and thus can be done away with. That leaves us with

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle.$$

If interpreted as radial coordinates, φ and θ determine a point on the three-dimensional unit sphere. We fix $|0\rangle$ and $|1\rangle$ at the poles. This representation of a quantum state vector is what we call the Bloch sphere representation.

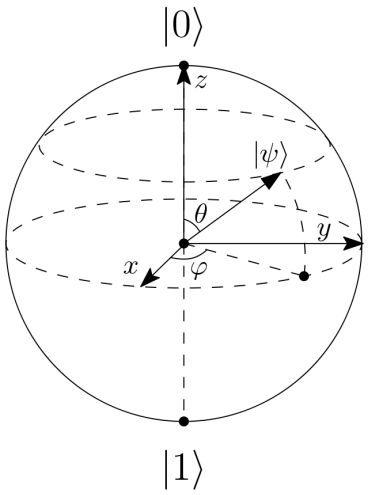


Figure 3.1: The Bloch sphere can be used to intuitively represent the state $|\psi\rangle$ of a single qubit.

It is now much easier to explain the effects of the Pauli gates. The X gate "flips" (rotates by π) a vector on the sphere around the x axis. It is clear that a vector at $|0\rangle$ is sent to $|1\rangle$ and vice-versa. The Y gate performs a similar operation, only this time around the y axis. Finally, the Z gate performs a π-degree rotation around the z (vertical) axis.

The Z gate can be generalized to the *rotation gate* R_φ, which will be fundamental when, in the next chapter, we discuss the quantum Fourier transform. The R_φ gate is

defined as follows:

$$R_\varphi = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix}.$$

If we set $\varphi = \pi$, we have indeed $e^{i\pi} = -1$ and thus $R_\pi = Z$. Intuitively, the R_φ gate performs a φ-degree rotation around the z axis of a vector on the Bloch sphere.

Another fundamental single-qubit gate is the *Hadamard* gate, which is defined as

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

The Hadamard gate is perhaps the most widely used single-qubit gate. The reason behind this is that the Hadamard gate is excellent for creating superpositions of states, as shown by its action on the basis states:

$$|0\rangle \xrightarrow{H} \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle,$$

$$|1\rangle \xrightarrow{H} \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle.$$

Both ending states are balanced superpositions which have equal probability of being observed in either basis state. The only difference between them is that they have different *phase*, i.e. they are rotated differently around the Bloch sphere's z axis. However, because phase does not affect probabilities, we will not concern ourselves with these details.

Note that the Hadamard gate closely resembles the coin toss operation that we tried to define in Example 3.1.8. In fact, if we apply H to the state of a quantum coin (and since a quantum coin is a qubit, we can, at least in principle), we find that it behaves exactly like a coin toss. Why is H a valid transformation, if T was not? The reason lies exactly in the phase shift operated by H, which does not affect probabilities, but allows the reconstruction of the original state.

So far we have only seen single-qubit gates. We start discussing two-qubit transformations with a quantum gate that allows us to introduce another fundamental concept of quantum computation. This gate is the *controlled NOT gate*, which is defined as follows:

$$\text{CNOT} = \begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}.$$

Let us try and decode this matrix. If the first qubit is 0, then CNOT does nothing (note that the upper-left 2×2 submatrix coincides with the I_2 identity matrix). On the other hand, if the first qubit is 1, then the second qubit is negated (the bottom-right 2×2 submatrix is exactly X). We can describe the action of CNOT on a generic two-qubit state $|x\rangle|y\rangle$ as

$$|x\rangle|y\rangle \xrightarrow{\text{CNOT}} |x\rangle|x \oplus y\rangle.$$

In other words, CNOT performs the NOT operation on a *target* qubit, conditionally on the state of a *control* qubit. The notion of *control* is general and can be applied to any reversible gate. For example, we can have a controlled R_φ gate:

$$CR_\varphi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\varphi} \end{bmatrix}.$$

In general, a n-qubit reversible gate U can be added a control qubit by defining a new $(n+1)$-qubit gate as the transformation performed by the matrix

$$CU = \begin{bmatrix} I_{2^n} & 0_{2^n} \\ 0_{2^n} & U \end{bmatrix},$$

where I_{2^n} is the $2^n \times 2^n$ identity matrix and 0_{2^n} is the $2^n \times 2^n$ null matrix. U is the $2^n \times 2^n$ matrix that describes gate U. This new gate acts on generic states as follows:

$$|0\rangle|\varphi^n\rangle \xrightarrow{CU} |0\rangle|\varphi^n\rangle,$$
$$|1\rangle|\varphi^n\rangle \xrightarrow{CU} |1\rangle \otimes U|\varphi^n\rangle.$$

That is, CU performs U on the n qubits denoted by φ^n only if the first qubit is 1. By adding further control qubits to a controlled gate, we can make the final result depend on the state of an arbitrarily large number of controls.

In fact, the next gate we are going to examine is the doubly-controlled NOT gate, also known as *Toffoli gate*. It is defined as follows:

$$\text{Toffoli} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

This gate behaves just like CNOT, with the only difference that both the first and second qubits must be 1 in order for the third qubit to be negated. Simply put, the Toffoli gate acts on a generic state as follows:

$$|xy\rangle|z\rangle \xrightarrow{\text{Toffoli}} |xy\rangle|(x \wedge y) \oplus z\rangle.$$

The Toffoli gate is particularly significant in that it is *universal* for classical computation. That is, every possible classical gate can be described in terms of Toffoli gates. We will avail ourselves of this property in the next chapter, when we try and translate classical circuits to their quantum counterparts.

The last gate we see is the *swap gate*. As one might expect, this gate acts on two qubits, swapping their states. A matrix representation is given as follows:

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

The action of SWAP on the basis states is straightforward:

$$|00\rangle \xrightarrow{\text{SWAP}} |00\rangle, \quad |01\rangle \xrightarrow{\text{SWAP}} |10\rangle,$$

$$|10\rangle \xrightarrow{\text{SWAP}} |01\rangle, \quad |11\rangle \xrightarrow{\text{SWAP}} |11\rangle.$$

3.2.2 Quantum Circuits

It is now time to start assembling reversible quantum gates to obtain the quantum counterpart of boolean circuits, i.e. *quantum circuits*. In Section 1.2 we saw that classical circuits are nothing more than directed acyclic graphs. Their nodes can be labeled with NOT, AND or OR gates and their output depends on these labels.

Can this model also be used to represent quantum circuits? The short answer is *no*. The reason behind this is that classical circuits allow the *fan-in* and the *fan-out* of wires. That is, respectively, the merging of two distinct wires into a single wire through a gate and the duplication of a wire along with the value it carries. It is fairly easy to show that these two operations go against the fundamental nature of quantum systems.

First of all, the fan-in of wires is clearly not an injective operation and as such cannot be a reversible transformation on the system. Furthermore, as we anticipated in Section 3.1.2, it is impossible to copy a generic quantum state exactly, so the fan-out of wires is also impossible in quantum circuits. Let us examine the *no-cloning theorem* in more detail:

Theorem 3.2.1 (No-cloning theorem)**.** *No reversible quantum gate exists that, given a qubit in a generic state and a blank qubit, copies the first qubit to the second.*

Proof. Suppose such gate exists. We call it C. A generic single-qubit state $|\varphi\rangle$ can be written as $|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle$, for some adequate $\alpha, \beta \in \mathbb{C}$. Because C can copy any quantum state, it must act on $|\varphi\rangle$ and $|0\rangle$ as follows:

$$(\alpha|0\rangle + \beta|1\rangle) \otimes |0\rangle \xrightarrow{C} (\alpha|0\rangle + \beta|1\rangle) \otimes (\alpha|0\rangle + \beta|1\rangle) = \alpha|00\rangle + \beta|11\rangle.$$

Furthermore, if C is a valid quantum gate, then it is also linear. A transformation U is linear if and only if

$$U(|\varphi\rangle + |\psi\rangle) = U|\varphi\rangle + U|\psi\rangle,$$
$$U(\lambda|\varphi\rangle) = \lambda U|\varphi\rangle.$$

However, consider the state $|\varphi\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and apply C to it. Thanks to the linear properties of C we must have

$$\begin{aligned} C\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle\right) &= \frac{1}{\sqrt{2}} C((|0\rangle + |1\rangle) \otimes |0\rangle) \\ &= \frac{1}{\sqrt{2}} C(|0\rangle|0\rangle + |1\rangle|0\rangle) \\ &= \frac{1}{\sqrt{2}}(C(|0\rangle|0\rangle) + C(|1\rangle|0\rangle)) \\ &= \frac{|00\rangle + |11\rangle}{\sqrt{2}} \neq \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle + |1\rangle}{\sqrt{2}}. \end{aligned}$$

This contradicts the very definition of C, which therefore cannot be a valid reversible quantum gate. □

Because wires in quantum circuits cannot be merged nor split, we have that a one-to-one correspondence exists between qubits and wires (whereas in classical circuits this is not always the case). In abstract terms, a quantum wire ends up representing a single qubit which evolves in time.

We can therefore apply reversible quantum gates to wires, which amounts to performing unitary transformations on the corresponding qubits. We can now give a formal definition of *quantum circuit*.

Definition 3.2.3 (Quantum circuit)**.** *A* quantum circuit Q *is an ordered list of instructions, which can include:*

- $\text{Input}(i_j)$*, i.e. the introduction of a new input wire* i_j*.*
- $\text{Init}_0(q)$ *and* $\text{Init}_1(q)$*, i.e. the introduction of a new wire* q*, in the* $|0\rangle$ *or* $|1\rangle$ *basis state, respectively.*
- $[\text{Gate}](q_1, q_2, \ldots, q_n)(c_1, c_2, \ldots, c_m)$*, i.e. the application of the* n*-qubit gate* $[\text{Gate}]$ *to* n *wires* $q_1, q_2, \ldots, q_n$*, with control qubits* $c_1, c_2, \ldots, c_m$*. In the case of uncontrolled gates, the second pair of parentheses is omitted.*
- $\text{Measure}(q_1, q_2, \ldots, q_n)$*, i.e. the measurement of one or more wires* $q_1, q_2, \ldots, q_n$*.*
- $\text{Output}(q)$*, i.e. the designation of a wire* q *as an output wire.*

Gates, measurements and output designations can only be applied to wires that have already been introduced, either as inputs or via initialization.

Because quantum gates are transformations on qubits, we need not specify where their outputs must go. This way, a circuit ceases to be a graph and can be represented as a sequence of operations. It is also clear, from this definition, that a quantum circuit is not necessarily reversible (inputs and outputs can be defined freely and without constraints). This is not a problem if the circuit is self-contained. However, if the circuit is to be used as a subroutine in a larger circuit, then it must be reversible.

One last note, on the Init instruction. The initial states of the wires introduced by this instruction only depend on the definition of the individual circuit, and not on the circuit's input. They are usually employed as an aid to computation and seldom figure in the circuit's output. Because of that, they are called *ancillary wires* or simply *ancillae*.

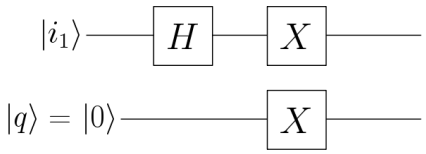
Although quantum circuits are just lists, we display them through a slightly more sophisticated graphical representation. Namely, each wire (it does not matter whether it is an input or an ancilla) is represented by a horizontal line. The following is the graphical representation of the $Q = (\text{Input}(i_1), \text{Init}_0(q), \text{Output}(i_1), \text{Output}(q))$ circuit:

$$|i_1\rangle \text{———————————}$$

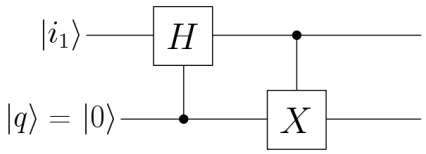
$$|q\rangle = |0\rangle \text{———————————}$$

Gates are represented as boxes and are identified by their symbol. As one might expect, they are placed directly on top of the wires they act upon. We add Hadamard and Pauli-X gates to the previous example, to obtain

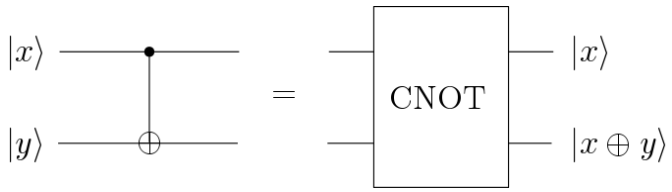
$$Q = (\mathrm{Input}(i_1), \mathrm{Init}_0(q), H(i_1), X(i_1), X(q), \mathrm{Output}(i_1), \mathrm{Output}(q)),$$

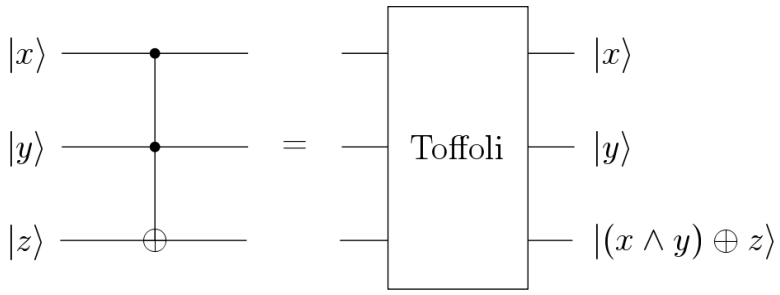


We know that quantum gates can have controls. A control is usually denoted by a small circle, connected by a vertical line to its gate. Take as an example the circuit $Q = (\mathrm{Input}(i_1), \mathrm{Init}_0(q), H(i_1)(q), X(q)(i_1), \mathrm{Output}(i_1), \mathrm{Output}(q))$:

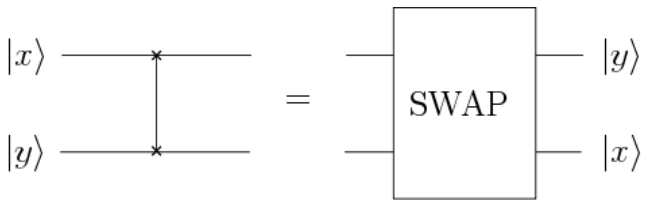


Because they are the most commonly employed controlled gates, CNOT and Toffoli have special, ad-hoc notation:

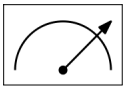




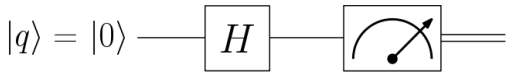
Even though both gates are just controlled negations (and therefore could be written as controlled X gates), we will refer to their application as $\mathrm{CNOT}(y)(x)$ and $\mathrm{Toffoli}(z)(x, y)$, for the sake of clarity. The SWAP gate also has a symbol of its own:



Lastly, measurements are represented by the following symbol:

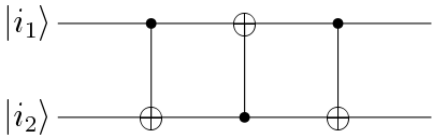


Consider, for example, the circuit $Q = (\text{Init}_0(q), H(q), \text{Measure}(q), \text{Output}(q))$. Q can be represented as follows:



This circuit has no inputs, and it outputs one of the basis states, with equal probability (it outputs *exactly* one of the basis states, *not* a balanced superposition). Note the double line at the end, which is commonly employed to denote the collapse of a wire to a classical state. We conclude this section with some examples.

Example 3.2.4: Consider the following circuit Q:



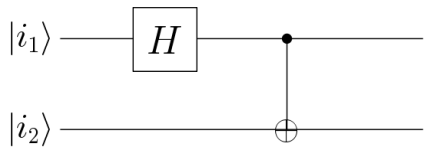
On input $|0\rangle|1\rangle$, Q produces the following output:

$$|0 \oplus (1 \oplus 0)\rangle \otimes |(1 \oplus 0) \oplus (0 \oplus (1 \oplus 0))\rangle = |1\rangle|0\rangle.$$

In other words, the circuit swapped the two inputs. If we test Q on the remaining basis states, we find that it acts *exactly* like the SWAP gate. In fact, swap operations are usually implemented this way, using three CNOT gates.

Example 3.2.5: Consider the following circuit:

$$Q = (\text{Input}(i_1), \text{Input}(i_2), H(i_1), \text{CNOT}(i_2)(i_1), \text{Output}(i_1), \text{Output}(i_2)),$$



On input $|00\rangle$, Q outputs the following superposition:

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}}.$$

Here we can recognize an abstraction in terms of qubits of the entangled state presented in Section 3.1.4. This is one of the four states known as *Bell states*, named after physicist John S. Bell. These are the following maximally entangled states:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}, \quad |\Phi^-\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}},$$

$$|\Psi^+\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}, \quad |\Psi^-\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}.$$

Note that by varying the inputs of Q we can obtain the first three states and $-|\Psi^-\rangle$.

Chapter 4

A Quantum Solution for Abelian Groups

Now that we have an essential, but solid foundation of quantum computing, it is time to dwell into some of the quantum algorithms for solving the HSP. In Chapter 2 we announced that efficient quantum solutions exists for the HSP on specific kinds of groups. This is the case with Abelian groups. We remind the reader of Definition 1.1.5, in which we say that a group is Abelian when its operation is commutative.

In this section, we first present an algorithm that solves the HSP on the cyclic additive groups of integers modulo N. Later, we generalize this case to obtain an algorithm for generic Abelian groups.

4.1 Preliminaries

Before we start, however, there are two important matters that need to be discussed. The first one is the *quantum Fourier transform*, a quantum transformation which will be essential in both algorithms. The second one is the conversion of classical circuits to quantum circuits, a process that allows us to keep using coset-separating circuits as a means to pass a coset-separating function to an HSP functional.

4.1.1 The Quantum Fourier Transform

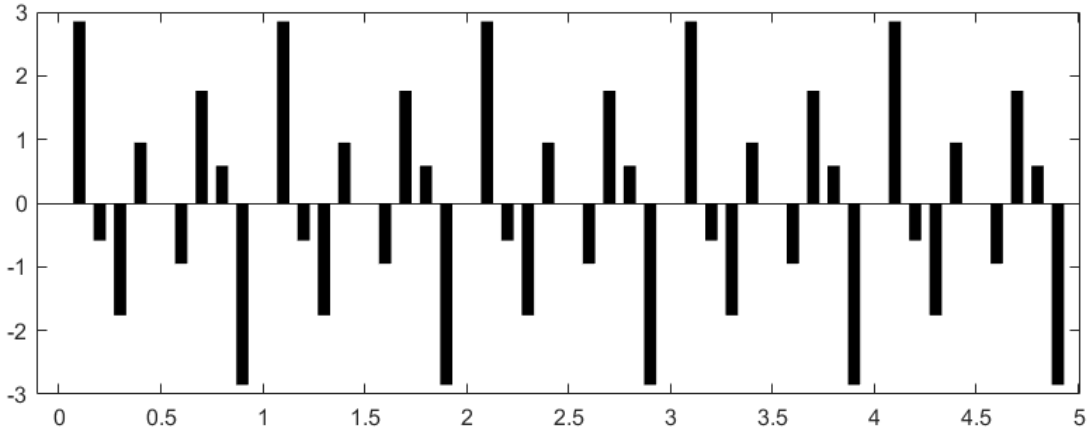
The *quantum Fourier transform* (QFT) is a transformation whose role is paramount in quantum algorithmics. As a matter of fact, the QFT lies at the heart of quantum algorithms such as Shor's factoring algorithm [12], the quantum phase estimation algorithm and, as one might imagine, the forthcoming algorithms for the Abelian HSP.

Starting, as usual, from a purely mathematical standpoint, we give the definition of *discrete Fourier transform* (DFT). What the DFT does is, given a complex vector $[x_0, x_1, \ldots, x_{N-1}]^T$ of length N, return a complex vector $[y_0, y_1, \ldots, y_{N-1}]^T$ such that:

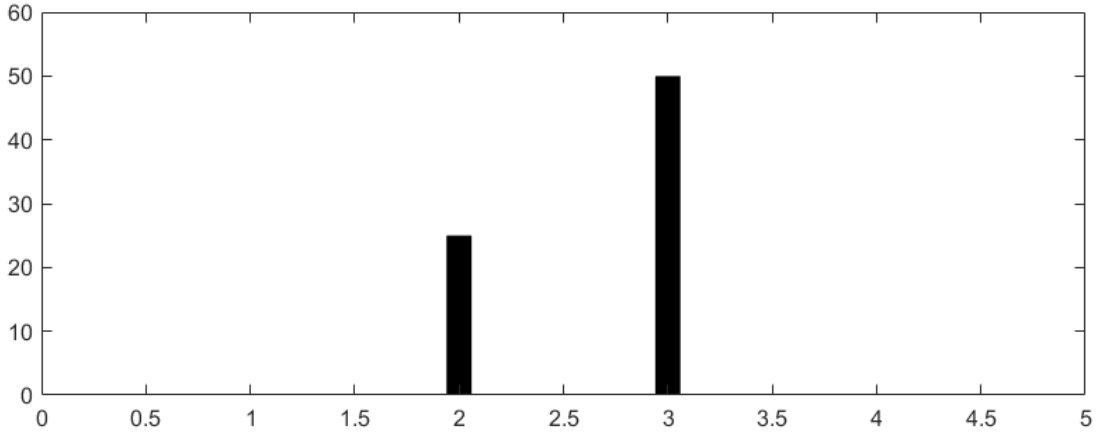
$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{\frac{2\pi i j k}{N}}. \tag{4.1}$$

What does the DFT compute? Intuitively, if we take the various $x_0, x_1, \ldots, x_{N-1}$ to be equally spaced samples of a wave-like function of time $f : \mathbb{R} \to \mathbb{C}$, then the elements $y_0, y_1, \ldots, y_{N-1}$ represent the amplitudes with which equally spaced frequencies appear in f. To more easily grasp this idea, consider the following example:

Example 4.1.1: Let $f(t) = \sin(2\pi 2t) + 2\sin(2\pi 3t)$ be a wave-like function of time. One can already tell, by looking at this definition, that f's main frequencies are 2 Hz and 3 Hz. Suppose we sample f every 0.1 s for 5 s. We obtain 50 values, which we can plot as follows:



We now apply the DFT on $[f(0.1), f(0.2), \ldots, f(4.9)]^T$ to obtain a vector of 50 complex values. We plot the magnitudes of these values and we obtain the following plot (which we scale and clip to make the results more comprehensible):



Note how all values but the ones corresponding to 2 Hz and 3 Hz are negligible and do not show in the results. This is exactly the outcome we anticipated. Also note that the magnitude corresponding to 3 Hz is (correctly) twice the magnitude of 2 Hz.

Now, the quantum Fourier transform computes the exact same transformation as the DFT, the only difference being that the QFT is applied on quantum state vectors, whereas the DFT is applied to generic complex vectors. Given an orthonormal basis $|0\rangle, |1\rangle, \ldots, |N-1\rangle$, the QFT is a linear operator that performs the following action on each of the basis states:

$$|j\rangle \longrightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{\frac{2\pi i j k}{N}} |k\rangle. \tag{4.2}$$

Alternatively, given an arbitrary state $|\gamma\rangle$ in the orthonormal basis, we have:

$$|\gamma\rangle = \sum_{j=0}^{N-1} x_j |j\rangle \longrightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} y_k |k\rangle,$$

where the y_k are the results of the application of the DFT on the amplitudes of $|\gamma\rangle$, as described in Equation 4.1. Of course, we need a computational definition of the QFT. Namely, we need to prove that for a generic group family $\{\mathbb{G}_N\}_{N\in\mathbb{N}}$ encoded through a suitable ρ, there exists a quantum circuit family $\{Q_N\}_{N\in\mathbb{N}}$ such that Q_N efficiently computes the QFT on the encoded elements of $\mathbb{G}_N$.

Here *efficiently* means that if ρ is LP for some function f, then the number of gates required to build Q_N is polynomial in $f(N)$, i.e. polynomial in the size of the input. We proceed to show how to build such circuit family for groups of order $|\mathbb{G}_N| = 2^n$ for some n (in this case, ρ can be surjective), leaving the more general case for later.

Let $|0\rangle, |1\rangle, \ldots, |2^n - 1\rangle$ be the computational basis of the n-qubit quantum computer on which we wish to implement Q_N. For any basis state $|j\rangle$, let $j_1 j_2 \ldots j_n$ denote the binary representation of j according to ρ and let $0.j_l j_{l+1} \ldots j_m$ denote the binary fraction $j_l/2 + j_{l+1}/4 + \cdots + j_m/2^{m-l+1}$. We can expand Equation 4.2 as follows:

$$\begin{aligned}
|j\rangle \longrightarrow & \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{\frac{2\pi i j k}{2^n}} |k\rangle \\
= & \frac{1}{2^{n/2}} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} e^{2\pi i j \left(\sum_{l=1}^{n} k_l 2^{-l}\right)} |k_1 \ldots k_n\rangle \\
= & \frac{1}{2^{n/2}} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} \bigotimes_{l=1}^{n} e^{2\pi i j k_l 2^{-l}} |k_l\rangle \\
= & \frac{1}{2^{n/2}} \bigotimes_{l=1}^{n} \left(\sum_{k_l=0}^{1} e^{2\pi i j k_l 2^{-l}} |k_l\rangle \right) \\
= & \frac{1}{2^{n/2}} \bigotimes_{l=1}^{n} \left(|0\rangle + e^{2\pi i j 2^{-l}} |1\rangle \right) \\
= & \frac{(|0\rangle + e^{2\pi i 0.j_n}|1\rangle)(|0\rangle + e^{2\pi i 0.j_{n-1} j_n}|1\rangle) \ldots (|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle)}{2^{n/2}}.
\end{aligned}$$

The last line of this expansion is known as the *product representation* of the QFT. This form of the QFT is particularly useful in that a definition of the Q_N circuit we seek follows naturally from it. Let us begin the construction of Q_N. Consider the Hadamard gate H and the R_k gate (a more specific form of the rotation gate). These gates are defined as

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}.$$

Let us apply H to the first qubit of a state $|j\rangle = |j_1 j_2 \ldots j_n\rangle$. The Hadamard gate sends $|j\rangle$ to the state

$$\frac{1}{\sqrt{2}} \left(|0\rangle + e^{2\pi i 0.j_1} |1\rangle \right) |j_2 \ldots j_n\rangle.$$

This is true because if $j_1 = 1$, then $0.j_1 = 1/2$ and $e^{\pi i} = -1$, whereas if $j_1 = 0$ we have $e^0 = 1$. It is clear that this result coincides with the definition of H. Let us apply the R_2 gate on $|j_1\rangle$, using $|j_2\rangle$ as control. We have a resulting state of

$$\frac{1}{\sqrt{2}} \left(|0\rangle + e^{2\pi i 0.j_1 j_2} |1\rangle \right) |j_2 \ldots j_n\rangle.$$

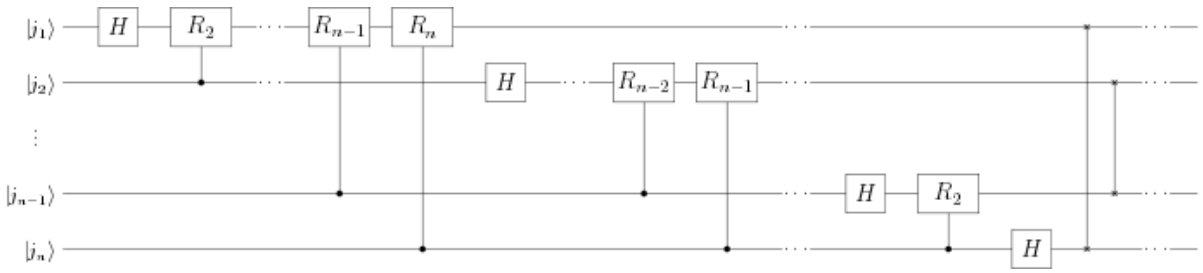


Figure 4.1: The quantum circuit that computes the QFT on n qubits. The outputs are exactly the factors of the product representation.

Note that the controlled-R_2 added j_2 to the binary fraction. We repeat this operation, applying R_i on $|j_1\rangle$ with $|j_i\rangle$ as control for all the remaining i. We end up in the state

$$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle\right)|j_2 \ldots j_n\rangle.$$

In this state we recognize $(|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle)$, the last factor of the product representation. Now we operate similarly on $|j_2\rangle$. We first we apply H, followed by R_{i-1} controlled by $|j_i\rangle$ for all $2 < i \leq n$. We end up in the state

$$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle\right)\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i 0.j_2 j_3 \ldots j_n}|1\rangle\right)|j_3 \ldots j_n\rangle,$$

where we recognize yet another factor of the product representation. By operating similarly with H and R_k on each of the remaining qubits, we obtain the final state

$$\frac{(|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle)\,(|0\rangle + e^{2\pi i 0.j_2 j_3 \ldots j_n}|1\rangle)\ldots(|0\rangle + e^{2\pi i 0.j_n}|1\rangle)}{\sqrt{2^n}}.$$

This is exactly the product representation of the QFT acting on $|j\rangle$, up to the order of the factors. We can easily reorder the resulting state by applying swap gates between the first and last elements, the second and second-to-last elements and so on. Note that we have in fact built Q_N for $N = 2^n$ using only unitary quantum gates. As such, Q_N is a valid unitary transformation.

It is time to address the efficiency of this construction of Q_N. For each qubit $|j_i\rangle$, we employed one Hadamard gate and $(n - i)$ controlled-R_k gates. Thus the main body of the computation requires $n(n + 1)/2$ gates. We also employed $n/2$ swap gates, each implemented using three CNOT gates. The total amount of gates required adds up to $n(n + 1)/2 + 3n/2$, which is $\Theta(n^2)$. In other words, Q_N is efficient, as it provides a polytime algorithm for computing the QFT on groups of order $|\mathbb{G}_N| = 2^n$.

Of course, not all the groups we have handled so far have order 2^n for some $n \in \mathbb{N}$ (in fact, only the group family $\{\mathbb{B}^n, \oplus\}_{n\in\mathbb{N}}$ from Simon's problem does, for all n). We need to find a way to build Q_N for generic values N. The following result [8] tells us that this can be done for all odd N:

Theorem 4.1.1. *Given an odd integer $N \geq 13$, and any $0 < \epsilon \leq \sqrt{2}$, we can compute Q_N with error bounded by ϵ, using at most $\lceil 12.53 + 3\log\frac{\sqrt{N}}{\epsilon}\rceil$ qubits. The algorithm has an operation complexity of*

$$O\left(\log\frac{\sqrt{N}}{\epsilon}\left(\log\log\frac{\sqrt{N}}{\epsilon} + \log\frac{1}{\epsilon}\right)\right).$$

Furthermore, the induced probability distributions D_v from the output and D from $Q_N|\varphi\rangle|\psi\rangle$ satisfy

$$|D_v - D| \leq 2\epsilon + \epsilon^2.$$

That is, we can approximate the QFT very well with an efficient circuit. Furthermore, for odd values of $N < 13$, ad-hoc circuits can be built. Now that we have an efficient Q_N for $N = 2^n$ and odd N, we can build Q_N for any N. Consider a composite N. There exist A and B that are either a power of 2 or odd and for which $N = AB$. We have [10] that computing Q_N amounts to applying the following transformation:

$$Q_N = (U_B \otimes U_A)(Q_A \otimes Q_B),$$

where $U_B : |x \mod A\rangle \to |xB \mod A\rangle$ and $U_A : |x \mod B\rangle \to |xA \mod B\rangle$. Q_A and Q_B are covered by the previous results, so it follows that we can efficiently compute the QFT on any group of any order, with arbitrary precision.

4.1.2 Converting Classical Circuits

Another obstacle we need to overcome is the discrepancy between classical and quantum circuits. A significant part of the input to the HSP functional is the representation of a (classical) coset-separating circuit C. Therefore, if we wish to use and evaluate C in the context of a quantum solution to the HSP, we need it to comply with the requirements of the quantum circuit model.

In Section 1.2.2 we defined a boolean circuit as a labeled directed acyclic graph, whereas in Section 3.2.2 we saw that quantum circuits are essentially lists of operations on qubits, which must obey much stricter constraints. Namely:

- **Reversibility:** The gates employed in a quantum circuit must be reversible. However, this is not the case with the AND and OR gates in classical circuits.

- **Fan-out:** Due to the no-cloning theorem, the output of a quantum gate cannot be duplicated to be used as the input of multiple subsequent gates. However, in classical circuits, this technique (fan-out) is allowed and commonly employed.

We address these two problems in order. As for reversibility, we already know that the NOT operation is reversible and that it corresponds to the X gate. For convenience, however, we decide to express it through a CNOT gate. We remind the reader that the CNOT gate performs the following operation:

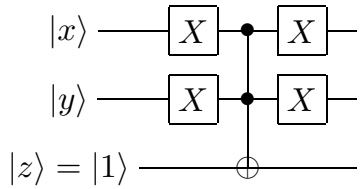
$$|x\rangle|y\rangle \xrightarrow{\text{CNOT}} |x\rangle|x \oplus y\rangle$$

By setting $y = 1$, we obtain as output $|x\rangle|x \oplus 1\rangle = |x\rangle|\neg x\rangle$, which corresponds to a copy of the input, plus the output expected from the NOT gate.

What we need to do now is express AND and OR in terms of reversible operations. We are helped in this task by the Toffoli gate. We remind the reader that the Toffoli gate performs the following operation:

$$|xy\rangle|z\rangle \xrightarrow{\text{Toffoli}} |xy\rangle|(x \wedge y) \oplus z\rangle$$

By setting $z = 0$, we obtain as output $|xy\rangle|x \wedge y\rangle$, i.e. we can implement the AND gate at the cost of one Toffoli gate and one ancilla. The construction of a subcircuit that computes the OR gate follows naturally once we apply De Morgan's laws:



With $z = 1$, we obtain an output of $|xy\rangle|1 \oplus (\neg x \wedge \neg y)\rangle = |xy\rangle|\neg(\neg x \wedge \neg y)\rangle = |xy\rangle|x \vee y\rangle$. This means that we can implement the OR gate by using one Toffoli gate, four Pauli-X (uncontrolled NOT) gates and one ancilla.

It might seem unnecessary to express NOT gates through CNOT gates, just like it might seem unnecessary to negate the first two outputs in the conversion of an OR gate. These two choices are justified when we consider gates with fan-out. The reversible versions of NOT, AND and OR preserve the state they act upon, as the result is output to a new ancilla every time. This way, the output of a gate can be acted upon multiple times by subsequent gates, without the need to copy any information.

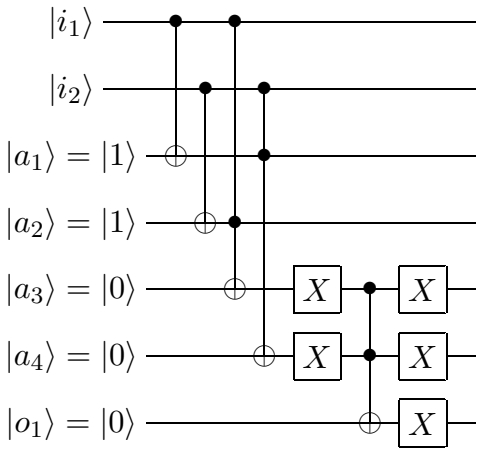


Figure 4.2: The circuit from Figure 1.1 once converted to its quantum counterpart. The inputs are qubits i_1 and i_2, the output is qubit o_1.

Let us explain this idea through an example. Consider, without loss of generality, an AND gate on x and y with a fan-out of m. The no-cloning theorem forbids us from copying its output, so what we do is:

1. Apply Toffoli on $|xy\rangle|0\rangle$ to obtain $|xy\rangle|x \wedge y\rangle$.

2. Apply U_0 on $|x \wedge y\rangle$, saving the result to a new ancilla a_0 and leaving $|x \wedge y\rangle$ unaltered.

3. Apply the remaining gates $U_1, U_2, \ldots, U_{m-1}$ to $|x \wedge y\rangle$, each time saving the result to a new ancilla $a_1, a_2, \ldots, a_{m-1}$ and leaving $|x \wedge y\rangle$ unaltered.

This way, n generic classical gates can be replaced by $O(n)$ reversible gates (X, CNOT and Toffoli) using $O(n)$ ancillae. Before we can present an actual algorithm, however, we need to address one last detail. Specifically, given a circuit C, we need to decide *in which order* to replace C's gates. A classical circuit is nothing more than a directed acyclic graph. As such, it can be *topologically sorted*.

Definition 4.1.1 (Topological sorting)**.** *Let $G = (V, E)$ be a directed acyclic graph such that $|V| = n$. A* topological sorting *of G consists of an ordering $S = v_{s_0}, v_{s_1}, \ldots, v_{s_{n-1}}$ (where $0 \leq s_i < n$ for all i) of all the vertices in V such that if $(v_{s_i}, v_{s_j}) \in E$, then v_{s_i} occurs before v_{s_j} in S.*

It is clear that once C is topologically sorted, we can convert its gates in an orderly fashion, without the risk of disrupting the correct flow of computation. More specifically, we proceed as follows:

Algorithm 3: Conversion of classical circuits

Input: A classical circuit C, given as a labeled DAG.
Output: A reversible circuit U_C that computes the same function as C.

Compute L, a topological sorting of C;
Initialize U_C as an empty list;

for $v \in L$ **do**
  switch v's label **do**
    case i_k **do**
      Append Input(i_k) to U_C;
    case NOT **do**
      Let a be the input node;
      Append $(\text{Init}_1(v), \text{CNOT}(v)(a))$ to U_C;
    case AND **do**
      Let a and b be the input nodes;
      Append $(\text{Init}_0(v), \text{Toffoli}(v)(a, b))$ to U_C;
    case OR **do**
      Let a and b be the input nodes;
      Append $(\text{Init}_1(v), X(a), X(b), \text{Toffoli}(v)(a, b), X(a), X(b))$ to U_C;
  if v is an output node **then**
    Append Output(v) to U_C;

return U_C;

Note that a directed acyclic graph $G = (V, E)$ can be sorted in $O(|V|^2)$ time [4]. Therefore, the algorithm converts a circuit C of n classical gates into a quantum circuit U_C of $O(n)$ gates in $O(n^2)$ time.

Now that we have quantum circuits that compute the quantum Fourier transform and a way to convert coset-separating circuits, we can move on to the next section, where we put these results to use.

4.2 Cyclic Additive Case

We start to examine some of the quantum algorithms that make use of the QFT to efficiently implement the HSP functional. We start with the simplest of cases, that is, the HSP functional on the family $\{\mathbb{Z}_N\}_{N \in \mathbb{N}}$ of additive groups of integers modulo N.

In Section 1.2.1, we saw that we can easily build a PLP function ρ that encodes this group family using simple binary representations of integers. Let $\text{bin}(N), \lceil C_f \rceil$ be the inputs to an instance of the HSP functional on $\{\mathbb{Z}_N\}_{N\in\mathbb{N}}$, where C_f is a coset-separating circuit for some $\mathbb{H}_N = \langle d \rangle \leq \mathbb{Z}_N$ with respect to ρ. Let $|\mathbb{H}_N| = M$.

We know, from the previous section, that we can convert C_f to an equivalent quantum circuit U_f of size linear in the number of gates of C_f. In particular, in the following discussion we will assume that we already have U_f at our disposal and that it performs the following transformation:

$$|x\rangle|y\rangle \xrightarrow{U_f} |x\rangle|y \oplus f(x)\rangle,$$

where $|x\rangle$ and $|y\rangle$ are two n-qubit registers such that $n \geq \log_2 N$. Note that this kind of transformation can be easily obtained by slightly modifying Algorithm 3, without affecting its complexity.

Now, before we present the actual algorithm, let us provide a specialization of the QFT on cyclic groups:

Definition 4.2.1 (Cyclic QFT)**.** *The* Cyclic QFT *(CQFT) is the unitary operator Q_N on a register with $n \geq \log_2 N$ qubits defined as*

$$Q_N = \frac{1}{\sqrt{N}} \sum_{j,k=0}^{N-1} e^{\frac{2\pi i j k}{N}} |k\rangle\langle j|.$$

As one can see from this definition, we are describing our computation in terms of the $\mathbb{G} = \{|0\rangle, |1\rangle, \ldots, |N-1\rangle\}$ basis. Note that the elements of $\mathbb{H}$ can also be described through the $\mathbb{H} = \{|0\rangle, |d\rangle, |2d\rangle, \ldots, |(M-1)d\rangle\}$ basis. Consider a quantum computer with two n-qubit registers, both initialized to $|0\rangle$. We start with the application of Q_N to the first register:

$$|0\rangle|0\rangle \xrightarrow{Q_N \text{ on 1st}} \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} |j\rangle|0\rangle. \tag{4.3}$$

The state we obtain follows from the definition of the QFT on a basis state (see expression 4.2). We now apply U_f on the two registers. Note that because the second register is $|0\rangle$, this amounts to computing f on the superposition generated by Q_N and storing the result in the second register. We obtain the following entangled state:

$$\frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} |j\rangle|0\rangle \xrightarrow{U_f} \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} |j\rangle|f(j)\rangle. \tag{4.4}$$

Next, we measure the second register, collapsing the state. The measured register is left with $f(x)$, for some $x \in \{0, 1, \ldots, N-1\}$, while the first register is left with a superposition of only those values j for which $f(j) = f(x)$. That is, the first register contains a superposition of all the values in the coset $x + \mathbb{H}$. We no longer need the second register, so we do away with it. We end up in the state

$$\frac{1}{\sqrt{M}} \sum_{h \in \mathbb{H}} |x + h\rangle = \frac{1}{\sqrt{M}} \sum_{s=0}^{M-1} |x + sd\rangle. \tag{4.5}$$

This follows from $\mathbb{H} = \langle d \rangle$. We apply Q_N one more time to obtain:

$$\frac{1}{\sqrt{M}} \sum_{s=0}^{M-1} |x + sd\rangle \xrightarrow{Q_N} \frac{1}{\sqrt{M}} \sum_{s=0}^{M-1} \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{\frac{2\pi i(x+sd)k}{N}} |k\rangle \tag{4.6}$$

$$= \frac{1}{\sqrt{MN}} \sum_{k=0}^{N-1} e^{\frac{2\pi i x k}{N}} |k\rangle \sum_{s=0}^{M-1} e^{\frac{2\pi i s d k}{N}} \tag{4.7}$$

$$= \frac{1}{\sqrt{MN}} \sum_{k=0}^{N-1} e^{\frac{2\pi i x k}{N}} |k\rangle \sum_{s=0}^{M-1} \left(e^{\frac{2\pi i k}{M}}\right)^s. \tag{4.8}$$

This last step is true because $|\mathbb{G}_N|/|\mathbb{H}_N| = N/M = d$ and thus $d/N = 1/M$. The last sum constitutes a geometric series, which evaluates to

$$\sum_{s=0}^{M-1} \left(e^{\frac{2\pi i k}{M}}\right)^s = \begin{cases} 0 & \text{if } M \nmid k, \\ M & \text{if } M | k. \end{cases}$$

Which means that in expression 4.8 we can ignore all the k that are not multiples of M. We obtain the following simplified final state:

$$\frac{1}{\sqrt{d}} \sum_{t=0}^{d-1} e^{\frac{2\pi i x t M}{N}} |tM\rangle.$$

We proceed to measure the one remaining register. What we get is a multiple of M, i.e. an integer in $\{0, M, \ldots, (d-1)M\}$, with uniform probability. It is clear that if we were to obtain M, we could easily output a generator $d = N/M$ for $\mathbb{H}$. We therefore run the previous procedure multiple times, each time obtaining a (not necessarily different) multiple of M. We then compute the gcd of these multiples, in the hope of finding *exactly* M. Let us estimate how many trials we need to have a high probability of success.

Assume we already have k multiples $t_1M, t_2M, \ldots, t_kM$, where $t_i \in \{0, 1, \ldots, d-1\}$ for all i. If $\gcd(t_1, \ldots, t_k) = 1$, then $\gcd(t_1M, \ldots, t_kM) = M$ and we can find a generator for $\mathbb{H}$. It can be proven [8] that:

Lemma 4.2.1. *Let $d \geq 2$ and $k \geq 2$. Let $t_1, t_2, \ldots, t_k$ be a number of randomly distributed integers such that $t_i \in \{0, 1, \ldots, d-1\}$, for all i. We have*

$$P(\gcd(t_1, t_2, \ldots, t_k) = 1) \geq 1 - \left(\frac{1}{2}\right)^{k/2}.$$

So few iterations are enough to obtain M with high probability. We now have all the pieces we need to implement the HSP functional on the $\{\mathbb{Z}_N\}_{N\in\mathbb{N}}$ group family:

Algorithm 4: F_{HSP} – Cyclic Abelian Case

Input: A binary representation $\text{bin}(N)$ and $\lceil C_f \rceil$ such that C_f separates cosets for $\mathbb{H}_N = \langle d \rangle \leq \mathbb{Z}_N$, for some $d \in \mathbb{Z}_N$, with respect to ρ.
Output: The representation $\rho(d, N)$, with probability at least $3/4$.

Use $\lceil C_f \rceil$ to build U_f, a reversible quantum circuit that computes C_f;
Initialize a list L;
for 8 times **do**
    Initialize $|\varphi\rangle|\psi\rangle$ to $|0\rangle|0\rangle$;
    Apply Q_N to $|\varphi\rangle$ with an approximation error of at most $\epsilon = 0.01$;
    Apply U_f to $|\varphi\rangle|\psi\rangle$;
    Apply Q_N to $|\varphi\rangle$ with an approximation error of at most $\epsilon = 0.01$;
    Measure $|\varphi\rangle$ to obtain tM, for some $t \in \{0, 1, \ldots, d-1\}$;
    Add tM to L;
$M \leftarrow \gcd(L)$;
$d \leftarrow N/M$;
return $\rho(d, N)$;

The probability of each of the eight iterations of returning a valid tM is at least $1 - (2\epsilon + \epsilon^2)$, by Theorem 4.1.1. Now, assume we have eight valid samples. By Lemma 4.2.1, the probability of getting the actual M from computing the GCD on the elements of L is $1 - (1/2)^4 = 15/16$. Therefore, the overall success probability of the previous algorithm is $(1 - 0.0201)^8(15/16) \approx 0.796 > 3/4$.

Let us consider the complexity of the algorithm. We consider $N = |\mathbb{Z}_N|$ as the size of the problem. Let $n = O(\log N)$ be the size of the elements of $\mathbb{Z}_N$, according to ρ. Thanks to Algorithm 3 from the previous section, we know that building U_f from C_f

requires time polynomial in the number of gates in C_f. Therefore, if C_f has an operation complexity of $p(n)$ for some polynomial p, then we can build U_f with $O(p(n))$ gates in $O(p^2)$ time. This entails that U_f has operation complexity (and can be built in time) $O(\text{polylog}(N))$.

After that, the *for* loop is iterated a constant number of times. Inside the loop, the most expensive operations are the applications of Q_N and U_f. As for U_f, we know already that its complexity is polynomial in n. Thanks to Theorem 4.1.1, we also know that Q_N has polynomial complexity (in n) as well. Lastly, the gcd can be computed with $O(\log N)$ divisions, each taking $O(\log N)$ time [2].

Putting these three pieces together, we can clearly see that the overall time complexity of the algorithm is $O(\text{polylog}(N))$, which is efficient.

4.3 Representation Theory

When, in Section 1.2, we discussed the encoding of group families, we saw that we could interpret generic Abelian groups as a direct sum of simple cyclic groups of the form $\mathbb{Z}_N$ (Theorem 1.2.1). It is therefore sensible, at this point, to wonder whether *a solution* to the HSP on families of generic Abelian groups can be obtained from the solution to the $\mathbb{Z}_N$ case.

Luckily, the answer to this question is *yes*. However, such construction of an implementation of the Abelian HSP functional is not as straightforward as one may imagine and requires some non-trivial concepts of *representation theory*, which we present in this section. We start with a couple of definitions.

Definition 4.3.1 (Automorphism)**.** *Let V be a vector space over a field $\mathbb{F}$. An* automorphism *on V is a bijective linear transformation of the form $\varphi : V \to V$.*

Definition 4.3.2 (General linear group)**.** *Let V be a vector space over a field $\mathbb{F}$. We write $GL(V)$ to denote the* general linear group *of V, i.e. the group of all the automorphisms on V under composition.*

With these definitions in mind, we can define the very concept of *representation*:

Definition 4.3.3 (Representation)**.** *Let $\mathbb{G}$ be a group. A* representation *of $\mathbb{G}$ consists of a vector space V (over a field $\mathbb{F}$) along with an homomorphism $\varrho : \mathbb{G} \to GL(V)$.*

For our purposes, $\mathbb{F}$ will be the field of complex numbers $\mathbb{C}$ and V will be finite-dimensional of dimension d. A representation allows us to represent the elements of $\mathbb{G}$ as square matrices. In fact, once we fix a basis for V, we have that for every $g \in \mathbb{G}$, $\varrho(g)$ corresponds to a unitary $d \times d$ matrix.

Notice that we employed ϱ, instead of the traditional ρ, to denote a representation. This is to distinguish a *representation* function from an *encoding* function as defined in Section 1.2.1. Although the two concepts may appear to be related (they both produce some kind of representation of group elements), they have nothing to do with each other. Specifically, a representation ϱ is a purely mathematical concept and has nothing to do with how group elements can be encoded to be processed by a computer.

Once we have a representation ϱ, we can associate a *character* to it.

Definition 4.3.4 (Character)**.** *Let ϱ be a representation for a group $\mathbb{G}$. The* character *associated to ϱ is referred to as χ_ϱ and is defined as $\chi_\varrho(g) = \mathrm{Tr}(\varrho(g))$ for all $g \in \mathbb{G}$.*

Where $\mathrm{Tr}(A)$ is the trace of matrix A. An equivalent definition exists, which characterizes a character χ on a group $\mathbb{G}$ as a homomorphism of the form $\chi : \mathbb{G} \to \mathbb{C}^*$, where $\mathbb{C}^*$ is the group of complex numbers of unit length under multiplication. Although the two definitions are equivalent, it is the second one that will be of particular help in the next section.

Soon we will prove more results regarding characters and general representation theory. However, since these are results specific to a certain group family, we cover them in the next section, where such family is also introduced.

4.4 General Abelian Case

Our main goal in this first part of the section is to use representation theory to generalize the CQFT given in Definition 4.2.1 to a QFT that acts on generic Abelian groups. As we said earlier, Theorem 1.2.1 guarantees that for every Abelian group $\mathbb{G}$ there exist $N_1, N_2, \ldots, N_k$ such that

$$\mathbb{G} \cong \mathbb{Z}_{N_1} \oplus \mathbb{Z}_{N_2} \oplus \cdots \oplus \mathbb{Z}_{N_k}.$$

Although this theorem proves the *existence* of such decomposition, it does not tell us how to *compute* it. Actually, the problem of finding $N_1, N_2, \ldots, N_k$ given $\mathbb{G}$ is known to be classically hard. Fortunately, a quantum algorithm exists [3] that can compute this decomposition efficiently:

Theorem 4.4.1 (Cheung and Mosca)**.** *Given a finite Abelian black-box group $\mathbb{G}$ with unique encoding, the decomposition of $\mathbb{G}$ into a direct sum of cyclic groups of prime power order can be computed in time polynomial in the input size by a quantum computer.*

4.4.1 More Representation Theory

Let us now see some representation theory results specific to this kind of group. We assume to be already working with a group of the form $\mathbb{G} = \mathbb{Z}_{N_1} \oplus \cdots \oplus \mathbb{Z}_{N_k}$. The elements of $\mathbb{G}$ are the k-tuples of the form $(g_1, g_2, \ldots, g_k)$, where $g_j \in \mathbb{Z}_{N_j}$ for all j. Let $\beta_1 = (1, 0, \ldots, 0), \beta_2 = (0, 1, \ldots, 0), \ldots, \beta_k(0, 0, \ldots, 1)$ be elements of a basis for $\mathbb{G}$. For a generic character χ on $\mathbb{G}$, we find that for all $g \in \mathbb{G}$

$$\chi(g) = \chi\left(\sum_{j=1}^{k} g_j \beta_j\right) = \prod_{j=1}^{k} \chi(\beta_j)^{g_j}. \tag{4.9}$$

This is because χ is by definition a homomorphism between $\mathbb{G}$ and $\mathbb{C}^*$, and as such $\chi(ng) = \chi(g)^n$ for all $g \in \mathbb{G}$. As a result, χ is determined entirely by its values on the basis elements.

Since every β_j is zero in every position but the jth one (in which it is unitary), every β_j has order exactly N_j. It follows that $\chi(\beta_j)$ must have order dividing N_j, for every j. Let us employ the following notation:

$$\omega_N = e^{\frac{2\pi i}{N}}.$$

Then, for every $\chi(\beta_j)$, an integer h_j must exist for which

$$\chi(\beta_j) = \omega_{N_j}^{h_j},$$

as every ω_N is a primitive Nth root of unity. Furthermore, h_j can be chosen in the restricted range $\{0, 1, \ldots, N_j - 1\}$, since the values of $\omega_{N_j}^{h_j}$ are periodic. Therefore, every distinct character on $\mathbb{G}$ can be identified by a k-tuple $(h_1, h_2, \ldots, h_k)$, which is also an element $h \in \mathbb{G}$. Every $h \in \mathbb{G}$ determines a character χ_h as follows:

$$\chi_h(g) = \prod_{j=1}^{k} \omega_{N_j}^{h_j g_j}.$$

From this definition it is also evident, for all $h, g \in \mathbb{G}$, that $\chi_h(g) = \chi_g(h)$ and $\chi_h(-g) = \chi_h(g)^{-1}$. By $\chi(\mathbb{G})$ we denote the set of all homomorphisms χ_g such that $g \in \mathbb{G}$. Note that $\chi(\mathbb{G})$ is a group under $\chi_{g_1}\chi_{g_2} = \chi_{g_1+g_2}$, with identity χ_e. We have the following theorem:

Theorem 4.4.2. *Let $\mathbb{G}$ be a finite Abelian group and let $\chi(\mathbb{G})$ be a group as defined above. We have $\mathbb{G} \cong \chi(\mathbb{G})$.*

Proof. Consider $\alpha : \mathbb{G} \to \chi(\mathbb{G})$ such that $\alpha(g) = \chi_g$. The identity element $e \in \mathbb{G}$ is sent to the identity element $\chi_e \in \chi(\mathbb{G})$. Furthermore, given $g_1 + g_2$ we find $\alpha(g_1 + g_2) = \chi_{g_1+g_2} = \chi_{g_1}\chi_{g_2} = \alpha(g_1)\alpha(g_2)$. Being an homomorphism and a set bijection, α is an isomorphism between $\mathbb{G}$ and $\chi(\mathbb{G})$. □

Now we can introduce the concept of *orthogonal subgroup*. This form of subgroup is significant because whereas the algorithm for the cyclic HSP gave as output elements uniformly distributed in $\mathbb{H}$, the algorithm we are going to define for Abelian groups will output elements uniformly distributed in $\mathbb{H}$'s orthogonal subgroup.

Definition 4.4.1 (Orthogonal subgroup)**.** *Let $\mathbb{G}$ be a finite Abelian group and let $\mathbb{H} \leq \mathbb{G}$. The* orthogonal subgroup *of $\mathbb{H}$ is written $\mathbb{H}^\perp$ and is defined as*

$$\mathbb{H}^\perp = \{g \in \mathbb{G} \mid \forall h \in \mathbb{H} : \chi_g(h) = 1\}.$$

Lemma 4.4.1. *For every subgroup $\mathbb{H}$ of $\mathbb{G}$, $\mathbb{H}^\perp$ is a subgroup of $\mathbb{G}$.*

Proof. Since $\chi_e(g) = 1$ for all $g \in \mathbb{G}$ (and therefore for all $g \in \mathbb{H}$), the identity element is in $\mathbb{H}^\perp$. Furthermore, if $h'_1, h'_2 \in \mathbb{H}^\perp$, then $\chi_{h'_1}(h) = \chi_{h'_2}(h) = 1$ for all $h \in \mathbb{H}$. It follows that $\chi_{h'_1+h'_2}(h) = \chi_{h'_1}(h)\chi_{h'_2}(h) = 1$ for all $h \in \mathbb{H}$ and $h'_1 + h'_2$ is in $\mathbb{H}^\perp$. Finally, if $h' \in \mathbb{H}^\perp$, we have $\chi_{-h'}(h) = \chi_h(-h') = \chi_h(h')^{-1} = \chi_{h'}(h)^{-1} = 1$ and $-h' \in \mathbb{H}^\perp$. □

We will also use the following result:

Lemma 4.4.2. *Let $\mathbb{G}$ be a finite Abelian group and let $\chi \in \chi(\mathbb{G})$ be one of its characters. We have*

$$\sum_{g\in\mathbb{G}} \chi(g) = \begin{cases} |\mathbb{G}| & \text{if } \chi = \chi_e, \\ 0 & \text{otherwise.} \end{cases}$$

Proof. We avail ourselves of Theorem 4.4.2 and we choose $h \in \mathbb{G}$ such that $\chi = \chi_h$. With $\mathbb{G} \cong \mathbb{Z}_{N_1} \oplus \cdots \oplus \mathbb{Z}_{N_k}$, we have

$$\begin{aligned} \sum_{g\in\mathbb{G}} \chi_h(g) &= \sum_{g_1\in\mathbb{Z}_{N_1}} \sum_{g_2\in\mathbb{Z}_{N_2}} \cdots \sum_{g_k\in\mathbb{Z}_{N_k}} \prod_{j=1}^{k} \omega_{N_j}^{h_j g_j} \\ &= \left(\sum_{g_1\in\mathbb{Z}_{N_1}} \omega_{N_1}^{h_1 g_1}\right)\left(\sum_{g_2\in\mathbb{Z}_{N_2}} \omega_{N_2}^{h_2 g_2}\right)\cdots\left(\sum_{g_k\in\mathbb{Z}_{N_k}} \omega_{N_k}^{h_k g_k}\right). \end{aligned}$$

If $\chi_h \neq \chi_e$, then there is at least one $\omega_{N_j}^{h_j} \neq 1$, which causes the geometric series $\sum_{g_j \in \mathbb{Z}_{N_j}} \left(\omega_{N_j}^{h_j}\right)^{g_j}$, and therefore the entire product, to amount to 0. On the other hand, if $\chi_h = \chi_e$, the result is $|\mathbb{G}|$. □

Finally, we need two more results, the first of which requires us to define the concept of *quotient group*.

Definition 4.4.2 (Quotient group)**.** *Let $\mathbb{G}$ be a group and let $\mathbb{H} \leq \mathbb{G}$ be a normal subgroup. The* quotient group *of $\mathbb{G}$ and $\mathbb{H}$ is written $\mathbb{G}/\mathbb{H}$ and is defined as*

$$\mathbb{G}/\mathbb{H} = \{g\mathbb{H} \mid g \in \mathbb{G}\}.$$

In other words, the quotient group is the group of cosets of $\mathbb{H}$ under $g_1\mathbb{H}g_2\mathbb{H} = (g_1g_2)\mathbb{H}$.

Lemma 4.4.3. *Let $\mathbb{G}$ be a finite Abelian group and let $\mathbb{H} \leq \mathbb{G}$. We have*

$$\mathbb{H}^\perp \cong \mathbb{G}/\mathbb{H}.$$

Proof. Thanks to Theorem 4.4.2 it is sufficient to show that $\chi(\mathbb{H}^\perp) \cong \chi(\mathbb{G}/\mathbb{H})$. Let $\bar{g}$ denote the image of a generic $g \in \mathbb{G}$ through the projection map $\pi : \mathbb{G} \to \mathbb{G}/\mathbb{H}$. Define a map of the form $\alpha : \chi(\mathbb{H}^\perp) \to \chi(\mathbb{G}/\mathbb{H})$ such that

$$(\alpha\chi)(\bar{g}) = \chi_{h'}(g),$$

where $h' \in \mathbb{H}^\perp$ and $\bar{g} = g + \mathbb{H} \in \mathbb{G}/\mathbb{H}$ for some coset representative g. We show that α is a group isomorphism in three steps:

1. α is well-defined: if g_1 and g_2 are different representatives of the same coset $\bar{g_1} = \bar{g_2}$, then there exists $h \in \mathbb{H}$ such that $h = g_1 - g_2$ and $(\alpha\chi_{h'})(\bar{g_1}) = \chi_{h'}(g_1) = \chi_{h'}(g_1 + h) = \chi_{h'}(g_2) = (\alpha\chi_{h'})(\bar{g_2})$. Furthermore, for the identity $\chi_e \in \chi(\mathbb{H}^\perp)$ and any $\bar{g} \in \mathbb{G}/\mathbb{H}$, we have $(\alpha\chi_e)(\bar{g}) = \chi_e(g) = 1$, so $\alpha\chi_e$ is the identity in $\chi(\mathbb{G}/\mathbb{H})$. Finally, for $g \in \mathbb{G}$ we have $(\alpha(\chi_{h_1}\chi_{h_2}))(\bar{g}) = (\alpha\chi_{h_1+h_2})(\bar{g}) = \chi_{h_1+h_2}(g) = \chi_{h_1}(g)\chi_{h_2}(g) = ((\alpha\chi_{h_1})(\alpha\chi_{h_2}))(\bar{g})$, so α is a homomorphism.

2. α is injective: suppose for $h' \in \mathbb{H}^\perp$ that $\chi_{h'}$ is the identity element in $\chi(\mathbb{G}/\mathbb{H})$. Then, for every $g \in \mathbb{G}$, $(\alpha\chi_{h'})(\bar{g}) = 1$ implies $\chi_{h'}(g) = 1$. It follows that $\chi_{h'} = \chi_e$ and, since $\mathbb{G} \cong \chi(\mathbb{G})$, that $h' = e$, so α is injective.

3. α is surjective: let $\bar{\chi} \in \chi(\mathbb{G}/\mathbb{H})$ and remember π. The composite map $\chi = \bar{\chi} \circ \pi : \mathbb{G} \to \mathbb{C}^*$ is a homomorphism and therefore a character χ_t for some $t \in \mathbb{G}$. If $h \in \mathbb{H}$, then $\chi_t(h) = \bar{\chi}(\bar{e}) = 1$, so $t \in \mathbb{H}^\perp$ and $\chi_t \in \chi(\mathbb{H}^\perp)$. Now, let $\bar{g} \in \mathbb{G}/\mathbb{H}$. We have $(\alpha\chi_t)(\bar{g}) = \chi_t(g) = \bar{\chi}\pi(g) = \bar{\chi}(\bar{g})$. Thus α is surjective and a group isomorphism.

□

Lemma 4.4.4. *With the same premises of the previous lemma, we have*

$$\mathbb{H}^{\perp\perp} = \mathbb{H}.$$

Proof. From the previous lemma, we know that $\mathbb{G}/\mathbb{H} \cong \mathbb{H}^{\perp}$. It follows that $|\mathbb{G}/\mathbb{H}| = |\mathbb{H}^{\perp}|$ and therefore $|\mathbb{G}/\mathbb{H}^{\perp}| = |\mathbb{H}|$, but also $|\mathbb{G}/\mathbb{H}^{\perp}| = |\mathbb{H}^{\perp\perp}|$, so $|\mathbb{H}| = |\mathbb{H}^{\perp\perp}|$. Take $h \in \mathbb{H}$. By definition, $\mathbb{H}^{\perp\perp} = \{g \in \mathbb{G} \mid \forall h' \in \mathbb{H}^{\perp} : \chi_g(h') = 1\}$. In particular, $\chi_h(h') = \chi_{h'}(h) = 1$ for all $h' \in \mathbb{H}^{\perp}$, by the definition of $\mathbb{H}^{\perp}$. It follows that $h \in \mathbb{H}^{\perp\perp}$ and therefore $\mathbb{H} \subseteq \mathbb{H}^{\perp\perp}$. This, together with $|\mathbb{H}| = |\mathbb{H}^{\perp\perp}|$, implies $\mathbb{H} = \mathbb{H}^{\perp\perp}$. □

4.4.2 The QFT on Abelian Groups

We can now start to actually move towards a solution for the problem at hand. We do so by defining three operations called *the* $\mathbb{G}$*-operators*, since their definitions depend on a finite Abelian group $\mathbb{G}$. The first one is the the QFT over $\mathbb{G}$.

Definition 4.4.3 (Abelian QFT)**.** *The* Abelian QFT *(AQFT) over a finite Abelian group $\mathbb{G}$ is the quantum operator defined as*

$$Q_{\mathbb{G}} = \frac{1}{\sqrt{|\mathbb{G}|}} \sum_{g,h \in \mathbb{G}} \chi_g(h) |g\rangle\langle h|.$$

We can immediately se that if $\mathbb{G}$ is of the form $\mathbb{Z}_N$ for some $N \in \mathbb{N}$ (a cyclic additive group as seen in Section 4.2), then $\chi_g(h) = \omega_N^{gh} = e^{\frac{2\pi i g h}{N}}$ and $Q_{\mathbb{G}}$ coincides with Q_N, the cyclic QFT as given in Definition 4.2.1. In fact, if $\mathbb{G} = \mathbb{Z}_1 \oplus \mathbb{Z}_2 \oplus \cdots \oplus \mathbb{Z}_k$, then

$$Q_{\mathbb{G}} = \bigotimes_{j=1}^{k} Q_{N_j},$$

where each Q_{N_j} is the CQFT on $\mathbb{Z}_{N_j}$, as described in the previous section. This means that the AQFT can easily be built once we have the CQFT at our disposal.

The other two operators are the *translation operator* and the *phase-change operator*. They are defined as follows:

Definition 4.4.4 (Translation operator)**.** *Let $\mathbb{G}$ be a finite Abelian group. For $t \in \mathbb{G}$, the* translation operator τ_t *over $\mathbb{G}$ is the quantum operator defined as*

$$\tau_t = \sum_{g \in \mathbb{G}} |t + g\rangle\langle g|.$$

Definition 4.4.5 (Phase-change operator)**.** *Let $\mathbb{G}$ be a finite Abelian group. For $h \in \mathbb{G}$, the* phase-change operator φ_h *over $\mathbb{G}$ is the quantum operator defined as*

$$\varphi_h = \sum_{g \in \mathbb{G}} \chi_g(h) |g\rangle\langle g|.$$

The three $\mathbb{G}$-operators exhibit interesting relationships with one another. We are interested in two specific properties, which we state and prove:

Lemma 4.4.5. *Let $\mathbb{G}$ be a finite Abelian group and let $\mathbb{H} \leq \mathbb{G}$. We have*

$$Q_{\mathbb{G}} |\mathbb{H}\rangle = |\mathbb{H}^{\perp}\rangle,$$

where

$$|\mathbb{H}\rangle = \frac{1}{\sqrt{|\mathbb{H}|}} \sum_{h \in \mathbb{H}} |h\rangle.$$

Proof.

$$\begin{aligned}
Q_{\mathbb{G}} |\mathbb{H}\rangle &= \frac{1}{\sqrt{|\mathbb{G}|}} \sum_{u,v \in \mathbb{G}} \chi_u(v) |u\rangle\langle v| \frac{1}{\sqrt{|\mathbb{H}|}} \sum_{h \in \mathbb{H}} |h\rangle \\
&= \frac{1}{\sqrt{|\mathbb{G}||\mathbb{H}|}} \sum_{\substack{u,v \in \mathbb{G} \\ h \in \mathbb{H}}} \chi_u(v) |u\rangle\langle v|h\rangle \\
&= \frac{1}{\sqrt{|\mathbb{G}||\mathbb{H}|}} \sum_{\substack{u \in \mathbb{G} \\ h \in \mathbb{H}}} \chi_u(h) |u\rangle \\
&= \frac{1}{\sqrt{|\mathbb{G}||\mathbb{H}|}} \sum_{u \in \mathbb{G}} \left(\sum_{h \in \mathbb{H}} \chi_u(h) \right) |u\rangle.
\end{aligned}$$

By Lemma 4.4.2, $\sum_{h \in \mathbb{H}} \chi_u(h)$ is non-zero only if $\chi_u = \chi_e$, in which case the sum is equal to $|\mathbb{H}|$. Furthermore, if χ_u is the identity, then $u \in \mathbb{H}^{\perp}$, as $\chi_u(h) = 1$ for all $h \in \mathbb{H}$. Our equation thus becomes

$$\frac{1}{\sqrt{|\mathbb{G}||\mathbb{H}|}} \sum_{u \in \mathbb{H}^{\perp}} |\mathbb{H}||u\rangle = \sqrt{\frac{|\mathbb{H}|}{|\mathbb{G}|}} \sum_{u \in \mathbb{H}^{\perp}} |u\rangle = |\mathbb{H}^{\perp}\rangle,$$

as by Lemma 4.4.3, $|\mathbb{H}|/|\mathbb{G}| = |\mathbb{H}^{\perp}|^{-1}$. □

Lemma 4.4.6. *Let $\mathbb{G}$ be a finite Abelian group and let $t \in \mathbb{G}$. We have*

$$Q_{\mathbb{G}} \tau_t = \varphi_t Q_{\mathbb{G}}.$$

Proof.

$$\begin{aligned}
Q_{\mathbb{G}}\tau_t &= \left(\frac{1}{\sqrt{|\mathbb{G}|}}\sum_{u,v\in\mathbb{G}}\chi_u(v)|u\rangle\langle v|\right)\left(\sum_{g\in\mathbb{G}}|t+g\rangle\langle g|\right)\\
&= \frac{1}{\sqrt{|\mathbb{G}|}}\sum_{u,v,g\in\mathbb{G}}\chi_u(v)|u\rangle\langle v|t+g\rangle\langle g|\\
&= \frac{1}{\sqrt{|\mathbb{G}|}}\sum_{u,g\in\mathbb{G}}\chi_u(t+g)|u\rangle\langle g|\\
&= \frac{1}{\sqrt{|\mathbb{G}|}}\sum_{u,g\in\mathbb{G}}\chi_u(t)\chi_u(g)|u\rangle\langle g|.
\end{aligned}$$

At this point, we insert the identity transformation $I = \sum_{a\in\mathbb{G}}|a\rangle\langle a|$ into the equation, obtaining

$$\begin{aligned}
&\frac{1}{\sqrt{|\mathbb{G}|}}\sum_{u,g,a\in\mathbb{G}}\chi_u(t)\chi_u(g)|a\rangle\langle a|u\rangle\langle g|\\
=&\frac{1}{\sqrt{|\mathbb{G}|}}\sum_{u,g,a\in\mathbb{G}}\chi_a(t)\chi_u(g)|a\rangle\langle a|u\rangle\langle g|\\
=&\left(\sum_{a\in\mathbb{G}}\chi_a(t)|a\rangle\langle a|\right)\left(\frac{1}{\sqrt{|\mathbb{G}|}}\sum_{u,g\in\mathbb{G}}\chi_u(g)|u\rangle\langle g|\right) = \varphi_t Q_{\mathbb{G}}.
\end{aligned}$$

□

4.4.3 The Algorithm

Now we have a sound foundation of representation theory *and* we have the AQFT that works on generic Abelian groups. We will now try and mimic the flow of computation that we followed in Section 4.2, to see where we end up.

This time, we are working with a family $\{\mathbb{G}_N\}_{N\in\mathbb{N}}$ of Abelian groups. We assume, as we did in the case of the DLP, that each group $\mathbb{G}_N$ comes with its order, $|\mathbb{G}_N|$. Each $\mathbb{G}_N$ can be efficiently decomposed, by Theorem 4.4.1, into a direct sum of cyclic additive groups $\mathbb{Z}_{N_1}\oplus\mathbb{Z}_{N_2}\oplus\cdots\oplus\mathbb{Z}_{N_k}$. Every $g\in\mathbb{G}_N$ is thus encoded as a k-tuple $(g_1, g_2, \ldots, g_k)$, which, from now on, we refer to as simply g. We also have U_f, which is a quantum circuit that computes

$$|x\rangle|y\rangle \xrightarrow{U_f} |x\rangle|y\oplus f(x)\rangle,$$

such that f separates cosets for some $\mathbb{H}_N \leq \mathbb{G}_N$.

So, once again we consider a quantum computer with two n-qubit registers (for a big enough n), both initialized to $|0\rangle$, and we once again start by applying the AQFT on the first register:

$$|0\rangle|0\rangle \xrightarrow{Q_{\mathbb{G}_N} \text{ on 1st}} \frac{1}{\sqrt{|\mathbb{G}_N|}} \sum_{g \in \mathbb{G}_N} |g\rangle|0\rangle. \tag{4.10}$$

We then proceed to apply U_f on both registers, obtaining

$$\frac{1}{\sqrt{|\mathbb{G}_N|}} \sum_{g \in \mathbb{G}_N} |g\rangle|0\rangle \xrightarrow{U_f} \frac{1}{\sqrt{|\mathbb{G}_N|}} \sum_{g \in \mathbb{G}_N} |g\rangle|f(g)\rangle. \tag{4.11}$$

This time, the simplification process is slightly less trivial. Consider $T = \{t_1, t_2, \ldots, t_m\}$, a set of coset representatives for the $\mathbb{H}_N \leq \mathbb{G}_N$ separated by f (T is called a *transversal* for $\mathbb{H}_N$). Our equation becomes

$$\frac{1}{\sqrt{|T|}} \sum_{t \in T} |t + \mathbb{H}_N\rangle|f(t)\rangle = \frac{1}{\sqrt{|T|}} \sum_{t \in T} \tau_t |\mathbb{H}_N\rangle|f(t)\rangle \tag{4.12}$$

At this point, in the algorithm for the cyclic additive case, we measured the second register to collapse the state. However, thanks to a principle known as *principle of deferred measurement*, we can skip this step and measure only once at the end. We apply the AQFT once more on the first register, to obtain the following state:

$$\frac{1}{\sqrt{|T|}} \sum_{t \in T} Q_{\mathbb{G}} \tau_t |\mathbb{H}_N\rangle|f(t)\rangle \tag{4.13}$$

$$= \frac{1}{\sqrt{|T|}} \sum_{t \in T} \varphi_t Q_{\mathbb{G}} |\mathbb{H}_N\rangle|f(t)\rangle \tag{4.14}$$

$$= \frac{1}{\sqrt{|\mathbb{H}_N^\perp|}} \sum_{t \in T} \varphi_t |\mathbb{H}_N^\perp\rangle|f(t)\rangle. \tag{4.15}$$

In this simplification we used lemmata 4.4.6 and 4.4.5, in this order, as well as the fact that $|T| = |\mathbb{G}_N|/|\mathbb{H}_N| = |\mathbb{H}_N^\perp|$. The last step consists in measuring the first register to obtain a random element of $\mathbb{H}_N^\perp$, with uniform probability (being a phase operator, φ_t does not affect amplitudes).

At this point, two questions must be asked. The first one is: *how many* of these elements do we need before we can be sufficiently confident that we have a generating set for $\mathbb{H}_N^\perp$? The answer is provided by the following theorem [8]:

Theorem 4.4.3. *Let $\mathbb{G}$ be a finite group. For an integer $k \geq 0$, the probability that $k + \lceil \log |\mathbb{G}| \rceil$ elements chosen uniformly at random from $\mathbb{G}$ will generate the group is bounded by*

$$P(\langle g_1, g_2, \ldots, g_{k+\lceil \log |\mathbb{G}| \rceil} \rangle = \mathbb{G}) \geq 1 - \left(\frac{1}{2}\right)^k.$$

This result is similar to the one expressed by Lemma 4.2.1, so once again relatively few samples are needed to obtain a generating set for $\mathbb{H}_N^\perp$ with high probability. The second question is: *how* do we obtain a generating set for $\mathbb{H}_N$ (which is the output we seek) from a generating set for $\mathbb{H}_N^\perp$? Of course, since $(\mathbb{H}_N^\perp)^\perp = \mathbb{H}_N$, we have that $\mathbb{H}_N^\perp$ uniquely determines $\mathbb{H}_N$, but the actual process of finding a generating set for $\mathbb{H}_N$ is non-trivial.

Suppose we already have the $g_1, g_2, \ldots, g_t$ that generate $\mathbb{H}_N^\perp$. Since $(\mathbb{H}_N^\perp)^\perp = \mathbb{H}_N$, it must be, for all $h \in \mathbb{H}_N$, $h' \in \mathbb{H}_N^\perp$ and $j \in \{1, 2, \ldots, t\}$, that $\chi_h(h'_j) = 1$. Next, let $d = \text{lcm}(N_1, \ldots, N_k)$, where the N_j come from the decomposition of the original group. If we set $\alpha_l = d/N_l$, then $\omega_{N_l} = \omega_d^{\alpha_l}$. At this point we have that $\chi_h(g_j) = \prod_{l=1}^k \omega_d^{\alpha_l (g_j)_l h_l} = 1$ if and only if $\sum_{l=1}^k \alpha_l (g_j)_l h_l \equiv 0 \mod d$. So all we need to do to compute elements of $\mathbb{H}_N$ is find random solutions to the following linear system of t equations:

$$\begin{cases} \alpha_1(g_1)_1 x_1 + \alpha_2(g_1)_2 x_2 + \cdots + \alpha_k(g_1)_k x_k \equiv 0 \mod d \\ \alpha_1(g_2)_1 x_1 + \alpha_2(g_2)_2 x_2 + \cdots + \alpha_k(g_2)_k x_k \equiv 0 \mod d \\ \ldots \\ \alpha_1(g_t)_1 x_1 + \alpha_2(g_t)_2 x_2 + \cdots + \alpha_k(g_t)_k x_k \equiv 0 \mod d \end{cases}$$

To do so, consider the system in matrix form: $Ax \equiv 0 \mod d$. Compute the Smith normal form of A, that is,

$$D = UAV,$$

where D is diagonal and U and V are integer-valued. Computing the normal form for a $t \times k$ matrix such as A requires $O(k^2 t)$ time [14]. At this point, finding random solutions to $Dy \equiv 0 \mod d$ amounts to solving simple linear congruences. Once we have a y, we compute $x = Vy$, thus yielding an element $x \in \mathbb{H}_N$. Note that recovering V from D requires time $O(k^2 t \log^c(k^2 t))$[14], for some constant c.

At this point, assume we have run the above procedure $t = t_1 + \lceil \log |\mathbb{G}_N| \rceil$ to obtain $g_1, g_2, \ldots, g_t \in \mathbb{H}_N^\perp$, which generate $\mathbb{H}_N^\perp$ with probability $p_1 \geq 1 - 1/2^{t_1}$. We find $s = t_2 + \lceil \log |\mathbb{G}_N| \rceil$ samples in $\mathbb{H}_N$ using the procedure described above. We have that the s samples generate $\mathbb{H}_N$ with probability $p \geq (1 - 1/2^{t_1})(1 - 1/2^{t_2})$.

That being said, we can now give the full definition of the algorithm that solves the HSP on Abelian groups:

Algorithm 5: F_{HSP} – Abelian Case

Input: A binary representation $\text{bin}(N)$ and $\lceil C_f \rceil$ such that C_f separates cosets for $\mathbb{H}_N \leq \mathbb{G}_N$ with respect to ρ.

Output: The representations $\rho(h_1, N), \rho(h_2, N), \ldots, \rho(h_s, N)$ of the elements of a generating set for $\mathbb{H}_N$, with probability at least $1 - \frac{1}{|\mathbb{G}_N|}$.

Compute $N_1, N_2, \ldots, N_k$ such that $G_N \cong \mathbb{Z}_{N_1} \oplus \mathbb{Z}_{N_2} \oplus \cdots \oplus \mathbb{Z}_{N_k}$;
Use $N_1, \ldots, N_k$ to build $Q_{\mathbb{G}_N} = \bigotimes_{j=1}^{k} Q_{N_j}$;
Use $\lceil C_f \rceil$ to build U_f, a reversible quantum circuit that computes C_f;

$t \leftarrow 2\lceil \log |\mathbb{G}_N| \rceil + 1$;
Initialize an array g of size t;
for $j = 1$ to t **do**
    Initialize $|\varphi\rangle|\psi\rangle$ to $|0\rangle|0\rangle$;
    Apply $Q_{\mathbb{G}_N}$ to $|\varphi\rangle$;
    Apply U_f on $|\varphi\rangle|\psi\rangle$;
    Apply $Q_{\mathbb{G}_N}$ to $|\varphi\rangle$;
    Measure $|\varphi\rangle$ to obtain a random $h' \in \mathbb{H}_N^{\perp}$;
    $g_j \leftarrow h'$

$d \leftarrow \text{lcm}(N_1, \ldots, N_k)$;
Initialize an array α of size k;
for $j = 1$ to k **do**
    $\alpha_j \leftarrow d/N_j$;
Construct the $t \times k$ matrix A such that $A_{i,j} = \alpha_j (g_i)_j$;
Compute A's Smith normal form $D = UAV$;
Retrieve V from D;

Initialize a set S;
for t times **do**
    Compute a random solution y of $Dy \equiv 0 \mod d$;
    $x \leftarrow Vy$;
    Add x to S;
Map ρ over S;
return S;

Here we chose $t_1 = t_2 = \lceil \log |\mathbb{G}_N| \rceil + 1$, so the probability that S is a generating set for $\mathbb{H}_N$ is at least $(1 - 2^{-\lceil \log |\mathbb{G}_N| \rceil + 1})^2 \geq 1 - \frac{1}{|\mathbb{G}_N|}$ and the claim of our algorithm is correct.

Complexity-wise, we take $n = |\mathbb{G}_N|$ to be the size of the problem. We furthermore assume that ρ represents the elements of each $\mathbb{G}_N$ in $O(\log n)$ space. We already know (Theorems 4.4.1 and 4.1.1 and Algorithm 3) that the first three steps can be carried out in $O(\text{polylog}(n))$ time. Furthermore, the first *for* loop requires $O(\log n)$ iterations, each with complexity dominated by $Q_{\mathbb{G}_N}$ and U_f, which we know run in polylog time.

From Lemma 1.2.2, we know that k is $O(\log n)$, so the second loop requires $O(\log n)$ time. At this point, we build matrix A, whose dimensions are both $O(\log n)$. Next, computing D and V requires $O(\text{polylog}(n))$ time, as both t and k are $O(\log n)$. Lastly, the fourth loop iterates $O(\log n)$ times. Finding y and x is efficient, thus the entire loop requires $O(\text{polylog}(n))$ time.

It is fairly evident that the resulting time complexity of the algorithm is $O(\text{polylog}(n))$ in the size $n = |\mathbb{G}_N|$ of the input, which is efficient.

Application to concrete problems We conclude this chapter with a brief remark on the problems that we discussed in Section 2.2. In particular, we want to point out that both $\{\mathbb{B}^N, \oplus\}_{N\in\mathbb{N}}$, the group family employed in the reduction of Simon's problem, and $\{\mathbb{Z}_N \oplus \mathbb{Z}_N\}_{N\in\mathbb{N}}$, the one used in the reduction of the discrete logarithm problem, are families of *Abelian* groups. It follows that both SP and the DLP are reduced to instances of the HSP that can be solved by Algorithm 5.

Conclusions

In our discussion on the hidden subgroup problem, we examined its importance as the generalization of those problems for which quantum algorithms exist that are exponentially faster than the classical ones. We also presented the actual reduction of two of these problems to the HSP, namely Simon's problem and the discrete logarithm problem. Lastly, we showed that efficient solutions to some instances of the HSP already exist, and that they can therefore be used to solve the very problems we reduced.

We also hope that our work succeeded in convincing the reader of the flexibility and generality of the HSP, two properties that make it a prime candidate for the discovery of new and improved results in quantum complexity theory.

We conclude this thesis with a quick review of some of the topics that we did not cover, either because of a willful choice of scope or due to time and space constraints.

Integer Factorization

In multiple occasions we anticipated that integer factorization could be reduced to the hidden subgroup problem. However, when we actually showed some of the reductions, we omitted such key problem entirely.

The reason behind this choice is that integer factorization is reducible to an instance of the HSP that eludes our definition of HSP functional. Namely, factorization can be reduced to a problem of *order finding*, which can be further reduced to an instance of the HSP on $(\mathbb{Z}, +)$, the additive group of integers [11]. Despite being Abelian and finitely generated, this group is not *finite*. Our definition of the HSP functional, on the other hand, is limited by construction to finite groups and is therefore inadequate to handle this case. Note, however, that solutions to the HSP on finitely generated Abelian groups exist.

A second approach exists, which reduces the factorization of an integer n to an instance of the HSP on the additive group $\mathbb{Z}_{\varphi(n)}$, where φ is Euler's totient function [9]. However, finding $\varphi(n)$ given n is just as hard as factoring n. Therefore, either solutions to a weaker version of the HSP (in which the actual $\mathbb{G}$ is unknown) are found, or this approach turns out to be of very little use.

Non-Abelian Groups

It is also worth mentioning the current state of the art with regard to solutions to the HSP on finite *non-Abelian* groups, since this is the kind of HSP which many heterogeneous problems are reducible to. The graph isomorphism problem, for example, is reducible to the HSP on the family $\{S_N\}_{N\in\mathbb{N}}$ of *simmetric groups* of N symbols, which are not generally Abelian [8].

On this front, we have the results of Ettinger, Høyer and Knill [5], who proved that the HSP on generic groups of finite order can be solved with polynomial *query complexity*, i.e. with a number of calls to U_f polynomial in the size of the input. Formally:

Theorem. *There exists a quantum algorithm that, given a finite group $\mathbb{G}$ and an oracle f on $\mathbb{G}$ promised to be strictly $\mathbb{H}$-periodic for some subgroup $\mathbb{H} \leq \mathbb{G}$, calls the oracle $O(\log^4 |\mathbb{G}|)$ times and outputs a generating set for $\mathbb{H}$. The algorithm fails with probability exponentially small in $\log |\mathbb{G}|$. The algorithm can be made exact in any model allowing arbitrary one-qubit gates.*

This solution, however, requires some form of classical preprocessing, as an adequate quantum network must be built from a specification of $\mathbb{G}$. Unfortunately, the preprocessing routines and the quantum networks they produce are inefficient. As a result, in spite of the polynomial query complexity, the whole algorithm has time complexity exponential in the size of the input.

Bibliography

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] Paul W. Beame, Stephen A. Cook, and H. James Hoover. Log depth circuits for division and related problems. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 1984.

[3] Kevin K. H. Cheung and Michele Mosca. Decomposing finite abelian groups. *J. Quantum Inf. Comp.*, 2001.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, third edition edition, 2009.

[5] Mark Ettinger, Peter Høyer, and Emanuel Knill. The quantum query complexity of the hidden subgroup problem is polynomial. *Information Processing Letters*, July 2004.

[6] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC)*, May 1996.

[7] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 2015.

[8] Chris Lomont. The hidden subgroup problem – review and open problems, November 2004.

[9] Stephen McAdam. The abelian hidden subgroup problem, 2013.

[10] Michele Mosca. *Quantum Computer Algorithms*. Ph.d. thesis, Wolfson College, University of Oxford, 1999.

[11] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2004.

[12] Peter Williston Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of 35th Annual Symposium on Foundations of Computer Science*. IEEE Press, 1994.

[13] Daniel R. Simon. On the power of quantum computation. *SIAM Journal on Computing*, 26(5), October 1997.

[14] Arne Storjohann. Near optimal algorithms for computing smith normal forms of integer matrices. In *Proceedings of the 1996 international symposium on Symbolic and algebraic computation*. ACM Press, 1996.

[15] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008.