Informatics

Quantum Algorithms for the Discrete Logarithm Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Alexander Mandl, BSc
Matrikelnummer 01429186

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao. Univ-Prof. Dr. Uwe Egly

Wien, 12. Oktober 2021

Alexander Mandl	Uwe Egly

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Quantum Algorithms for the Discrete Logarithm Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Alexander Mandl, BSc
Registration Number 01429186

to the Faculty of Informatics

at the TU Wien

Advisor: Ao. Univ-Prof. Dr. Uwe Egly

Vienna, 12th October, 2021

Alexander Mandl Uwe Egly

Erklärung zur Verfassung der Arbeit

Alexander Mandl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Oktober 2021

Alexander Mandl

Danksagung

An dieser Stelle möchte ich meinen aufrichtigsten Dank an meinen Betreuer Prof. Dr. Uwe Egly aussprechen. Für die kontinuierliche Unterstützung mit Ideen, Korrekturen und anregenden Diskussionen über das Thema während dem Verfassen der Arbeit bin ich zutiefst dankbar.

Außerdem möchte ich mich bei Philipp Mandl und Tommy Tschida für deren Korrekturen und Begleitmaterial zum Thema bedanken.

Kurzfassung

Shor's Algorithmen für die Primfaktorzerlegung und für das Problem des diskreten Logarithmus zählen zu den bahnbrechendsten Entwicklungen im Bereich der Quanteninformatik. Sie lösen Probleme in polynomieller Zeit mithilfe eines Quantencomputers, für welche keine effizienten klassischen Algorithmen bekannt sind. Diese Tatsache unterstreicht den Nutzen von Quantencomputern im Allgemeinen.

Heutzutage existieren Bibliotheken und Simulatoren, die es ermöglichen, Quantenalgorithmen zu implementieren und zu analysieren. Eine dieser Bibliotheken, welche außerdem noch die Möglichkeit bietet, die Algorithmen auf echten Maschinen auszuführen, ist Qiskit. Qiskit beinhaltet zurzeit jedoch keine Referenzimplementierung für Shor's Algorithmus für den diskreten Logarithmus.

Für eine Implementierung müssen zusätzlich zur allgemeinen Beschreibung des Algorithmus noch Prozeduren für modulare Arithmetik bereitgestellt werden. Für diese Berechnungen gibt es viele unterschiedliche Algorithmen, wobei manche sehr klassischen Prozeduren ähneln und andere wiederum die Eigenheiten von Quantencomputern ausnutzen.

Diese Arbeit liefert eine Referenzimplementierung von Shor's Algorithmus für das Problem des diskreten Logarithmus. Dabei werden drei verschiedene Versionen der benötigten Routinen für modulare Arithmetik verwendet. Da aktuelle Quantencomputer immer noch relativ wenige Quantenbits, oder Qubits, für die Berechnung zur Verfügung stellen, liegt der Hauptfokus der Arbeit auf Möglichkeiten der Implementierung, die eine minimale Anzahl an Qubits benötigen.

Mithilfe der Simulationsfunktionen von Qiskit wird zusätzlich zum Vergleich der Implementierungen bezüglich der asymptotischen Zeitkomplexität ein Vergleich der verschiedenen Versionen anhand der konkreten Anzahl an elementaren Operationen für kleine Probleminstanzen vorgenommen.

Außerdem wird diese Analyse um eine genaue Untersuchung der Erfolgswahrscheinlichkeit ergänzt, da diese für die Laufzeit insgesamt ausschlaggebend ist.

Abstract

One of the groundbreaking results in the field of quantum computing is Shor's work describing algorithms for prime factorization and for the discrete logarithm problem. These algorithms solve problems in polynomial time on a quantum computer, for which no efficient solution is known on a classical computer, which highlights the utility of quantum computers.

Today there are libraries and simulators available that enable the implementation and analysis of quantum algorithms. One of these development environments is Qiskit, which furthermore enables its users to execute quantum algorithms on real devices. However, this library currently does not contain a reference implementation of Shor's algorithm for the discrete logarithm problem.

For such an implementation, not only the structure of the overall algorithm but also procedures for performing modular arithmetic on a quantum computer have to be provided. There are various ways of performing these calculations. Some are very similar to classical procedures and some utilize the peculiarities of quantum systems.

This work provides a reference implementation of Shor's algorithm for the discrete logarithm problem. This quantum program makes use of three different implementations of the required modular arithmetic procedures. Since current quantum computers are still small in the number of qubits available for computation, this thesis focuses on implementation proposals that optimize the space complexity of the algorithm for the discrete logarithm problem.

The presented implementations are compared using their asymptotic complexity bounds. Additionally, using Qiskit's simulation capabilities, a comparison of the different versions using concrete values for the number of elementary instructions for small problem instances is presented.

Furthermore, this analysis is extended to also investigate the actual success probability of the algorithm as this value is crucial for the overall runtime.

Contents

CHAPTER 1

Introduction

Shor's factoring algorithm and the algorithm for the discrete logarithm problem, which were both first presented in [31], sparked a lot of interest in the field of quantum computing. They were some of the first algorithms to solve problems in polynomial time on a quantum computer that are not known to be solvable efficiently on a classical computer.

The mathematical problems these algorithms find answers to are of interest for cryptography. Both the RSA cryptosystem and the Diffie-Hellman key exchange protocol use the fact that some efficiently computable functions are hard to reverse [10, 31]. As an example, part of the public information that is shared in the Diffie-Hellman key exchange protocol is the integer $g^X \bmod m$, where both g and m are publicly known. This value can be computed efficiently on a classical computer. However, to be able to break this protocol and obtain the private key, an attacker has to compute the hidden value X from the publicly available information.

The security of this protocol stems from the fact that obtaining X, i.e., computing the discrete logarithm, is a computationally costly task as the fastest known algorithms for solving this task on a classical computer run in superpolynomial time [31]. Efficient solutions to this problem, such as Shor's algorithm for the discrete logarithm problem, therefore diminish the security of such protocols. The speedup that is obtained by the use of quantum computers makes these algorithms and quantum computers in general a very important topic to study.

Both, Shor's factoring algorithm and the algorithm for discrete logarithms, make use of a quantum computer's ability to efficiently approximate the period of a function given only a quantum program, or *circuit* that computes its result for any input. A simple example of this process is given by the function $f(x) = a^x \bmod m$ as it is used in the factoring algorithm. If a and m are coprime, there is some smallest positive integer r such that $a^r \equiv 1 \bmod m$. For this function $f(x) = f(x + r)$ holds, meaning this function is periodic with period r. A quantum computer can be used to obtain this period r.

In recent years, the nature of the problems solved by Shor's algorithms has been extensively studied. The problem of finding the period of the function above, or in other words the multiplicative order of a, as well as the problem of finding the discrete logarithm, have been generalized as the *hidden subgroup problem* [24]. Explained roughly, there is a function $f : G \to X$ that maps each group element $x \in G$ to an element of the set X. Furthermore it is given that f is constant on the cosets of some subgroup K of G [24]. The task is to specify the subgroup K, given a way of computing the function f. For the example of the order finding problem, this subgroup of the integers is given by the multiples of the multiplicative order of a [25].

Additionally, the algorithm for solving such problems can be generalized to the process of *phase estimation* [24]. Herein, the eigenvalues of a unitary operator are approximated using a quantum computer. This generalization will also be used to present the algorithm for finding the discrete logarithm in this text.

Both algorithms presented by Shor are not only of interest because of their applications to cryptography. They also suggest that there might be problems that can be solved more efficiently on quantum computers than on classical computers, which has implications for theoretical computer science. In particular, the algorithms cast doubt on the complexity-theoretic strengthening of the Church-Turing thesis: "Any algorithmic process can be simulated efficiently using a Turing machine" [25]. As the word *efficiently* is usually understood as only allowing a polynomial simulation overhead, the thesis would imply that there also is a polynomial-time algorithm for prime factorization and for the discrete logarithm problem on classical computers. However, such an algorithm still has not been found, which poses the question whether quantum computers are more powerful than classical computers.

Even though the concept of a quantum computer was purely hypothetical when Shor's results were first presented, there have been significant advances in this field in recent years. Libraries and development environments for quantum algorithms have been developed and, albeit small, there are now quantum computers available for the general public to use. One such library is Qiskit[1]. It enables its users to implement, analyze and simulate quantum algorithms as well as execute them on real quantum devices. Qiskit also provides a reference implementation for Shor's algorithm for prime factorization which is based on a version of this algorithm proposed by Beauregard [2] and was implemented by Rui Maia and Tiago Leão [21]. As of now, Qiskit does not contain a program solving the discrete logarithm problem, which prompts for the implementation of Shor's algorithm for the discrete logarithm problem in Qiskit.

The goal of this work is to provide such a reference implementation of Shor's algorithm for the discrete logarithm problem in Qiskit. Several variants of this algorithm have been proposed, some of which are presented and implemented as part of this work. One particular challenge at the moment is that existing quantum computers are very small in the number of quantum bits, also referred to as *qubits*, that can be used for computation.

[1] `https://qiskit.org/`

Furthermore, simulating quantum algorithms using a large number of qubits on a classical computer is very inefficient, if not practically impossible, because of the exponential overhead required to represent a quantum state. Therefore, proposals for implementations of quantum algorithms aim to keep the number of used qubits as small as possible [2, 11, 16].

Both of Shor's algorithms employ a subroutine that performs the modular exponentiation operation $a^x \bmod m$, where x is an input stored in a quantum register. The overall space complexity of the algorithm, as given by the number of used qubits, is influenced both by the amount of qubits required to perform phase estimation and by the number of qubits needed to perform this modular exponentiation operation. Quantum algorithms, similar to classical algorithms, often use temporary storage space. Keeping the number of temporary storage qubits, or ancilla qubits, that are used for modular exponentiation minimal, also improves the space complexity of the overall algorithm.

There are various different approaches on how to perform arithmetic on a quantum computer which can be employed to perform modular exponentiation. Some rely on circuits that behave very similar to classical processes and, for example, employ carry qubits to perform addition [16]. Others make use of peculiarities of quantum systems to perform operations that are not possible on classical computers to produce a result [11]. For this work, three different proposals for implementations of modular exponentiation are examined. In addition to providing implementations in Qiskit, a goal of this work is to compare these approaches regarding their resource usage.

Decreasing the number of qubits used in an algorithm sometimes goes hand in hand with an increase in its time complexity. Therefore, the comparison of the proposed implementations discusses these effects as well.

Ultimately, the thesis provides a reference implementation of Shor's algorithm of the discrete logarithm problem in Qiskit. This implementation is able to compute the discrete logarithm using all implementation variants and all modular exponentiation gates presented in this text. This enables an overall comparison of the performance of the different versions.

This thesis starts out by giving the required number-theoretic background for the discrete logarithm problem as well as by introducing the concept of quantum computing and the utilized complexity measures in Chapter 2. It continues with presenting work related to this topic and gives a brief overview of existing implementation proposals for this algorithm in Chapter 3. Chapter 4 covers Shor's algorithm as an application of the phase estimation process in detail and examines its success probability as well as its complexity. A detailed description of the different modular exponentiation circuits used for the algorithm, along with an analysis of their complexity in the number of qubits used and in the number of operations applied, is then given in Chapter 5. Finally, these implementations and the different variants of the overall algorithm presented in this text are compared in Chapter 6.

CHAPTER 2

Background

To aid the presentation of phase estimation and its application to the discrete logarithm problem, relevant background information on both the nature of the problem and on quantum computing is given in this chapter.

The discrete logarithm problem as it is presented in [31] concerns the fields of integers modulo a prime p. These fields and in particular its multiplicative group share some interesting properties which are repeatedly applied both in Chapter 4 and in Chapter 5 of this text. For this reason, this chapter provides a recapitulation of the theorems and properties used in this thesis before introducing the discrete logarithm problem formally.

As Shor's algorithm for the discrete logarithm problem is a quantum algorithm, the most important concepts in quantum computing are introduced in Section 2.2. At the time when Shor's work was presented, the notion of a quantum computer still was purely hypothetical. However, today there are, albeit small in the number of available quantum bits, functioning quantum computers as well as powerful simulators which can be used to implement and study quantum algorithms. In Section 2.3 one such development environment, Qiskit[1], is introduced.

By providing different variants of the algorithm as implementations in Qiskit, their complexity both in the number of operations performed as well as in the number of quantum bits required to perform the algorithm is studied. A summary of the theoretical foundation of quantum complexity is therefore given in Section 2.4.

2.1 The Discrete Logarithm Problem

Before presenting the discrete logarithm problem in detail, some number-theoretic theorems and properties will be presented, which are used repeatedly in this work.

[1]`https://qiskit.org/`

Some proofs are omitted to aid readability, although references where those proofs can be found are given.

The first property of importance for this work is the algebraic structure of the integers modulo a prime number p.

Theorem 2.1.1. *For a prime number p the integers modulo p, denoted as $(\mathbb{Z}_p, +, \cdot)$, together with the operations $+$ and $\cdot$ representing modular addition and multiplication, form a field.*

Of particular interest is the property that each element $a \in \mathbb{Z}_p \setminus \{0\}$ has a multiplicative inverse element. Such an element $a^{-1} \in \mathbb{Z}_p \setminus \{0\}$ exists if and only if the congruence

$$aa^{-1} \equiv 1 \mod p$$

admits a solution. Using Euclid's algorithm (see Chapter 1 in [1]) it is possible to find integers e and f such that

$$\gcd(x, y) = ex + fy$$

for any integers x, y. Since p is prime, $\gcd(a, p) = 1$, there are integers e and f such that

$$1 = ea + fp.$$

Rewriting this equation yields $(-f)p = ea - 1$ which is equivalent to the congruence $ea \equiv 1 \mod p$, giving the inverse element $a^{-1} \equiv e \mod p$.

In slightly other words, the fact that every nonzero element in this structure has an inverse element shows that $\mathbb{Z}_p \setminus \{0\}$, together with modular multiplication forms a group, denoted by $(\mathbb{Z}_p \setminus \{0\}, \cdot)$.

However, some of the definitions presented next hold not only in a field but also in a ring, where not every nonzero element is invertible.

Definition 2.1.1 (Group of units)**.** For any integer module m, let $\mathbb{Z}_m^*$ be the set of invertible elements with respect to multiplication modulo m:

$$\begin{aligned}\mathbb{Z}_m^* &= \{a \in \mathbb{Z}_m \mid \exists a^{-1} : aa^{-1} \equiv 1 \bmod m\} \\ &= \{a \in \mathbb{Z}_m \mid \gcd(a, m) = 1\}.\end{aligned}$$

Together with modular multiplication, this set forms the commutative group $(\mathbb{Z}_m^*, \cdot)$, referred to as the *group of units*.

To judge the size of the group of units for arbitrary modules, Euler's totient function [1] can be used.

Definition 2.1.2. *Euler's totient function* $\phi(m)$ gives the number of integers $1 \leq x < m$ that are relatively prime to m. More formally,

$$\phi(m) = \Big|\{x \in \mathbb{N} \mid x < m, \gcd(x, m) = 1\}\Big|.$$

The definition shows that $\phi(m)$ gives the number of elements in the group of units $\mathbb{Z}_m^*$.

Euler's totient function can be calculated by using the following two facts [1].

- If $\gcd(a,b) = 1$, then $\phi(ab) = \phi(a)\phi(b)$.
- For a prime p and $e \in \mathbb{N}^+$, $\phi(p^e) = p^e\left(1 - \frac{1}{p}\right)$.

The group of units, as well as subgroups of it, are of particular interest for the problem studied in this text. Let g be an element from the group of units. By repeatedly multiplying g with itself a sequence of integers

$$g^1 \bmod m,\ g^2 \bmod m,\ g^3 \bmod m,\ \ldots$$

is generated. Since $\mathbb{Z}_m^*$ is a group, it is closed under application of the group operation, which, since $\mathbb{Z}_m^*$ itself is finite, implies that this generated set is also finite. In fact, it can even be shown that this set contains the identity element and forms a group with modular multiplication.

Definition 2.1.3 [20]. A *cyclic* group $G = \langle g \rangle$ is a multiplicative group G containing an element $g \in G$ such that, for all $b \in G$, there is an $i \in \mathbb{N}^+$ with $g^i = b$. The element g is referred to as the *generator*.

Theorem 2.1.2. *Any $g \in \mathbb{Z}_m^*$ generates a cyclic subgroup $\langle g \rangle \subseteq \mathbb{Z}_m^*$.*

Proof. As noted above, the set of elements generated by g is finite and closed under modular multiplication. Also, the associative law holds because it holds in $(\mathbb{Z}_m^*, \cdot)$.

The first property that remains to be proven is the existence of an identity element. As this set is finite, $|\langle g \rangle| = k$ for some $k \in \mathbb{N}$ can be asserted. The way of constructing the elements for this set as powers of g allows listing $k+1$ elements as

$$g^1 \bmod m, g^2 \bmod m, \ldots, g^k \bmod m, g^{k+1} \bmod m.$$

Obviously, there has to be a duplicate element $g^i \equiv g^j \mod m$, $1 \le i < j \le k+1$. As $g^i \in \mathbb{Z}_m^*$, there exists an inverse element $g^{-i} \in \mathbb{Z}_m^*$. Multiplying with this element on both sides yields

$$\begin{aligned} g^i g^{-i} &\equiv g^j g^{-i} \mod m \quad \text{if and only if} \\ 1 &\equiv g^{j-i} \mod m. \end{aligned}$$

Since for $i < j$, $1 \le j - i \le k+1$ holds, the element $g^{j-i} \equiv 1 \mod m$ was part of this list and therefore is in $\langle g \rangle$.

The second property that remains to be shown is the existence of inverse elements for all elements in $\langle g \rangle$. There is an $r \in \mathbb{N}$ such that $g^r \equiv 1 \mod m$. Let $g^e \in \langle g \rangle$ be arbitrary and let f be an integer such that $e + f \equiv 0 \mod r$, which implies $e + f = rq$, for some $q \in \mathbb{Z}$. Examining the element g^{e+f} gives

$$\begin{aligned} g^{e+f} &\equiv g^{rq} \mod m \quad \text{if and only if} \\ g^e g^f &\equiv 1^q \mod m \quad \text{if and only if} \\ g^e g^f &\equiv 1 \mod m, \end{aligned}$$

which shows that the inverse element of g^e can be expressed as a power of g and is therefore also in $\langle g \rangle$. □

Note that in this text G or $(G, \cdot)$ refers to an arbitrary group with some group operation $\cdot$, while $\langle g \rangle$ usually only refers to the subgroup generated by g. Some of the following properties refer to arbitrary groups $(G, \cdot)$ and can therefore also be applied to the special case $\langle g \rangle$.

Not all units generate subgroups of the same size, and to express these differences the concept of the *order* is introduced.

Definition 2.1.4 (Order of a group element [20])**.** Let $(G, \cdot)$ be a finite group and $x \in G$ be an arbitrary element of that group. The size of the subgroup $\langle x \rangle$ generated by x is referred to as the *order* of x.

$$\mathrm{ord}_{(G,\cdot)}(x) = |\langle x \rangle| = \min\{r \geq 1 \mid x^r = 1\}$$

The order of an element in $\mathbb{Z}_p^*$ is of particular interest for the algorithms proposed by Shor [31]. For the algorithm for prime factorization, the quantum computer is responsible for finding the multiplicative order of a group element modulo the integer that is to be factored, as this seems to be a computationally difficult task for a classical computer. Furthermore, in the algorithm for the discrete logarithm problem, the order of a generator is part of the eigenvalue that is estimated as described in Section 4.3.

A useful relation can be given between the order of a particular element and the number of elements in a group.

Definition 2.1.5 (Order of a group)**.** Let $(G, \cdot)$ be a finite group, the number of elements in the group $|G|$ is referred to as the *order* of G.

Theorem 2.1.3. *Let $(G, \cdot)$ be a finite group with order $|G|$ and let H be a subgroup of G. Then the order of H divides the order of G.*

For the proof of this theorem refer to [20]. In particular, this theorem shows that the order of any generator in a group divides the order of the whole group. This fact allows for a characterization of the orders of elements in the group studied in this thesis, $(\mathbb{Z}_p^*, \cdot)$.

As every integer $0 < x < p$ is coprime to the prime p, $\mathbb{Z}_p^*$ contains exactly $p - 1$ elements. Furthermore, each element $g \in \mathbb{Z}_p^*$ is a generator of some subgroup $\langle g \rangle \subseteq \mathbb{Z}_p^*$ with order $|\langle g \rangle| = r$ such that $r \mid p - 1$.

Having covered the required background, the discrete logarithm, as well as the problem of finding the discrete logarithm, are presented. The discrete logarithm can be seen as the number-theoretical equivalent to the logarithm in analysis.

Definition 2.1.6 (Discrete logarithm)**.** If p is prime, $g \in \mathbb{Z}_p^*$ is a generator and b is an integer such that $g^m \equiv b \mod p$, then m is referred to as the discrete logarithm.

The goal of the discrete logarithm problem (DLP) is to find the value m given p, g and b.

To give an example, consider the field of integers modulo 13:

$$\mathbb{Z}_{13} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}.$$

Since 13 is prime, every nonzero element of this field generates a subgroup of the group of units $\mathbb{Z}_{13}^* = \{1, 2, \ldots, 11, 12\}$ (see Theorem 2.1.2). Let the generator g be 5 for this example, then the generated subgroup has the form $\langle 5 \rangle = \{5^0, 5^1, 5^2, 5^3\} = \{1, 5, 12, 8\}$. Using this representation it is trivial to see that the discrete logarithm of the integer 12 relative to the base 5 and the module 13 is 2, since $5^2 \equiv 12 \bmod 13$.

Unfortunately, in the worst case, this subgroup contains $\phi(p) = p - 1$ elements. For this reason, the naive solution of simply enumerating the elements to find the discrete logarithm would take exponential time in the number of bits required to represent p.

There are other more advanced approaches to solving the problem of finding m. Yet there still is no polynomial-time algorithm for solving this problem on a classical computer, which is why algorithms for the DLP using quantum computers have been proposed.

To conclude the introduction of the DLP, one application of this problem is given to highlight the significance of fast algorithms for solving it. This application is the Diffie-Hellman key exchange protocol [10]. This system ensures that two communication partners A and B can agree on an encryption key over an insecure medium without the key itself ever being transmitted. To do that, they pick a sufficiently large prime p as well as some generator g. As the size of the subgroup generated by g is important for the security of the protocol, $|\langle g \rangle| = p - 1$ should hold. This negotiation is performed in plaintext, meaning both g and p are public.

Both A and B then individually pick a private integer X_A and X_B from $\{1, \ldots, p-1\}$. Using these, A computes $Y_A = g^{X_A} \bmod p$ and B computes $Y_B = g^{X_B} \bmod p$. By sharing Y_A and Y_B publicly, both A and B can obtain their shared key $K_A = Y_B^{X_A} \bmod p$ and $K_B = Y_A^{X_B} \bmod p$ respectively.

By substituting the definition of Y_A and Y_B it can be shown that the key each communication partner computes is equal to the key the other party obtains:

$$K_A = Y_B^{X_A} \bmod p = g^{X_B X_A} \bmod p = Y_A^{X_B} \bmod p = K_B.$$

This equality holds since the multiplication is commutative and it can be defined similarly for other structures where this is the case.

This system ensures that no third party can obtain the shared key $K_A = K_B$, since it can only be computed using the private integers X_A or X_B. However, as Diffie and Hellman note [10], if an efficient algorithm for computing the discrete logarithm is found, the private integers X_A and X_B can be obtained from Y_A and Y_B, which would allow a third party to compute the shared key $K_A = K_B$.

2.2 Quantum Computing

While in a classical computer each bit is always in either the state 0 or 1, the state of a quantum bit, or qubit, can be described as

$$\alpha |0\rangle + \beta |1\rangle .$$

Here α and β are complex numbers, for which the relation

$$|\alpha|^2 + |\beta|^2 = 1$$

holds, where $|\cdot|$ is the absolute value. The state of a qubit can therefore be expressed as an element of a two-dimensional complex vector space with basis $\{|0\rangle, |1\rangle\}$, which is also referred to as $\mathbb{H}$.

As in classical computation, algorithms usually require more than a single qubit. Such a collection of qubits is referred to as a quantum *register*. For a register of size n, its state can be expressed as a linear combination of elements of the *computational basis* $\{|0\rangle, |1\rangle, \ldots, |2^n - 1\rangle\}$ using complex coefficients α_x [7].

$$|\psi\rangle = \sum_{x=0}^{2^n-1} \alpha_x |x\rangle$$

Similar to the one qubit case, the following relation

$$\sum_{x=0}^{2^n-1} |\alpha_x|^2 = 1 \tag{2.1}$$

has to hold for these amplitudes. The collection of coefficients together forms an element in a 2^n-dimensional vector space. To more formally describe this vector space, the tensor product [25] is introduced.

Definition 2.2.1. Let V and W be n- and m-dimensional vector spaces over the same field with orthonormal bases $\{|v_1\rangle, \ldots, |v_n\rangle\}$ and $\{|w_1\rangle, \ldots, |w_m\rangle\}$ respectively. Then $V \otimes W$ gives an nm-dimensional vector space with basis $\{|v_i\rangle \otimes |w_j\rangle \mid 1 \leq i \leq n, 1 \leq j \leq m\}$.

Just as the elements in both V and W can be expressed as linear combinations of the given basis vectors, the elements in $V \otimes W$ can be expressed as linear combinations of the combined basis vectors $|v_i\rangle \otimes |w_j\rangle$.

Concrete elements of $V \otimes W$ can also be obtained using the tensor product of vectors $|v\rangle \in V$ and $|w\rangle \in W$ as

$$|v\rangle \otimes |w\rangle = \left(\sum_{|v_i\rangle} a_i |v_i\rangle\right) \otimes \left(\sum_{|w_j\rangle} b_j |w_j\rangle\right) = \sum_{|v_i\rangle} \sum_{|w_j\rangle} a_i b_j |v_i\rangle \otimes |w_j\rangle .$$

If it is clear from the context, the tensor product of two vectors is also sometimes given as $|v\rangle |w\rangle$, $|v, w\rangle$ or even $|vw\rangle$ if the binary representation of the state is of interest.

A similar operation is defined for matrices A and B [25]:

$$A \otimes B = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{pmatrix} .$$

Here the entries of the resulting matrix are all elements of A multiplied with all elements of B. As with vectors, the dimension of the resulting matrix is the product of the dimensions of the original matrices. This operation is particularly useful as it allows for a parallel combination of operators when examining quantum algorithms as will be shown later.

Extending a quantum register by additional qubits extends the dimension of the vector space representing the register states. Using the concept of the tensor product, an n-qubit register state is an element in the vector space obtained by applying the tensor product of $\mathbb{H}$ with itself n times:

$$|\psi\rangle \in \mathbb{H} \otimes \dots \otimes \mathbb{H}$$

or equivalently

$$|\psi\rangle \in \mathbb{H}^{\otimes n}.$$

This also gives another explanation of the nature of the computational basis: The entries of the computational basis are the decimal representations of the basis vectors obtained using the tensor product.

$$\begin{aligned} &\{|0\rangle \otimes \dots \otimes |0\rangle , |0\rangle \otimes \dots \otimes |0\rangle \otimes |1\rangle , \dots , |1\rangle \otimes \dots \otimes |1\rangle\} \\ &= \{|0\rangle , |1\rangle , \dots , |2^n - 1\rangle\} \end{aligned}$$

Similar to the state of one register, the state of a whole m qubit system can also be described as an element in $\mathbb{H}^{\otimes m}$.

Before going on to show how computational results can be extracted from a quantum system, one important property of the nature of quantum states is defined. Up until now, it has been shown that a state of an n qubit system $|\psi\rangle$ can be expressed as a linear combination using the states in a basis of $\mathbb{H}^{\otimes n}$. Furthermore it was noted that these basis states can be obtained by the tensor product of basis states in $\{|0\rangle , |1\rangle\}$. It is, however, important to note that not all states can themselves be described as the tensor product of states in lower-dimensional vector spaces.

Definition 2.2.2 (Entanglement [25])**.** Consider $\mathbb{H}^{\otimes n}$ and $\mathbb{H}^{\otimes m}$ and the tensor product space $\mathbb{H}^{\otimes n} \otimes \mathbb{H}^{\otimes m} = \mathbb{H}^{\otimes(n+m)}$. Let $|\psi\rangle \in \mathbb{H}^{\otimes(n+m)}$, then $|\psi\rangle$ is *separable* if and only if there are $|\phi_1\rangle \in \mathbb{H}^{\otimes n}$ and $|\phi_2\rangle \in \mathbb{H}^{\otimes m}$ such that $|\psi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle$. If there are no such elements $|\phi_1\rangle$ and $|\phi_2\rangle$, then $|\psi\rangle$ is *entangled.*

It depends on how a specific vector space $\mathbb{H}^{\otimes n}$ is factored to determine if an element of this vector space is entangled. Consider the three-qubit state $|\psi\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |110\rangle) \in \mathbb{H}^{\otimes 3}$. This state can be separated into $|\phi_1\rangle \in \mathbb{H}^{\otimes 2}$ and $|\phi_2\rangle \in \mathbb{H}^{\otimes 1}$ as

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \otimes |0\rangle = |\phi_1\rangle \otimes |\phi_2\rangle \,,$$

however there is no possible separation into $\mathbb{H}^{\otimes 1} \otimes \mathbb{H}^{\otimes 2}$.

The state $|\phi_1\rangle$ in this example is one of the so-called Bell states [25] and is more commonly referred to as $|\beta_{00}\rangle$. It will be used repeatedly in this chapter for various examples. The fact that states, such as the Bell states, are entangled, has ramifications for the outcome of measurements as it is shown in the next subsection.

2.2.1 Measurement

Unlike classical computation, the state of a quantum system as described by the coefficients of the basis vectors can generally not be fully extracted, or observed, during the computation. Assuming the system is in state $|\psi\rangle = \sum_{b\in B} \alpha_b \, |b\rangle$, where B is any basis of $\mathbb{H}^{\otimes n}$, a measurement operation is only able to determine one basis vector. The exact basis vector $|b\rangle$ that is determined is chosen using a probability proportional to the absolute value squared of the coefficient α_b [25].

As the description suggests, measurements can be performed in various bases. However, for the purposes of this text, measurements are always performed in the computational basis. When measuring all qubits of an arbitrary n qubit state

$$|\psi\rangle = \sum_{x=0}^{2^n-1} \alpha_x \, |x\rangle \,,$$

the measurement determines one element $|x\rangle$ from the computational basis. This means that some binary integer $0 \leq d < 2^n$ is returned. The exact element that is chosen is then determined by the coefficient α_x. Using this coefficient the probability of measuring each $|x\rangle$ can be expressed as

$$\Pr(d = x) = |\alpha_x|^2.$$

As Equation (2.1) has to hold for any state of a quantum system, it is assured that the probabilities for measurement sum up to 1.

It is, however, not necessary to always measure the whole system. To properly define the measurement outcome for the measurement of only a subset of the qubits in a system, the process of *projective measurement* is introduced for the computational basis.

It is defined by an *observable*, which is a Hermitian operator. Such an operator only possesses real eigenvalues and this observable can be given by its decomposition into its eigenvalues λ and projections into the respective eigenspaces:

$$H = \sum_{\lambda} \lambda P_{\lambda}.$$

These projections $P_{\lambda} : \mathbb{H}^{\otimes n} \to E_{\lambda}$, where E_{λ} is the eigenspace for the eigenvalue λ, are essential for the mathematical description of the measurement.

Definition 2.2.3 [25]**.** The probability that a measurement returns the specific eigenvalue λ for a state $|\psi\rangle$ is given by $\Pr(\lambda) = \langle\psi|P_{\lambda}|\psi\rangle = \|P_{\lambda}|\psi\rangle\|^2$, where $\||v\rangle\| = \sqrt{\langle v|v\rangle}$ is the norm of a vector $|v\rangle$ and $\langle \cdot | \cdot \rangle$ denotes the inner product.

As an example, consider the state $|\beta_{00}\rangle$ as presented above. To find the probability that the first qubit in the computational basis is $|0\rangle$, a projection into the eigenspace identified by the basis $\{|00\rangle, |01\rangle\}$ is defined, where in each basis vector the first qubit is $|0\rangle$.

$$P_{\lambda} = |00\rangle\langle 00| + |01\rangle\langle 01|$$

Applied to $|\beta_{00}\rangle$, the result of the projection is

$$\begin{aligned} P_{\lambda}|\beta_{00}\rangle &= (|00\rangle\langle 00| + |01\rangle\langle 01|)\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\ &= \frac{1}{\sqrt{2}}(|00\rangle\langle 00|00\rangle + |01\rangle\langle 01|00\rangle + |00\rangle\langle 00|11\rangle + |01\rangle\langle 01|11\rangle) \\ &= \frac{1}{\sqrt{2}}|00\rangle. \end{aligned}$$

Here the fact that $\langle x|y\rangle = 1$ for elements $|x\rangle$ and $|y\rangle$ of an orthonormal basis if and only if $|x\rangle = |y\rangle$, and it is zero otherwise, is used.

Now the probability to measure λ can be calculated as $\Pr(\lambda) = \|\frac{1}{\sqrt{2}}|00\rangle\|^2 = 1/2$. Therefore, half of the performed measurements of this state should lead to λ as the result. Usually not the eigenvalue of the observable is of interest, but the properties of the state it represents. In this case, it can be expressed as

$$\Pr(\text{the leftmost qubit is } |0\rangle) = \frac{1}{2}.$$

Another important property of quantum computation is the fact that this observation of the state itself influences the state of the system after the measurement. In a sense, a measurement of a subset of qubits might influence the state of other qubits in the resulting state. This fact can also be described using the projector given above.

Definition 2.2.4 (successor state [25])**.** The state immediately after a measurement of λ given the state $|\psi\rangle$ is given by

$$\frac{P_{\lambda}|\psi\rangle}{\|P_{\lambda}|\psi\rangle\|} = \frac{P_{\lambda}|\psi\rangle}{\sqrt{\Pr(\lambda)}}.$$

The ramifications of this rule are best illustrated by an example. Consider the state

$$|\gamma\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle),$$

where both qubits are in a superposition of both basis states.

A measurement of the first qubit can be conducted similar to the measurement performed above. The projection into E_λ gives $P_\lambda |\gamma\rangle = \frac{1}{2}(|00\rangle + |01\rangle)$, which implies the measurement probability

$$\Pr(\text{the leftmost qubit is } |0\rangle) = \left\| \frac{1}{2}(|00\rangle + |01\rangle) \right\|^2 = \sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2}^{\,2} = \frac{1}{2}.$$

By using the formula above, the state that remains after $|0\rangle$ has been measured in the leftmost qubit is

$$\frac{1/2\,(|00\rangle + |01\rangle)}{\sqrt{\frac{1}{2}}} = \frac{\sqrt{2}}{2}(|00\rangle + |01\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle) = |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

From this representation it is evident that the second qubit remains in a superposition of both basis states and each measurement outcome for the second qubit can be obtained with equal probability.

Furthermore, as the first qubit is now in state $|0\rangle$, repeating this measurement after once obtaining $|0\rangle$ for the first qubit will never change the measurement outcome. Although this qubit started in a state comprised of both basis states, only the basis state that has been measured can now be obtained when the measurement is repeated. The superposition has *collapsed* into the $|0\rangle$ state.

For entangled states, this rule has even further implications. Consider again the Bell state $|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. For both the first and second qubit, a measurement similar to above reveals that the probability of measuring $|0\rangle$ is $1/2$. Therefore both qubits are not in any basis state and the measurement outcome for both is uncertain.

Now apply the rule from Definition 2.2.4 assuming the first qubit was $|0\rangle$ when measured.

$$\frac{P_\lambda |\beta_{00}\rangle}{\sqrt{\frac{1}{2}}} = \frac{\frac{1}{\sqrt{2}}|00\rangle}{\frac{1}{\sqrt{2}}} = |00\rangle$$

After having measured the first qubit only, the state of the whole system is now $|00\rangle$. Therefore any subsequent measurement performed on both the first and second qubit leads to $|0\rangle$ with certainty. Although only one qubit was measured, the fact that they were entangled means that the state of the second 'untouched' qubit is now known without measuring.

2.2.2 Operations

Quantum algorithms are, similar to boolean circuits for example, represented by circuits that start with individual qubits in an initial state (usually $|0\rangle$), then perform a series of single or multi-qubit transformations and finally measure the result, which should give the solution to the problem that is to solve.

The individual qubits involved in the computation are represented by wires, while the performed transformations are represented by boxes, or in some cases circles, on those wires. For example, the circuit in Figure 2.1, when read left to right, performs the H operation on $|q_0\rangle$ as well as two X operations on $|q_1\rangle$, then it performs a transformation involving both qubits, before finally measuring the result.

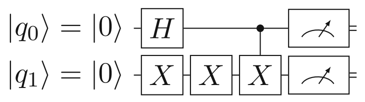


Figure 2.1: A quantum circuit performing transformations on two qubits before measuring the result.

As qubit states can be described as complex vectors, these transformations, or quantum *gates* can be described as complex matrices. Specifically, they are described as unitary matrices.

Definition 2.2.5 [7]. An operator U is unitary if and only if $U^\dagger U = I$, where $U^\dagger$ is the conjugate transpose matrix of U obtained as $\left(\overline{U}\right)^T$.

Using this representation of the gates in the circuit it is possible to mathematically describe how the state of the quantum system evolves during the execution of the algorithm. Figure 2.1 serves as an example of how this is done and how some elementary gates are described. For a more comprehensive overview of the most important gates refer to [33] and Chapter 4 in [25].

The first such transformation is the Hadamard gate H [25]. It is represented by the unitary

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

As seen in the example circuit in Figure 2.1, this operation is applied to the topmost qubit which starts in state $|0\rangle$. The state $|0\rangle = 1\,|0\rangle + 0\,|1\rangle$ is represented as a column vector $(1,0)^T$ and the result of the application of this gate is then obtained by the matrix multiplication

$$H\,|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)\,.$$

Similarly the application of the two X gates on the qubit $|q_1\rangle$ can then be described by the multiplications $XX|0\rangle$. However, to highlight an important property of quantum operations, the calculation of the state in the shown circuit will proceed differently.

Note that by Definition 2.2.5 any operator has an inverse obtained by the conjugate transpose. The gate X can be described as the gate that swaps the coefficients of $|0\rangle$ and $|1\rangle$ in the description of a state and is given by the matrix

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

Note that this gate is sometimes also visualized by the $\bigoplus$ symbol in circuit diagrams.

For X it holds that $X^\dagger = \left(\overline{X}\right)^T = X$, and therefore $X = X^{-1}$. This simplifies the derivation of the state after applying both operations as then $XX = I$ and $XX|0\rangle = I|0\rangle = |0\rangle$.

If the state of the whole system before the application of the transformations on both qubits is separable, these gates can be applied independently. However, should this not be the case, the gates can still be applied using their tensor product. In the example in Figure 2.1 the system evolves starting from the two-qubit input state $|0\rangle \otimes |0\rangle$ by first applying $(H \otimes X)$ and then applying $(I \otimes X)$. This resulting state can in total be described as $(I \otimes X)\big[(H \otimes X)(|0\rangle \otimes |0\rangle)\big]$, which is further simplified using the argument above to $(H \otimes I)(|0\rangle \otimes |0\rangle)$.

Therefore, after the application of all single-qubit gates in Figure 2.1, the system is in the state

$$(H \otimes I)(|0\rangle \otimes |0\rangle) = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \otimes |0\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |10\rangle\right).$$

The next gate that is applied in Figure 2.1 is a multi-qubit operation: $CNOT$, also referred to as CX [25]. This is a controlled operation, meaning another qubit controls the application of the X gate. In particular, it takes the state $|0\rangle \otimes |\psi\rangle$ to $|0\rangle \otimes |\psi\rangle$ and the state $|1\rangle \otimes |\psi\rangle$ to $|1\rangle \otimes X|\psi\rangle$.

Therefore the $CNOT$ gate is only applied if the control qubit is $|1\rangle$. Of course, the state of the control qubit could be in a superposition of both $|1\rangle$ and $|0\rangle$. To calculate the result of the application of this gate on such a state, either the matrix representation of the gate, or the fact that it can be expressed by its basis vectors is used.

Consider the arbitrary two qubit state $|\psi\rangle$ given by the coefficients $\alpha, \beta, \gamma, \delta \in \mathbb{C}$ in the computational basis as

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle.$$

Applying $CNOT$ using the first qubit as the control qubit and utilizing the fact that this is a linear transformation gives the resulting state

$$\begin{aligned} CNOT\,|\psi\rangle &= \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|1\rangle \otimes X\,|0\rangle + \delta\,|1\rangle \otimes X\,|1\rangle \\ &= \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|11\rangle + \delta\,|10\rangle\,. \end{aligned}$$

This scheme of determining the application of some gate by a control qubit is used repeatedly during various quantum algorithms. Furthermore, it is also possible to use multiple control qubits, in which case the gate is only applied for those basis vectors where all control qubits are one.

For the sake of completeness, the matrix representation of $CNOT$ is given by

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

Continuing with the example from Figure 2.1 by applying $CNOT$ after the application of all single-qubit gates gives

$$\begin{aligned} CNOT(I \otimes X)(H \otimes X)\left(|0\rangle \otimes |0\rangle\right) &= CNOT\left(\frac{1}{\sqrt{2}}\left(|00\rangle + |10\rangle\right)\right) \\ &= \frac{1}{\sqrt{2}}\left(CNOT\,|00\rangle + CNOT\,|10\rangle\right) \\ &= \frac{1}{\sqrt{2}}\left(|00\rangle + |1\rangle \otimes X\,|0\rangle\right) \\ &= \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right). \end{aligned}$$

Because this is exactly the Bell state $|\beta_{00}\rangle$ presented in the previous subsection, the circuit in Figure 2.1 can be used to prepare this state.

Lastly, measurement operations are performed. These measurements, given the state $|\beta_{00}\rangle$, according to the calculations in Section 2.2.1 give $|00\rangle$ or $|11\rangle$ with equal probability of $1/2$.

Reversible computation

Examining the properties of unitary operators (see Definition 2.2.5) more closely reveals another important characteristic of quantum computation: the state transformations are *reversible* [25]. Since $U^\dagger U = I$, the inverse U^{-1} of any transformation U always exists and can be obtained by $U^\dagger$.

In particular, since compositions and serial applications of unitary transformations are obtained by the tensor and matrix product as described above, any circuit can, excluding non-unitary measurement operations, be given as a unitary operator.

This means that for any circuit C performing $C\,|\phi_0\rangle = |\phi_1\rangle$, the inverse C^{-1} performing $C^{-1}\,|\phi_1\rangle = |\phi_0\rangle$ has to exist and is easily obtained.

On the one hand, this fact simplifies the construction of quantum algorithms as the inverse operation is often of interest. Consider a gate performing the addition $ADD(|a\rangle\,|b\rangle) = |a\rangle\,|a+b\rangle$. The inverse of this operation gives $ADD^{-1}(|a\rangle\,|a+b\rangle) = |a\rangle\,|b\rangle$, or expressed differently:

$$ADD^{-1}(|a\rangle\,|b\rangle) = |a\rangle\,|b-a\rangle\,.$$

This is a subtraction gate obtained by simply inverting the addition gate using the fact that this addition is reversible.

On the other hand, this need for reversible gates sometimes complicates the construction of algorithms considerably. Simply put, for a gate to be reversible, after the application of this gate, all information that is needed to restore the initial state has to be present. To give a simple example, consider the classical conjunction. It outputs 1 if and only if both inputs are 1, and outputs 0 otherwise. The truth table for this operation is given in Table 2.1.

A	B	$A \wedge B$
0	0	0
1	0	0
0	1	0
1	1	1

Table 2.1: Truth table for the classical conjunction.

If the result of this operation is 0, there are three different input states that could have caused this result. But from the result alone it is not possible to determine which one it actually was. Thus, this gate is not reversible.

More generally, such an operation can be represented by a function $f(\boldsymbol{x})$, with $\boldsymbol{x} \in \{0,1\}^n$. One measure that is taken to make this operation reversible, is to keep the input as part of the output in separate qubits and perform the map

$$|\boldsymbol{x}\rangle\,|y\rangle \rightarrow |\boldsymbol{x}\rangle\,|f(\boldsymbol{x})\rangle\,.$$

Herein, $|y\rangle$ represents a qubit in an arbitrary state before the operation is applied. This qubit contains the result afterwards. However from $|\boldsymbol{x}\rangle\,|f(\boldsymbol{x})\rangle$ the content of $|y\rangle$ still cannot be inferred. Therefore another measure has to be taken: The output qubit should not contain the result of the operation itself, but it is mapped to $|y \oplus f(\boldsymbol{x})\rangle$, where $\oplus$ represents the classical exclusive disjunction.

$\|a\rangle$	$\|b\rangle$	$\|c\rangle$	$\|a\rangle$	$\|b\rangle$	$\|c \oplus (a \wedge b)\rangle$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	**1**	**0**	**1**	**1**	**1**
1	**1**	**1**	**1**	**1**	**0**

Table 2.2: Truth table for the reversible version of the conjunction gate, the Toffoli gate.

For the classical conjunction this leads to a reversible operator with the truth table shown in Table 2.2.

The gate with this truth table is referred to as the Toffoli gate [25]. This table also gives another interpretation for this gate. The gate flips the last qubit if and only if both of the first two qubits are $|1\rangle$, as indicated by the two bold rows in the table. So this gate can be considered a version of the $CNOT$ gate that uses two control qubits instead of one.

2.3 Qiskit and IBM Quantum

As the development of quantum computers progresses there is also the need for software development kits. One of them is Qiskit[2], an open-source library[3] containing tools for the development and analysis of quantum algorithms.

Aided by these development tools, Qiskit also provides a comprehensive introduction into quantum computing, which contains programming examples as well as explanations of their functionality [29].

Using a simple Python interface, it is possible to build and visualize quantum circuits using a vast array of prebuilt gates. As an example, Listing 2.1 shows how a two-qubit quantum circuit which takes the system into the Bell state

$$|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle),$$

as shown in Figure 2.2, can be constructed.

[2]https://qiskit.org/
[3]https://github.com/Qiskit

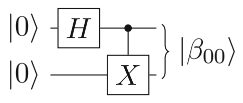


Figure 2.2: A circuit which takes the system to the Bell state $|\beta_{00}\rangle$.

```
reg = QuantumRegister(2)
circ = QuantumCircuit(reg)

circ.h(reg[0])
circ.cx(reg[0], reg[1])
```

Listing 2.1: Constructing a circuit that takes two qubits to the Bell state $|\beta_{00}\rangle$.

The constructed circuits can then be simulated in various ways to test their functionality and to gather more information on their properties.

One way of simulating the behavior of the circuit is to perform a state vector simulation [32]. This type of simulation returns a vector of complex numbers representing the coefficients of the basis states at the end of the circuit.

Performing such a simulation with the circuit as created with the code in Listing 2.1, returns a vector which matches $|\beta_{00}\rangle$ up to rounding differences due to the representation as floating-point numbers.

$$\begin{pmatrix} 0.70710678 + 0i \\ 0 + 0i \\ 0 + 0i \\ 0.70710678 + 0i \end{pmatrix} \approx \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

This state vector, and even other manually built state vectors, can further be used to initialize the state for another circuit, which helps in testing the stages of the algorithm presented in Section 4.3 separately [32]. The second stage can be executed and analyzed independently from the first stage, by using the state vector at the end of the first stage to initialize the second stage. In a sense, the simulator can be parametrized to take care of state preperation which of course is not possible on a real device.

The second important simulation result is obtained by saving the unitary matrix for a circuit [32]. For the circuit in Listing 2.1, the returned matrix has the form

$$U = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 \end{pmatrix}.$$

This matrix is obtained by using the fact that parallel executions of gates can be described by the tensor product of those gates and subsequent applications of gates can be seen as the matrix product (see Section 2.2.2). To verify this result, the unitary for the presented circuit can be manually calculated.

Note that care has to be taken since the tensor product order of Qiskit's simulation result is different from the order as presented until now. The topmost qubit in the circuit always represents the least significant bit b_0 of a binary number $b_n b_{n-1} \dots b_1 b_0$ and the tensor product is given accordingly as $|b_n\rangle \otimes |b_{n-1}\rangle \otimes \dots \otimes |b_1\rangle \otimes |b_0\rangle$. Here the topmost qubit in the circuit is the rightmost vector in the tensor product. This contrasts the convention used until now where the topmost qubit was the leftmost qubit in the tensor product representation. Therefore, in the following derivation of the state as simulated by Qiskit, the order of the operators in the tensor product has to be changed accordingly.

$$\begin{aligned}
CNOT(I \otimes H) &= (I \otimes |0\rangle\langle 0| + X \otimes |1\rangle\langle 1|)(I \otimes H) \\
&= I \otimes (|0\rangle\langle 0| H) + X \otimes (|1\rangle\langle 1| H) \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} + \\
&\quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 0 \\ 1 & -1 \end{pmatrix} \\
&= \frac{1}{\sqrt{2}} \left[\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix} \right] \\
&= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 \end{pmatrix}
\end{aligned}$$

The representation of the circuit by its unitary matrix U also shows that the other Bell states can be obtained by the same circuit by preparing a different input state. Similar to above, the order of the tensor product differes in this representation:

$$U(|1\rangle \otimes |0\rangle) = U \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = |\beta_{01}\rangle \,.$$

Obtaining the unitary matrix via the simulator is especially helpful when verifying circuits performing arithmetic as presented in Chapter 5. The matrix shows which input state in the computational basis is mapped to which output state, and therefore shows whether the performed calculation is correct.

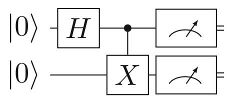


Figure 2.3: A circuit which takes the system to the Bell state $|\beta_{00}\rangle$ and then measures the result.

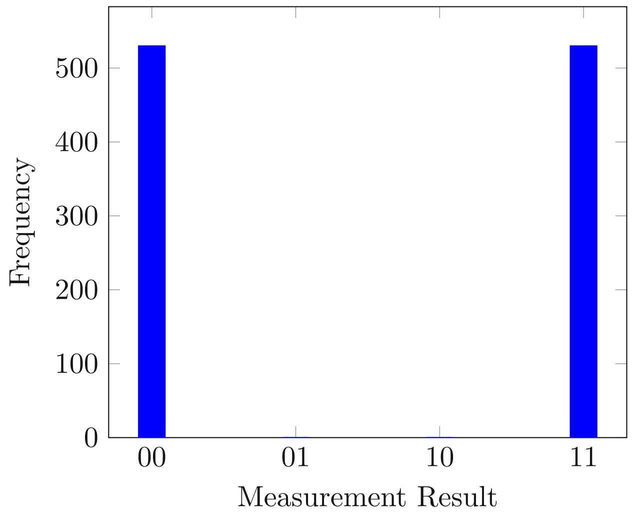


Figure 2.4: The simulated measurement result for the circuit in Figure 2.3.

Finally, it is also possible to obtain measurement results, similar to a real device, for a circuit.

Proceeding with the example presented in Listing 2.1, the function `circ.measure_all()` appends measurement operations to the end of the circuit, giving the circuit shown in Figure 2.3.

Qiskit, for both the simulation and for actual runs of the circuit on a quantum computing device, performs multiple shots and measures the result for each shot. It returns a dictionary mapping the measurement result to its observed frequency. For the circuit in Figure 2.3 the result for 1024 shots is plotted in Figure 2.4, which shows that $|00\rangle$ and $|11\rangle$ are observed with approximately equal probability.

2.3.1 Transpiler and IBM Quantum

One of the major features of Qiskit is the ability to transpile and run the circuits on actual quantum computers. IBM provides a number of quantum systems, which can be used remotely either using their online development tools[4], or by selecting their machines as the backend in Qiskit.

[4]`https://quantum-computing.ibm.com/`

Quantum computers generally only implement a small set of gates directly in hardware. For example, in the `ibmq_santiago` system these elementary gates are $CNOT$, I, RZ (the rotation around the Z-axis by arbitrary angles), $\sqrt{X}$ and X [26]. It has been shown that using only such a small universal set of operations, all gates can be approximated up to an arbitrarily small error [7, 25]. Therefore, one responsibility of the transpiler is to translate the conceptual circuit as described in Figure 2.3 to an equivalent circuit composed of these basis gates [37]. The transpiled circuit for this example is shown in Figure 2.5. As shown, this circuit after transpilation applies more gates to the top qubit as the Hadamard gate is transpiled into three different gates. This suggests that the circuit that is actually executed generally is more complex than the circuit constructed using Qiskit. How this affects the runtime bounds will be examined in Section 2.4.

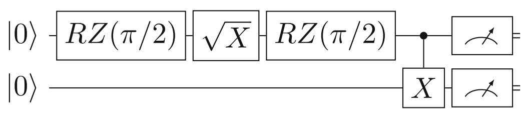


Figure 2.5: A circuit which takes the system to the Bell state $|\beta_{00}\rangle$ and then measures the result.

The only multi-qubit gate implemented in hardware is the $CNOT$ gate. Other multi-qubit gates such as the Toffoli gate are translated into a series of $CNOT$ and single-qubit gates. Furthermore, in circuits utilizing multiple qubits, the transpiler has to take special care regarding the assignment of the qubits in the conceptual circuit and in the actual circuit that is executed on the hardware [37]. This stems from the fact that the circuits constructed in Qiskit can assume to be executed on an 'ideal' system, where each qubit can be used with all other qubits to execute a $CNOT$. However this is not the case as the qubit map in [26] shows: Only qubits directly connected in this map can be used in the same $CNOT$ instruction.

This leads to another important task of the transpiler. It has to map the virtual qubits to their physical counterparts and introduce $SWAP$ gates if two disconnected qubits have to participate in a $CNOT$ operation. As introducing $SWAP$s again increases the complexity of the circuit it is beneficial if the transpiler finds an assignment to the physical system that minimizes the number of required $SWAP$s. Unfortunately solving this problem is NP-hard [37], which is why the transpiler in Qiskit employs a heuristic algorithm to find a good, but not necessarily the best, solution.

Finally, the transpiler also tries to apply certain optimizations to remove unnecessary operations and to decrease the size of the resulting circuit [37].

2.3.2 Reset

One operation that has not yet been presented and that can be performed on both the IBM Quantum systems as well as on the simulator is the reset instruction [8]. This instruction is used in the version of the algorithm studied in this text that uses only one input register (see Section 4.3.4). Hereby the reset instruction enables the system to use the same input register for both stages of the algorithm.

The reset instruction is comprised of a measurement followed by an X gate conditioned on the outcome of the measurement [8]. This way the X gate is applied only if $|1\rangle$ is measured, and is omitted if $|0\rangle$ is measured. After the application of the reset gate, the qubit continues in state $|0\rangle$ in all cases. Since information about the state before the application of this instruction is lost, this operation is not unitary and must be treated similar to a measurement.

2.4 Complexity

An important result of Shor's work [31] is that there is a quantum computer algorithm that solves the factorization problem in polynomial time, as opposed to superpolynomial time as it is required for the best-known classical algorithms.

To show this property for the algorithms presented in this work, first the notions of runtime and space complexity based on the representation of quantum algorithms as circuits are examined.

In classical computation, space complexity describes the number of bits of storage that are required to perform a given algorithm. More formally, it can be described as the number of tape cells a Turing Machine requires to perform the algorithm. Very similarly, for a given quantum circuit, the number of qubits involved in the computation defines the space complexity of a quantum algorithm.

Time complexity refers to the number of elementary operations that are performed before the algorithm terminates. For a Turing Machine, this is the number of steps, or applications of the state transition function, required to reach a halting state.

This measure can be described in terms of steps performed by a quantum Turing Machine (QTM) [4]. The definition of a QTM and its properties, however, exceeds the scope of this work, thus a different, yet equivalent model [38] is used to define time and space complexity. Since up until now quantum algorithms were described as quantum circuits, the *quantum circuit model* serves as the base for defining the complexity of the studied algorithms.

This model is assumed to be capable of preparing a state in the computational basis. Furthermore, quantum circuits are able to perform elementary operations that together form a universal set of transformations and finally measure the result of the computation in the computational basis [25].

Even though the algorithms presented in this text can be used independently of the size of the input, the quantum circuits involved in this computation are always constructed for a specific input size. This contrasts classical algorithms as they are usually defined, which perform computations independent of the input size. For this reason, it is vital to specify properties of algorithms not depending on a specific quantum circuit but on a *family* of quantum circuits [25, 38]. When studying the time complexity of Shor's algorithm for the discrete logarithm problem, it is implicitly assumed that the size of the circuits in the family of circuits solving this problem for different input sizes n is considered.

To give properties regarding the size of such a quantum circuit, the size of its encoding as a binary string can then be used [38]. This number only depends on the number of different gate operations that are applied to calculate the result.

There has to be a process that constructs the specific circuits in the familiy of circuits. To ensure that the problem can be solved in polynomial time, this process must also be performed in polynomial time. Extending upon the notion of a family of circuits, such a collection is referred to as *polynomial-time uniform* if this is the case for each $n \in \mathbb{N}$ [38]. Note that since a polynomial-time algorithm can only construct circuits of polynomial size, therefore this notion also implies that the circuit size is polynomial.

Throughout this work, all examined quantum algorithms are given as descriptions on how to construct them. Therefore, although usually only the size of the quantum circuit itself is examined when specifying the complexity, the descriptions implicitly give a polynomial-time algorithm responsible for their construction. Thus the family of quantum circuits solving the discrete logarithm problem presented in this thesis is polynomial-time uniform.

As shown in the previous section, a circuit can increase in size after it has been transpiled. Due to the fact that only a limited universal set of instructions is available on the device this circuit is executed on, every transformation has to be described as series of those. Therefore the increase in the number of gate operations due to this translation has to be accounted for.

For this reason, all circuits presented in this thesis are composed of a series of operations which can be transpiled into a constant number of elementary gates. The circuit depth after transpilation then increases only by a constant factor, which is irrelevant for the complexity bounds since $O(c \cdot f(n))$ and $O(f(n))$ are identical for any c independent of n.

Similarly, also the classical postprocessing that is required for the algorithms in this text has to be examined with respect to its runtime complexity.

The complexity class generalizing polynomial-time algorithms for quantum computers is **BQP**. In terms of the quantum Turing Machine **BQP** is defined as "the set of languages that are accepted with probability $\frac{2}{3}$ by some polynomial-time QTM" [4]. Quoting essentially verbatim from [38], a similar definition for the circuit model can be given:

Definition 2.4.1. A language $L \subseteq \{0,1\}^*$ is in **BQP** if there exists a polynomial-time uniform family of quantum circuits $\{Q_n\}$ with the following properties.

- Each circuit Q_n takes n qubits as input and produces one output qubit.
- If $x \in L$, then $\Pr(Q_{|x|}(x) = 1) \geq 2/3$.
- If $x \notin L$, then $\Pr(Q_{|x|}(x) = 0) \geq 2/3$.

Herein, $\Pr(Q_{|x|}(x) = y)$ represents the probability of measuring y after applying the circuit $Q_{|x|}$ to the input state $|x\rangle$.

For this definition to be used regarding the problem of finding the discrete logarithm, the problem has to be redefined as a decision problem first. This can be achieved similar to the construction presented for the factoring problem in [38] by giving an upper bound for the result as follows.

Definition 2.4.2 (DLP as decision problem)
Input: A prime p, a generator $g \in \mathbb{Z}_p$, $b \in \mathbb{N}$ and $M \in \mathbb{N}$.
Question: Is there an $m \in \mathbb{N}$ with $m < M$ such that $g^m \equiv b \mod p$?

This version of the problem can be solved using the algorithm presented in this text with an additional step that compares the result with this upper bound.

The definition of the class **BQP** only concerns quantum circuits. However, as presented in Chapter 4, there is some classical postprocessing that has to be performed before the discrete logarithm is obtained. Therefore the involved quantum circuit in itself does not solve the discrete logarithm problem. The fact that, given a classical circuit, a reversible version of this circuit can be obtained with an overhead that is limited to a constant factor only [25], is used to address this problem. The quantum circuits in this thesis can be extended by reversible versions of the classical operations that have to be performed in the postprocessing phase to obtain a circuit that fits Definition 2.4.1.

Of course, this argument is only used for the sake of showing **BQP** membership and in actual implementations the postprocessing steps are performed classically.

The class **BQP** reflects the probabilistic nature of quantum computation and is chosen as the quantum equivalent of the class **BPP** which is defined similarly for probabilistic Turing Machines. Since their success probability can be increased to near certainty by repeating the algorithm, these classes represent problems that are efficiently solvable [25].

Bernstein and Vazirani show in [4] that **BPP** $\subseteq$ **BQP** and Shor's algorithm shows that there are problems in **BQP** that are not yet proven to also be in **BPP**. The pressing question is whether **BQP** $\neq$ **BPP**, in which case quantum computers are provably more powerful than classical computers. However, although there is evidence suggesting this fact [4], it is not yet shown whether or not **BQP** $\not\subseteq$ **BPP**.

To conclude this chapter another measure is introduced that, although it is not relevant for the definition of the circuit complexity, is of practical significance regarding the execution of quantum circuits: the circuit *depth*.

By dividing a given quantum circuit into 'time slices', where it is assumed that all gates in a given time slice are performed simultaneously, the number of such slices gives the circuit depth. In each of these slices, every qubit in the circuit can only participate in one quantum gate.

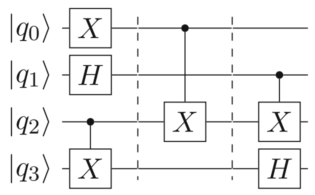


Figure 2.6: A quantum circuit with circuit depth 3. Note that although the qubits $|q_1\rangle$ and $|q_3\rangle$ are idle in the second time slice it is not possible to perform the next $CNOT$ operation in this time slice since the other qubit is participating in another operation. However it would be possible to perform the Hadamard transformation on $|q_3\rangle$ also in the second time-slice.

There is a close relationship between polynomial-time and polynomial-depth quantum circuits. Consider some circuit family $\{G_n\}$ with circuit depth given by the polynomial $f(n)$, trivially at most $f(n) \cdot n$ elementary operations can be performed, when each qubit participates in an operation in each time-slice. Conversely, let $g(n)$ be a polynomial that describes the circuit size in the number of elementary gate operations. The circuit depth for this circuit can be at most $g(n)$ if in each time-slice only one operation is executed.

CHAPTER 3

Related Work

Shor's work shows that two very important problems for the security of cryptographic protocols can theoretically be solved efficiently on a quantum computer [31]. This fact prompted a lot of interest in this field of study and lead to a number of works examining the properties of Shor's algorithm for prime factorization and Shor's algorithm for the discrete logarithm problem.

One important characteristic of both algorithms is that they can be seen as the application of a more general process called phase estimation [6, 23–25], which is covered in more detail in Section 4.2. Mosca describes the application of phase estimation to the discrete logarithm problem in [23], which, although the states during the execution of the algorithm are described differently, gives an equivalent quantum program to Shor's algorithm for the discrete logarithm problem. This text, as well as the descriptions of the process in [24] and [19], serve as the basis for the description and implementation of the algorithm in Chapter 4.

A main component of the phase estimation process is the quantum Fourier transform (QFT). This operation and its implementation on a quantum computer is described in detail for example by Cleve et al. [6], who also provide an overview of other important quantum algorithms. Furthermore, the details of the QFT and its applications in arithmetic algorithms are given by Draper [11].

There are various adaptions of the phase estimation algorithm aimed at reducing the size of the control register by restructuring the operations performed to obtain the estimate. Mosca and Ekert describe one such optimization based on the sequential nature of the quantum Fourier transform in [24]. Another optimization, which is presented by Kaliski in [19], reduces the quantum part of the algorithm to only obtain an estimate of one bit of the result and computes the overall discrete logarithm classically using the reduced circuit as an oracle.

To specify the runtime of both algorithms presented by Shor, the exact success probability of the quantum circuit is of interest. The description of the algorithms in [31] shows that, for both the factorization problem as well as the discrete logarithm problem, a polynomial number of repetitions of the quantum part leads to the result. Although Shor's lower bound for the success probability suffices to prove that this is a polynomial-time algorithm, it has been shown that the actual success probability is still underestimated and various works aim to prove more accurate bounds [13, 23].

Gerjuoy [14] as well as Bourdon and Williams [5] examine the probability of obtaining a divisor of the order of an element in Shor's algorithm for prime factorization in detail. They prove lower bounds of above 90% for the success probability of this process.

Both problems studied in Shor's paper [31] can be generalized as instances of the hidden subgroup problem [23, 24]. This of course leads to the question of whether or not other problems can be described using this formulation. A thorough overview of different problems that can be categorized as special cases of the hidden subgroup problem, as well as a general algorithm for this problem, is given in [23] and [24].

An integral part of the algorithm for prime factorization as well as the algorithm for the discrete logarithm problem is a subroutine that performs modular exponentiation. There are various implementation proposals for the algorithm for prime factorization which also describe how to build a circuit to perform this operation [2, 16, 34]. Since the algorithm for the discrete logarithm problem also makes use of such a modular exponentiation circuit, these proposals serve as important references for this thesis.

The modular exponentiation algorithm is usually composed of a series of modular multiplications, which in turn perform modular additions [2, 16, 34]. In recent years a number of ways of how to implement the addition of two quantum registers [11, 35] as well as the addition of a classical constant with a quantum register [2, 16, 30, 34] have been studied. For Shor's algorithm, the value that is added is usually known at the time when the quantum circuit is constructed. This addition with a classical constant is also sometimes referred to as *quantum-classical* addition [30] and the fact that it requires no second quantum register to hold the classical summand can be used to decrease the complexity of the circuit.

One way of performing arithmetic on a quantum computer is to use constructions similar to algorithms for classical computers. However, these algorithms usually use instructions that are not reversible. For this reason, a lot of research has been conducted in recent years on how to adapt these processes to obtain reversible quantum addition circuits.

One approach that was presented by Häner, Roetteler and Svore [16] is covered in more detail in Section 5.2.2. It is based on a gate for computing the final carry of an addition of a constant that is known at the time of construction and a quantum value. There are also other carry-based addition circuits and a good overview including their time and space complexity can be found in [30].

Although bitwise addition circuits, similar to those for classical systems, can be adapted for quantum computers, there is also research concerning the question of whether the special properties of quantum states can be used to perform arithmetic in other ways.

Draper [11] describes one such algorithm that employs the quantum Fourier transform, which is explained in more detail in the next chapter and in Chapter 5. Using the transformed representation of the input, the process of addition itself can be simplified to a series of controlled rotations for each qubit. In the case where quantum-classical addition is performed, and the other summand is known at the time of construction of the circuit, it even can be simplified to only one controlled rotation per qubit as shown in Section 5.2.1.

Rines and Chuang give a detailed description and comparison of various modular multiplication algorithms in [30], one of which, the Montgomery modular multiplication [22] using Fourier space addition, is presented in Section 5.4. They present three different modular multiplication algorithms: modular multiplication based on a division algorithm, modular multiplication based on Montgomery reduction and modular multiplication based on Barrett reduction. Montgomery reduction and Barrett reduction aim to speed up the calculation of the representative of the residue class modulo some integer n by avoiding or simplifying costly processes such as the trial divisions required for exact division [30]. The authors furthermore present implementations of these multiplication algorithms for carry-based adders as well as for Fourier transform based adders.

To find fast quantum multiplication algorithms it is helpful to examine the fastest known classical multiplication algorithms. The fastest multiplication algorithm for very large numbers is Schönhage-Strassen multiplication [31] and an adaption for quantum computers that performs two recursive steps of this process is proposed by Zalka [39]. However, as Zalka's version of the Schönhage-Strasse multiplication algorithm still uses significantly more qubits than the algorithms described above it was not considered for this work. Zalka also briefly investigates the possibility of implementing another fast classical multiplication algorithm, the Karatsuba-Ofman algorithm [39], but concludes that the very extensive space requirement outweighs its benefits.

This thesis concentrates on the discrete logarithm problem in the multiplicative group of integers modulo some prime p as described by Shor [31]. However, this problem can be defined for other groups as well. A trivial example is the additive group of integers modulo some integer n, $(\mathbb{Z}_n, +)$, where the solution can be obtained in polynomial time on a classical computer using Euclid's algorithm [27]. Other more complicated groups, where an efficient solution impairs the security of cryptographic protocols, are the elliptic curves over finite fields. Proos and Zalka present an adaption of Shor's algorithm for the discrete logarithm problem for elliptic curves over $\mathrm{GF}(p)$ in [27]. They make use of the fact that Shor's original algorithm does not use properties of the group itself to solve the problem, thus an adaption of the modular exponentiation gate, as well as an adapted representation of the group elements, leads to a solution [27].

To accurately estimate the runtime of the algorithms studied in this thesis, a solid theoretical foundation regarding computational complexity on quantum computers is required. The conceptual model that is usually employed to study computability and complexity for classical algorithms is the Turing machine. Its quantum equivalent, the quantum Turing machine, as well as complexity-theoretic results concerning the class **BQP** are described by Bernstein and Vazirani [4]. Cleve furthermore gives a good overview of different complexity scenarios that can be examined for quantum algorithms in [7]. Additionally, Watrous presents an introduction of quantum computation and quantum complexity using the circuit model in [38], which is also used in this thesis.

Lastly, Nielsen and Chuang [25] give an extensive introduction to the topic of quantum computing. Their book furthermore provides a comprehensive overview of most of the concepts discussed in Chapters 2 and 4 of this text and is therefore repeatedly referenced.

CHAPTER 4

Shor's Algorithm for the DLP

A quantum computer's ability to efficiently determine the period of a function enables it to solve some problems, such as the discrete logarithm problem, efficiently where no such efficient solution on classical computers is known [31]. This process relies heavily on the quantum Fourier transform (QFT) presented in Section 4.1. In addition to its ability to extract certain information about a unitary operator, as it is used in Shor's algorithm for the discrete logarithm problem, the QFT is also a vital part of many other algorithms, some of which are presented in Chapter 5.

The quantum algorithm used to compute the discrete logarithm will be presented as the application of a more general process: *phase estimation* (see Section 4.2). This operation is able to approximate eigenvalues of a unitary operator. This chapter will go on to show that a certain operator, modular exponentiation, will contain the discrete logarithm in its eigenvalues. This suggests that phase estimation can be used to extract this value and solve the discrete logarithm problem.

How this is achieved will be explained in detail in Section 4.3. Additionally, the success probability of this process, and based upon that, the runtime in the number of performed elementary operations is examined. Lastly, this chapter also covers some optimizations of this process which allows it to be performed using fewer qubits, thus reducing the overall space complexity.

4.1 The Quantum Fourier Transform

A vital part of many quantum algorithms is the quantum Fourier transform (QFT). On the one hand, the QFT enables quantum computers to transform particular tasks into other problems that can be solved more efficiently as is done in Section 5.2.1. The problem of addition is transformed into the task of phase shifting the input, which can be performed efficiently on a quantum computer. On the other hand, the QFT can be used to recover useful information about the state of the system as it is done in *phase estimation*.

The QFT performs a transformation that closely resembles the discrete Fourier transform in its mathematical description [25]:

$$|a\rangle \rightarrow \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} \exp\left(2\pi i \frac{ax}{2^n}\right) |x\rangle .$$

In this representation $|a\rangle$ represents an arbitrary element of the basis $\{|0\rangle, \dots, |2^n-1\rangle\}$ that is transformed into a superposition of those states. The fact that the matrix representation of the discrete Fourier transform is a unitary matrix suggests that this transformation can be implemented on a quantum computer.

To show how this gate can be constructed, the fact that the state after the transformation is separable [6, 25] can be used. Given the input $|a\rangle$, where a is a binary integer given by its individual bits a_i such that $a = a_n 2^{n-1} + a_{n-1}2^{n-2} + \dots + a_1 2^0$, the result $QFT|a\rangle$ can be expressed as follows:

$$\begin{aligned} \frac{1}{\sqrt{2^n}}\Big[\Big(|0\rangle + e^{2\pi i(0.a_n a_{n-1}\dots a_1)}|1\rangle\Big) \otimes \Big(|0\rangle + e^{2\pi i(0.a_{n-1}\dots a_1)}|1\rangle\Big) \otimes \\ \cdots \\ \otimes \Big(|0\rangle + e^{2\pi i(0.a_2 a_1)}|1\rangle\Big) \otimes \Big(|0\rangle + e^{2\pi i(0.a_1)}|1\rangle\Big)\Big]. \end{aligned} \tag{4.1}$$

The individual phases are given in a notation that represents binary fractions which is often used in literature to highlight the structure of the QFT. Hereby $0.a_l a_{l-1}\dots a_m$ represents the fraction $a_l/2 + a_{l-1}/4 + \dots + a_m/2^{l-m+1}$ [6, 11, 25]. This representation shows that the phase introduced for each qubit is only influenced by the value of the qubit itself and the values of the less significant qubits.

The quantum Fourier transform can be performed by using controlled phase gates[1], which map an arbitrary qubit state $\alpha|0\rangle + \beta|1\rangle$ to $\alpha|0\rangle + \exp(i\lambda)\beta|1\rangle$. The construction of the QFT circuit relies on phase gates that introduce a phase of the form $\lambda = 2\pi/2^k$, which will in the following figure be referred to as R_k, to reflect representations in literature [25].

To give an argument why this circuit performs the correct transformation consider some qubit $|a_j\rangle$, $1 \le j \le n$ in the input state $|a_n \dots a_2 a_1\rangle$. First, the Hadamard gate is applied which maps the state to $|0\rangle + (-1)^{a_j}|1\rangle$, since $a_j \in \{0,1\}$. Using the fact that $(-1)^{a_j} = \exp(\pi i a_j) = \exp(2\pi i(a_j/2))$ for $a_j \in \{0,1\}$, this expression can further be rewritten as $|0\rangle + \exp(2\pi i(a_j/2))|1\rangle$. Note that in this argument the factor $1/\sqrt{2}$ is omitted to improve readability.

[1] `https://qiskit.org/documentation/stubs/qiskit.circuit.library.PhaseGate.html`

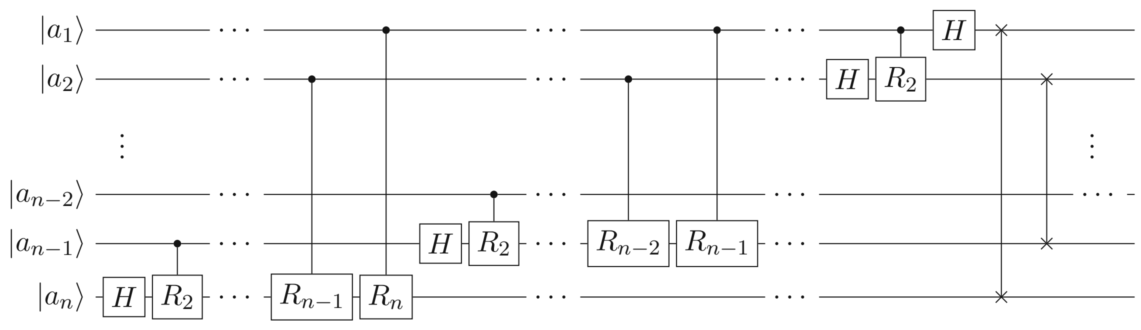


Figure 4.1: Circuit for the quantum Fourier transform with input $|a\rangle = |a_n \dots a_2 a_1\rangle$ as presented in [25]. Note that in contrast to usual representations, the least significant qubit a_1 is the topmost qubit, which reflects the qubit order in Qiskit.

Next, a phase gate with $\lambda = 2\pi/2^2$ is applied which is controlled by a_{j-1}. Since $\exp(2\pi i 0/2^2) = 1$ and because this expression extends the coefficient of $|1\rangle$ as a multiplicative factor, the state after this conditional rotation can be expressed as

$$|0\rangle + \exp\left(2\pi i \frac{a_j}{2}\right) \exp\left(2\pi i \frac{a_{j-1}}{2^2}\right) |1\rangle = |0\rangle + \exp\left[2\pi i \left(\frac{a_j}{2} + \frac{a_{j-1}}{4}\right)\right] |1\rangle .$$

Continuing until the last conditional rotation, the state is transformed to

$$|0\rangle + \exp\left[2\pi i \left(\frac{a_j}{2} + \frac{a_{j-1}}{4} + \dots + \frac{a_1}{2^j}\right)\right] |1\rangle ,$$

which can be represented using the notation for binary fractions again as

$$|0\rangle + \exp\left(2\pi i \left(0.a_j a_{j-1} \dots a_1\right)\right) |1\rangle .$$

After performing the conditional rotations as shown in Figure 4.1, all correct output states are present, yet they are not in the correct order. For example, the least significant qubit $|a_1\rangle$ is transformed to $|0\rangle + \exp(2\pi i(0.a_1))\,|1\rangle$, which should be the result for the most significant qubit $|a_n\rangle$ as shown in Equation (4.1). Therefore the outputs are swapped to their correct positions before concluding the circuit for the QFT.

However, this swap operation can be omitted in some instances. If the overall quantum algorithm finishes after the application of the QFT and the register is measured, then this reordering can be done classically. Also, if it is known that the order of the outputs has not been corrected and the rest of the quantum algorithm is adapted accordingly, the reordering can be omitted similarly, as it is done when performing addition using the QFT as shown in Section 5.2.1.

To give upper bounds for the runtime complexity of the QFT, it first has to be noted that the controlled rotations can be performed in $O(1)$ time using $CNOT$s and uncontrolled rotations. On each qubit $|q_i\rangle$, $i-1$ rotations and one Hadamard gate are applied, before finally performing $\lfloor n/2 \rfloor$ *SWAP*s. Therefore in total, the number of gates is given by

$$\left\lfloor \frac{n}{2} \right\rfloor + \sum_{i=1}^{n} i \leq \frac{n}{2} + \frac{n(n+1)}{2},$$

which shows that the runtime of the QFT is in $O(n^2)$.

Of course, since the QFT can be implemented on a quantum computer, there is an also an inverse operation. The inverse quantum Fourier transform, herein denoted as QFT^{-1}, is described as:

$$|a\rangle \to \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} \exp\left(-2\pi i \frac{ax}{2^n}\right) |x\rangle \, .$$

To conclude the introduction of the QFT, another useful equivalent representation [11] of the individual qubits in the separable state after the transformation is given here. This representation is used in Chapter 5.

Lemma 4.1.1. *Similar to the tensor product representation given in Equation* (4.1)*, the state $QFT|a\rangle$ can also be represented as the tensor product of individual qubit states denoted as $|\phi_k(a)\rangle$, where*

$$|\phi_k(a)\rangle = \frac{1}{\sqrt{2}} \left(|0\rangle + \exp\left(2\pi i \frac{a}{2^k}\right) |1\rangle \right) . \tag{4.2}$$

Proof. First, the value of $\frac{a}{2^k}$ is calculated as follows.

$$\begin{aligned}
\frac{a}{2^k} &= \frac{a_n 2^{n-1} + \cdots + a_1 2^0}{2^k} \\
&= a_n 2^{n-1-k} + \cdots + a_1 2^{-k} \\
&= a_n 2^{n-1-k} + \cdots + a_{k+2} 2^{k+2-1-k} + a_{k+1} 2^{k+1-1-k} + a_k 2^{k-1-k} + \cdots + a_1 2^{-k} \\
&= a_n 2^{n-1-k} + \cdots + a_{k+2} 2^1 + a_{k+1} 2^0 + a_k 2^{-1} + \cdots + a_1 2^{-k} \\
&= a_n 2^{n-1-k} + \cdots + a_{k+2} 2^1 + a_{k+1} 2^0 + \frac{a_k}{2} + \cdots + \frac{a_1}{2^{-k}} \\
&= a_n 2^{n-1-k} + \cdots + a_{k+2} 2^1 + a_{k+1} 2^0 + 0.a_k \ldots a_1
\end{aligned}$$

Substituted into the coefficient of $|1\rangle$ in $|\phi_k(a)\rangle$, the expression can further be simplified:

$$\begin{aligned}
\exp\left(2\pi i \frac{a}{2^k}\right) &= \exp\left(2\pi i a_n 2^{n-1-k}\right) \cdots \exp\left(2\pi i a_{k+2} 2^1\right) \cdot \exp\left(2\pi i a_{k+1} 2^0\right) \\
&\quad \cdot \exp\left(2\pi i (0.a_k \ldots a_1)\right) \\
&= \exp\left(\pi i a_n\right)^{2^{n-k}} \cdots \exp\left(\pi i a_{k+2}\right)^{2^2} \cdot \exp\left(\pi i a_{k+1}\right)^2 \cdot \exp\left(2\pi i (0.a_k \ldots a_1)\right) .
\end{aligned}$$

Since for any $a_i \in \{0, 1\}$, $\exp(\pi i a_i) \in \{1, -1\}$ and using the fact that the exponents for all terms but the last are even this simplifies to

$$\exp\left(2\pi i \frac{a}{2^k}\right) = \exp\left(2\pi i (0.a_k \dots a_1)\right),$$

which shows that in total the equality

$$\begin{aligned} |\phi_k(a)\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle + \exp\left(2\pi i \frac{a}{2^k}\right) |1\rangle \right) \\ &= \frac{1}{\sqrt{2}}\left(|0\rangle + \exp\left(2\pi i (0.a_k \dots a_1)\right) |1\rangle \right) \end{aligned}$$

holds. $\square$

4.2 Phase Estimation

Instead of immediately examining Shor's algorithm for the discrete logarithm problem, first a generalization of the approach used in the algorithm is presented: *phase estimation* [6, 24, 25].

This process is in itself not a complete description of a quantum algorithm as it assumes certain properties about the circuit that will be executed. First, it is assumed that a way to perform some transformation U in a controlled manner is given. This controlled gate is referred to as an *oracle*, and the goal of phase estimation is to estimate properties of the eigenvalues of this oracle.

In particular, it is assumed that U has an eigenvector $|\psi\rangle$ and that the corresponding eigenvalue is described as $e^{2\pi i \phi}$. Furthermore, it is required that there is a process that prepares the state $|\psi\rangle$.

The result of the phase estimation algorithm is then an estimate of the value ϕ and therefore an estimate of the eigenvalue as a whole.

The description of the algorithm as presented for example in [6] furthermore uses oracles that also perform controlled U^{2^i} transformations. Although performing such a transformation using a single gate is important for performance estimates as described later on, for the purpose of explaining phase estimation in general, it can be assumed that these transformations are simply repeated applications of the gate U using the same control input.

The process as shown in Figure 4.2 uses two registers, where the first register starts in the state $|0\rangle$ and the second register is prepared to start in the state $|\psi\rangle$. The size n of the first register is important for the quality of the estimate and will be examined more closely in the later part of this subsection.

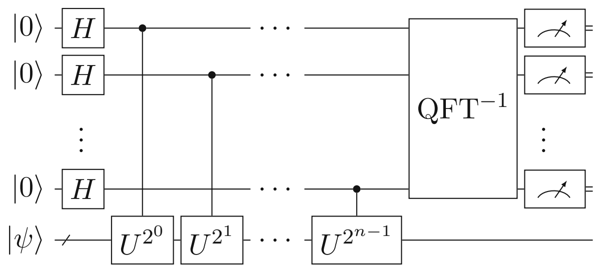


Figure 4.2: Quantum circuit for phase estimation, assuming the bottom register is prepared in the eigenstate $|\psi\rangle$ [6, 25]. The slashed line represents a register containing multiple qubits. If the actual register size is of interest, this slash is annotated with the number of qubits this line represents.

After applying Hadamard gates to the first register, the state of the system can be described as

$$\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle|\psi\rangle = \frac{1}{\sqrt{2}}(\,|0\rangle+|1\rangle\,)\otimes\cdots\otimes\frac{1}{\sqrt{2}}(\,|0\rangle+|1\rangle\,)\otimes|\psi\rangle\,.$$

The application of the ith controlled gate using the ith control qubit then gives

$$\begin{aligned} cU^{2^i}\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)|\psi\rangle &= cU^{2^i}\frac{1}{\sqrt{2}}(|0\rangle|\psi\rangle+|1\rangle|\psi\rangle) \\ &= \frac{1}{\sqrt{2}}\left(|0\rangle|\psi\rangle+|1\rangle U^{2^i}|\psi\rangle\right) \\ &= \frac{1}{\sqrt{2}}\left(|0\rangle|\psi\rangle+\exp\left(2\pi i\phi 2^i\right)|1\rangle|\psi\rangle\right) \\ &= \frac{1}{\sqrt{2}}\left(|0\rangle|\psi\rangle+\exp\left(2\pi i\frac{\phi 2^n}{2^{n-i}}\right)|1\rangle|\psi\rangle\right) \\ &= \frac{1}{\sqrt{2}}\left(|0\rangle+\exp\left(2\pi i\frac{\phi 2^n}{2^{n-i}}\right)|1\rangle\right)\otimes|\psi\rangle\,. \end{aligned}$$

In this description cU^{2^i} represents the controlled application of the gate U^{2^i}. The result of the application of the controlled gate shows that the phase factor that is introduced by U^{2^i} propagates from the second register to the first register.

The description of the state for one qubit after the application of the oracle is very similar to the description of the state after the QFT as presented in Lemma 4.1.1. Indeed, the state of the whole system after the application of the oracle gates using all control qubits can be given as

$$\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}\exp\left(2\pi i\phi x\right)|x\rangle = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}\exp\left(2\pi i\frac{\phi 2^n x}{2^n}\right)|x\rangle\,.$$

This fact suggests that in the case where $\phi 2^n$ is an integer, after applying the inverse QFT on the top register, ϕ can be acquired exactly by dividing the measured value by 2^n. However, if this is not the case, obtaining a good estimate of the value ϕ is still possible as the following argument shows.

Theorem 4.2.1 [6]. *The measurement result after phase estimation will be the best n-bit approximation of the phase ϕ with probability $\geq 4/\pi^2$.*

Proof. Consider the circuit implementing phase estimation in Figure 4.2 where the top register holds n qubits. The state of the system evolves as follows.

1. The system starts in the state $|0\rangle\,|\psi\rangle$.
2. A Hadamard gate is applied to each qubit in the top register. The resulting state is
$$\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle\,|\psi\rangle\,.$$
3. The oracle function is executed, thus introducing the eigenvalue as described above. The state is
$$\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}e^{2\pi i\phi x}\,|x\rangle\,|\psi\rangle\,.$$
4. Finally, QFT^{-1} is applied to the top register before measuring it, resulting in the state
$$\frac{1}{2^n}\sum_{x=0}^{2^n-1}\sum_{y=0}^{2^n-1}e^{\frac{-2\pi ixy}{2^n}}e^{2\pi i\phi x}\,|y\rangle\,|\psi\rangle\,.$$

Since $\phi 2^n$ is not an integer, this value can not be measured exactly. Therefore the best n-bit approximation of the phase is the closest integer to $\phi 2^n$, which is expressed as $y^* = \phi 2^n + \Delta$, where $0 < |\Delta| \leq \frac{1}{2}$. This representation allows for the phase to be expressed in terms of its best n-bit estimate
$$\phi = \frac{y^*}{2^n} + \delta.$$

Since $\delta = \Delta/2^n$, this error term is bounded by $0 < |\delta| \leq \frac{1}{2^{n+1}}$. Therefore, the size of the top register influences the quality of the estimate as a larger size n decreases the error term $|\delta|$. By using this representation of the phase in the description of the state, the probability for measuring y^*, denoted as $\Pr(y = y^*)$, can be given in terms of this error term:
$$\frac{1}{2^n}\sum_{x=0}^{2^n-1}\sum_{y=0}^{2^n-1}e^{\frac{-2\pi ixy}{2^n}}e^{2\pi i\left(\frac{y^*}{2^n}+\delta\right)x}\,|y\rangle\,|\psi\rangle = \frac{1}{2^n}\sum_{x=0}^{2^n-1}\sum_{y=0}^{2^n-1}e^{\frac{2\pi i(y^*-y)x}{2^n}}e^{2\pi i\delta x}\,|y\rangle\,|\psi\rangle\,.$$

Then this probability is

$$\Pr(y = y^*) = \left| \frac{1}{2^n} \sum_{x=0}^{2^n-1} e^{2\pi i \delta x} \right|^2 .$$

By using formulas for the sum of a geometric series and introducing upper and lower bounds for the numerator and denominator, the minimal success probability can then be given as

$$\begin{aligned}\Pr(y = y^*) &= \left| \frac{1}{2^n} \frac{1 - e^{2\pi i \delta 2^n}}{1 - e^{2\pi i \delta}} \right|^2 \\ &= \left(\frac{1}{2^n} \frac{\left|1 - e^{2\pi i \delta 2^n}\right|}{\left|1 - e^{2\pi i \delta}\right|} \right)^2 .\end{aligned}$$

Now the following two inequalities

$$\left|1 - e^{2\pi i \delta 2^n}\right| \geq 4|\delta| 2^n \quad \text{and} \quad \left|1 - e^{2\pi i \delta}\right| \leq 2\pi|\delta|$$

can be applied [3, 6, 12], which finally gives a lower bound for the success probability as

$$\Pr(y = y^*) \geq \left(\frac{4|\delta|2^n}{2^n 2\pi|\delta|} \right)^2 = \left(\frac{2}{\pi} \right)^2 = \frac{4}{\pi^2} \approx 0.405.$$

$\square$

4.3 Finding the Discrete Logarithm

The principles presented in the previous section can be applied to solve the discrete logarithm problem. How this is done was first presented by Peter W. Shor [31], and properties, as well as generalizations of the algorithm, have been extensively studied in the previous years [6, 19, 24].

In this thesis, a variant of the algorithm described by Michele Mosca and Artur Ekert [19, 24] is presented. This algorithm is based on repeated phase estimation using the same input register.

To show how the process of phase estimation can be applied to this problem, some properties of the required oracle have to be examined first. The notation used here follows the notation presented in [24].

Let p be a prime and c be an element in the field $\mathbb{Z}_p$. The modular exponentiation oracle will perform the transformation

$$U_{c^X} \left|x\right\rangle \left|y\right\rangle = \left|x\right\rangle U_{c^x} \left|y\right\rangle = \left|x\right\rangle \left|c^x y \bmod p\right\rangle .$$

In this description U_{c^X}, is the controlled version performing the multiplication with c^x on the second register based on the value of x in the first register. Conceptually, this process can also be seen as the repeated application of the operator U_c, which performs the modular multiplication with c:

$$|x\rangle\, U_{c^x}\, |y\rangle = |x\rangle\, (U_c)^x\, |y\rangle\,. \tag{4.3}$$

Of course, for the process of phase estimation, the eigenvalues and eigenvectors of this operator are of interest. In particular, the properties of the oracle for the case of the multiplication with a generator are useful for the algorithm.

Lemma 4.3.1. *Let $g \in \mathbb{Z}_p$ be a generator with order r such that $g^r \equiv 1 \bmod p$. U_g has eigenvalues $e^{\frac{2\pi i k}{r}}$ with corresponding eigenvectors*

$$|\psi_k\rangle = \frac{1}{\sqrt{r}} \sum_{t=0}^{r-1} e^{\frac{-2\pi i k t}{r}}\, |g^t \bmod p\rangle\,,$$

for $k = 0, 1, \ldots, r-1$.

Proof. To show this property, the effect of the modular multiplication with g that is performed by U_g on the eigenvector $|\psi_k\rangle$ is examined.

$$\begin{aligned} U_g\, |\psi_k\rangle &= \frac{1}{\sqrt{r}} \sum_{t=0}^{r-1} e^{\frac{-2\pi i k t}{r}}\, |g^{t+1} \bmod p\rangle \\ &= \frac{1}{\sqrt{r}} \sum_{t=1}^{r} e^{\frac{-2\pi i k (t-1)}{r}}\, |g^t \bmod p\rangle \\ &= \frac{1}{\sqrt{r}} \sum_{t=1}^{r} e^{\frac{-2\pi i k t}{r}} e^{\frac{2\pi i k}{r}}\, |g^t \bmod p\rangle \\ &= e^{\frac{2\pi i k}{r}} \frac{1}{\sqrt{r}} \sum_{t=1}^{r} e^{\frac{-2\pi i k t}{r}}\, |g^t \bmod p\rangle \end{aligned}$$

Since $e^{\frac{-2\pi i k r}{r}} = e^{\frac{-2\pi i k 0}{r}}$ and $g^r \equiv g^0 \bmod p$, the sum can be rewritten to give

$$\begin{aligned} U_g\, |\psi_k\rangle &= e^{\frac{2\pi i k}{r}} \frac{1}{\sqrt{r}} \sum_{t=0}^{r-1} e^{\frac{-2\pi i k t}{r}}\, |g^t \bmod p\rangle \\ &= e^{\frac{2\pi i k}{r}}\, |\psi_k\rangle\,. \end{aligned}$$ □

As U_{g^x} can be thought of as the repeated application of U_g (see Equation (4.3)), Lemma 4.3.1 can be used to specify the eigenvalues of U_{g^x}.

Lemma 4.3.2. *For the generator g with $g^r \equiv 1 \bmod p$, U_{g^x} has eigenvalues $e^{\frac{2\pi i k x}{r}}$ with corresponding eigenvectors*

$$|\psi_k\rangle = \frac{1}{\sqrt{r}} \sum_{t=0}^{r-1} e^{\frac{-2\pi i k t}{r}}\, |g^t \bmod p\rangle\,,$$

for $k = 0, 1, \ldots, r-1$.

Proof. This lemma is proven by induction on x.

Base case: $x = 0$. The resulting state is

$$U_{g^0} |\psi_k\rangle = (U_g)^0 |\psi_k\rangle = |\psi_k\rangle = e^{\frac{2\pi i k 0}{r}} |\psi_k\rangle$$

as desired.

Induction hypothesis: Let x be an arbitrary integer ≥ 0 and assume that $U_{g^x} |\psi_k\rangle = e^{\frac{2\pi i k x}{r}} |\psi_k\rangle$ holds.

Induction step: Consider the statement for $x + 1$. Then

$$\begin{aligned} U_{g^{x+1}} |\psi_k\rangle &= (U_g)^{x+1} |\psi_k\rangle \\ &= U_g (U_g)^x |\psi_k\rangle \\ &= e^{\frac{2\pi i k x}{r}} U_g |\psi_k\rangle && \text{(using the induction hypothesis)} \\ &= e^{\frac{2\pi i k x}{r}} e^{\frac{2\pi i k}{r}} |\psi_k\rangle && \text{(using Lemma 4.3.1)} \\ &= e^{\frac{2\pi i k (x+1)}{r}} |\psi_k\rangle . \end{aligned}$$

□

For the discrete logarithm problem as described in Section 2.1, the properties of the oracle given the input b are of interest. By expressing b in terms of the generator g, these properties can be examined in detail.

Lemma 4.3.3. *For the generator g with $g^r \equiv 1 \bmod p$ and the integer $b \in \langle g \rangle$ with $g^m \equiv b \bmod p$, U_{b^x} has the eigenvalues $e^{\frac{2\pi i k m x}{r}}$ and the eigenvectors*

$$|\psi_k\rangle = \frac{1}{\sqrt{r}} \sum_{t=0}^{r-1} e^{\frac{-2\pi i k t}{r}} |g^t \bmod p\rangle ,$$

for $k = 0, 1, \ldots, r - 1$.

Proof. Since b can be expressed in terms of the generator as $b \equiv g^m \bmod p$, the operator U_{b^x} also can be expressed in terms of the generator:

$$U_{b^x} = U_{g^{mx}} = (U_{g^x})^m .$$

The lemma can then be proven similar to Lemma 4.3.2 by induction on m.

Base case: $m = 0$. Then the resulting state is

$$(U_{g^x})^0 |\psi_k\rangle = |\psi_k\rangle = e^{\frac{2\pi i k 0 x}{r}} |\psi_k\rangle$$

as desired.

Induction hypothesis: Let m be an arbitrary integer ≥ 0 and assume that $(U_{g^x})^m |\psi_k\rangle = e^{\frac{2\pi i k m x}{r}} |\psi_k\rangle$ holds.

Induction step: Consider the statement for $m + 1$. Then

$$\begin{aligned}(U_{g^x})^{m+1} |\psi_k\rangle &= U_{g^x} (U_{g^x})^m |\psi_k\rangle \\ &= U_{g^x}\, e^{\frac{2\pi ikmx}{r}} |\psi_k\rangle && \text{(using the induction hypothesis)} \\ &= e^{\frac{2\pi ikx}{r}} e^{\frac{2\pi ikmx}{r}} |\psi_k\rangle && \text{(using Lemma 4.3.2)} \\ &= e^{\frac{2\pi ik(m+1)x}{r}} |\psi_k\rangle . \end{aligned}$$ □

Lemma 4.3.3 shows that the eigenvalue of U_{b^x} contains the value m, which is the sought-after discrete logarithm of b with respect to the generator g. This suggests that phase estimation can be used to extract this value, which is exactly what is done in Shor's algorithm for the discrete logarithm problem as it is presented in [24].

Unfortunately, there still are a few obstacles to overcome before being able to compute the discrete logarithm. First, it still has to be examined how the oracle computing U_{g^X} as well as U_{b^X} can be constructed. A lot of research is being done on how to perform this process, referred to as modular exponentiation (some examples can be found in [2, 16, 30] and [34]), since it is both an integral part of Shor's algorithm for prime factorization as well as Shor's algorithm for the discrete logarithm problem [31]. The ways of implementing this gate that are used for this work are described in Chapter 5.

Furthermore, as the description of the phase estimation process in Section 4.2 notes, it is required to prepare the eigenstate $|\psi_k\rangle$ for a known value k to be able to infer the discrete logarithm m from the resulting phase. Following the description of the algorithm in the appendix of [19], this preparation is the first stage of the algorithm while the phase estimation giving the final result is the second stage.

4.3.1 First stage

To prepare the eigenstate in the second register and to extract the value k, the first stage will perform phase estimation on U_{g^X} using a superposition of all eigenstates. This is done by noting that the sum of the eigenstates can easily be prepared in the second register [24, 25].

Lemma 4.3.4. *Let g be a generator modulo the prime p with order r such that $g^r \equiv 1 \bmod p$. Then*

$$|1\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |\psi_k\rangle .$$

Proof. Using the definition of $|\psi_k\rangle$ from Lemma 4.3.1 the superposition of the eigenstates is given as

$$\frac{1}{r} \sum_{t=0}^{r-1} \sum_{k=0}^{r-1} e^{\frac{-2\pi ikt}{r}} |g^t \bmod p\rangle .$$

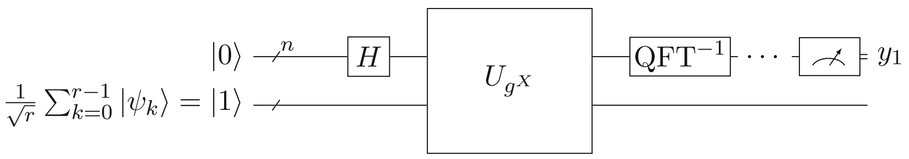

$|0\rangle$
n
H
U_{g^X}
QFT^{-1}
y_1
$\frac{1}{\sqrt{r}}\sum_{k=0}^{r-1}|\psi_k\rangle = |1\rangle$

Figure 4.3: First stage of the algorithm given in [24] and [19]. The number of qubits needed in the second register depends on the chosen implementation of modular exponentiation. The exact position of the measurement operation depends on the specific implementation of this algorithm (see Section 4.3.4 and 4.3.5) and might even be deferred to the end of the second stage, which is indicated by dots. The measurement outcome y_1 is used to infer the value $\tilde{k}$ which identifies the eigenstate remaining in the second register.

Since r is the order of g and $0 \le t \le r-1$ holds for the index in the first sum, all $|g^t \bmod p\rangle$ in the sum are pairwise distinct. The coefficient for a specific state $|g^{t'} \bmod p\rangle$ is given by

$$\frac{1}{r}\sum_{k=0}^{r-1}\left(e^{\frac{-2\pi i t'}{r}}\right)^k.$$

If $t' = 0$, this sum reduces to $\frac{1}{r}\sum_{k=0}^{r-1} 1 = 1$. For all other $1 \le t' < r$, the formula for the sum of a geometric series can be applied to give

$$\begin{aligned}\frac{1}{r}\sum_{k=0}^{r-1}\left(e^{\frac{-2\pi i t'}{r}}\right)^k &= \frac{1}{r}\left(\frac{1-e^{\frac{-2\pi i t' r}{r}}}{1-e^{\frac{-2\pi i t'}{r}}}\right)\\ &= \frac{1}{r}\left(\frac{1-1}{1-e^{\frac{-2\pi i t'}{r}}}\right)\\ &= 0.\end{aligned}$$

This shows that the only remaining basis state in the description of this superposition is $|g^0 \bmod p\rangle = |1\rangle$, which proves this lemma. □

By performing phase estimation using the superposition of eigenstates as input to the second register as shown in Figure 4.3, the resulting measurement in the first register will, with equal probability, be an estimate of the phase of some eigenstate $|\psi_k\rangle$, $k \in \{0, 1, \dots, r-1\}$ chosen uniformly at random [25].

Using this fact it is assumed for the rest of this explanation that the algorithm is performed using some specific eigenstate $|\psi_{\tilde{k}}\rangle$, since the argument for its correctness holds for each eigenstate and when measuring only the results for one specific $\tilde{k}$ are returned.

Since for any specific $\tilde{k}$ the eigenvalue is $e^{\frac{2\pi i \tilde{k} x}{r}}$, the measured value in the first register will be $y_1 = \frac{\tilde{k}}{r}2^n$ [19]. By dividing with 2^n, $\tilde{k}$ can be obtained using a multiplication with r if the order is known. Should this not be the case, the resulting value can be approximated as a fraction similar to Shor's algorithm for prime factorization [31] to obtain both $\tilde{k}$ and r.

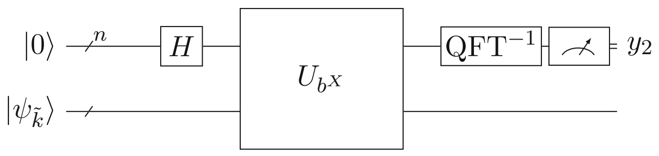


Figure 4.4: Second stage of the algorithm given in [24] and [19]. The measurement outcome y_2 is used to approximate the value $\tilde{k}m \mod r$, which can be used to infer the discrete logarithm m.

The content of the second register is then reused to ensure that the phase estimation performed in the second stage is performed using the same state $|\psi_{\tilde{k}}\rangle$. Of course, as phase estimation might not be exact, the value obtained in this stage might not be exactly $\tilde{k}$. This approximation error is reflected in the success probability given in Section 4.3.3.

4.3.2 Second stage

As the first stage prepares the eigenstate $|\psi_{\tilde{k}}\rangle$ in the second register, phase estimation using the oracle U_{b^X} can now be performed to extract the phase and in turn extract the discrete logarithm m.

As shown in Lemma 4.3.3, the eigenvalue of the operator U_{b^x}, for the eigenstate $|\psi_{\tilde{k}}\rangle$ is $\exp\left(\frac{2\pi i\tilde{k}mx}{r}\right)$. Therefore the estimated phase will be $\frac{\tilde{k}m}{r}$. Since in some cases $\tilde{k}m \geq r$ special care has to be taken since the coefficient $\exp\left(\frac{2\pi i\tilde{k}mx}{r}\right)$ is indistinguishable from $\exp\left(\frac{2\pi i(\tilde{k}m-qr)x}{r}\right)$, for any integer q.

The measured value y_2 after the second stage can therefore be described as

$$y_2 = \frac{(\tilde{k}m \bmod r)}{r}2^n.$$

By again dividing by 2^n and multiplying with r, m can be calculated by modular multiplication with the inverse of $\tilde{k}$, where $\tilde{k}$ is known from the measurement of the first stage.

4.3.3 Success probability and complexity

As the described algorithm for the discrete logarithm problem can be seen as two consecutive runs of phase estimation, the state before the final measurement can be described as

$$\frac{1}{2^n}\frac{1}{2^n}\sum_{x_1,y_1=0}^{2^n-1}\sum_{x_2,y_2=0}^{2^n-1} e^{\frac{2\pi i\tilde{k}x_1}{r}} e^{\frac{-2\pi i x_1 y_1}{2^n}} e^{\frac{2\pi i\tilde{k}mx_2}{r}} e^{\frac{-2\pi i x_2 y_2}{2^n}} |y_1\rangle |y_2\rangle |\psi_{\tilde{k}}\rangle .$$

In this description, x_1 and y_1 get introduced by the Hadamard gates and the inverse QFT in the first stage, while x_2 and y_2 are introduced by the respective gates in the second stage. This description of the state assumes that the measurements for both stages are deferred to the end of the circuit.

The best approximations of the phases of the respective modular exponentiation gates in each stage will be referred to as y_1^* and y_2^*. To be able to perform the rest of the algorithm, the following relationship has to hold for the measurement results for the first and second stages:

$$(y_1^* + \delta_1, y_2^* + \delta_2) = \left(\frac{\tilde{k}}{r}2^n, \frac{(\tilde{k}m \bmod r)}{r}2^n\right).$$

Here δ_1 and δ_2 refer to the approximation error of the first and second stage in case the phase can not be estimated exactly.

To calculate the probability that y_1^* and y_2^* will be measured, the success probability for phase estimation as shown in Theorem 4.2.1 is used for each stage separately [23]:

$$\Pr\left[(y_1, y_2) = (y_1^*, y_2^*)\right] \geq \left(\frac{4}{\pi^2}\right)^2.$$

As noted above, it can be assumed that at the beginning of the first stage, the target register is in state $|\psi_{\tilde{k}}\rangle$, because the target register actually starts in the superposition of all eigenstates. Therefore the value $\tilde{k}$ will be chosen uniformly at random and unfortunately not all values of $\tilde{k}$ eventually lead to a usable measurement result.

In particular, only those $\tilde{k}$ that have an inverse element $\tilde{k}^{-1} \in \mathbb{Z}_r$, or in other words, where $\tilde{k}m \bmod r$ has a solution for m will lead to a result. This is the case for all natural numbers $\tilde{k} < r$ that are coprime to r, the number of which can be calculated using Euler's totient function $\phi(r)$.

This contrasts the probability given in [23], where it is assumed that r is prime and therefore only $\tilde{k} = 0$ is excluded. This leads to a total success probability greater than or equal to

$$\frac{\phi(r)}{r} \cdot \left(\frac{4}{\pi^2}\right)^2.$$

Since the actual value of $\phi(r)$ depends on the prime factorization of r, an asymptotic lower bound for this value is used similar to [31] (see also Theorem 328 in [17]):

$$\frac{\phi(r)}{r} \in \Omega\left(\frac{1}{\log\log r}\right).$$

Since the factor $(4/\pi^2)^2$ is constant, the lower bound for the success probability for this algorithm is given by

$$\Omega\left(\frac{1}{\log\log r}\right).$$

The number of repetitions required to obtain a number $\tilde{k}$ that is coprime to r can be given as the reciprocal value of the success probability. As for functions f and g, $f(x) \in \Omega(g(x))$ implies that $1/f(x) \in O(1/g(x))$ and applying the fact that $r < p$, this shows that after $O(\log \log p)$ repetitions the algorithm is successful with high probability.

Using this expression for the number of repetitions, the upper bound for the complexity of the algorithm can be given. First, examining the circuits for both stages shows that the number of qubits in each input register is n. Herein, n is chosen to be the number of bits required to represent the module p. Additionally the circuit needs a certain number of qubits for the actual computation of the modular exponentiation as indicated by the bottom register. This number depends on the actual implementation of the modular exponentiation circuit that is chosen in Chapter 5 and shall be referred to as $S(n)$ for now.

Therefore the circuit uses $2n + S(n)$ qubits in total. This number can be decreased using optimizations explained in the next subsections.

The remarkable property of Shor's algorithm for the discrete logarithm problem is that its runtime is polynomial. As the first stage and the second stage are almost identical in the operations that are applied, the runtime for one stage can be examined to infer the runtime of the whole algorithm.

It is assumed for this argument that the number $T(n)$ of elementary gate operations applied in the modular exponentiation circuit is polynomial. Note that for this argument, the modular exponentiation is treated as a single gate performing $|x\rangle |y\rangle \rightarrow |x\rangle |y \cdot a^x \mod p\rangle$, and not a series of gates each performing multiplications.

One stage of the algorithm performs $O(n)$ Hadamard operations on the top register, followed by $O(T(n))$ elementary operations to perform modular exponentiation as well as $O(n^2)$ transformations to compute the inverse QFT. In summary, as each of these runtime bounds is polynomial, one stage can be performed in polynomial time.

As will be shown in the next chapter, the time required to perform modular exponentiation $O(T(n))$ dominates the time required to perform the other steps of each stage of the algorithm. Therefore the number of elementary gate operations can be given as $O(T(n))$. The second stage of the algorithm then amounts to a repetition of all steps in terms of the runtime, which is why the asymptotic runtime for the quantum computing part of the algorithm is given by $O(T(n))$.

The only non-constant term in the success probability is the number of repetitions required due to the fact that the inferred value $\tilde{k}$ might not have an inverse modulo r. It is asserted that after $O(\log \log p)$ repetitions a $\tilde{k}$ coprime to r is found. Since all other terms of the success probability are constant, the time required until the algorithm yields the result with near certainty is given by

$$O((\log \log p) T(n)).$$

In addition to the quantum part of the algorithm, some classical postprocessing is performed. To make sure the classical part does not outweigh the quantum part in its complexity, arguments that this part can be performed in polynomial time are given here. For this algorithm four steps have to be performed:

- If the period r is not known it has to be inferred from the measurement result using the continued fraction algorithm which can be carried out in polynomial time [31].
- Perform divisions with 2^n and multiplications with r to obtain $\tilde{k}$ and $\tilde{k}m \bmod r$. These arithmetic operations can be performed in polynomial time.
- Check whether the inverse of $\tilde{k}$ exists and calculate it in polynomial time using the Euclidean algorithm. If it does not exist, restart the algorithm.
- Multiply with $\tilde{k}^{-1} \bmod r$ and check if this result is correct by performing modular exponentiation classically in polynomial time.

In summary, since the quantum circuit can be constructed and executed in polynomial time, and the classical postprocessing can be performed efficiently, the whole algorithm as presented here runs in polynomial time. It gives the correct result with near certainty which, using the argument for the classical postprocessing from Section 2.4, shows the membership of the discrete logarithm problem in **BQP**.

4.3.4 Reusing the input register

A major challenge for quantum computer development is handling problem instances with an increased number of qubits. It is beneficial for todays hardware if the number of used qubits in any circuit is minimized. One approach for achieving this goal is minimizing the resources required for modular exponentiation as presented in Chapter 5. But there are also variants of the overall algorithm that aim to reduce the number of used qubits.

One such variant [19, 24] uses the fact that the two applications of modular exponentiation are performed in succession, which allows the top register to be reused.

The circuit performs both stages as described in Section 4.3.1 and Section 4.3.2 using the same top register, thus saving n qubits and reducing the number of qubits needed from $2n + S(n)$ to $n + S(n)$.

A requirement for this variant is that there is a way of resetting the register content after a measurement was performed, which can be done in Qiskit using the reset gate as presented in Section 2.3.2.

4.3.5 Reducing the size of the control register

This subsection will introduce two optimizations of the phase estimation algorithm which can be applied to the algorithm for the DLP above. The first one performs the QFT at the end of the circuit *semi-classically* [15, 23, 24], which simplifies the implementation as it eliminates the need for multi-qubit quantum gates. Building upon this circuit, another optimization becomes evident. By exploiting the sequential nature of the QFT, it is possible to perform phase estimation using only a single control qubit, thus eliminating $n-1$ qubits from the circuit [2, 23, 24].

Examining the circuit for the inverse QFT, i.e., the inverse of the circuit in Figure 4.1, shows that the most significant qubit $|\phi_n(a)\rangle$, which will be transformed to $|a_1\rangle$, is only used to control other conditional rotations after the Hadamard operation is applied. Similarly, all other qubits will only be used to control other conditional operations after the Hadamard gate has been applied.

By employing the deferred measurement principle (see Section 4.4 in [25]), the measurement operations on each qubit can be moved forward to just after the Hadamard gate has been applied to obtain an equivalent circuit. Since the subsequent operations are then controlled classically, this approach is referred to as the semi-classical QFT.

Figure 4.5 shows the circuit performing the semi-classical inverse QFT. It takes the input $|\phi(a)\rangle$, performs the inverse QFT to obtain $|a\rangle$, while at the same time measuring to obtain a binary measurement result $y_n y_{n-1} \dots y_2 y_1$.

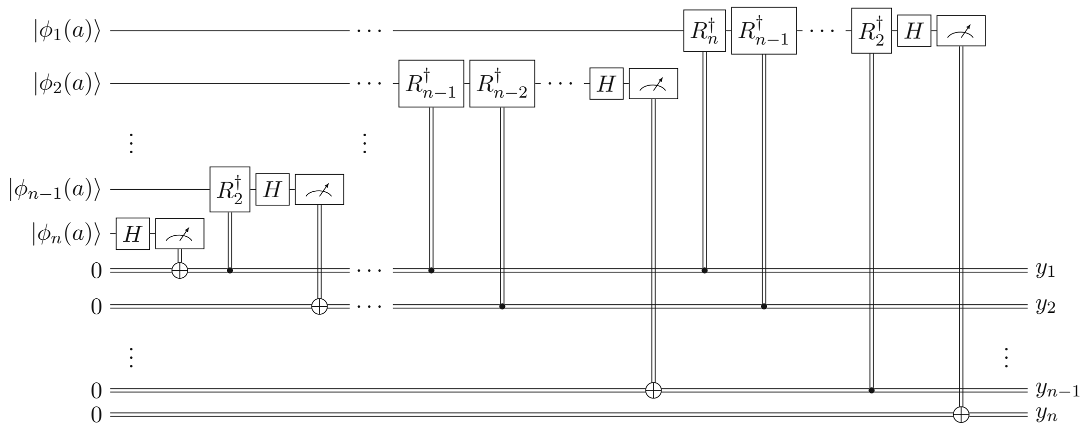


Figure 4.5: The semi-classical version of the inverse QFT. Note that in addition to the measurements being moved forward, the swap operations at the beginning of the inverse QFT are performed implicitly by changing which qubits the rotations are applied to as well as the classical bit y_i, they are measured into. The double wires indicate that not the qubit, but the classical value that was measured is used to control an operation.

A noteworthy property of this circuit is that each qubit is idle after it has been measured, since all operations that would have been controlled by it are now controlled by its corresponding classical bit. Furthermore, as the circuit for phase estimation in Figure 4.2 shows, each control qubit is only used to control one of the oracle gates while remaining idle the rest of the time.

In summary, the lifecycle of the control qubit $|q_i\rangle$ in the phase estimation algorithm with semi-classical inverse QFT is as follows.

1. Apply the Hadamard gate.
2. Control the $U^{2^{i-1}}$ oracle.
3. Apply conditional rotations which all are controlled by classical values.
4. Apply Hadamard again.
5. Measure.

Performing these operations serially leads to the second optimization: phase estimation with one control qubit [2, 23, 24]. Starting with $i = n$, the steps listed above are performed in sequence. This is possible since the initial value of each control qubit is known to be $H\,|0\rangle$, therefore it can repeatedly be reinitialized after those steps have been performed to continue with the next oracle. The circuit for this process in shown in Figure 4.6.

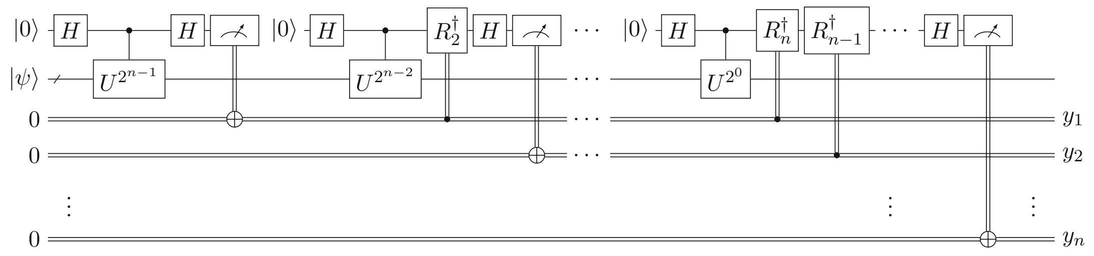


Figure 4.6: Phase estimation using only one control qubit. The sequential nature of the inverse QFT is employed to be able to reuse one control qubit for each step in the execution of the oracle. Before being measured, the rotations required for the inverse QFT are performed based on the previously measured values. The non-unitary operation denoted as $|0\rangle$ represents a reinitialization of the qubit to $|0\rangle$ as described in Section 2.3.2.

The fact that, instead of n control qubits, only one qubit is needed greatly reduces the resource demand for the algorithm. The number of used qubits for this optimization is given as $1 + S(n)$, which is almost entirely dependent on the space complexity of modular exponentiation. This shows that, in addition to optimizing the process of phase estimation, it is vital to perform the oracle transformations presented in the next chapter efficiently.

Although it is possible to implement and simulate the circuit presented in Figure 4.6, the rotations conditioned on a classical bit can at the moment not be performed on the available quantum computers. This feature, however, is planned for the near future [36].

CHAPTER 5

Modular Exponentiation

Both Shor's algorithm for prime factorization as well as its counterpart for the discrete logarithm problem utilize a gate performing modular exponentiation as a central component of the circuit. As this operation is the most expensive operation of both algorithms both in terms of the number of applied unitary transformations as well as in the number of ancilla qubits required, various approaches have already been examined as to how this operation can be performed most efficiently [2, 16, 30, 34].

As shown in Chapter 4, Shor's algorithm requires a gate U_{a^X} that, ignoring ancilla qubits, takes some input $|x\rangle |y\rangle$ and computes the result $|x\rangle |y \cdot a^x \mod p\rangle$. In this chapter, we show in Section 5.1 how this calculation is performed using repeated application of modular multiplication. This serves as the motivation for introducing different approaches to performing modular multiplication in the later sections.

As a modular multiplier can be built using circuits that perform modular addition, Section 5.2 presents two approaches solving this task. The first one is an adder that uses the properties of the Quantum Fourier transform (Section 5.2.1), while the second algorithm more closely resembles addition on a classical computer (Section 5.2.2).

Section 5.3, continues to show how these algorithms for modular addition can be extended to perform modular multiplication. Of course, any algorithm for non-modular multiplication can be extended to an algorithm for modular multiplication by performing a division to calculate the residue. As this division is a costly operation, a different approach is presented in Section 5.4: Montgomery modular multiplication [22]. Here a representation of the result is used that allows this division to be performed more efficiently.

For the argument in Section 4.3.3 that the discrete logarithm problem can be solved in polynomial time, it was assumed that modular exponentiation can be computed in polynomial time. This claim now has to be examined. Therefore, the complexity of these algorithms both in the number of performed gate operations $T(n)$ and in the number of required qubits $S(n)$ is given.

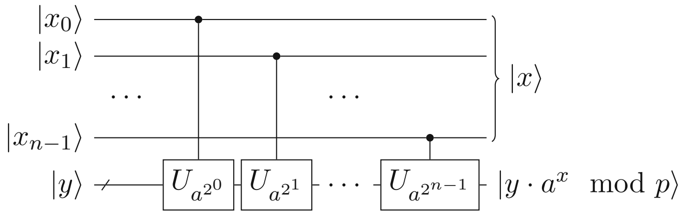


Figure 5.1: Modular exponentiation using a built-in constant a as a series of controlled multiplications.

5.1 Modular Exponentiation

As described in Shor's original paper [31], modular exponentiation can be performed by simply performing a series of modular multiplications with a constant that is built into the circuit. The multiplications are controlled by the individual qubits of the input.

It is assumed that there is a gate U_a which performs the multiplication

$$U_a |y\rangle = |ay \mod p\rangle\,, \tag{5.1}$$

where $|y\rangle$ is a register large enough to hold any value $0 \leq y < p$ for the prime p and a is a constant known at the time when the circuit is constructed. Since $c \in \{0,1\}$, the controlled version of this gate can further be represented as

$$U_{a^c} |c\rangle |y\rangle = |c\rangle |a^c y \mod p\rangle\,. \tag{5.2}$$

How such a gate and its corresponding controlled version can be obtained for any constant a is explained in the later parts of this chapter.

Notice that although for Shor's algorithm for prime factorization it cannot be assumed that p is prime, this is always the case for Shor's algorithm for the discrete logarithm problem. Therefore for the remainder of this chapter p is always an odd prime, as this simplifies the presented algorithms.

The circuit will serially apply gates U_a for different constants a which are built into the circuit. These gates are controlled by the input x as shown in Figure 5.1. In total, the circuit will perform the following transformation

$$\begin{aligned}
|x\rangle |y\rangle \to |x\rangle U_{a^{2^0 x_0}} |y\rangle &= |x\rangle |y \cdot a^{2^0 x_0} \mod p\rangle \\
&\to |x\rangle |y \cdot a^{2^0 x_0} \cdot a^{2^1 x_1} \mod p\rangle \\
&\dots \\
&\to |x\rangle |y \cdot a^{2^0 x_0} \cdot a^{2^1 x_1} \cdots a^{2^{n-1} x_{n-1}} \mod p\rangle \\
&= |x\rangle |y \cdot a^{2^0 x_0 + 2^1 x_1 + \dots + 2^{n-1} x_{n-1}} \mod p\rangle \\
&= |x\rangle |y \cdot a^x \mod p\rangle\,.
\end{aligned}$$

Here the fact that x (represented as a binary integer) can be expressed as

$$x_0 \cdot 2^0 + x_1 \cdot 2^1 + \cdots + x_{n-1} \cdot 2^{n-1}$$

is used.

This shows that, using a modular multiplier that performs a multiplication with some built-in constant, the modular exponentiation can be constructed with gate complexity $O(n \cdot T_{\text{mul}}(n))$, where $T_{\text{mul}}(n)$ specifies the number of elementary operations required to perform multiplication. The modular exponentiation circuit uses n control qubits as well as $S_{\text{mul}}(n)$ qubits holding the result.

The remainder of this chapter investigates how the multiplication gate that is required for this algorithm as described in Equation (5.1) and Equation (5.2) is constructed.

5.2 Modular Addition

Modular multiplication on a quantum computer is usually performed by a series of modular addition gates, which in turn are built using conditional additions. There are various proposals for implementing the addition $a + b$, both where b is a variable stored in a quantum register as well as where b is a constant built into the circuit.

For both algorithms presented by Shor's [31], b is known at the time of construction. Therefore, the addition gates presented in this section can use this fact to save instructions and qubits. Unfortunately, the straightforward approach of implementing the addition of some integer a with a constant b similar to a classical computer, while only using one qubit to save the intermediate carries is not possible as this gate would not be reversible. As an example, the following mappings $|a_k, c_{in}\rangle \rightarrow |(a+b)_k, c_{out}\rangle$ are required for adding the bit $b_k = 1$ to the input qubit a_k, while respecting the previous carry c_{in} and saving the new carry c_{out}:

$$\begin{aligned}
|0,0\rangle &\rightarrow |1,0\rangle \\
|0,1\rangle &\rightarrow |0,1\rangle \\
|1,0\rangle &\rightarrow |0,1\rangle \\
|1,1\rangle &\rightarrow |1,1\rangle .
\end{aligned}$$

As both $|0,1\rangle$ and $|1,0\rangle$ would be transformed into the same state $|0,1\rangle$, this gate cannot be reversible.

But there are other ways of performing this addition, some of which are examined in the following subsections. Using these implementations, ways of performing the modular addition $(a + b) \bmod p$ and finally the modular multiplication $ax \bmod p$ are presented in the later sections.

5.2.1 Addition using the Quantum Fourier Transfrom

Based on an implementation of the addition proposed by Draper [11], Beauregard proposed a full circuit for Shor's algorithm for prime factorization [2]. Rui Maia and Tiago Leão provided an implementation of this circuit [21], which now serves as the reference implementation of the algorithm for prime factorization in Qiskit.

As opposed to the adder presented in Section 5.2.2, which uses an approach very similar to addition on a classical computer, Draper provides a circuit that makes use of operations specific to quantum computers.

First, the input a is transformed into Fourier space using the Quantum Fourier transform (QFT). Following the notation presented in Section 4.1, the resulting state is denoted as $|\phi(a)\rangle$. Afterwards rotations on the qubits of $|\phi(a)\rangle$ are performed, which give the state $|\phi(a+b)\rangle$. Since b is known at the time of construction of the circuit, the angles for these rotations can be precomputed. Lastly the inverse QFT is applied to $|\phi(a+b)\rangle$ resulting in $|a+b\rangle$ in the register.

A significant advantage of this approach is the elimination of ancilla qubits that save intermediate carry values as it would be required when performing the addition in a more classical way. To handle overflows, only one additional qubit is added to the register containing the input a.

To illustrate how the addition is performed, the state of the system starting from the $m = n+1$ qubit register containing the n qubit integer a and the ancillary qubit initialized to $|0\rangle$ is examined. Applying the QFT to this registers gives

$$QFT\,|a\rangle = |\phi(a)\rangle = \frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} \exp\left(2\pi i \frac{ay}{2^m}\right) |y\rangle\,.$$

As noted in Section 4.1, this state is separable [11] and can be expressed as $\bigotimes_{k=1}^{m} |\phi_k(a)\rangle$, where

$$|\phi_k(a)\rangle = \frac{1}{\sqrt{2}} \left(|0\rangle + \exp\left(2\pi i \frac{a}{2^k}\right)|1\rangle\right).$$

To perform the actual addition, each qubit is now rotated around the Z-axis on the Bloch sphere using Qiskit's phase gate[1]. This gate maps an arbitrary state $\alpha\,|0\rangle + \beta\,|1\rangle$ to $\alpha\,|0\rangle + exp(i\lambda)\beta\,|1\rangle$, where λ can be supplied at the time of construction. Depending on the constant b and the position k of the qubit in the register, λ is set to $\frac{2\pi b}{2^k}$. Therefore each qubit $|\phi_k(a)\rangle$ is rotated to become

$$\begin{aligned}\frac{1}{\sqrt{2}}\left(|0\rangle + \exp\left(2\pi i\frac{b}{2^k}\right)\cdot\exp\left(2\pi i\frac{a}{2^k}\right)|1\rangle\right) &= \frac{1}{\sqrt{2}}\left(|0\rangle + \exp\left(2\pi i\frac{(a+b)}{2^k}\right)|1\rangle\right)\\ &= |\phi_k(a+b)\rangle\,.\end{aligned}$$

[1]https://qiskit.org/documentation/stubs/qiskit.circuit.library.PhaseGate.html

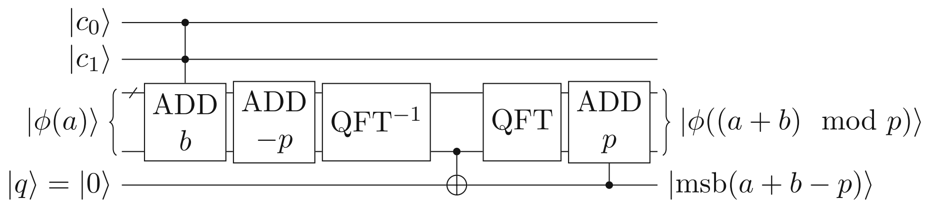


Figure 5.2: Modular addition $|\phi((a+b) \bmod p)\rangle$ for a built-in constant b, excluding the uncomputation of the ancilla qubit $|q\rangle$. The qubits $|c_0\rangle$ and $|c_1\rangle$ are used as control qubits. The circuit is taken from [2].

Consequently, the whole register then contains

$$\frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} \exp\left(2\pi i \frac{(a+b)y}{2^m}\right) |y\rangle = |\phi(a+b)\rangle \,.$$

By multiplying the input for the phase gate λ by -1, the addition can also easily be transformed into a subtraction.

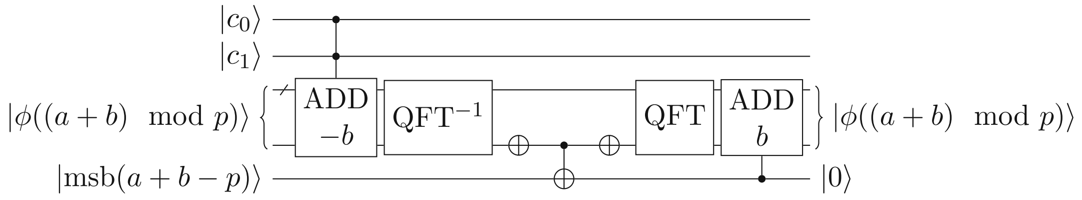
The inverse QFT can then be used to extract the result $|a+b\rangle$.

Modular Addition using the Quantum Fourier Transform

For the extension to modular multiplication and modular exponentiation, a modular addition providing two control bits is required. Beauregard [2] proposes such a circuit that saves the information on whether the result of the addition needs to be adapted with respect to the module p in an additional qubit. The circuit for modular addition assumes that the register containing a is already transformed to Fourier space. The register is denoted as $|\phi(a)\rangle$. This is valid since the additions are performed in succession and the intermediate results do not have to be transformed back from Fourier space.

Before presenting the circuit for modular addition, a doubly controlled version of the adder has to be constructed. This is done by controlling the rotations with two control bits using rotations by half the desired angle. These rotations cancel out should one control bit not be set. How such a gate can be constructed is shown in [28].

Assuming both control qubits are set, the circuit in Figure 5.2 first adds b and subtracts p, by changing the factor of the rotations as explained above. Should the result be greater than zero, then the register already contains the desired value $|\phi((a+b) \bmod p)\rangle$. If this is not the case, the subtraction of p has to be reversed. To perform this case distinction, the inverse QFT is applied to be able to extract the most significant bit of the register content $a+b-p$. As the register now contains the result in two's complement, the most significant bit encodes the sign. In the case $a+b-p<0$, the most significant bit will be one. This bit is saved in an ancilla qubit which is used to control a conditional addition of p after the register content has been transformed to Fourier space again.



$|c_0\rangle$
$|c_1\rangle$
$|\phi((a+b) \mod p)\rangle$
ADD $-b$
QFT^{-1}
QFT
ADD b
$|\phi((a+b) \mod p)\rangle$
$|\text{msb}(a+b-p)\rangle$
$|0\rangle$

Figure 5.3: Uncomputation of the ancilla qubit after modular addition as shown in Figure 5.2 [2].

At this point, the input register itself will always contain $|\phi((a+b) \bmod p)\rangle$. Unfortunately before the computation can be completed, the ancilla qubit still has to be restored to zero. This is achieved by using the fact that, if $a + b - p < 0$, then $a + b \bmod p > b$. This shows that the ancilla qubit was flipped in the previous operation if and only if $a + b \bmod p > b$.

More formally, the relationship

$$\text{msb}(a+b-p) = 1 \oplus \text{msb}(((a+b) \bmod p) - b)$$

is used, where $\text{msb}(x) \in \{0,1\}$ refers to the most significant bit of x and $\oplus$ is the exclusive *OR* operator. Consequently, another comparison with b can be used with the only difference that the complement of the result is used to flip the ancilla qubit. Of course, to restore the result in the register after this comparison, it has to be transformed to Fourier space and b has to be added again.

5.2.2 Bitwise Carry Addition

Originally used for the algorithm for prime factorization, Häner, Roetteler and Svore propose in [16] an implementation for a modular adder for quantum computers, which can be extended similar to Beauregard's version to perform modular multiplication and in turn modular exponentiation.

The adder itself utilizes a gate that calculates the final carry after addition of two integers. As opposed to Beauregard's modular adder this gate can be used to implement a simpler comparison, which in turn allows for performing the modular addition with one less ancilla qubit. Another difference is the fact that this implementation performs the addition in a more "classical" way: For a subset of the bits, the carry is computed and added to the next subset of bits, as opposed to the approach of calculating in Fourier space presented in Section 5.2.1.

Computing the carry

The innermost operation calculates the carry of an addition of an n-bit constant b to an n-bit input a, where b is built into the circuit. If the individual bits of the integers a and b and the result $r = a + b$ are represented as a_i, b_i and r_i $(0 \leq i < n)$, then this operation calculates r_n, the carry after the last bitwise addition of a_{n-1} and b_{n-1}.

Conditioning operations on toggling

To save information on the intermediate carry values for each combination of qubits, the proposed implementation in [16] uses *borrowed* qubits. These qubits are assumed to be in an unknown state before the execution of the algorithm and have to be returned to exactly this state after the computation. Therefore each operation that might be applied to those qubits has to be reversed before the computation of the overall carry finishes.

This is done by conditioning certain parts of the circuit not on the actual value of the qubit, but on the fact that it has been toggled during the execution of the operation.

For each bit of the input, the following operations are applied depending on the given constant b. If $b_i = 0$, and there is a carry c_{i-1} from bit $i-1$ to bit i, there will be a carry c_i after the addition if and only if the input $a_i = 1$. If there is no carry c_{i-1}, then there never will be a carry after adding a_i and $b_i = 0$. On the other hand, if $b_i = 1$, then there is a carry if either $c_{i-1} = 1$ or $a_i = 1$, or both. This gives the following relationships between input and output for $|c_{i-1}, a_i, c_i\rangle$, assuming the carry bits are ordinary ancilla qubits and not borrowed qubits:

If $b_i = 0$:

$$\begin{aligned}|0,0,0\rangle &\rightarrow |0,0,0\rangle\\ |1,0,0\rangle &\rightarrow |1,0,0\rangle\\ |0,1,0\rangle &\rightarrow |0,1,0\rangle\\ |1,1,0\rangle &\rightarrow |1,1,1\rangle\,.\end{aligned}$$

If $b_i = 1$:

$$\begin{aligned}|0,0,0\rangle &\rightarrow |0,0,0\rangle\\ |1,0,0\rangle &\rightarrow |1,0,1\rangle\\ |0,1,0\rangle &\rightarrow |0,1,1\rangle\\ |1,1,0\rangle &\rightarrow |1,1,1\rangle\,.\end{aligned}$$

With ancilla qubits c_i, where the state is fully known, this operation is just one Toffoli gate in the case $b_i = 0$. In the case $b_i = 1$ this operation flips the resulting carry bit in all cases except for the case $a_i = 0, c_{i-1} = 0$. To achieve this, c_i is flipped conditioned on a_i first and again conditioned on c_{i-1} and the complement of a_i. Figure 5.4 shows this circuit without the uncomputation of a_i.

To facilitate the usage of borrowed qubits, the operations in Figure 5.4 have to be conditioned not on the value of the qubits c_i but on the fact that they have been toggled. Also afterwards, they have to be returned to their original state. This is done by executing this procedure once before the carry-bits have been toggled and then repeating all operations that are controlled by carry-bits after they have been toggled. This way, if a qubit is not toggled, the operation is repeated with the same control input and therefore

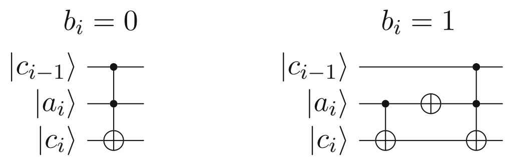
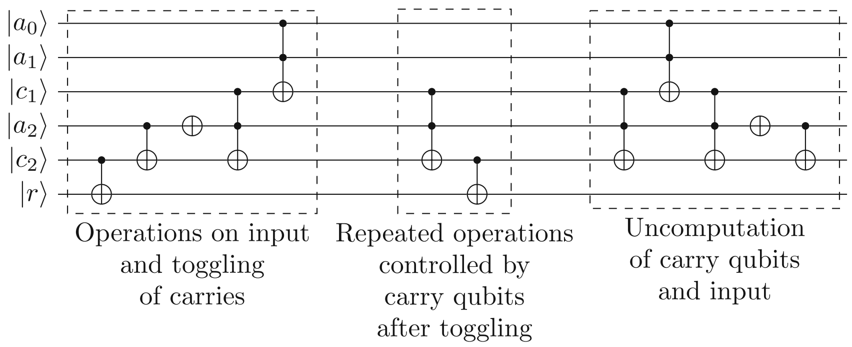


Figure 5.4: The carry computation without the uncomputation of a_i and the reset of borrowed qubits for both cases $b_i = 0$ and $b_i = 1$.



$|a_0\rangle$
$|a_1\rangle$
$|c_1\rangle$
$|a_2\rangle$
$|c_2\rangle$
$|r\rangle$
Operations on input and toggling of carries
Repeated operations controlled by carry qubits after toggling
Uncomputation of carry qubits and input

Figure 5.5: The carry computation for the addition with a constant $b = 5$ taken from [16]. The circuit has been optimized using the fact that the first carry c_0 from a_0 to a_1 is only conditioned on the qubit a_0 and can therefore be eliminated and a_0 can be used instead.

reversed. In the case that the corresponding qubit has been toggled it is only executed once. Finally, the circuit is run in reverse while ignoring all operations on the result r_n. This resets both the carry qubits c_i and the input a_i to their original state. Figure 5.5 shows the full carry gate for the example constant $b = 5$.

To give a more thorough argument for the correctness with respect to borrowed qubits, one possible variant of the execution excluding the uncomputiation is examined more closely. Assume the constant $b_i = 0$ and during the execution the borrowed qubit containing the carry from the previous bitwise addition is flipped, indicating that there is a carry from the previous addition. The circuit for this example is shown in Figure 5.6.

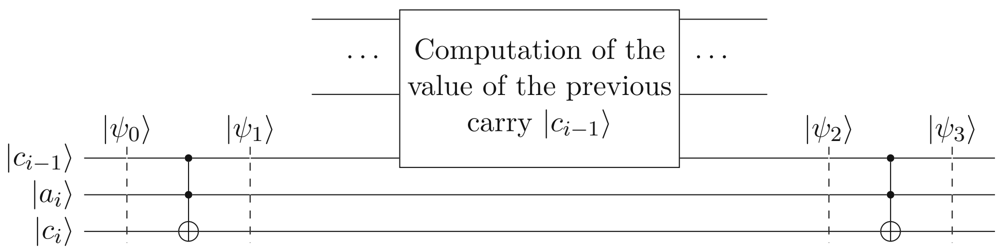


Figure 5.6: Computing the carry bit for one input bit in the case $b_i = 0$. The intermediate states $|\psi_i\rangle$ are examined in Table 5.1.

Coefficients	$\lvert\psi_0\rangle$	$\lvert\psi_1\rangle$	$\lvert\psi_2\rangle$	$\lvert\psi_3\rangle$
$\alpha(1-a_i)\gamma$	$\lvert 000\rangle$		$\lvert 100\rangle$	
$\beta(1-a_i)\gamma$	$\lvert 100\rangle$		$\lvert 000\rangle$	
$\alpha a_i\gamma$	$\lvert 010\rangle$		$\lvert 110\rangle$	$\lvert 111\rangle$
$\beta a_i\gamma$	$\lvert 110\rangle$	$\lvert 111\rangle$	$\lvert 011\rangle$	
$\alpha(1-a_i)\delta$	$\lvert 001\rangle$		$\lvert 101\rangle$	
$\beta(1-a_i)\delta$	$\lvert 101\rangle$		$\lvert 001\rangle$	
$\alpha a_i\delta$	$\lvert 011\rangle$		$\lvert 111\rangle$	$\lvert 110\rangle$
$\beta a_i\delta$	$\lvert 111\rangle$	$\lvert 110\rangle$	$\lvert 010\rangle$	

Table 5.1: State during the computation of the carry for one bit for the intermediate steps $\lvert\psi_i\rangle$ in Figure 5.6. This represents the case where $b_i = 0$ and the previous computations resulted in a carry, thus c_{i-1} is flipped. For simplicity, each column shows only the cases where the coefficient of a specific basis state changes. For example, column $\lvert\psi_1\rangle$ gives the state $\lvert\psi_1\rangle = \alpha(1-a_i)\gamma\lvert 000\rangle + \beta(1-a_i)\gamma\lvert 100\rangle + \alpha a_i\gamma\lvert 010\rangle + \beta a_i\gamma\lvert 111\rangle + \cdots + \beta a_i\delta\lvert 110\rangle$.

The system starts in an initial state, where the content of the borrowed qubits c_{i-1} and c_i is unknown. Such a state can be described as

$$\lvert\psi_0\rangle = \lvert c_{i-1}\rangle \otimes \lvert a_i\rangle \otimes \lvert c_i\rangle = (\alpha\lvert 0\rangle + \beta\lvert 1\rangle) \otimes ((1-a_i)\lvert 0\rangle + a_i\lvert 1\rangle) \otimes (\gamma\lvert 0\rangle + \delta\lvert 1\rangle).$$

As for a basis state $\lvert a_i\rangle$ it holds that $a_i \in \{0,1\}$, the corresponding coefficients are simplified to a_i and $(1-a_i)$.

Table 5.1 shows how the state of the previous carry c_{i-1}, the input a_i and the output carry c_i evolve during the execution of the circuit shown in Figure 5.6. As the X gate can be thought of as swapping the coefficients of different basis states in the linear combination, the table only shows which basis states are associated with different coefficients during execution. For simplicity only changes from the previous state are shown.

The state after the computation of the carry and before the uncomputation can, according to Table 5.1, then be described as

$$\begin{aligned}\lvert\psi_3\rangle &= \beta a_i\gamma\lvert 011\rangle + \beta a_i\delta\lvert 010\rangle + \beta(1-a_i)\gamma\lvert 000\rangle + \beta(1-a_i)\delta\lvert 001\rangle \\ &\qquad + \alpha a_i\gamma\lvert 111\rangle + \alpha a_i\delta\lvert 110\rangle + \alpha(1-a_i)\gamma\lvert 100\rangle + \alpha(1-a_i)\delta\lvert 101\rangle \\ &= (\beta\lvert 0\rangle + \alpha\lvert 1\rangle) \otimes (a_i\gamma\lvert 11\rangle + a_i\delta\lvert 10\rangle + (1-a_i)\gamma\lvert 00\rangle + (1-a_i)\delta\lvert 01\rangle).\end{aligned}$$

From this representation, it is evident that the outgoing carry qubit c_i is only flipped if $a_i = 1$, which is correct since the incoming carry c_{i-1} has been flipped and the circuit was constructed for $b_i = 0$.

In total, this gate now maps $\lvert a, c, 0\rangle$ to $\lvert a, c, r_n\rangle$, where a is the n-bit input and c is the register containing $n-2$ borrowed qubits.

A naive implementation of an addition gate using the described carry computation would simply calculate the carry for each subset of qubits in the input and then flip the result bit by bit accordingly. But Häner, Roetteler and Svore propose a different implementation in [16] which is able to use the qubits that are not used in the current iteration of the algorithm as borrowed qubits. Moreover fewer gate operations are used overall.

This divide and conquer approach recursively computes the carry for a subset of the input bits and then increments the remaining bits by one accordingly.

Incrementer

To treat carry values for a subset of qubits at once, an incrementer is used. There are many ways of implementing such a gate. One example would be to use Qiskit's multi-control X-Gate[2] to flip a qubit – starting from the most significant one – if all less significant qubits are one. Unfortunately using this gate would introduce unneccessary complexity as it transpiles into a number of operations that depends on the number of control qubits.

Fortunately, while the incrementer is used, half of the qubits required for addition are idle. These qubits can be used as borrowed qubits to create a more efficient version. As presented in [16], the incrementer is built using a circuit performing quantum-quantum addition, meaning – in contrast to the addition gates presented in this chapter – not a built-in constant is added, but the content of another quantum register.

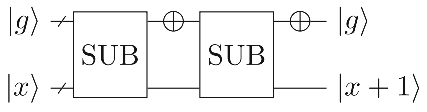


Figure 5.7: The incrementer as shown in [16] using n borrowed qubits in the register $|g\rangle$.

The quantum-quantum adder that is used for this implementation is taken from [35]. This circuit computes

$$|a\rangle\,|b\rangle \rightarrow |a\rangle\,|a+b\rangle\,,$$

where both registers are large enough to hold an n-qubit integer, while using a linear amount of Toffoli and $CNOT$ gates. Consequently, its inverse can then be used for subtraction:

$$|a\rangle\,|b\rangle \rightarrow |a\rangle\,|b-a\rangle\,.$$

As explained in [16], the circuit as shown in Figure 5.7 then realizes an incrementer. To give an argument for its correctness, assume the register containing n borrowed qubits resides in some arbitrary basis state $|g\rangle$. The circuit performs the following

[2]https://qiskit.org/documentation/stubs/qiskit.circuit.library.MCXGate.html#qiskit.circuit.library.MCXGate

transformation:

$$\begin{aligned} |g\rangle\,|x\rangle &\to |g\rangle\,|x-g\rangle \\ &\to |\overline{g}\rangle\,|x-g\rangle \\ &= |-g-1\rangle\,|x-g\rangle \\ &\to |-g-1\rangle\,|x-g-(-g-1)\rangle \\ &\to |g\rangle\,|x+1\rangle\,. \end{aligned} \tag{5.3}$$

Since the bitwise complement of g, denoted as $\overline{g}$, is equivalent to $-g-1$, where $-g$ is the two's complement representation of g, the second register in Equation (5.3) simplifies to $x-g+g+1$. As this derivation is valid for any basis state $|g\rangle$, it is furthermore valid for arbitrary content in the borrowed register, as those can be expressed as a superposition of those basis states.

Finally, a controlled incrementer using the incrementer presented above is constructed by simply treating the control qubit as the least significant qubit of the input, and therefore using one more borrowed qubit for the circuit. Should the control qubit be $|0\rangle$, only itself is incremented, and should it be $|1\rangle$, the actual value $|x\rangle$ is incremented. In the latter case the control qubit is also flipped. Therefore, to return the control qubit to its initial state, in both cases one X gate has to be appended.

Divide and Conquer Addition

Using both the carry gate and the incrementer, subsets of the input are examined recursively by splitting the current input x into x_L and x_H, where x_L is comprised of the lower $\lceil n/2 \rceil$ qubits and x_H represents the remaining $\lfloor n/2 \rfloor$ qubits in the upper half.

The circuit first calculates the carry for the addition of the constant to the lower half of the input using the upper half as borrowed qubits. The information if there is a carry is saved by toggling an ancilla qubit. This qubit now decides if one has to be added to the upper half of the input, or if the upper half of the input is left unchanged. This operation is performed by a controlled version of the incrementer.

At this point, the resulting state is $|x_L, x_H + r_{\lceil n/2 \rceil}, r_{\lceil n/2 \rceil}\rangle$ and to be able to reuse the ancilla qubit it has to be reset to zero. Since the carry operation flips the ancilla qubit if there is a carry and leaves the qubit unchanged if not, the carry gate can just be appended to the circuit again to revert the computation of $r_{\lceil n/2 \rceil}$.

Summing up, if $l = \lceil n/2 \rceil$, one recursion step can be described as follows.

$$\begin{aligned} |x_L, x_H, 0\rangle &\to |x_L, x_H, r_l\rangle \\ &\to |x_L, x_H + r_l, r_l\rangle \\ &\to |x_L, x_H + r_l, r_l \oplus r_l\rangle \\ &= |x_L, x_H + r_l, 0\rangle \end{aligned}$$

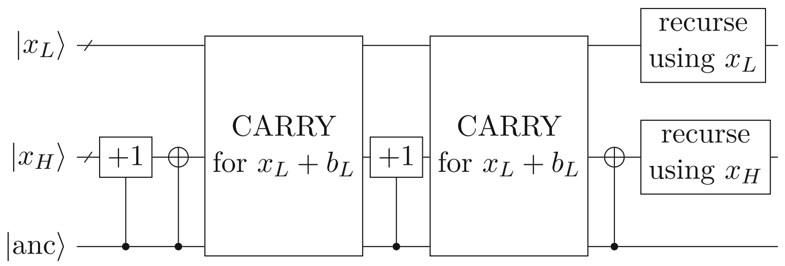


Figure 5.8: A recursive step of the computation of $x + b$ for a constant b known at the time of construction using one borrowed qubit [16].

Similar to above, the borrowed ancilla qubit cannot be assumed to be zero initially. Therefore using a similar technique the incrementation of x_H is conditioned on the toggling of this qubit by first conditionally incrementing x_H and then flipping all qubits, before performing the actual calculation. After the calculation is performed all qubits are flipped again and the ancilla qubit is reset by repeating the carry computation. Figure 5.8 shows the circuit performing this process.

To illustrate why this operation is correct, assume the ancillary qubit is in some unknown state $\alpha\,|0\rangle + \beta\,|1\rangle$. In the case where the carry is zero and the ancillary qubit is not flipped, the state evolves in the following way (x_L is left out since the content of x_L is unchanged during the computation):

$$\begin{aligned}
\alpha\,|x_H, 0\rangle + \beta\,|x_H, 1\rangle &\rightarrow \alpha\,|x_H, 0\rangle + \beta\,|(x_H + 1), 1\rangle \\
&\rightarrow \alpha\,|x_H, 0\rangle + \beta\,|{-(x_H + 1)}, 1\rangle \\
&\rightarrow \alpha\,|x_H, 0\rangle + \beta\,|{-(x_H + 1) + 1}, 1\rangle = \alpha\,|x_H, 0\rangle + \beta\,|{-x_H}, 1\rangle \\
&\rightarrow \alpha\,|x_H, 0\rangle + \beta\,|x_H, 1\rangle\,.
\end{aligned}$$

In the case where the ancilla qubit is flipped because there is a carry the state is:

$$\begin{aligned}
\alpha\,|x_H, 0\rangle + \beta\,|x_H, 1\rangle &\rightarrow \alpha\,|x_H, 0\rangle + \beta\,|(x_H + 1), 1\rangle \\
&\rightarrow \alpha\,|x_H, 0\rangle + \beta\,|{-(x_H + 1)}, 1\rangle \\
&\rightarrow \alpha\,|x_H, 1\rangle + \beta\,|{-(x_H + 1)}, 0\rangle && \text{(toggle by the first carry gate)} \\
&\rightarrow \alpha\,|x_H + 1, 1\rangle + \beta\,|{-(x_H + 1)}, 0\rangle \\
&\rightarrow \alpha\,|x_H + 1, 0\rangle + \beta\,|{-(x_H + 1)}, 1\rangle && \text{(toggle by the second carry gate)} \\
&\rightarrow \alpha\,|x_H + 1, 0\rangle + \beta\,|x_H + 1, 1\rangle\,.
\end{aligned}$$

This process is then repeated by splitting x_L and x_H recursively and applying the same operations until the base case $n = 1$ is reached.

At this point, all carry qubits that influence a particular qubit are already applied by the increment operation. Therefore the base case is only the toggling of the qubit conditioned on the input b_i.

Ultimately, this approach computes the sum $x + b$ using only one borrowed ancillary qubit.

Controlled addition

As modular exponentiation will be implemented using a series of controlled modular multiplications, which in turn are based on a series of doubly controlled modular additions, the gates shown up until now need to be adapted slightly.

The modular addition that will be used to perform modular multiplication will make use of both a single controlled carry gate and a doubly controlled carry gate.

To facilitate the use of control bits, the carry gate simply is adapted by controlling all operations that are performed on the resulting bit r_n. In the version with one control bit, the *CNOT* gates are replaced by Toffoli gates and in the version with two control bits, Qiskit's multi-control Toffoli gate can be used, as the number of control qubits stays constant.

The addition is then adapted by using this controlled carry gate.

Extension to Modular Addition

The addition and the carry computation itself can be extended to implement modular addition by performing the addition conditionally.

The modular addition has to make sure that the result $a + b$ is smaller than the modulus p. Since $a < p - b$ implies that $a + b < p$, the addition can be carried out as expected in this case. If on the other hand $a \geq p - b$, this implies that $a + b \geq p$. In this case, p has to be subtracted from the result to arrive at $(a + b) \bmod p$.

Häner, Roetteler, and Svore use an approach similar to the modular addition in [34] and [2] that implements exactly this case distinction by adding b if $a < p - b$ and subtracting $p - b$ if $a \geq p - b$.

Figure 5.9 shows the process for modular addition as described in the following paragraphs.

To arrive at a doubly controlled version of this modular addition, the doubly controlled carry gate is used as the comparator. Comparing some input a with the known constant $p - b$ is equivalent to calculating the most significant bit of the addition $a + (-(p - b))$. In the case $a < p - b$, the result is negative and the ancilla qubit is flipped.

This qubit is then used to control all subsequent operations. First, a controlled addition with b is applied. Afterwards, the ancilla qubit is flipped according to both control bits (i.e., a Toffoli gate is applied to this qubit), before finally performing the controlled subtraction by $p - b$.

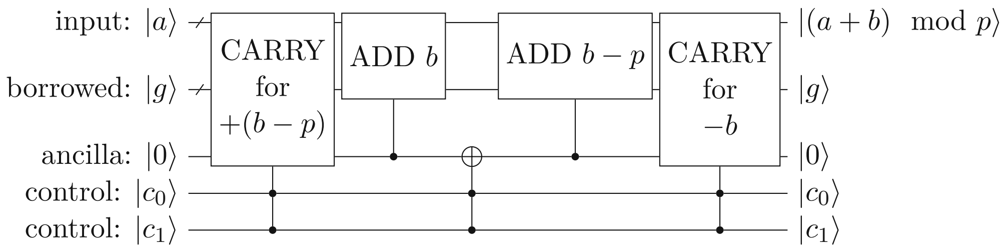


Figure 5.9: Circuit for the computation of $a + b \mod p$ using the carry gate as a comparator [16].

A subtraction can either be performed by using the fact that the addition is reversible, or by simply adding $b - p$ instead of subtracting $p - b$.

The ancilla qubit that saves the result of the comparator is expected to be zero at the beginning of the operation in Figure 5.9. This leads to a circuit that performs no operation unless both control bits $|c_0\rangle$ and $|c_1\rangle$ are set to one since both the addition and the subtraction are conditioned on the result of the comparator as saved in the ancilla qubit. If both control bits are set and the comparator still gives zero, the Toffoli gate after the addition will flip the result bit to perform the subtraction with $p - b$.

Finally, the ancilla qubit indicating the result of the comparator has to be reset to zero. This can be done by exploiting the fact that if b was added to the register, the result will always be greater or equal than b and if $p - b$ was subtracted, the result will always be less than b. As after both the conditional addition as well as the conditional subtraction have been applied the ancilla qubit is one if $p - b$ was subtracted, it can again be flipped by another comparison with b.

5.2.3 Time and Space Complexity

First, the complexity of modular addition using the QFT from Section 5.2.1 is examined. The number of qubits, excluding control qubits, this gate uses can be given as $S_{\text{qft\_mod\_add}}(n) = n + 2$ since one ancilla qubit is needed to control addition and substraction and the addition in Fourier space uses $n + 1$ qubits to handle overflows.

The non-modular addition in Fourier space is comprised of one conditional rotation for each qubit, which is a linear time algorithm. However, the time complexity of the quantum Fourier transform in the algorithm for modular addition dominates the runtime of the circuit. The circuit for modular addition is comprised of 5 additions and subtractions on $n + 1$ qubits as well as 4 applications of the QFT or its inverse on $n + 1$ qubits and some constant number of operations manipulating the ancilla qubit. Its runtime can be given as

$$T_{\text{qft\_mod\_add}}(n) \in O(5(n + 1) + 4(n + 1)^2) = O(n^2).$$

In the following only the leading most significant term of the polynomial representing the runtime bound will be examined.

The carry-based addition uses a gate computing the final carry after the addition of a constant as its main building block. In the worst case, this gate uses 6 operations per input qubit which gives a time complexity of $T_{\text{carry}}(n) \in O(n)$. Note that in contrast to the analysis in [16] where the number of applied Toffoli gates is of interest and all other operations in the carry gate are treated as constant overhead, here the number of elementary operations is considered.

The next building block is an incrementer gate. This gate uses an addition gate acting on two quantum registers which computes the result in $O(n)$ [35]. In the incrementer, the adder is used twice on $n+1$ qubits (to include the control qubit). Additionally $O(n)$ X gates are applied, which gives the time complexity of the incrementer as $T_{\text{inc}}(n) \in O(n)$.

In each recursive step of the addition, the carry gate as well as the incrementer are applied twice for half the current qubits. Additionally the upper half of the register is conditionally flipped twice. This leads to a complexity of each recursive step given by $T_{\text{step}}(n) \in O(T_{\text{carry}}(n/2) + T_{\text{inc}}(n/2) + 2n/2) = O(n)$. The total time complexity of the addition can then be given using a recursive formula similar to [16]:

$$T_{\text{carry\_add}}(n) = T_{\text{carry\_add}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{\text{carry\_add}}\left(\left\lceil \frac{n}{2} \right\rceil\right) + T_{\text{step}}(n).$$

This gives the runtime of the carry-based addition as

$$T_{\text{carry\_add}}(n) \in O(n \log_2(n)).$$

Finally, the extension of this gate to perform modular addition (see Figure 5.9), uses two adders as well as two carry gates to perform the comparison which leads to a total runtime of

$$T_{\text{carry\_mod\_add}}(n) \in O(2T_{\text{carry\_add}(n)} + 2T_{\text{carry}}(n)) = O(n \log_2(n)).$$

This shows that although both approaches give polynomial time algorithms the carry-based approach, ignoring multiplicative constants, is more efficient.

What remains to be shown is the number of qubits used by the carry-based addition. This number, excluding control qubits, is greater than the corresponding number for the version using the quantum Fourier transform:

$$S_{\text{carry\_mod\_add}}(n) = n + (n-1) + 1 = 2n.$$

This is due to the fact that $n-1$ borrowed qubits are needed to perform the comparisons using the carry gate. However, as will be shown in the next section, there are enough idle qubits that can be borrowed during multiplication so this shortcoming does not affect the number of used qubits for the multiplication negatively.

5.3 Multiplication

As described in [2, 16, 34], both implementations presented in Section 5.2 can be utilized to perform modular multiplication. Since the goal is to perform modular exponentiation, a controlled version of the multiplication is needed. Ultimately, assuming the control qubit c is set, the following operation will be carried out:

$$|c, x\rangle \rightarrow |c, bx \bmod p\rangle \, .$$

5.3.1 Partial Multiplication

The multiplication circuit is built using two partial multiplications and a swap of registers. This partial multiplication computes $|x, y\rangle \rightarrow |x, (y + bx) \bmod p\rangle$ for an integer constant b. The first quantum register contains the n bit integer x and the second register will contain the n bit result. Additionally, the ancilla qubits required for the modular addition as well as one control qubit are part of the circuit as well.

The top register containing x will, in addition to the overall control bit, be used to control modular additions in the second register. Using the decomposition of x into its binary representation $x = 2^0 x_0 + 2^1 x_1 + \cdots + 2^{n-1} x_{n-1}$, the circuit adds the precomputed constant $b \cdot 2^i$ conditioned on the qubit x_i, which results in the following state transitions:

$$\begin{aligned}
|x, y\rangle &\rightarrow |x, (y + b \cdot 2^0 x_0) \bmod p\rangle \\
&\rightarrow |x, ((y + b \cdot 2^0 x_0 \bmod p) + b \cdot 2^1 x_1) \bmod p\rangle \\
&\ldots \\
&\rightarrow |x, (\ldots(y + b \cdot 2^0 x_0 \bmod p) + b \cdot 2^1 x_1 \bmod p) + \cdots) + b \cdot 2^{n-1} x_{n-1}) \bmod p\rangle \\
&= |x, (y + b \cdot (2^0 x_0 + 2^1 x_1 + \cdots + 2^{n-1} x_{n-1})) \bmod p\rangle \\
&= |x, (y + bx) \bmod p\rangle \, .
\end{aligned}$$

These additions can be carried out using a series of the doubly controlled gates presented in Section 5.2 conditioned on the individual bits x_i in the first register.

5.3.2 Complete Multiplication

To arrive at the desired result containing only $ax \bmod p$ in the first register, some additional work has to be done. First, the multiplication as described above is performed, which takes the system from $|x, 0\rangle$ to the state $|x, (0 + bx) \bmod p\rangle$. Then a swap of both registers conditioned on a control bit is performed. Finally, another application of the multiplication described in the previous subsection with the additive inverse of the multiplicative inverse of b modulo p gives the final result. Since for the purpose of this algorithm p is always prime, there will always be such a multiplicative inverse.

To illustrate how the circuit will arrive at the desired state $|bx \bmod p, 0\rangle$, the individual transitions, assuming the control qubit is set, are the following:

$$\begin{aligned}|x, 0\rangle &\to |x, bx \bmod p\rangle \\ &\to |bx \bmod p, x\rangle\,.\end{aligned}$$

At this point, another partial multiplication $|x', y'\rangle \to |x', (y' + (-b^{-1})x') \bmod p\rangle$, with $x' = bx \bmod p$ and $y' = x$ is performed. This completes the uncompuation of x in the second register because, as the following calculation shows, the second register is $|0\rangle$.

$$\begin{aligned}(y' + (-b^{-1})x') \bmod p &= (x - b^{-1}(bx \bmod p)) \bmod p \\ &= (x - b^{-1}bx) \bmod p \\ &= (x - x) \bmod p \\ &= 0\end{aligned}$$

Since the second register is initially set to zero and will arrive at zero after the computation, it can be seen as n additional ancilla bits that are used for multiplication.

5.3.3 Time and Space Complexity

The multiplication algorithm presented in Section 5.3 can be based on any of the modular addition circuits presented above. Consequently its runtime will be given depending on $T_{\text{mod\_add}}(n)$ which can be substituted with the respective measure.

The multiplication starts with performing n modular additions, then swaps both registers before finally performing another n modular additions. Each swap operation can be performed using a constant number of *CNOT* gates. The complexity of this operation is dominated by the time needed to perform the additions, which is why the swaps can be ignored for the final measure. This leads to a runtime complexity of

$$T_{\text{mod\_mul}}(n) \in O(n \cdot T_{\text{mod\_add}}(n)).$$

Substituting the runtime bounds for both addition circuits above yields

$$T_{\text{qft\_mod\_mul}}(n) \in O(n^3)$$

and

$$T_{\text{carry\_mod\_mul}}(n) \in O(n^2 \log_2(n)).$$

The multiplication circuit uses two registers, where both are big enough to hold an n qubit integer. In addition to that, the QFT-based modular addition needs one qubit to hold the addition overflow and another qubit to analyze the most significant qubit after addition. In total $S_{\text{qft\_mod\_mul}}(n) = 2n + 2$ excluding the control qubit. The carry-based addition circuit only uses one ancilla qubit to hold the most significant qubit after addition but uses $n-1$ borrowed qubits. However, each addition step in the partial multiplication shown in Section 5.3.1 only uses one qubit of the left register as control. Therefore there are $n-1$ idle qubits that can be used as borrowed qubits without increasing the number of total qubits used. This leads to the space complexity of $S_{\text{carry\_mod\_mul}}(n) = 2n+1$.

Having examined the circuit size for both of these implementations, the overall time-complexity for modular exponentiation can be given as

$$T_{\text{qft\_mod\_exp}}(n) \in O(n^4)$$

and

$$T_{\text{carry\_mod\_exp}}(n) \in O(n^3 \log(n)).$$

5.4 Montgomery Modular Multiplication

The multiplication algorithm presented in the previous section is able to perform multiplication modulo a prime p by using modular adders in succession. This creates overhead, as it requires a conditional subtraction or addition in each addition step to transform the result in its representation modulo p. Of course a naive implementation of modular multiplication would be to multiply the inputs in a register sized appropriately to hold the result and then perform a division by p to obtain the residue at the end only.

Implementing such a division is quite cumbersome, as it requires trial subtractions by multiples of p and corrections if the result becomes negative. Additionally, these operations have to be conditioned on the most significant bit of the currently examined value, which in case of the addition in Fourier space as presented in Section 5.2.1 neccesitates performing a Quantum Fourier transform and its inverse in each step, thus further increasing the overhead.

Montgomery modular multiplication [22] speeds up the modular multiplication process by changing the representation of the input to allow for easier reduction of the result to a representation modulo p.

For the algorithm, the performed multiplication is changed slightly. Instead of performing the multiplication $x \cdot b$ for a constant b, the algorithm performs the multiplication $t = x \cdot (bR \bmod p)$, for some R such that $gcd(R, p) = 1$.

The factor $bR \bmod p$ is then called the *Montgomery form* of b. Of course to use the result in further calculations, a final reduction, the *Montgomery reduction*

$$\text{REDC}(x \cdot (bR \bmod p) \mid p, R) = x \cdot (bR \bmod p)R^{-1} \pmod{p}$$

is needed.

Rines and Chuang [30] propose an algorithm that performs the multiplication as well as the subsequent Montgomery reduction by using addition in Fourier space as presented in Section 5.2.1. However, the authors employed an adapted version of the Fourier transform that uses rotations around the Y-axis. This version might introduce unwanted phase differences, which is why in this text a version using the usual QFT is presented.

For the reduction, R is chosen to be 2^n, where n is the smallest n such that $2^n > p$. In the first step, the values $u = tp^{-1} \bmod 2^n$ and $s = (t - up)/2^n$ are calculated. For the second value s, it can be shown that this result is congruent to the desired result $t2^{-n} (\bmod p)$:

$$\begin{aligned}(t - up)2^{-n} &\equiv t2^{-n} - up2^{-n} \pmod p\\ &\equiv t2^{-n} \pmod p.\end{aligned}$$

In this congruence that fact that $p \equiv 0 \pmod p$ is used. Furthermore, the division in the calculation for s will always give an integer result as

$$t - up \equiv t - tp^{-1}p \equiv 0 \pmod{2^n}.$$

Finally, it also holds that s is not only congruent to the result, but at most one correcting addition away from the desired result.

$$\frac{t - up}{2^n} \geq \frac{-up}{2^n} = \frac{u}{2^n}(-p)$$

The inequality holds since t is always greater than or equal to 0, as it is the result of the multiplication of the positive integer x and the positive Montgomery form of b. Furthermore, both u and p are positive integers.

Since $u < 2^n$, u can be substituted by 2^n to show that $\frac{t-up}{2^n} \geq \frac{2^n}{2^n}(-p) = -p$. Using the fact that both u and p are positive, an upper bound can be given:

$$\frac{t - up}{2^n} < \frac{t}{2^n}.$$

As t is the result of a multiplication of two integers smaller than p and $2^n > p$ is set by construction, the following holds for the result:

$$-p \leq \frac{t - up}{2^n} < \frac{p \cdot p}{p} = p. \tag{5.4}$$

To sum up, this step gives an integer s which is congruent to the desired result $t2^{-n} \pmod p$ and lies between $-p$ and p. Therefore to get the final result only a conditional addition of p is needed, in case p is negative.

5.4.1 Quantum Algorithm

Of course for the implementation of the algorithm outlined above, any addition algorithm can be used. However, using the addition in Fourier space as described in Section 5.2.1, allows for certain optimizations that will be described here, which is why this addition algorithm is chosen.

The algorithm presented in [30] transforms the result of the multiplication $t = x(bR \bmod p)$ to $xb \bmod p$. To perform such a multiplication, repeated additions similar to the ones described in Section 5.3.1 can be used, with the only difference that regular addition instead of modular addition is used and the result register is sized appropriately to hold the result of the multiplication.

Given the input $|c\rangle_1 |x\rangle_n |0\rangle_{2n+1}$, the first register contains the control qubit and the second register contains the input x and has size n, such that $2^n > p$. Since the third register will have to hold the result of the multiplication, $2n$ qubits have to be used. Additionally one ancilla qubit that will be required later is also included.

To keep the description concise, the control qubit c and the input x will be left out of the description of the state. Furthermore, since the register is subdivided in varying ways during the algorithm, the size of a register will be given as subscript in the descriptions, in case the size is greater than one.

To compute t in the second register, repeated additions are performed. By adding in Fourier space, only one QFT at the beginning is needed. Afterwards, the controlled additions (see Section 5.2.1) can be performed in sequence without leaving Fourier space. The factor $bR \bmod p$ is known at the time of construction and can therefore be build into the circuit.

After this initial multiplication, the system resides in the state

$$|\phi(t)\rangle_{2n+1} = |\phi(x \cdot (bR \bmod p))\rangle_{2n+1} .$$

The quantum Montgomery reduction will now take this state and transform it to

$$|0\rangle_n |xb \bmod p\rangle_n |0\rangle_1 .$$

Estimation Stage

The first stage of the Montgomery reduction calculates the result $s = (t - up)/2^n$. From s alone, the input t cannot be reconstructed. Consequently, the value u is also part of the output:

$$|\phi(t)\rangle_{2n+1} \to |\phi((t - up)/2^n)\rangle_{n+1} |u\rangle_n . \tag{5.5}$$

In each iteration, one qubit u_i of the value in the second result register is fixed by simply assigning the current least significant qubit of the first register as the next qubit of the second register. Of course the left register contains the value in Fourier space and it therefore has to be transformed first, as the register content at the beginning of the ith iteration is as follows.

$$|\phi(t')\rangle_l |u\rangle_{i-1} = |\phi_l(t')\rangle \dots |\phi_1(t')\rangle |u_{i-1}\rangle \dots |u_1\rangle$$

Here, l is the integer such that $l + (i - 1) = 2n + 1$.

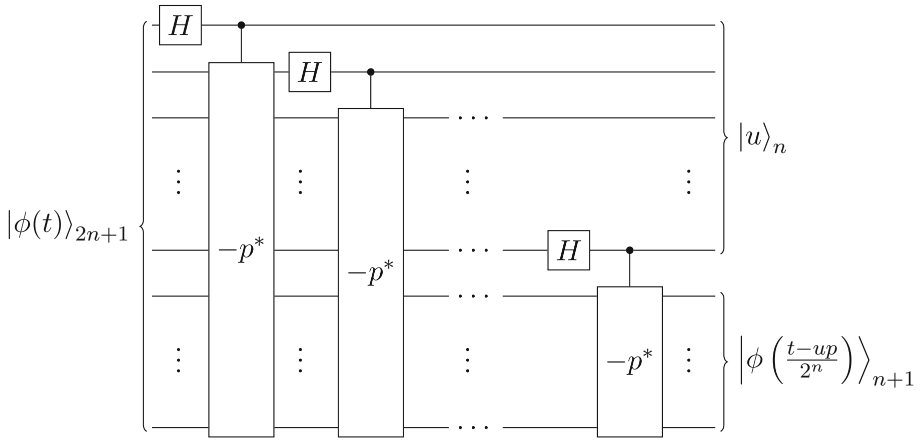


Figure 5.10: Estimation stage for quantum Montgomery reduction, as shown in [30]. The $2n+1$ qubit register containing the input is successively reduced in size, which implicitly divides its content by 2. Here $-p^*$ denotes subtractions that ignore the least significant qubit. Note that the register order differs from Equation (5.5) to highlight the fact that the least significant qubit of the bottom register is reused as the most significant qubit of the top register.

Fortunately, the properties of the Quantum Fourier transform suggest that not the whole register, but only the least significant qubit, has to be transformed. Examining this particular qubit at the beginning of each iteration, where $|\phi(t')\rangle_l$ is the register content at this moment, reveals that a single Hadamard gate will suffice to transform this qubit back to the computational basis.

$$\begin{aligned} H\,|\phi_1(t')\rangle &= H\left(\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i t'}{2}}\,|1\rangle\right)\right) \\ &= H\left(\frac{1}{\sqrt{2}}\left(|0\rangle + (-1)^{(t' \bmod 2)}\,|1\rangle\right)\right) \\ &= |t'_1\rangle \end{aligned} \tag{5.6}$$

The Montgomery reduction thus only requires a simple extraction of the least significant qubit, whereas the trial subtractions, required for division, need a full QFT on the whole register.

As the least significant bit, from now on referred to as u_i, has been extracted and added to the second register, the value in the first register has effectively been divided by two.

In the last step of each iteration, the extracted qubit u_i is then used to control a subtraction by $\lfloor p/2 \rfloor$. Making use of the properties for the QFT yet another time, the subtraction can be performed by not subtracting $p/2$ from the resulting $l-1$ qubit register, but instead subtracting p from the left register in combination with the just now extracted u_i, but ignoring all operations that would be performed on u_i.

To show the correctness for all iterations, a formal argument is now given.

Lemma 5.4.1. *One iteration of the estimation stage performs the transformation*

$$\begin{aligned}|\phi(t')\rangle_l |u\rangle_{i-1} &\to |\phi(t'/2 - t'_1 \cdot (p/2))\rangle_{l-1} |t'_1\rangle_1 |u\rangle_{i-1}\\ &= |\phi(t'/2 - u_i \cdot (p/2))\rangle_{l-1} |u_i\rangle_1 |u\rangle_{i-1}\\ &= |\phi(t'/2 - u_i \cdot (p/2))\rangle_{l-1} |u\rangle_i .\end{aligned}$$

Proof. In the proof, the cases where t' is even and where t' is odd are distinguished.

Case 1: t' is even. It follows from Equation (5.6) that $|\phi_1(t')\rangle = H|0\rangle$, and therefore $|t'_1\rangle = |u_i\rangle = |0\rangle$ after the transformation of the least significant qubit. Therefore, no subtraction will be performed and after the iteration, the system will be in the state

$$|\phi_l(t')\rangle \dots |\phi_2(t')\rangle |0\rangle |u\rangle_{i-1} .$$

Examining only the leftmost $l-1$ qubits, this state can further be rewritten as

$$\left[\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i t'}{2^l}}|1\rangle\right)\right] \otimes \dots \otimes \left[\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i t'}{2^2}}|1\rangle\right)\right].$$

As t' is even it further holds that $\frac{t'}{2^l} = \frac{\frac{t'}{2}}{2^{l-1}}$ and $\frac{t'}{2} \in \mathbb{Z}$. Using this equality, the former state is expressable as

$$\left[\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i \frac{t'}{2}}{2^{l-1}}}|1\rangle\right)\right] \otimes \dots \otimes \left[\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i \frac{t'}{2}}{2}}|1\rangle\right)\right]$$

$$\begin{aligned}&= |\phi_{l-1}(t'/2)\rangle \otimes \dots \otimes |\phi_1(t'/2)\rangle\\ &= |\phi((t'/2) - 0 \cdot (p/2))\rangle_{l-1}\\ &= |\phi((t'/2) - u_i \cdot (p/2))\rangle_{l-1} .\end{aligned}$$

Case 2: t' is odd. It holds that $|\phi_1(t')\rangle = H|1\rangle$ and therefore $|t'_1\rangle = |u_i\rangle = |1\rangle$. As $u_i = 1$, the subtraction with p, ignoring the least significant bit as described above, is performed on the leftmost $l-1$ qubits:

$$|\phi_l(t')\rangle \dots |\phi_2(t')\rangle |1\rangle \to |\phi_l(t'-p)\rangle \dots |\phi_2(t'-p)\rangle |1\rangle .$$

As in this case for any prime $p > 2$, $t' - p$ is even and $\frac{t'-p}{2} \in \mathbb{Z}$, a similar equality can be given:

$$\frac{t'-p}{2^l} = \frac{\frac{t'}{2} - \frac{p}{2}}{2^{l-1}}.$$

Applying this equality in a similar manner as in the first case gives the desired representation of the leftmost $l-1$ qubits of the result after the subtraction.

$$\begin{aligned}
&\left[\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i(t'-p)}{2^l}}|1\rangle\right)\right] \otimes \cdots \otimes \left[\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i(t'-p)}{2^2}}|1\rangle\right)\right] \\
= &\left[\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i\left(\frac{t'}{2}-\frac{p}{2}\right)}{2^{l-1}}}|1\rangle\right)\right] \otimes \cdots \otimes \left[\frac{1}{\sqrt{2}}\left(|0\rangle + e^{\frac{2\pi i\left(\frac{t'}{2}-\frac{p}{2}\right)}{2}}|1\rangle\right)\right] \\
&= |\phi((t'/2) - 1 \cdot (p/2))\rangle_{l-1} \\
&= |\phi((t'/2) - u_i \cdot (p/2))\rangle_{l-1}
\end{aligned}$$

Again, the fact that this is the representation of the subtraction result in Fourier space (see Section 4.1) is used.

Summing up, this shows that in both cases the result of the iteration is

$$\begin{aligned}
|\phi(t'/2 - u_i \cdot (p/2))\rangle_{l-1} |t'_1\rangle |u\rangle_{i-1} &= |\phi(t'/2 - u_i \cdot (p/2))\rangle_{l-1} |u_i\rangle |u\rangle_{i-1} \\
&= |\phi(t'/2 - u_i \cdot (p/2))\rangle_{l-1} |u\rangle_i \,. \qquad \square
\end{aligned}$$

The estimation stage starts with a register containing the Fourier-transformed result of the initial multiplication of the integer x with the Montgomery form of the constant b. This product is denoted by $|\phi(t)\rangle_{2n+1} = |\phi(x \cdot (b2^n \bmod p))\rangle_{2n+1}$. In total, the transformation described by Lemma 5.4.1 will be performed n times for a decreasing number of qubits in the register. It remains to show that ultimately, the state

$$|\phi((t-up)/2^n)\rangle_{n+1} |u\rangle_n$$

as described at the beginning of Section 5.4 is reached.

Theorem 5.4.2. *After n iterations as described above, the estimation stage performs the transformation*

$$|\phi(t)\rangle_{2n+1} \to |\phi((t-up)/2^n)\rangle_{n+1} |u\rangle_n \,,$$

where $u = tp^{-1} \bmod 2^n$.

Proof. The proof will proceed by induction on the number i of performed iterations. In constrast to u_i which describes the ith bit of the register containing u, $t^{(i)}$ and $u^{(i)}$ describe the contents of the first and second register after the ith iteration in this proof.

Base case: $i = 1$. In the base case it has to be shown that $u^{(1)} = tp^{-1} \bmod 2^1$ and $t^{(1)} = (t - u^{(1)}p)/2^1$.

As p is assumed to be odd, the equation $pp^{-1} \equiv 1 \bmod 2^n$ implies that p^{-1} is odd also, which in turn shows that the least significant bit of tp^{-1} depends solely on t. Therefore the equality

$$tp^{-1} \bmod 2 = t \bmod 2 = t_1$$

applies. Lemma 5.4.1 shows that the least significant qubit t_1 is extracted in the iteration and assigned in the base case as the first bit of u. Using this fact $u^{(1)} = t_1 = tp^{-1} \bmod 2^1$ is shown.

Furthermore, Lemma 5.4.1 asserts that the state $|\phi(t)\rangle$ is transformed to $|\phi(t/2 - u_1 \cdot (p/2))\rangle$. Since $u^{(1)}$ consists only of the single bit u_1 also the equation $t^{(1)} = (t - u^{(1)}p)/2^1$ holds.

Induction hypothesis: Suppose $i \geq 1$ and $u^{(i)} = tp^{-1} \bmod 2^i$ and $t^{(i)} = (t - u^{(i)}p)/2^i$ holds.

Induction step: Lemma 5.4.1 states that in one iteration the state $|\phi(t^{(i)})\rangle\,|u^{(i)}\rangle$ is transformed to $|\phi(t^{(i)}/2 - u_{i+1} \cdot (p/2))\rangle\,|t_1^{(i)}\rangle\,|u^{(i)}\rangle$. It first will be shown that the extension of $|u^{(i)}\rangle$ with the new most significant qubit $|u_{i+1}\rangle = |t_1^{(i)}\rangle$ preserves the condition $u^{(i+1)} = tp^{-1} \bmod 2^{i+1}$.

As $t_1^{(i)}$ is the least significant bit of $t^{(i)}$, it can also be expressed as $t^{(i)} \bmod 2$. Therefore the extension of $u^{(i)}$ with this bit as its most significant bit is given by

$$\begin{aligned}
u^{(i+1)} &= u_{i+1}2^i + u^{(i)} \\
&= (t^{(i)} \bmod 2)2^i + u^{(i)} \\
&= \left(\frac{t - u^{(i)}p}{2^i} \bmod 2\right)2^i + u^{(i)} && \text{(using ind. hyp. for } t^{(i)}\text{)} \\
&= \left(\frac{t - (tp^{-1} \bmod 2^i) \cdot p}{2^i} \bmod 2\right)2^i + tp^{-1} \bmod 2^i. && \text{(using ind. hyp. for } u^{(i)}\text{)}
\end{aligned}$$

Now the congruence $t \equiv tpp^{-1} \bmod 2$ can be applied resulting in

$$u^{(i+1)} = \left(\frac{tpp^{-1} - (tp^{-1} \bmod 2^i) \cdot p}{2^i} \bmod 2\right)2^i + tp^{-1} \bmod 2^i.$$

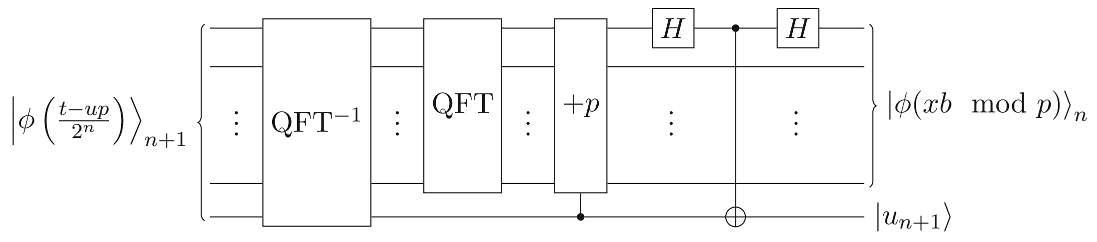
Aditionally, the fact that for odd p, $p \equiv 1 \bmod 2$ is used to reduce the equation to

$$u^{(i+1)} = \left(\frac{tp^{-1} - (tp^{-1} \bmod 2^i)}{2^i} \bmod 2\right)2^i + tp^{-1} \bmod 2^i.$$

The first half of this expression represents exactly the $(i+1)$th bit of tp^{-1}, which is then shifted by i bits to the left before being added to $tp^{-1} \bmod 2^i$. In total this gives

$$\begin{aligned}
u^{(i+1)} &= (tp^{-1})_{i+1}2^i + (tp^{-1} \bmod 2^i) \\
&= tp^{-1} \bmod 2^{i+1}.
\end{aligned}$$

It remains to be shown that $t^{(i+1)} = (t - u^{(i+1)}p)/2^{i+1}$. As mentioned above, the value in the leftmost register after this iteration is given by $|\phi(t^{(i)}/2 - u_{i+1} \cdot (p/2))\rangle$. Therefore



$\left|\phi\left(\frac{t-up}{2^n}\right)\right\rangle_{n+1}$
QFT^{-1}
QFT
$+p$
H
H
$|\phi(xb \mod p)\rangle_n$
$|u_{n+1}\rangle$

Figure 5.11: From a value that is congruent to the desired result, the correction stage computes the representative of the residue class.

$t^{(i+1)}$ can be expressed as

$$\begin{aligned} t^{(i+1)} &= t^{(i)}/2 - u_{i+1} \cdot (p/2) \\ &= \frac{\frac{t-u^{(i)}p}{2^i}}{2} - u_{i+1}\frac{p}{2} && \text{(using the induction hypothesis)} \\ &= \frac{t-u^{(i)}p}{2^{i+1}} - \frac{2^i u_{i+1} p}{2^{i+1}} \\ &= \frac{t-(u_{i+1}2^i + u^{(i)})p}{2^{i+1}}. \end{aligned}$$

Similar to the argument for $u^{(i+1)}$, the extension with the new most significant bit can be expressed as $u^{(i+1)} = u_{i+1}2^i + u^{(i)}$. This fact can be applied to show that the value in the leftmost register after the iteration is

$$t^{(i+1)} = \frac{t-u^{(i+1)}p}{2^{i+1}},$$

which concludes this argument. □

In total this estimation stage is performed only using Hadamard gates and conditional subtractions, without ever having to fully transform the result back to the computational basis. As noted at the end of the introduction of this section, a conditional addition is still needed to calculate the representative of the residue class.

Correction and Uncomputation

After the estimation stage, the system resides in the state

$$|\phi((t-up)/2^n)\rangle_{n+1} |u\rangle_n ,$$

where the first register contains a value that is congruent to the desired representative of the residue class, and (by Equation (5.4)) lies between $-p$ and p. In case the value is negative, a correcting addition with p is still needed. Unfortunately, to examine the sign of the value, the most significant qubit s_{n+1} has to be extracted. This requires a complete inverse QFT on the left register and another QFT on all but the sign-qubit in the left register giving

$$|s_{n+1}\rangle\,|\phi(s)\rangle_n\,|u\rangle_n\,.$$

So after an addition controlled by $|s_{n+1}\rangle$, the second register finally will contain the result

$$|s_{n+1}\rangle\,|\phi(xb \bmod p)\rangle_n\,|u\rangle_n\,.$$

Now two parts have to be uncomputed, the sign qubit $|s_{n+1}\rangle$ and the value $|u\rangle_n$. Rines and Chuang [30] first modify the sign bit to be able to include it in the last register, thus extending $|u\rangle_n$ to $|u\rangle_{n+1}$ and finally uncompute $|u\rangle_{n+1}$. This is achieved by using the fact that, if the conditional addition with the odd value p has been performed, the least significant qubit of the result $|(xb \bmod p)_1\rangle$ will change. By flipping $|s_{n+1}\rangle$, conditioned on this qubit, the least significant qubit $|s_1\rangle$ before this correction can be restored. Summing up, a Hadamard gate will be used to extract $|(xb \bmod p)_1\rangle$. Afterwards, this qubit will control a *CNOT* operation on $|s_{n+1}\rangle$, before restoring $|\phi_1(xb \bmod p)\rangle$ using another Hadamard gate.

As the newly calculated qubit $|s_1\rangle$ represents the least significant qubit before the correction it can be treated similar to the estimation stage and reinterpreted as the most significant bit $|u_{n+1}\rangle$ of the last register. In total, the state can now be described as

$$|u_{n+1}\rangle\,|\phi(xb \bmod p)\rangle_n\,|u\rangle_n\,.$$

This way, only the value $u = tp^{-1} \bmod 2^{n+1}$ has to be uncomputed. For this task, the manner in which t was computed at the beginning of this section, can be applied in reverse. Since t is the result of conditional additions with a value $bR \cdot 2^i \bmod p$ that is known at the time of construction a similar representation can be used to express u:

$$u = \sum_{i=0}^{n-1} x_i (b2^n \cdot 2^i \bmod p)p^{-1} \mod 2^{n+1}.$$

Therefore, n conditional subtractions by $(b2^n \cdot 2^i \bmod p)p^{-1}$ will restore the value of the sign qubit as well as the last register to zero.

Finally, the result of the full Montgomery reduction is the result of the multiplication (including the input $|x\rangle_n$):

$$|x\rangle_n\,|0\rangle\,|\phi(xb \bmod p)\rangle_n\,|0\rangle_n\,.$$

A final inverse QFT can now be used to obtain

$$|x\rangle_n\,|0\rangle\,|xb \bmod p\rangle_n\,|0\rangle_n\,.$$

In-place Multiplication

In the last step, this multiplication algorithm has to be extended to allow in-place multiplication as this is required to use it in Shor's algorithm. As described in [30], this is achieved by examining some properties of the out-of-place multiplication an its inverse.

The out-of-place algorithm performs the multiplication

$$|a\rangle_n |0\rangle_{2n+1} \to |a\rangle_n |0\rangle |ab \bmod p\rangle_n |0\rangle_n$$

for a classical value b that is known at the time of construction. Ignoring the ancilla qubits from now on, using the input $b^{-1} \bmod p$, the gate transforms $|a\rangle_n |0\rangle_n$ to

$$|a\rangle_n |ab^{-1} \bmod p\rangle_n .$$

Therefore its inverse can be described as

$$|a\rangle_n |ab^{-1} \bmod p\rangle_n \to |a\rangle_n |0\rangle_n . \tag{5.7}$$

Since for any x, $x = (xb \bmod p) \cdot b^{-1} \bmod p$, the description of the inverse gate in Equation (5.7) can be used to show that the following transformation can be performed:

$$\begin{aligned} |xb \bmod p\rangle_n |x\rangle_n &= |xb \bmod p\rangle_n |(xb \bmod p) \cdot b^{-1} \bmod p\rangle_n \\ &\to |xb \bmod p\rangle_n |0\rangle_n . \end{aligned}$$

Therefore, to obtain an algorithm for in-place multiplication, first an out-of-place multiplication with b is performed, then both registers are swapped, before finally performing the inverse of the out-of-place multiplication with b^{-1} as input.

As a controlled version of this gate is needed, the out-of-place mutliplication can be controlled by conditioning the additions not only on the input x but also on the control bit. This requires doubly controlled additions as described in Section 5.2.1. Additionally the swap has to be performed based on the control bit.

In total the gate can then be described as

$$\text{MONT\_MUL}_b(|1\rangle |x\rangle_n |0\rangle_{2n+1}) = |1\rangle |xb \bmod p\rangle_n |0\rangle_{2n+1}$$

and

$$\text{MONT\_MUL}_b(|0\rangle |x\rangle_n |0\rangle_{2n+1}) = |0\rangle |x\rangle_n |0\rangle_{2n+1} .$$

5.4.2 Time and Space Complexity

A major drawback in terms of runtime efficiency for the implementation of the modular multiplication based on the adder in Fourier space (see Section 5.2.1 and 5.3) is that for each modular addition the most significant qubit needs to be extracted using a full quantum Fourier transform.

Since in the algorithm for Montgomery multiplication not the most but the least significant qubit is extracted, and since this operation can be done in constant time with only one Hadamard gate, the circuit size tends to be significantly smaller.

In summary, the circuit presented here performs the following operations:

- Starting from the input in the computational basis, the QFT is applied which is executed in quadratic runtime.
- The multiplication is then performed using n additions on $O(n)$ qubits giving $O(n^2)$.
- The estimation stage then applies n Hadamard gates and subtracts n times on a subregister on decreasing size of at most $O(2n)$ which results in a total runtime of this stage as $O(n^2)$.
- In the correction stage, one QFT, one inverse QFT and an addition each on $O(n)$ qubits is applied. Additionally a constant number of gates are required giving $O(n^2)$.
- Finally, the uncomputation stage is composed of n subtractions on a register of size $O(n)$, which is also performed in quadratic time.

This shows that the largest term in the polynomial representing the runtime is a quadratic term and the runtime can be given as

$$T_{\text{montgomery\_mod\_mul}}(n) \in O(n^2).$$

Substituting into the corresponding formula for modular exponentiation, this yields the following bound:

$$T_{\text{montgomery\_mod\_exp}}(n) \in O(n^3).$$

CHAPTER 6

Results and Discussion

For this thesis, the algorithm presented in Chapter 4 including its optimized variants has been implemented in Qiskit[1]. The implementation utilizes the multiplication and modular exponentiation circuits presented in Chapter 5 as oracles.

Also, other implementation variants were considered for this work. One notable example is presented in [19]. Here the fact that given an oracle that estimates one bit of the discrete logarithm with success probability $1/2+\epsilon$, where ϵ is small but positive, the discrete logarithm can be obtained after a polynomial number of queries [19]. The quantum part of the algorithm is therefore adapted to only obtain one bit of the result.

In this quantum algorithm, the second stage is initialized not with the value 0 as in the version used for this text, but with a value that is classically computed based on the measurement of the first stage. As this is currently not possible in Qiskit, the only way of implementing this algorithm would be to implement the classical computation of the initialization value in the quantum circuit itself. Since this process would prompt the need for even more ancilla qubits and therefore would increase space complexity drastically, this algorithm was not chosen for implementation.

For the implementation of Shor's algorithm, a gate computing modular exponentiation based on modular multiplication is required. The fastest classical multiplication algorithm, for large n, is the Schönhage-Strassen multiplication based on the fast Fourier transform [31]. This algorithm however uses a lot of additional space as ancilla qubits [30, 39]. Since the goal of this work was to keep the number of qubits low, the multiplication gates presented in Chapter 5 have been chosen.

This chapter examines the success probability of Shor's algorithm for the discrete logarithm problem using simulations performed in Qiskit. For this comparison, a fixed number of trials was performed and it was recorded how many of them were successful, by

[1]`https://github.com/mhinkie/ShorDiscreteLog`

performing modular exponentiation with the resulting discrete logarithm classically to verify the result. Additionally, the different implementations of modular exponentiation are compared with respect to the number of used qubits as well as the number of applied gate operations.

Using the optimizations from Section 4.3.5, the number of used qubits for the overall algorithm can be reduced to $S(n) + 1$, where $S(n)$ is the space required to perform modular exponentiation, excluding control qubits, for an n-bit module. Therefore the smallest circuit presented in this thesis, measured by its space complexity, is based on the modular exponentiation gate by Häner, Roetteler and Svore [16], where $S(n) = 2n + 1$. Unfortunately, as noted in the description of the optimization in Section 4.3.5, it is not yet possible to execute this particular circuit on the quantum computers used for this work as the feature of controlling quantum gates by classical bits is not yet available. However there have been promising results indicating that it is possible to use this variant of phase estimation on quantum computers [9], which is why it was implemented in this work aswell. Furthermore, according to IBM's roadmap for quantum computer development [18, 36], this feature, referred to as 'dynamic circuits', should be available soon.

Modular exponentiation gate	# of qubits $S(n)$	Gate operations
Beauregard [2]	$2n + 2$	$O(n^4)$
Häner, Roetteler and Svore [16]	$2n + 1$	$O(n^3 \log(n))$
Rines and Chuang [30]	$3n + 1$	$O(n^3)$

Table 6.1: Number of qubits, excluding control qubits, and gate complexity for the different gates implemented in Chapter 5.

Using these implementations and the simulators it was possible, in addition to the correctness proofs presented in Chapter 4, to experimentally verify the correctness of the algorithm for small examples. Refer to Figure 6.1 of a comparison of the success probabilities for the example $p = 11$. For this experimental verification not only the expected performance on an actual quantum computer is of interest, but also to a certain degree the performance of each version of the algorithm in the simulator. To obtain a simulation result, Qiskit calculates the state vector for each step in the algorithm (see Section 2.3) and, as explained in Section 2.2, such a vector will have exponential size with respect to the register size. Therefore, when performing simulations, keeping the number of qubits as small as possible is vital for improving the runtime of the simulation, even for small instances.

The experiment results in Figure 6.1 also highlight the role of the order r of the different generators for the success probability of this algorithm. For some values of r, such as for the generators $3, 4, 5$ and 9 with order $r = 5$ in the example, the chance of obtaining the correct result is notably higher than for the others. This fact highlights the role of the factor $\phi(r)/r$ in the formula for the success probability given in Section 4.3.3. It defines a strict upper bound that depends on the order r. This is because even in the

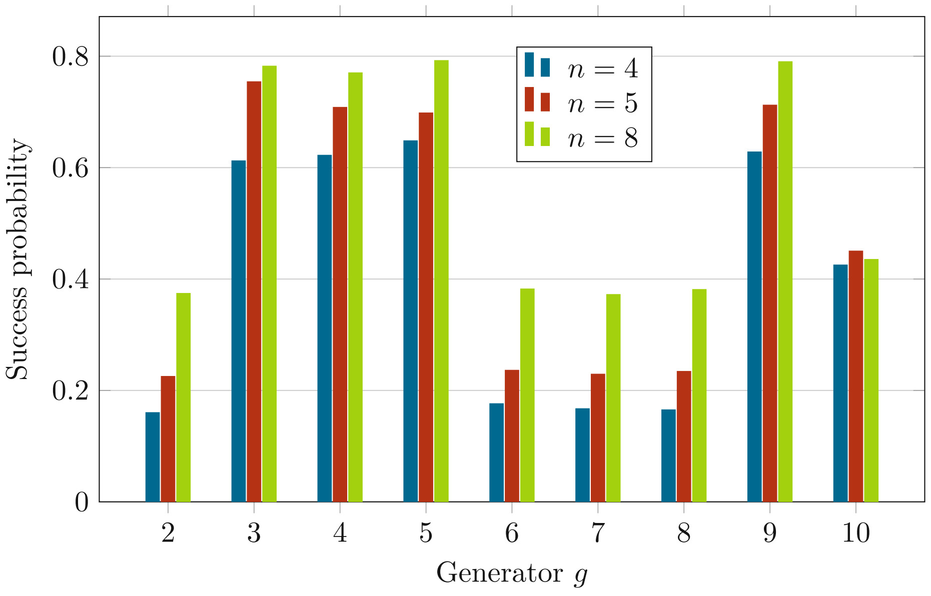


Figure 6.1: Comparison of the average success probability per generator with $p = 11$. The probabilities have been obtained using different top register sizes n.

cases where the phase is estimated exactly, in some trials the returned value $\tilde{k}$ identifying the eigenvector is not invertible modulo r. In these cases the calculation of the discrete logarithm fails.

For some instances, especially those where r is prime, $\phi(r)$ is close to r and as a result $\phi(r)/r$ is close to 1. This is the case for the generators mentioned above where $r = 5$ and $\phi(r)/r = 4/5$. On the contrary, for the generators $2, 6, 7$ and 8, the order is composite with $r = 10$, which results in the significantly lower success probability of at most $4/10$. The remaining generator $g = 10$ is a notable exception, since its order $r = 2$ is prime but the probability seems to be rather low aswell. Although it is prime there is only one possible value of $\tilde{k}$ that leads to a result, which gives a probability of at most $1/2$.

In addition to the comparison of success probabilities for different generators, Figure 6.1 also highlights the improvement of the phase estimation process when a larger top register is used. As mentioned in Section 4.2, the size of this register n influences the quality of the estimate and the figure shows that the larger this register is, the better the chance of successfully obtaining the discrete logarithm. The figure indicates that increasing the size of the top register improves the probability of obtaining the discrete logarithm almost to the described strict upper bound $\phi(r)/r$ for this example.

Using Qiskit it is also easily possible to not only compare the implementations using their asymptotic runtime bounds but also using the number and type of gate operations that are applied after transpilation. One way to get comparable measures for the number of gates used in these circuits would be to transpile them using a real quantum computer as the backend. This forces the transpiler to use exactly the set of elementary gates available on this quantum computer, which would allow for an estimation of the actual number of gates applied. However, using a real device as the backend also prompts the transpiler to adhere to this device's qubit layout (for example the map shown in [26]). This introduces $SWAP$ operations which in turn increase the number of applied $CNOT$ operations. As some algorithms might be better suited for certain qubit layouts this overhead might be different for each circuit and therefore could distort the comparison.

For this reason, another way of obtaining the number of elementary gates is chosen. Qiskit's transpiler is parametrized to use a specific set of elementary gates that are allowed to persist after transpilation. For the comparison in Figure 6.2 and Figure 6.3 the set $\{RZ, I, \sqrt{X}, X\}$ of one-qubit gates together with the two-qubit operation $CNOT$ is chosen. This mimics the gate set of actual devices while still transpiling for the simulator which is not restricted regarding the qubit layout. This way no additional $SWAP$s are introduced.

Of course, the exact gate set that is chosen for transpilation influences the exact number of gate operations in the final circuit. Since the quantum computers that are available through Qiskit usually support the set described above, this set is used for the comparison.

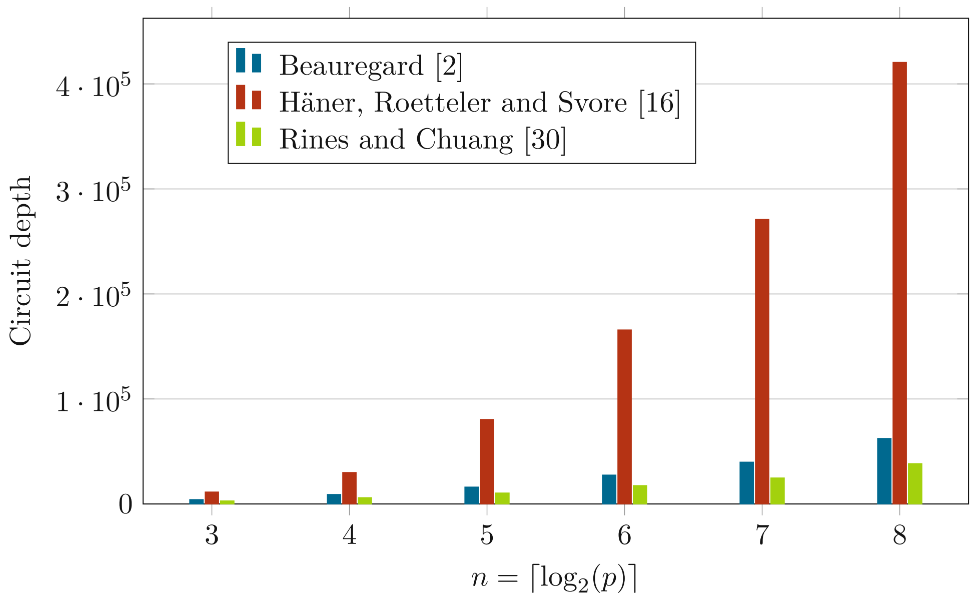


Figure 6.2: Comparison of the circuit depth of the modular exponentiation gates presented in Chapter 5 for different input sizes n.

Figure 6.2 and Figure 6.3 show that although the circuit based on the carry gate and divide and conquer addition [16] seems to be the second best circut regarding the asymptotic gate complexity in Table 6.1, it has to apply the most operations by a wide margin for the small examples considered here. This is due to the fact that the addition gate and the carry gate it is based on are relatively complex when compared to the addition in

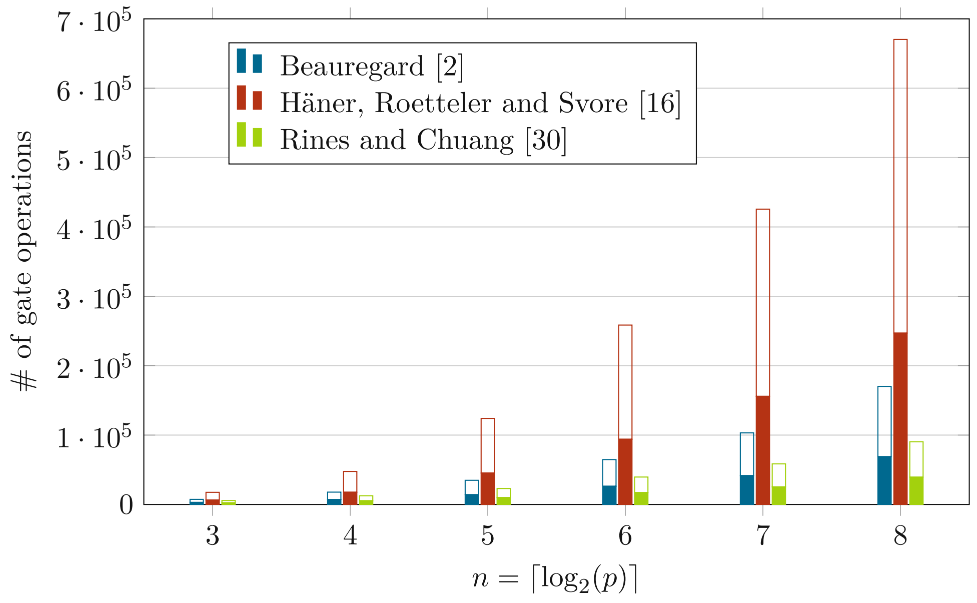


Figure 6.3: Comparison of the total gate count of the modular exponentiation gates presented in Chapter 5 for different input sizes n. The filled part of each bar indicates the number of *CNOT* gates, while the rest represents all remaining one-qubit operations.

Fourier space. Also, the overhead due to the quantum Fourier transform that is required for both other implementations is not substantial for small n.

Furthermore, the figures show that also for these examples the gate based on Montgomery multiplication in Fourier space [30] uses the least amount of elementary gates and therefore should perform best. The drawback of this approach, however, is the increased number of ancilla qubits that are required. This might hinder its usage on a current quantum computer as the number of qubits is available is rather low.

CHAPTER 7

Conclusion

By employing various implementation proposals for the overall algorithm and by reusing ideas for modular exponentiation circuits from works regarding Shor's factoring algorithm, this thesis provides implementations of Shor's algorithm for the discrete logarithm using Qiskit[1]. Furthermore, the algorithm is improved by using a number of optimization techniques found in the literature as well as by utilizing efficient circuits for modular arithmetic.

Using Qiskit's transpiler it was possible to not only compare these implementations regarding their asymptotic runtime bounds but it was also easily possible to obtain the circuit depth and circuit size for small examples.

In summary, the most efficient circuit in this work, measured by its number of qubits, uses $2n+2$ qubits for n-bit modules. This version employs an optimized phase estimation procedure which reduces the number of control qubits for the circuit to only a single qubit. Furthermore, to perform modular exponentiation it uses the carry-based addition and multiplication approach presented by Häner, Roetteler and Svore [16].

However, the most efficient implementation regarding its runtime complexity uses the modular exponentiation circuit by Rines and Chuang [30], which performs a quantum version of Montgomery modular multiplication.

Of course, there are many other classical multiplication and addition algorithms whose implementation on a quantum computer would lead to better runtime bounds than those chosen for this work. But since one goal is to minimize the space complexity to support current quantum computers with a small number of available qubits, the presented algorithms were chosen.

How the proposed circuits behave for small instances of the problem was compared using Qiskit's transpiler. This comparison shows that the circuit with the smallest number of

[1]`https://github.com/mhinkie/ShorDiscreteLog`

qubits, unfortunately, results in the largest number of applied gate operations, which suggests that the other algorithms in this thesis might be better suited for a reference implementation.

In total, from these experiments it was evident that the circuit by Rines and Chuang not only behaves best with respect to its runtime complexity for large n, but also for the small instances that were studied.

Finally, the fact that Shor's algorithm solves the discrete logarithm problem in polynomial time as opposed to superpolynomial time of all known classical algorithms is a groundbreaking result in the field of quantum computing. To investigate this fact, the success probability of the quantum algorithm using simulations for small instances was examined. This work gives a comparison of the probabilities and therefore of the number of repetitions needed to obtain a usable result which also highlights the importance of the size of the measured quantum register for the overall precision.

Although there already are usable quantum computers they are still not powerful enough to run these algorithms for practically meaningful inputs. To execute the algorithm presented here, they also require resources to perform error detection and error correction in addition to the qubits required for the algorithms themselves. However, the implementations presented in this work can be used to find the discrete logarithm for larger modules as soon as the required computing power is available.

Bibliography

[1] A. Baker, *A Comprehensive Course in Number Theory*. Cambridge University Press, 2012. DOI: `10.1017/cbo9781139093835`.

[2] S. Beauregard, Circuit for Shor's algorithm using 2n+3 qubits, *Quantum Information and Computation*, vol. 3, no. 2, pp. 175–185, May 2003. arXiv: `quant-ph/0205095`.

[3] G. Benenti, G. Casati, D. Rossini, and G. Strini, *Principles of Quantum Computation and Information*. WORLD SCIENTIFIC, Jan. 2018. DOI: `10.1142/10909`.

[4] E. Bernstein and U. Vazirani, Quantum Complexity Theory, *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1411–1473, Oct. 1997. DOI: `10.1137/s0097539796300921`.

[5] P. S. Bourdon and H. T. Williams, Sharp Probability Estimates for Shor's Order-Finding Algorithm, *Quantum Information and Computation*, vol. 7, no. 5, pp. 522–550, Jul. 2007, ISSN: 1533-7146. arXiv: `quant-ph/0607148`.

[6] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, Quantum algorithms revisited, *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1969, pp. 339–354, Jan. 1998, ISSN: 1471-2946. DOI: `10.1098/rspa.1998.0164`.

[7] R. Cleve, An Introduction to Quantum Complexity Theory, *Quantum Computation and Quantum Information Theory*, pp. 103–127, Jan. 2001. DOI: `10.1142/9789810248185_0004`.

[8] Conditional Reset on IBM Quantum Systems - IBM Quantum. [Online]. Available: `https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/reset/backend_reset` (visited on 08/10/2021).

[9] A. D. Corcoles, M. Takita, K. Inoue, S. Lekuch, Z. K. Minev, J. M. Chow, and J. M. Gambetta, Exploiting dynamic quantum circuits in a quantum algorithm with superconducting qubits, 2021. arXiv: `2102.01682 [quant-ph]`.

[10] W. Diffie and M. Hellman, New directions in cryptography, *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. DOI: `10.1109/TIT.1976.1055638`.

[11] T. G. Draper, Addition on a Quantum Computer, Aug. 2000. arXiv: quant-ph/0008033.

[12] U. Egly, Quantum phase estimation: The measurement, Course on Quantum Computing 192.070 extra sheet, 2020.

[13] M. Ekerå, Revisiting Shor's quantum algorithm for computing general discrete logarithms, 2021. arXiv: 1905.09084 [cs.CR].

[14] E. Gerjuoy, Shor's factoring algorithm and modern cryptography. An illustration of the capabilities inherent in quantum computers, *American Journal of Physics*, vol. 73, pp. 521–540, 2005. DOI: https://doi.org/10.1119/1.1891170.

[15] R. B. Griffiths and C.-S. Niu, Semiclassical Fourier Transform for Quantum Computation, *Physical Review Letters*, vol. 76, no. 17, pp. 3228–3231, Apr. 1996, ISSN: 1079-7114. DOI: 10.1103/physrevlett.76.3228.

[16] T. Häner, M. Roetteler, and K. M. Svore, Factoring using 2n+2 qubits with Toffoli based modular multiplication, *Quantum Information and Computation*, vol. 17, no. 7 & 8, Jun. 2017. arXiv: 1611.07995 [quant-ph].

[17] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*. Oxford University Press, 1979, ISBN: 9780198531715.

[18] IBM's roadmap for building an open quantum software ecosystem, IBM Research Blog. [Online]. Available: https://research.ibm.com/blog/quantum-development-roadmap (visited on 09/02/2021).

[19] B. S. Kaliski Jr., A Quantum "Magic Box" for the Discrete Logarithm Problem, *Cryptology ePrint Archive, Report 2017/745*, 2017, https://ia.cr/2017/745.

[20] R. Lidl and H. Niederreiter, Algebraic Foundations, in *Introduction to Finite Fields and their Applications*, 2nd ed. Cambridge University Press, 1994, pp. 1–43. DOI: 10.1017/CBO9781139172769.003.

[21] R. Maia and T. Leão, ShorAlgQiskit, GitHub repository, https://github.com/ttlion/ShorAlgQiskit.

[22] P. L. Montgomery, Modular multiplication without trial division, *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[23] M. Mosca, Quantum computer algorithms, PhD thesis, University of Oxford, 1999.

[24] M. Mosca and A. Ekert, The Hidden Subgroup Problem and Eigenvalue Estimation on a Quantum Computer, *Lecture Notes in Computer Science*, vol. 1509, Apr. 1999. DOI: 10.1007/3-540-49208-9_15. arXiv: quant-ph/9903071.

[25] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. DOI: 10.1017/CBO9780511976667.

[26] Overview of ibmq_santiago system - IBM Quantum. [Online]. Available: https://quantum-computing.ibm.com/services?services=systems&system=ibmq_santiago (visited on 08/10/2021).

[27] J. Proos and C. Zalka, Shor's discrete logarithm quantum algorithm for elliptic curves, *Quantum Information and Computation*, vol. 3, no. 4, pp. 317–344, Jan. 2003. arXiv: quant-ph/0301141 [quant-ph].

[28] Qiskit - More Circuit Identities. [Online]. Available: https://qiskit.org/textbook/ch-gates/more-circuit-identities.html (visited on 06/21/2021).

[29] Qiskit Textbook - Learn Quantum Computation using Qiskit. [Online]. Available: https://qiskit.org/textbook/ (visited on 08/09/2021).

[30] R. Rines and I. Chuang, High Performance Quantum Modular Multipliers, Jan. 2018. arXiv: 1801.01081 [quant-ph].

[31] P. W. Shor, Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997, ISSN: 0097-5397. DOI: 10.1137/S0097539795293172.

[32] Simulators — Qiskit 0.29 documentation. [Online]. Available: https://qiskit.org/documentation/tutorials/simulators/1_aer_provider.html (visited on 08/09/2021).

[33] Summary of Quantum Operations — Qiskit 0.29.0 documentation. [Online]. Available: https://qiskit.org/documentation/tutorials/circuits/3_summary_of_quantum_operations.html (visited on 08/13/2021).

[34] Y. Takahashi and N. Kunihiro, A Quantum Circuit for Shor's Factoring Algorithm Using 2n + 2 Qubits, *Quantum Information and Computation*, vol. 6, no. 2, pp. 184–192, Mar. 2006, ISSN: 1533-7146. DOI: 10.26421/QIC6.2-4.

[35] Y. Takahashi, S. Tani, and N. Kunihiro, Quantum Addition Circuits and Unbounded Fan-Out, *Quantum Information and Computation*, vol. 10, no. 9, pp. 872–890, Sep. 2010, ISSN: 1533-7146. arXiv: 0910.2530 [quant-ph].

[36] M. Takita, S. Lekuch, K. Inoue, and A. Corcoles, Quantum circuits get a dynamic upgrade with the help of concurrent classical computation – IBM Research Blog. [Online]. Available: https://www.ibm.com/blogs/research/2021/02/quantum-phase-estimation/ (visited on 08/16/2021).

[37] Transpiler (qiskit.transpiler) — Qiskit 0.29.0 documentation. [Online]. Available: https://qiskit.org/documentation/apidoc/transpiler.html (visited on 08/10/2021).

[38] J. Watrous, Guest column: An Introduction to Quantum Information and Quantum Circuits, *ACM SIGACT News*, vol. 42, no. 2, pp. 52–67, Jun. 2011. DOI: 10.1145/1998037.1998053.

[39] C. Zalka, Fast versions of Shor's quantum factoring algorithm, 1998. arXiv: quant-ph/9806084.