-

Quantum Computing from Introductory Linear Algebra

Major Qualifying Project

Written By:

Em Beeler
Matthew Levine
Gwyneth Ormes
Avery Smith

A Major Qualifying Project
Worcester Polytechnic Institute

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.

May 2021 - March 2022

Authorship

This MQP project report has four authors: Em Beeler, Matthew Levine, Gwyneth Ormes, Avery Smith. All four authors contributed to Chapters 1–5 and hereby attest to equal co-authorship. The report will be submitted twice as a WPI degree requirement. At the end of C Term 2022, Em Beeler and Matthew Levine will submit this as their capstone project for the degree in Mathematical Sciences. At a later date, Gwyneth Ormes and Avery Smith will encorporate this material into a larger report and will submit this as their capstone project for the degrees in Computer Science and Mathematical Sciences.

Abstract

Since its conception in the early 1980s, quantum computing has rapidly grown as a field of study. This growth has placed it in the eye of the public, where it is often seen as inscrutable and restricted to post-graduate studies. Furthermore, the rapid development of physical quantum computers, the existence of algorithms that prove the distinct advantage that quantum computing holds over classical computing, and the threat to current cryptographic schemes has emphasized the need for "quantum computer literacy" now more than ever.

This paper aims to rigorously instruct the reader on the basics of quantum computing with only the assumption of introductory linear algebra—no physics background necessary. A collection of algorithms is discussed, including the Deutsch-Jozsa problem and Shor's algorithm. Alongside each algorithm is an explanation of how to implement and simulate that algorithm using Qiskit.

Contents

1 Introduction **7**

1.1 State of the Art 8

2 The Basics of Quantum Computing **11**

2.1 Notation 11

2.2 Complex Vector Spaces 11

2.3 Qubits 13

2.4 Quantum Registers 14

2.5 Measurement 16

2.6 Single Qubit Gates and Unitary Transformations 21

2.6.1 Not Gate 21

2.6.2 Z Gate 22

2.6.3 Y Gate 22

2.6.4 Hadamard Gate 23

2.6.5 Phase Shift Gate 24

2.7 Multi-Qubit Gates 24

2.7.1 Hadamard Gate on a Quantum Register 26

2.7.2 CNOT Gate 27

2.7.3 SWAP Gate 28

2.7.4 Controlled Gates 28

2.8 Entanglement 29

3 An Introduction to Qiskit **33**

3.1 Installation 33

3.2 Building Simple Circuits 39

4 Early Algorithms 44
4.1 The Hidden Subgroup Problem . . . 44
4.1.1 Problem Reductions . . . 45
4.2 Deutsch-Jozsa . . . 45
4.2.1 The Problem . . . 46
4.2.2 The Algorithm . . . 46
4.2.3 Qiskit . . . 49
4.3 Bernstein-Vazirani . . . 51
4.3.1 The Problem . . . 52
4.3.2 The Algorithm . . . 52
4.3.3 Qiskit . . . 54
4.4 Simon . . . 56
4.4.1 The Problem . . . 56
4.4.2 The Algorithm . . . 56
4.4.3 Query Complexity . . . 58
4.4.4 Qiskit . . . 60

5 Shor's Algorithm 66
5.1 The Problem . . . 66
5.2 Discrete Fourier Transform . . . 67
5.3 Quantum Fourier Transform . . . 67
5.4 Phase Estimation . . . 69
5.5 The Algorithm . . . 71
5.5.1 Classical Component . . . 72
5.5.2 Quantum Component . . . 75
5.5.3 Runtime Analysis . . . 77
5.6 Qiskit . . . 78
5.7 The Discrete Logarithm Problem . . . 82
5.7.1 Baby-Step Giant-Step . . . 83
5.7.2 Overview of Shor's Algorithm for Discrete Logs . . . 85
5.7.3 Quantum Component . . . 85
5.7.4 Classical Component . . . 86
5.7.5 Analysis . . . 87

A Glossary of Definitions 89

Bibliography 93

Chapter 1

Introduction

Scientists, engineers, and mathematicians alike have all striven for more powerful computers. Tasks such as simulation of a physical system, searching through a large list, or factoring a large semiprime number are all inefficient or even impossible to do by hand. Computers have been used to complete these arduous tasks. As our problems grow more and more complicated, our computers have grown more and more powerful. However, even a classical computer has its limits: a broad range of tasks of practical importance are widely believed to require exponential time on a Turing machine.

Seeking to circumvent these limitations, the quantum computer was proposed at three separate times by three separate notable researchers. The first was Paul Benioff, in 1979, who suggested that such a machine could be built. A year later, Yuri Manin, a mathematician, proposed the idea in his book *Computable and Non-Computable*, which was only available in Russian at the time. In 1981, Richard Feynman in one of his seminal lectures put forward the idea of a quantum computer as a device better suited for simulating quantum systems. [1].

The quantum computer was only a hypothetical device at that point, and yet it had captured the attention of many. This theoretical device would utilize the properties of quantum states, including superposition and entanglement. Such a device would be capable of performing tasks in less time than the classical computer—not due to its inherent power, but rather the utilization of these quantum properties in algorithms run on it. The fundamental idea of superposition of quantum states gives the impression of a data input that represents all possible values simultaneously—we will soon make this precise—opening the door to some sort of massive parallelization of algorithms.

It would be almost 20 years before Isaac L. Chuang, Neil Gershenfeld, and Mark Kubinec implemented the first publicly known physical quantum computer, with two hard-won qubits and no error correcting [2]. Since then, there has been significant progress in the development of a physical quantum computer. However,

as with any new technology, significant progress comes with significant setbacks. A handful of companies have risen to the challenge of implementing a scalable, error-correcting, useful quantum computer.

1.1 State of the Art

As with any new technology, early implementations usually provide proof of concept. The quantum computer created in 1998 [2] implemented a powerful algorithm of Grover that locates an item in a list far faster than exhaustive search would do, but that implementation handled a list of size four! The question that naturally follows is: how can we make quantum computers bigger, and therefore more useful?

By nature, quantum computers and the way they implement physical qubits are sensitive and prone to disruption via noise. For our physical qubits to even begin behaving like the abstract notion of the logical qubit, most known versions of the quantum computer must be kept at near-absolute zero temperatures [3].

Furthermore, it is necessary to utilize quantum error correction to prevent errors that nonetheless arise. The concept of "logical qubit" versus "physical qubit" is derived from this concept. Many implementations of quantum error correction have come forward. Peter Shor demonstrated one code that utilized 9 physical qubits to error correct 1 logical qubit, while Andrew Steane reduced the requirement to 7 physical qubits. Raymond LaFlamme showed that 5 physical qubits for 1 logical qubit is the smallest number that fulfills the requirements [4]. While the theory of quantum error correction does offer an economy of scale—for instance, ten physical qubits suffice to protect a system of four logical qubits against errors affecting any single qubit [5]—the number of logical qubits that can be relied upon for computational use is typically much smaller than the number of physical qubits. As the number of logical qubits thus depends on the specifics of the error correcting code, the quantum computers below are introduced with their number of physical qubits.

The rapid growth and scaling of quantum computers becomes all the more awe-inspiring with this knowledge. More and more companies and universities are rising to the challenge of creating a quantum computer that can outperform a classical one; to do so requires more qubits than the first 2-qubit machine from 1998.

In November 2021, IBM announced its 127-qubit processor, Eagle. The company also announced its plans for a 433-qubit processor, Osprey, to be released in 2022, and a 1,121-qubit processor, Condor, to be released in 2023 [6].

Companies that develop quantum computers often wish to demonstrate their superiority over leading classical supercomputers. However, physical implementations (as of the writing of this report) are incapable of tasks that would easily demonstrate this, such as factoring large semiprimes. Sampling distributions from randomly-chosen quantum circuits, known as random circuit sampling (RCS), has become a popular means to demonstrate "quantum supremacy". While RCS has not been proven to be classically hard, strong

evidence has been given by Bouland, Fefferman, Nirkhe, and Vazirani [7]. RCS remains the leading choice for demonstrating the quantum advantage.

The University of Science and Technology of China, in June 2021, also announced the completion of a 66-qubit processor, named Zuchongzhi. In the paper by Wu et al., they estimate that Zuchongzhi could complete a RCS task 58,400 times faster than IBM's Summit, the most powerful classical supercomputer at the time [8].

Google has also stepped into the quantum race. In 2021, Google's Quantum AI team published a paper claiming that their 54-qubit processor Sycamore firmly demonstrated "quantum supremacy" by outperforming a classical supercomputer. The paper claimed that Sycamore had completed a RCS task in 200 seconds that IBM's Summit would take 10,000 years to complete [9]. IBM, however, contested this claim, stating that Summit would only take 2.5 days to complete the task [10]. The question of "quantum supremacy" was once more unanswered.

Comparatively, smaller groups have created their own quantum processors, such as Starmon-5, a 5-qubit processor from QuTech, of the Netherlands' Delft University of Technology [11], and IonQ of Maryland's 11-qubit processor [12].

We see that quantum computing has rapidly gone from a hypothetical speedup to a very real development that promises immense speedups in simulation and computation, as well as threatens cryptography as we know it. Literacy in this topic is more imperative than ever. Perhaps surprisingly, once a student has studied the basics of linear algebra, they are prepared to study and understand many of the fundamental quantum algorithms, and prepare themselves for the future of computing.

The technological side of quantum computing—the application of physics and engineering to model the processes described only abstractly in this report—advances at a rapid pace. New announcements are made every month. To the best of our knowledge, the following table presents a list of all important physical quantum computers built in recent years.

Company	**Computer Name**	**Year**	**Physical Qubits**	**Computer Type**	
IBM	Eagle	2021	127	Superconducting	Source
Atom Computing	Pheonix	2021	100	Nuclear Spin	Source
UST China	Zuchongzhi	2021	66	Superconducting	Source
IBM	Hummingbird	2021	65	Superconducting	Source
Rigetti	Aspen-10	2021	32	Superconducting	Source
Rigetti	Aspen-9	2021	32	Superconducting	Source
Xanadu	X8	2020	-	Photonic	Source
ColdQuanta	Hilbert (Prototype)	2020	64	Cold Atom	Source
IonQ	-	2020	32	Trapped Ion	Source
Rigetti	Aspen-8	2020	31	Superconducting	Source
IBM	Falcon	2020	27	Superconducting	Source
Honeywell	System Model H1	2020	10	Trapped Ion	Source
QuTech	Starmon-5	2020	5	Superconducting	Source
Qutech	Spin-2	2020	2	Spin	Source
Rigetti	Aspen-7	2019	28	Superconducting	Source
Rigetti	Aspen-4	2019	13	Superconducting	Source
IonQ	-	2019	11	Trapped Ion	Source
Google	Sycamore	2018	56	Superconducting	Source
Rigetti	Aspen-1	2018	16	Superconducting	Source
Alibaba	-	2018	11	Superconducting	Source
Google	Bristlecone	2017	72	Superconducting	Source
Rigetti	Acorn	2017	19	Superconducting	Source
IBM	Canary	2017	16	Superconducting	Source
Rigetti	Agave	2017	8	Superconducting	Source
Raytheon	-	2017	5	Superconducting	Source
Google	Foxtail	2016	22	Superconducting	Source

Table 1.1: An incomplete list of announced, developed quantum computers.

Chapter 2

The Basics of Quantum Computing

2.1 Notation

In this paper, we will use Greek letters such as φ ("phi") for vectors. Column vectors will be written as "kets", $|\varphi\rangle$ ("ket phi") and the conjugate transpose of this is the row vector, or "bra", $\langle\varphi|$ ("bra phi"). The dimension of a complex vector space will often be denoted by N.

Any square $N \times N$ matrix A represents a linear transformation $\mathbb{C}^N \to \mathbb{C}^N$ with respect to a specified basis. For a vector φ, instead of writing $A\varphi$, we write $A|\varphi\rangle$ ("A ket phi").

We use † ("dagger") to represent the conjugate transpose: if A is an $M \times N$ matrix with entries a_{ij} and $B = A^\dagger$, then B is an $N \times M$ matrix with entries $b_{ij} = \overline{a_{ji}}$. The bar over $\overline{a_{ij}}$ denotes a scalar complex conjugate.

The correspondence between symbols familiar to linear algebra and the notation we will use includes

φ	$\varphi^\dagger$	$A\varphi$	$A^\dagger$	$\varphi^\dagger A$	$\varphi^\dagger A^\dagger$	$\varphi^\dagger A\psi$
$|\varphi\rangle$	$\langle\varphi|$	$A|\varphi\rangle$	$A^\dagger$	$\langle\varphi| A$	$\langle A\varphi|$	$\langle\varphi| A |\psi\rangle$

In quantum computing, we apply operations represented by unitary matrices with complex coefficients. A unitary $N \times N$ matrix U has the property $U^\dagger = U^{-1}$, and consequently, $UU^\dagger = I_N$.

2.2 Complex Vector Spaces

Everything we do with regard to quantum computing happens in complex vector spaces. This section gives a short overview of relevant terms that will often be seen throughout the text. It can be skipped by those

very familiar with the concepts. For ease of understanding, clickable references are included throughout the text which points back to these definitions in the glossary.

The complex counterpart of $\mathbb{R}^N$ is $\mathbb{C}^N$, consisting of ordered N-tuples of complex numbers. So a vector in $\mathbb{C}^N$ is in the form $|\varphi\rangle = (a_1 + b_1 i,\ a_2 + b_2 i,\ \ldots,\ a_N + b_N i)$ with $a_1, b_1, a_2, b_2, \ldots, a_N, b_N \in \mathbb{R}$. A more common representation of vectors in $\mathbb{C}^N$ is by $N \times 1$ matrices:

$$|\varphi\rangle = \begin{bmatrix} a_1 + b_1 i \\ a_2 + b_2 i \\ \vdots \\ a_N + b_N i \end{bmatrix}.$$

A *complex vector space* is a set V of *vectors* forming an abelian group under vector addition which is closed under multiplication by complex scalars in such a way that the following identities hold for all $|\varphi\rangle, |\psi\rangle \in V$ and all $c, d \in \mathbb{C}$:

- $(c + d)|\varphi\rangle = c|\varphi\rangle + d|\varphi\rangle$
- $c(|\varphi\rangle + |\psi\rangle) = c|\varphi\rangle + c|\psi\rangle$
- $(cd)|\varphi\rangle = c(d|\varphi\rangle)$
- $1|\varphi\rangle = |\varphi\rangle$

If $|\varphi_1\rangle$, $|\varphi_2\rangle$, $\ldots$, $|\varphi_m\rangle$ are vectors in a complex vector space and $c_1, c_2, \ldots, c_m$ are scalars in $\mathbb{C}$, then the sum $c_1|\varphi_1\rangle + c_2|\varphi_2\rangle + \cdots + c_m|\varphi_m\rangle$ is called a *linear combination* of $|\varphi_1\rangle, \ldots, |\varphi_m\rangle$. Because V is closed under vector addition and scalar multiplication, a linear combination of vectors in V is also a vector in V. As with vectors in the real vector space, addition and scalar multiplication in $\mathbb{C}^N$ are performed component by component.

When we have a set of vectors $S = \{|\varphi_1\rangle, |\varphi_2\rangle, \ldots, |\varphi_m\rangle\}$, the *span* of S is the set of all linear combinations $c_1|\varphi_1\rangle + c_2|\varphi_2\rangle + \cdots + c_m|\varphi_m\rangle$ with $c_1, c_2, \ldots, c_m \in \mathbb{C}$. A set of vectors S is *linearly independent* if $c_1|\varphi_1\rangle + c_2|\varphi_2\rangle + \cdots + c_m|\varphi_m\rangle = |0\rangle$ is true only when $c_1 = c_2 = \cdots = c_m = 0$. If there are multiple solutions to $c_1|\varphi_1\rangle + c_2|\varphi_2\rangle + \cdots + c_m|\varphi_m\rangle = |0\rangle$, then S is *linearly dependent*.

A *basis* of a vector space V is set of vectors $\mathcal{B} \subseteq V$ which is linearly independent and whose span is V. In this case, $\mathcal{B}$ has the following properties.

- $\mathcal{B}$ is a maximal linearly independent set. Adding any vector in V would cause the set to become linearly dependent. The set is maximal when there does not exist a linearly independent set S where $\mathcal{B} \subset S$

other than $S = \mathcal{B}$.

- $\mathcal{B}$ is a minimal spanning set. The span of $\mathcal{B}$ equals V but there does not exist a set S whose span equals V and $S \subset \mathcal{B}$ other than $S = \mathcal{B}$.

In the case where V has a finite spanning set, achieving one of these properties guarantees the other by a corollary to the Replacement Theorem [13, p. 45]. In the case where a vector space V has a finite spanning set, the *dimension* of V is the cardinality of any basis of V. In fact, if V has a finite basis, all bases of V contain the same number of vectors, also by that same corollary to the Replacement Theorem. Additionally, any vector in V is uniquely expressed as a linear combination of the basis vectors in basis $\mathcal{B}$.

Two vectors $|\varphi\rangle$ and $|\psi\rangle$ are *orthogonal* if their *Hermitian inner product* is zero. We use a *norm* function to determine the length of a vector. Norm functions always yield non-negative real "lengths" and obey three properties: they respect scalars, they satisfy the triangle inequality, and they output 0 only when the input is the zero vector. The norm function we will use is the Euclidean or 2-norm: $||\psi\rangle| = \sqrt{\langle\psi|\psi\rangle}$. A *unit vector* is a vector which has norm 1. We can convert any vector, $|\psi\rangle$, into a unit vector, $\frac{|\psi\rangle}{||\psi\rangle|}$, by dividing each of the vector components by its norm. An *orthonormal basis* is a basis whose vectors are all unit vectors and are all orthogonal to each other.

2.3 Qubits

In a classical system, a bit is in one of two states, 0 or 1. A *qubit* is the quantum equivalent of a classical bit. Instead of 0 and 1, a qubit is typically in *superposition*, and this is modeled as a linear combination of two basis states of qubits. We first look at the simplest basis, and we call its vectors $|0\rangle$ and $|1\rangle$:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The vectors $|0\rangle$ and $|1\rangle$ form the *computational basis* for $\mathbb{C}^2$. To represent a qubit in superposition, we interpret its coordinates with respect to a basis as "probability amplitudes" as we will shortly see.

A *qubit in state* $|\varphi\rangle$ is a two-dimensional complex vector space V with orthonormal basis $\{|0\rangle, |1\rangle\}$ together with a unit vector $|\varphi\rangle \in V$. This unit vector $|\varphi\rangle$ is expressed uniquely as a linear combination with respect to the basis:

$$|\varphi\rangle = c_1 |0\rangle + c_2 |1\rangle .$$

Since $|\varphi\rangle$ is a unit vector, we have $\langle\varphi|\varphi\rangle = 1$, that is,

$$c_1\overline{c_1} + c_2\overline{c_2} = 1.$$

Let $|\varphi\rangle$ be a state of a qubit and let (c_1, c_2) be its coordinate vector with respect to the computational basis. The scalar entries of the coordinate vector are called the *probability amplitudes* of the qubit.

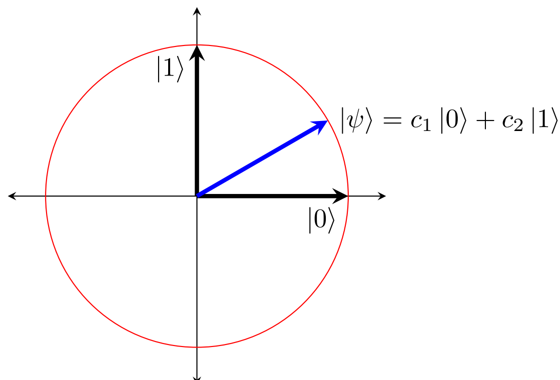


Figure 2.1: A visual of the unit circle on a slice of $\mathbb{C}^2$ with a real $|\psi\rangle$. A qubit is a vector on this unit circle.

2.4 Quantum Registers

An *n-qubit quantum register* is a 2^n-dimensional complex vector space with a prescribed orthonormal basis. The full vector space is a tensor product of n qubits:

$$\mathbb{C}^{2^n} \cong \mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2.$$

The most important basis for our registers is the *computational basis*

$$\mathcal{B} = \{|b\rangle \ | \ b \in \mathbb{Z}_2^n\}$$

which, when convenient, we will write as $\mathcal{B} = \{|b\rangle \ | \ 0 \leq b < 2^n\}$ interpreting bit strings as binary expansions of integers. For example, the 2-qubit register has computational basis,

$$\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}.$$

The computational basis is a nice basis to work with. Each element is a *Kronecker product* of the

single-qubit computational basis states:

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$|11\rangle = |1\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Similarly,

$$|111\rangle = |1\rangle \otimes |1\rangle \otimes |1\rangle$$

$$|011010\rangle = |0\rangle \otimes |1\rangle \otimes |1\rangle \otimes |0\rangle \otimes |1\rangle \otimes |0\rangle .$$

Often we will omit the $\otimes$ symbol and write $|x\rangle\,|y\rangle = |x\rangle \otimes |y\rangle$.

A *quantum register in state* $|\psi\rangle$ is an ordered pair $(V, |\psi\rangle)$ where V is a quantum register and $|\psi\rangle$ is a unit vector in V. We represent $|\psi\rangle$ as a linear combination of the elements of any basis:

$$|\psi\rangle = \sum_{|\phi_i\rangle \in \mathcal{B}} c_i \, |\phi_i\rangle .$$

Example: Using the computational basis for V and

$$|\psi\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

our state $|\psi\rangle$ can be represented as the linear combination:

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

2.5 Measurement

Measurements are how we get information from qubits when they are in superposition. We cannot directly look at any individual probability amplitude of a qubit. Instead, we must collect information differently than standard computations. We note that while the following paragraphs describe a measurement as a process playing out over time, this is merely for pedagogical purposes: we work only with the linear-algebraic model and make no claims about what happens in "reality".

To execute a measurement on a quantum register V, we define an orthogonal decomposition of V,

$$\mathcal{M} = \{S_1, S_2, \ldots, S_k\}$$

where S_i is a subspace of V for $1 \leq i \leq k$, and each subspace is orthogonal to each other, that is,

$$V = S_1 \perp S_2 \perp \cdots \perp S_k.$$

Now suppose we wish to measure V when it is in state $|\psi\rangle$. We represent $|\psi\rangle$ as a unit vector linear combination of elements from each of the subspaces of $\mathcal{M}$

$$|\psi\rangle = c_1 |\phi_1\rangle + c_2 |\phi_2\rangle + \cdots + c_k |\phi_k\rangle$$

with conditions

$$|\phi_i\rangle \in S_i, \quad \langle\phi_i|\phi_i\rangle = 1, \quad c_1\overline{c_1} + c_2\overline{c_2} + \cdots + c_k\overline{c_k} = 1.$$

We then measure the state against this decomposition. During measurement, we may imagine in our model three events occurring:

- An index $j \in \{1, 2, \ldots, k\}$ is chosen with probability $c_j\overline{c_j}$.
- The index j is returned as classical information.
- The quantum state collapses to $|\phi_j\rangle$.

The **only** information gained is the index j. At no point are we able to look at the probability amplitudes of the state $|\psi\rangle$. Additionally, if $|\psi\rangle \neq |\phi_j\rangle$, then we have permanently destroyed our previous state! Measuring again will continue to return $|\phi_j\rangle$ with probability 1.

Given a n-qubit quantum register $(V, |\psi\rangle)$, we can measure $|\psi\rangle$ using $\mathcal{M} = \{S_1, S_2, \ldots, S_k\}$. We represent $|\psi\rangle$ as a linear combination of unit length elements from the respective subspaces

$$|\psi\rangle = c_1 |\phi_1\rangle + c_2 |\phi_2\rangle + \cdots + c_k |\phi_k\rangle$$

and j is returned to us. We now have $|\psi\rangle$ replaced by $|\phi_j\rangle$. If we measure again using the same decomposition of V we have $|\psi\rangle$ represented as

$$|\psi\rangle = 0 |\phi_1\rangle + 0 |\phi_2\rangle + \cdots + 1 |\phi_j\rangle + \cdots + 0 |\phi_k\rangle .$$

Thus, the probability of returning j again is 1.

Example: Given a 2-qubit quantum register $(V, |\psi\rangle)$, and $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, measure $|\psi\rangle$. Choose $\mathcal{M}$ to be composed of subspaces spanned by the computational basis vectors:

$$\mathcal{M} = \{S_1, S_2, S_3, S_4\}$$

$$S_1 = \text{span}\{|00\rangle\}, \quad S_2 = \text{span}\{|01\rangle\}$$

$$S_3 = \text{span}\{|10\rangle\}, \quad S_4 = \text{span}\{|11\rangle\}.$$

Then we write $|\psi\rangle$ just as it is already written:

$$|\psi\rangle = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle$$

with $|00\rangle \in S_1$ and $|11\rangle \in S_4$. The probability of being given index 1 is $\left|\frac{1}{\sqrt{2}}\right|^2$:

$$\Pr(|\psi\rangle = |00\rangle) = \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}.$$

The probability of being given index 4 is the same. The remaining two probabilities are zero:

$$\Pr(|\psi\rangle = |11\rangle) = \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}.$$

After measurement, **either**

$|\psi\rangle$ will be $|00\rangle$ with 50% chance, and we are returned the index 1

or

$|\psi\rangle$ will be $|11\rangle$ with 50% chance, and we are returned the index 4.

While we had full freedom in choosing the measurement, we have no control over which of these two outcomes arises.

This act of using another basis to generate 1-dimensional subspaces is very common, so we will define it as *measuring in the specified basis.* In general, *measuring in basis* $\mathcal{B}$ is a decomposition of V into 1-dimensional subspaces created by the span of elements in $\mathcal{B}$:

$$\mathcal{M} = \{\text{span}\{|\phi\rangle\} \mid |\phi\rangle \in \mathcal{B}\}.$$

Basis $\mathcal{B}$ has 2^n elements to match the dimension of V. For the example measurement, we measured in the computational basis. Measuring in the computational basis is called the *standard measurement.* Also of importance is the fact that the number of subspaces k was equal to the dimension of the 2-qubit register. This means we did a *complete measurement*, as we measured every qubit.

Example: Given a 2-qubit quantum register $(V, |\psi\rangle)$, and $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, now let our chosen decomposition be $\mathcal{M} = \{S_1, S_2\}$ with S_1 and S_2 as

$$S_1 = \text{span}\{|00\rangle, |01\rangle\}$$
$$S_2 = \text{span}\{|10\rangle, |11\rangle\}.$$

We again write $|\psi\rangle$ just as it is already written,

$$|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle.$$

After measurement, **either**

$|\psi\rangle$ will be $|00\rangle$ with 50% chance, and we are returned the index 1

or

$|\psi\rangle$ will be $|11\rangle$ with 50% chance, and we are returned the index 2.

The results of measuring will be equivalent to the first example, but instead, this measurement can only return an index of 1 or 2, instead of the original indices of 1 and 4. The probabilities of returning index 1 and index 2 are also the same, at 50%.

Since the number of subspaces is 2 and the dimension of our quantum register is 4, we do not have a complete measurement. This is a *partial measurement.* We also notice that the first qubit is always 0 in S_1 and always 1 in S_2. Specifically, what we have just done is *measure the first qubit.* We interest ourselves only with the result of the first qubit. If the index returned is 1, we know the first qubit is in state 0; if the index returned is 2, we know the first qubit is in state 1.

Assume the index returned was 1. Then $|\psi\rangle$ is now in the state $|00\rangle$. Even though we only measured the first qubit, the second qubit was still affected. This is because the qubits were *entangled.* The definition of entanglement is covered in §2.8.

We could instead measure the second qubit by fixing the second qubit in each of the subspaces of $\mathcal{M}$:

$$S_1 = \text{span}\{|00\rangle, |10\rangle\}$$
$$S_2 = \text{span}\{|01\rangle, |11\rangle\}.$$

Partial measurements allow for the observer to be specific about which qubits they want to know the results of without destroying other information encoded in the state. For example, if we measured state

$$|\psi\rangle = \left(\frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right) = \frac{3}{5\sqrt{2}}|00\rangle + \frac{4}{5\sqrt{2}}|10\rangle - \frac{3}{5\sqrt{2}}|01\rangle - \frac{4}{5\sqrt{2}}|11\rangle$$

in this basis $\mathcal{M} = \{S_1, S_2\}$, the reader may check that, with 50% chance, the measurement will return 1 and $|\psi\rangle$ will collapse to $\frac{3}{5}|00\rangle + \frac{4}{5}|10\rangle$ and, with 50% chance, the measurement will return index 2 and $|\psi\rangle$ will collapse to $-\frac{3}{5}|01\rangle - \frac{4}{5}|11\rangle$. This is what we mean by saying that information in the first qubit is not destroyed.

Example: Now, let's measure $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ in the basis

$$\mathcal{B} = \left\{|\Phi^+\rangle, |\Phi^-\rangle, |\Psi^+\rangle, |\Psi^-\rangle\right\}$$

where

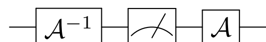
$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$
$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$
$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$
$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle).$$

This basis is called the *Bell basis*. Measuring $|\psi\rangle$ in $\mathcal{B}$ results in drawing a $|\Phi^+\rangle$ with probability 1, returning an index of 1, and the state remaining unchanged.

Now there is something odd going on here—the state collapsed in the first example, but not here. When we compute a non-standard measurement, we must perform a change of basis before and after measuring. We take the vectors of the Bell basis and place them into a matrix:

$$\mathcal{A} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \\ 1 & -1 & 0 & 0 \end{bmatrix}. \tag{2.1}$$

$\mathcal{A}$ is the change of basis matrix. Applying $\mathcal{A}^{-1}$ to our state vector before measuring, taking the standard measurement, then applying $\mathcal{A}$ will give us the desired results. In a circuit diagram, this looks like



where $\mathcal{A}$ is our unitary basis transformation, the "meter" is the standard measurement in V, and $\mathcal{A}^{-1}$ is the reverse computation of $\mathcal{A}$. Note that $\mathcal{A}$ is unitary, so the inverse is easily computed as $\mathcal{A}^{\dagger}$:

$$\mathcal{A}^{-1} |\psi\rangle = \frac{1}{\sqrt{2}} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Measuring with the computational basis returns the index 1 with probability 1 and leaves the register in state $|00\rangle$. Reapplying the change of basis yields

$$\mathcal{A}\,|00\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

and our register is left in state $|\Phi^{+}\rangle$.

This conjugation technique is what allows us to consider more general measurements $\mathcal{M}$. While we do not have full freedom to choose $S_1, \ldots, S_k$, we will gloss over this efficiency issue and restrict ourselves to fairly standard measurements.

Exercise:

1. Given a 2-qubit register $(V, |\psi\rangle)$, where V is the computational basis, and $|\psi\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$:

   (a) perform a complete measurement in the computational basis.

   (b) perform a complete measurement in the Bell basis.

   (c) perform a partial measurement on the first qubit.

      i. Did this measurement affect the second qubit?

2. Given a 3-qubit register $(V, |\psi\rangle)$, where V is the computational basis, and $|\psi\rangle = \frac{1}{\sqrt{3}}(|000\rangle + |011\rangle + |101\rangle)$:

   (a) perform a complete measurement in the computational basis.

   (b) perform a partial measurement on the second qubit.

2.6 Single Qubit Gates and Unitary Transformations

A *quantum logic gate* (or *quantum gate*) is a simple quantum circuit operating on a small number of qubits. These gates can be used to build complex quantum circuits, much like how classical logic gates form conventional digital circuits. Quantum gates are unitary operators described as unitary matrices relative to a given basis. The NOT gate, Z gate, and Hadamard gate are some of the standard single-bit gates used in quantum circuits.

2.6.1 Not Gate

The NOT gate or X gate is represented by the unitary matrix

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and is written in wire diagrams as $-\boxed{X}-$.

To see the effect a gate will have on a qubit, we multiply the qubit's state vector on the left by the matrix representing that gate. Let's take a look at how the NOT gate affects a qubit in state $|0\rangle$:

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

We can see that the NOT gate flips a qubit in state $|0\rangle$ to state $|1\rangle$. Now let's look at applying a NOT gate to $|1\rangle$:

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

So for a NOT gate,

$$X|0\rangle = |1\rangle \quad \text{and} \quad X|1\rangle = |0\rangle .$$

Consider a state other than a standard basis element:

$$|\psi\rangle = c_1 |0\rangle + c_2 |1\rangle .$$

So,

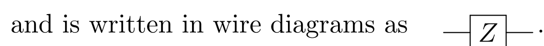
$$X|\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} c_2 \\ c_1 \end{bmatrix} = c_2 |0\rangle + c_1 |1\rangle .$$

The NOT gate swaps the probability amplitudes of $|0\rangle$ and $|1\rangle$.

2.6.2 Z Gate

The Z gate is represented by the unitary matrix

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

and is written in wire diagrams as  .

Let's apply a Z gate to both $|0\rangle$ and $|1\rangle$:

$$Z|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1+0 \\ 0+0 \end{bmatrix} = |0\rangle$$
$$Z|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0+0 \\ 0-1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} = -|1\rangle .$$

The Z gate did nothing to $|0\rangle$, but it changed the sign of $|1\rangle$. This corresponds to a reflection across the x-axis in our coordinate vector model of Figure 2.1.

2.6.3 Y Gate

It may seem strange to approach the gates outside of alphabetic order, but there is a reason for this. The Y gate is represented by the unitary matrix

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

and is written in wire diagrams as $-\boxed{Y}-$. We can represent the Y gate as a combination of the X and Z gates and a scalar multiplication by i:

$$Y = iXZ = i\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

The Y gate combines the concepts of the Z gate (reflecting across an axis) and the X gate (swapping the probability amplitudes) with a scalar multiplication by i. This interacts with our system:

$$Y|0\rangle = \begin{bmatrix} 0 \\ i \end{bmatrix} = i|1\rangle$$
$$Y|1\rangle = \begin{bmatrix} -i \\ 0 \end{bmatrix} = -i|0\rangle.$$

When we take $|Y|0\rangle|^2$, we get $i \cdot (-i) = 1$. Similarly, $|Y|1\rangle|^2 = -i \cdot (i) = 1$. Scaling a vector by a complex number of absolute value one does not change its length.

2.6.4 Hadamard Gate

A Hadamard gate is represented by the unitary matrix

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

and is written in wire diagrams as $-\boxed{H}-$.

Let's apply the Hadamard gate to $|0\rangle$:

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1+0 \\ 1+0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

We have just created a qubit in an equal superposition of $|0\rangle$ and $|1\rangle$. This specific vector is common, so have a special notation for it. We write it as

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Now let's apply a Hadamard gate to $|1\rangle$:

$$H|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0+1 \\ 0-1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

This yielded almost the same vector. We again have a special notation for this vector:

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

When applying a Hadamard gate to $|0\rangle$, the resulting state is $|+\rangle$. A Hadamard gate applied to $|1\rangle$ results in the state $|-\rangle$.

2.6.5 Phase Shift Gate

The phase shift gate is represented by the matrix

$$P(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}.$$

The symbol of the phase gate in a circuit diagram depends on the author. Some represent it with an 'R' (—$\boxed{R}$—), some with a 'P' (—$\boxed{P}$—), and some with the angle of rotation divided by 2 (—$\boxed{\theta/2}$—). The angular division representation means that a phase shift gate with $\theta = \frac{\pi}{4}$ is called $\frac{\pi}{8}$.

The phase shift gate leaves the probability amplitude of $|0\rangle$ as is, but changes the probability amplitude of $|1\rangle$. To show that this transformation preserves the unit vector condition of our qubit, we will use Euler's formula, $e^{i\theta} = \cos\theta + i\sin\theta$, and the Pythagorean identity, $\cos^2\theta + \sin^2\theta = 1$:

$$e^{i\theta}\overline{e^{i\theta}} = (\cos\theta + i\sin\theta)(\cos\theta - i\sin\theta) = \cos^2\theta + \sin^2\theta = 1.$$

Since the phase shift only affects $|1\rangle$, the probability of drawing a $|0\rangle$ or $|1\rangle$ after applying a phase shift gate is preserved.

2.7 Multi-Qubit Gates

The gates we have looked at so far only work on a single qubit. We need a way to extend them to n-qubit registers. One cannot apply a regular Hadamard gate to a 2-qubit register, since the dimensions do not match! The solution to this problem is to use the same technique as for creating multi-qubit registers: apply Kronecker products to our unitary transformations.

Our first gate was the NOT gate:

$$X|0\rangle = |1\rangle \quad \text{and} \quad |1\rangle = |0\rangle .$$

If we apply it to the second qubit of the computational basis states of the 3-qubit register, we get:

$$I_2 \otimes X \otimes I_2 |000\rangle = |010\rangle \qquad I_2 \otimes X \otimes I_2 |001\rangle = |011\rangle$$
$$I_2 \otimes X \otimes I_2 |010\rangle = |000\rangle \qquad I_2 \otimes X \otimes I_2 |011\rangle = |001\rangle$$
$$I_2 \otimes X \otimes I_2 |100\rangle = |110\rangle \qquad I_2 \otimes X \otimes I_2 |101\rangle = |111\rangle$$
$$I_2 \otimes X \otimes I_2 |110\rangle = |100\rangle \qquad I_2 \otimes X \otimes I_2 |111\rangle = |101\rangle$$

Thus, we can construct the full matrix transform for applying a NOT gate to just the second qubit:

$$I_2 \otimes X \otimes I_2 = \left[\begin{array}{cccc|cccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right].$$

Likewise, we can apply the Z gate to just the third qubit:

$$I_4 \otimes Z = \left[\begin{array}{cc|cc|cc|cc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{array}\right].$$

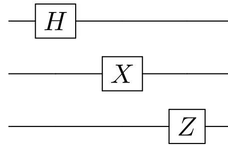
Lastly, apply the Hadamard gate to just the first qubit:

$$H \otimes I_4 = \frac{1}{\sqrt{2}} \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{array}\right].$$

Suppose we have a 3-qubit quantum register, and we want to apply H to the first qubit, X to the second qubit, and Z to the third qubit. Then the transformations on the entire register are

$$H \otimes I_4, \qquad I_2 \otimes X \otimes I_2, \qquad I_4 \otimes Z$$

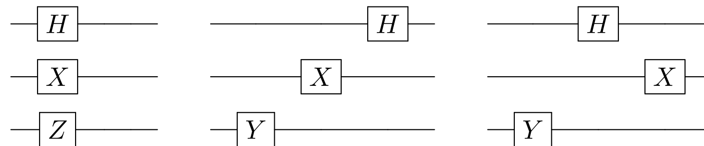
and the circuit diagram for this sequence of three gates is simply



By the mixed product property, this is equivalent to

$$(H \otimes I_2 \otimes I_2)(I_2 \otimes X \otimes I_2)(I_2 \otimes I_2 \otimes Z) = (HI_2I_2) \otimes (I_2XI_2) \otimes (I_2I_2Z) = H \otimes X \otimes Z.$$

So we could have applied them in any order or simultaneously. The circuit diagrams below are equivalent (note this is not an exhaustive list of arrangements):



2.7.1 Hadamard Gate on a Quantum Register

Performing a Hadamard transform on a register of n qubits can be represented by

$$H^{\otimes n} = \underbrace{H \otimes H \otimes \cdots \otimes H}_{n \text{ times}}.$$

If we apply this Hadamard transform to a register of n qubits initialized to $|0\rangle$, it will put the register in an equal superposition of all 2^n computational basis states. We can represent this as

$$\underbrace{\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\otimes\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\otimes\cdots\otimes\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)}_{n \text{ times}}=\frac{1}{\sqrt{2^n}}\sum_{i=0}^{2^n-1}|i\rangle\,.$$

The binary dot product of two vectors $a=(a_1,a_2,\ldots,a_n)$ in and $b=(b_1,b_2,\ldots,b_n)$ in $\mathbb{Z}_2^n$ plays a key role here: $a\cdot b=a_1b_1+a_2b_2+\cdots+a_nb_n \pmod 2$. The rows and columns of $H^{\otimes n}$ are indexed by binary n-tuples, and the (a,b)-entry is $(-1)^{a\cdot b}\frac{1}{2^{n/2}}$. What if we apply this Hadamard transform to some other state in the computational basis? We have

$$H^{\otimes n}|a\rangle=\frac{1}{2^{n/2}}\sum_{b=0}^{2^n-1}(-1)^{a\cdot b}|b\rangle\,.$$

Notice that exactly half of the states b result in a +1 coefficient and the other half result in a -1 coefficient, unless a is the zero state. So if we apply the transformation twice, we have

$$\begin{aligned}H^{\otimes n}H^{\otimes n}|a\rangle&=\frac{1}{2^{n/2}}\sum_{b=0}^{2^n-1}(-1)^{a\cdot b}H^{\otimes n}|b\rangle\\&=\frac{1}{2^n}\sum_{b=0}^{2^n-1}\left((-1)^{a\cdot b}\sum_{c=0}^{2^n-1}(-1)^{b\cdot c}|c\rangle\right)\\&=\frac{1}{2^n}\sum_{c=0}^{2^n-1}\left(\sum_{b=0}^{2^n-1}(-1)^{(a\oplus c)\cdot b}\right)|c\rangle\end{aligned}$$

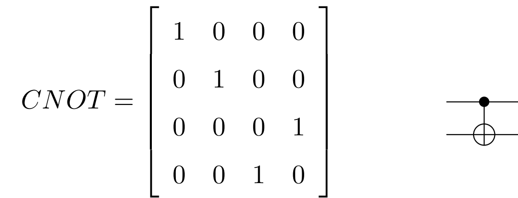
where $\oplus$ is vector addition in $\mathbb{Z}_2^n$. The sum $a\oplus c$ is the zero state if and only if $a=c$. As stated previously, unless this is the case, exactly half the tuples b result in a +1 coefficient and the other half result in a -1 coefficient. Therefore, these coefficients cancel, and we are left with only the case where $c=a$:

$$\frac{1}{2^n}\sum_{c=0}^{2^n-1}\left(\sum_{b=0}^{2^n-1}(-1)^{(a\oplus c)\cdot b}\right)|c\rangle=\frac{1}{2^n}\sum_{b=0}^{2^n}(-1)^{(a\oplus a)\cdot b}|c\rangle=\frac{1}{2^n}\sum_{b=0}^{2^n-1}|c\rangle=|c\rangle\,.$$

Thus $H^{\otimes n}H^{\otimes n}|a\rangle=|a\rangle$, and the square of this matrix is the identity.

2.7.2 CNOT Gate

We can also form unitary matrices that don't factor into Kronecker products of 2×2 single-qubit gates. Perhaps the most important one is the CNOT ("controlled not") gate, shown here with its matrix representation and its wire diagram notation:

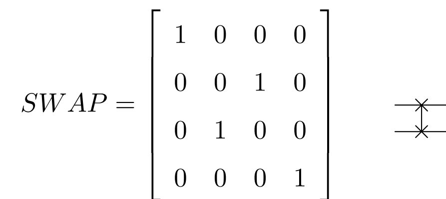


$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The gate maps $|00\rangle \mapsto |00\rangle$, $|01\rangle \mapsto |01\rangle$, $|10\rangle \mapsto |11\rangle$, and $|11\rangle \mapsto |10\rangle$. It flips the second bit if and only if the first bit is equal to one.

2.7.3 SWAP Gate

The swap gate has matrix representation and wire diagram notation:



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

This gate maps $|00\rangle \mapsto |00\rangle$, $|01\rangle \mapsto |10\rangle$, $|10\rangle \mapsto |01\rangle$, and $|11\rangle \mapsto |11\rangle$. It always swaps qubit 1 with qubit 2. If we look at a 2-qubit state in superposition and apply a swap gate, we see that only the probability amplitudes of $|01\rangle$ and $|10\rangle$ were swapped. This is because swapping qubit 1 and qubit 2 when they are both in state $|0\rangle$ has no effect on drawing $|00\rangle$. The same is true for $|11\rangle$.

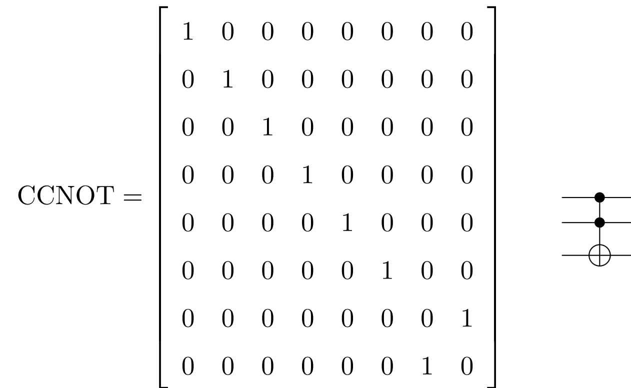
We can see the effect on the probability amplitudes of an arbitrary 2-qubit register:

$$\text{SWAP}\left(c_1 |00\rangle + c_2 |01\rangle + c_3 |10\rangle + c_4 |11\rangle\right) = c_1 |00\rangle + c_3 |01\rangle + c_2 |10\rangle + c_4 |11\rangle .$$

2.7.4 Controlled Gates

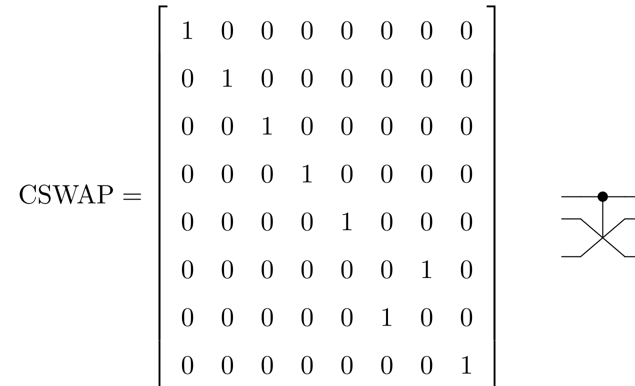
The Toffoli gate is a "controlled-controlled-not" gate. Its matrix representation and wire diagram notation are

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



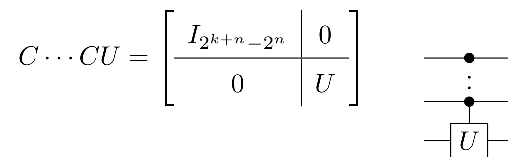
Similarly, the Fredkin gate is a "controlled swap" gate. Its matrix representation and wire diagram notation are

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



By now, a pattern may present itself. We can apply an arbitrary number of control qubits to a unitary matrix U which acts on n qubits. A controlled U gate with k control qubits can be represented as

$$C \cdots CU = \left[\begin{array}{c|c} I_{2^{k+n}-2^n} & 0 \\ \hline 0 & U \end{array}\right]$$



2.8 Entanglement

When the state of a quantum register is not expressible as a Kronecker product of single qubit states, we say that the state of that register is *entangled*.

For example,

$$\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

is not an entangled state because it can be represented by a Kronecker product of two other single qubit states. We can illustrate this:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

However, the state

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

is entangled. We can show this by supposing there is a Kronecker product which results in $|\Phi^+\rangle$:

$$\begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} a\begin{bmatrix} c \\ d \end{bmatrix} \\ \\ b\begin{bmatrix} c \\ d \end{bmatrix} \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}.$$

This gives us the system of equations:

$$\begin{aligned} ac &= \frac{1}{\sqrt{2}} \\ ad &= 0 \\ bc &= 0 \\ bd &= \frac{1}{\sqrt{2}} \end{aligned}$$

Which we can reduce to

$$\begin{aligned} ac = bd &= \frac{1}{\sqrt{2}} \\ ad = bc &= 0. \end{aligned}$$

We cannot have a equal to zero or else ac will equal zero, which violates our constraint. So d must be zero since $ad = 0$. But this causes bd to be zero, which also violates the constraints. Therefore, no Kronecker product of single qubit states can create the state $|\Phi^+\rangle$.

When we measure $|\Phi^+\rangle$ in the standard basis, the state collapses to either $|00\rangle$ or $|11\rangle$. The strange part of entanglement comes from the fact that in a physical implementation, we may measure individual qubits one at a time instead of looking at the whole register. If we measured just the first qubit, there would be an equal chance of it being a 0 or 1, but once the measurement is done, the second qubit is guaranteed to be equal to the first qubit upon measurement.

$|\Phi^+\rangle$ also exhibits another interesting property: it is *maximally* entangled. This means that the probabilities of obtaining any result are equal. There is an equal chance of drawing $|00\rangle$ or $|11\rangle$. These maximally entangled pairs in $\mathbb{C}^4$ form a basis called the *Bell basis*:

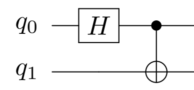
$$
\begin{aligned}
\mathcal{B} &= \left\{ |\Phi^+\rangle, |\Phi^-\rangle, |\Psi^+\rangle, |\Psi^-\rangle \right\} \\
|\Phi^+\rangle &= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\
|\Phi^-\rangle &= \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \\
|\Psi^+\rangle &= \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \\
|\Psi^-\rangle &= \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)
\end{aligned}
$$

Each of these vectors may also be referred to as an *EPR pair*, named after Einstein, Podolsky, and Rosen [14]. These pairs are useful in demonstrating the benefits of entanglement on a small scale. EPR pairs are used in quantum teleportation and cryptography.

To get a Bell state, we can use our change of basis transform from (2.1). In fact, this unitary transformation is represented by applying a Hadamard gate to the first qubit, then a CNOT gate from the first qubit targeting the second:

$$
\text{CNOT}\,(H \otimes I_2)\,|00\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).
$$

This gate combination on any state in the computational basis will generate its corresponding Bell state in the Bell basis:



$$|00\rangle \to |\Phi^+\rangle$$

$$|01\rangle \to |\Phi^-\rangle$$

$$|10\rangle \to |\Psi^+\rangle$$

$$|11\rangle \to |\Psi^-\rangle$$

Chapter 3

An Introduction to Qiskit

Qiskit [15] is an open-source development kit for working with quantum computers in Python. This package allows you to build and run quantum circuits either on local simulators or real IBM quantum machines [3].

3.1 Installation

The recommended configuration for Qiskit, and the configuration we will use here, involves installing the packages within an Anaconda environment. Anaconda is a distribution platform for Python that allows you to create separate Python environments (called *conda* environments). To download Qiskit and Anaconda:

1. **Download and install the latest version of Python**: You can find installers for Windows, Linux/UNIX, and macOS at

   `https://www.python.org`

   Ensure that you download Python version 3.6 or later; downloading the latest version is usually the best option.

Figure 3.1: Screenshot of the Python website highlighting the download link[16]

2. **Install Anaconda on your machine**: You can find installers for Windows, Linux/UNIX, and macOS at

https://www.anaconda.com/

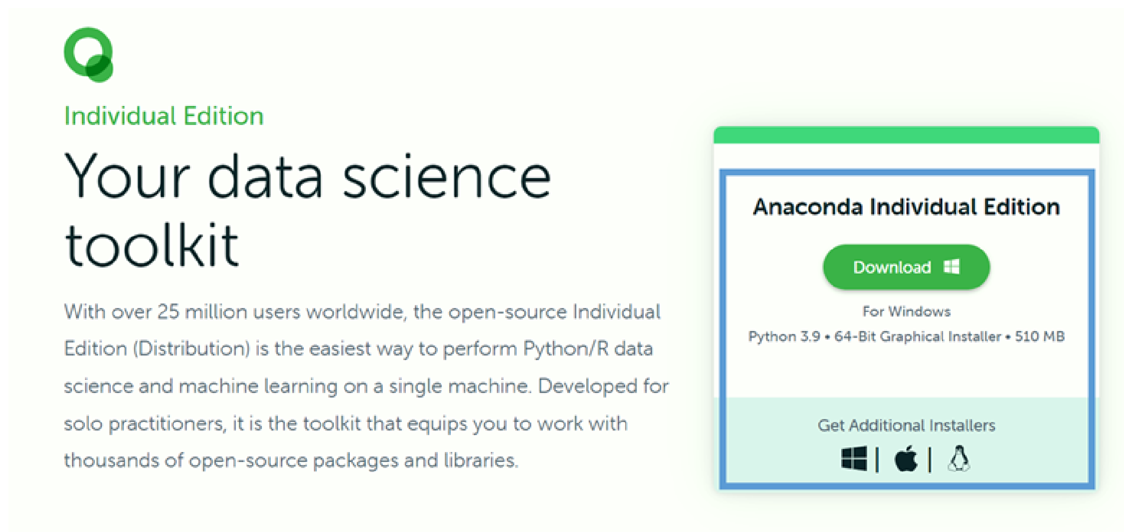


Figure 3.2: Screenshot of the Anaconda website highlighting the download link[17]

3. **Open an Anaconda environment**: In the Anaconda Command Prompt on a Windows machine,

or in the Terminal on a macOS or Linux/UNIX machine, create a new conda environment with the following command.

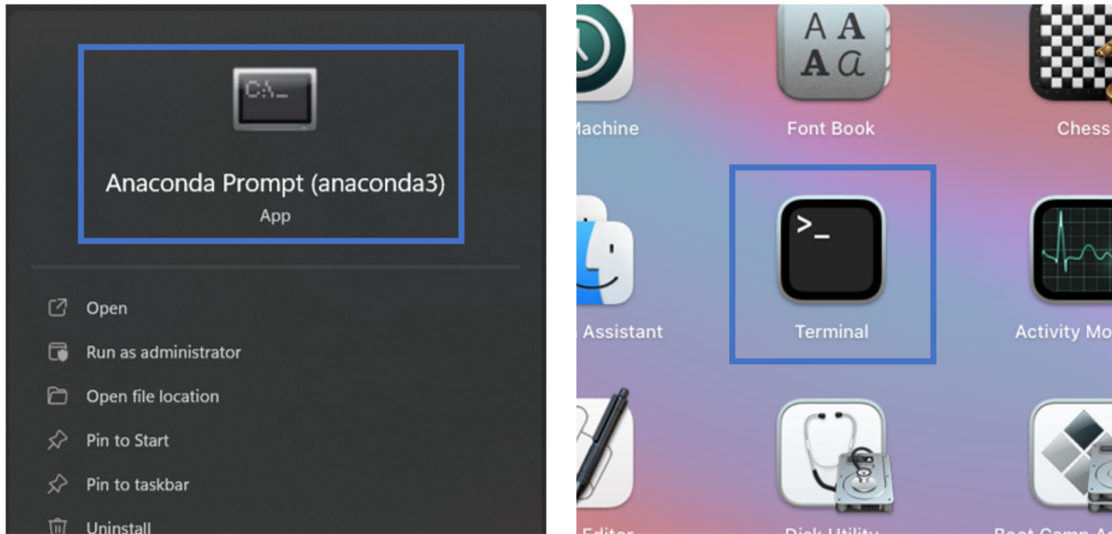


Figure 3.3: Opening the Anaconda Command Prompt in Windows and the Terminal in macOS

```
conda create -n environment-name anaconda
```

This command will create a new conda environment running Python 3 called *environment-name* and will automatically install all default packages. We will install and use Qiskit within this new environment.

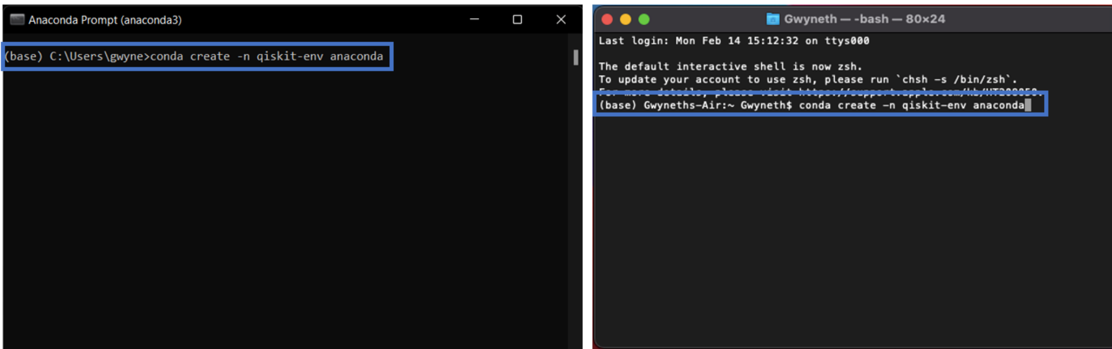


Figure 3.4: Creating a conda environment on Windows (left) and macOS (right)

4. **Open the environment and install the Qiskit package**: We can open our new environment with

```
conda activate environment-name
```

Figure 3.5: Activating a conda environment on Windows (left) and macOS (right)

When you activate an environment, you should see this change reflected in your command prompt/terminal.

Figure 3.6: Seeing active environment on Windows (left) and macOS (right)

Once you are in your environment, install Qiskit with the command

```
pip install qiskit
```

Figure 3.7: Installing Qiskit in a conda environment on Windows (left) and macOS (right)

As an optional step, you can install Qiskit's visualization package. This will make the visual outputs, for example the drawings of circuits, look much nicer. All the drawings in this report have been made using this visualization package. To install the package on a Windows, Linux/UNIX, and older macOS machines use

```
pip install qiskit[visualization]
```

To install on newer macOS (those which use zsh) machines use

```
pip install 'qiskit[visualization]'
```

You can verify that Qiskit was installed correctly with the command

```
conda list
```

This command lists the packages installed in your current environment.

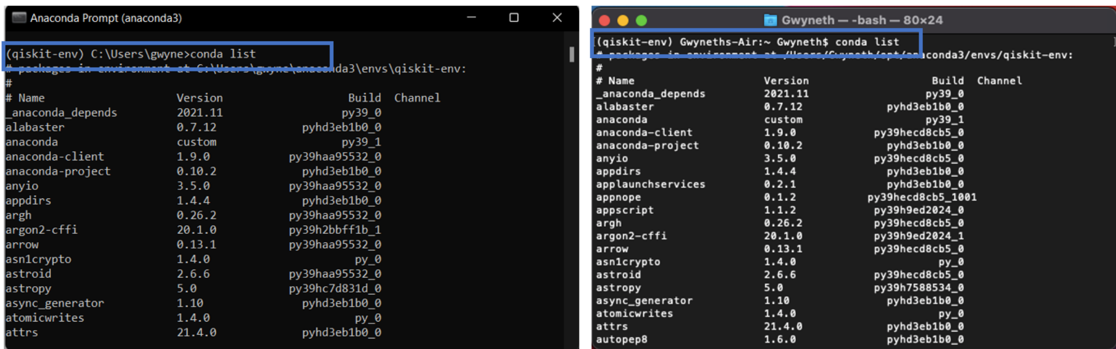


Figure 3.8: Output of the *conda list* command on Windows (left) and macOS (right)

If everything has been installed correctly, you should see Qiskit and its subsidiaries in this list.

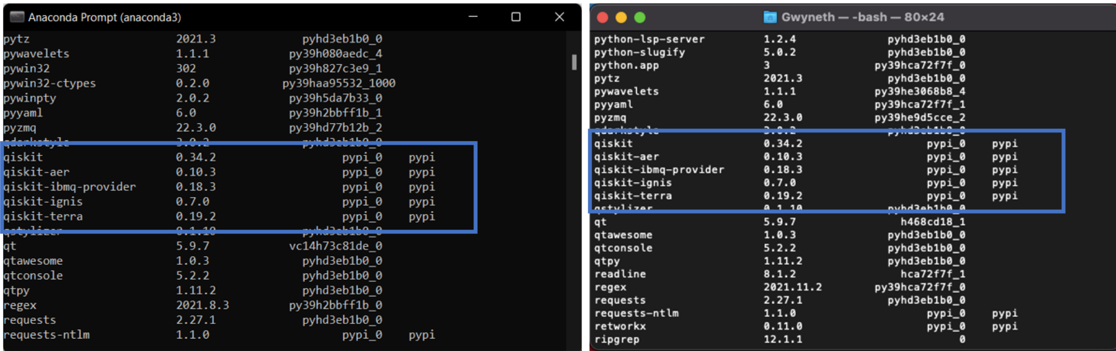


Figure 3.9: Qiskit package and its subsidiaries in the list of installed packages on Windows (left) and macOS (right)

5. **Open a new Jupyter Notebook:** A Jupyter Notebook is a special type of Python file that allows you to run lines of code in distinct chunks. Jupyter Notebook and JupyterLab can be installed from

   https://jupyter.org/

   To open a new Jupyter Notebook file and get started using Qiskit, use the command

```
jupyter notebook
```

   within your conda environment.

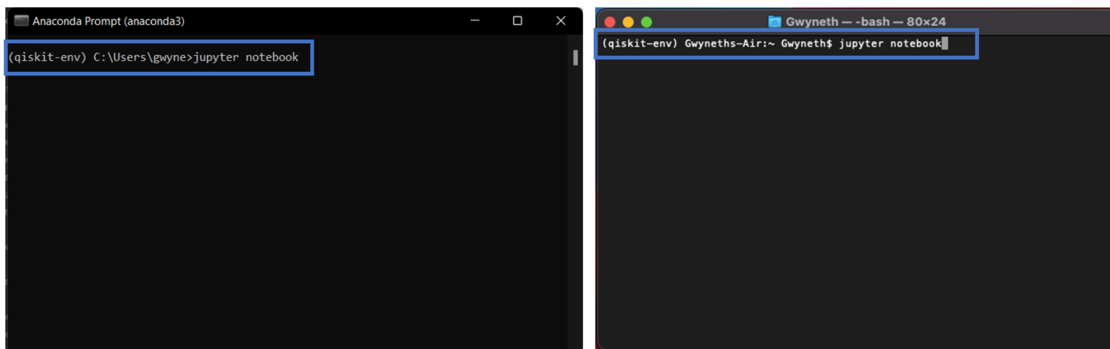


Figure 3.10: Command to open Jupyter Notebook on Windows(left) and macOS(right)

   This will launch Jupyter Notebook in your default browser at *localhost:8888*. You can then navigate to the folder of your choice and create a new script. To use Qiskit, make sure that the file you create is using the right version of python. You can also use the Anaconda Navigator GUI to launch Jupyter Notebook if you have it installed on your computer.

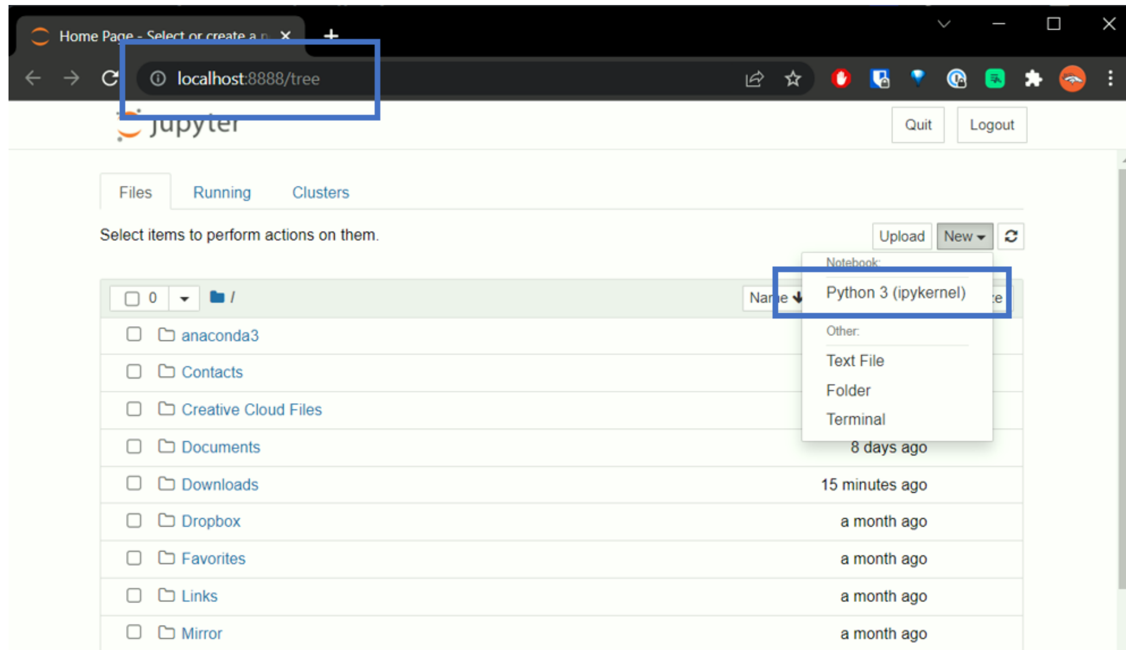


Figure 3.11: Creating a new Python 3 script with Jupyter Notebook launched in a browser

For more information, visit the installation page on the Qiskit website

https://qiskit.org/.

3.2 Building Simple Circuits

Now that we have installed Qiskit, let's build some simple circuits. To start out, we need to import some things from Qiskit. Open Jupyter Notebook and create a new cell using the menu bar or by pressing 'a' and add the following code to the cell.

```python
# Build and use quantum circuits
from qiskit import QuantumCircuit

# Create and use state vectors
from qiskit.quantum_info import Statevector

# Use the Qiskit's Aer quantum machine simulator
from qiskit.providers.aer import *

# Visualize quantum circuits in a nicer format
# If you did not install the visualization package,
# remove this import
from qiskit.visualization import *
```

Now run the first cell, so we can use these within this file. Next, let's build our first quantum circuit. Create a new cell and write the following.

```python
# QuantumCircuit(q,c) returns a new QuantumCircuit with
# q quantum bits and c classical bits
circuit = QuantumCircuit(2,2)

# QuantumCircuit.draw() outputs the circuit to the console
# The argument output='mpl' specifies it draw the circuit
# using matplotlib.
circuit.draw(output='mpl')
```

When you run this cell, it should output the following diagram.

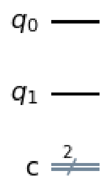


Figure 3.12: Quantum circuit with 2 qubits and 2 classical bits

This shows that our circuit has 2 quantum bits and 2 classical bits. Next, let's add some simple quantum gates to our new circuit. To add a Hadamard gate and a NOT gate, write and execute the following in another cell:

```
# QuantumCircuit.h(n) adds a Hadamard gate to qubit n
circuit.h(0)

# QuantumCircuit.x(n) adds a NOT gate to qubit n
circuit.x(1)

circuit.draw(output='mpl')
```

We can verify our gates were added correctly by looking at the resulting diagram.

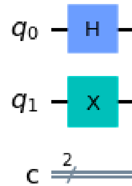


Figure 3.13: Quantum circuit after adding a Hadamard gate and a NOT gate

The 'Statevector' object in Qiskit holds the output state vector of an input state evolved by a given instruction or circuit. By default, this function assumes the starting state is the same number of qubits as the input instruction or circuit all in the state $|0\rangle$. In this case, we can find the state vector of the default register after being evolved by our quantum circuit.

```
# state = Statevector(c) Returns a Statevector for
# the QuantumCircuit c
state = Statevector(circuit)

# Output the state vector using the parameter
# 'latex' to format the output using LaTeX
state.draw('latex')
```

This should result in this output vector.

$$\begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

But this result is backwards! **Rather than numbering bits from left to right, as we were working with above, Qiskit numbers them right to left.** It's really important to take note of this for the calculations we will do in the future. Getting back to our circuit, we can add some entanglement with another gate.

```
# To create a CNOT gate use QuantumCircuit.cx(c,t) where
# c represents the control qubit's number, and
# t represents the target qubit's number
circuit.cx(0,1)

# Draw the circuit
circuit.draw(output.'mpl')

# Calculate and draw the Statevector
state2 = Statevector(circuit)
state2.draw('latex')
```

This cell outputs the following circuit diagram and output state vector. Recall that by default, the output state vector is calculated assuming that both qubits are in state $|0\rangle$.

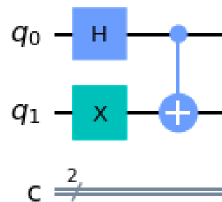


Figure 3.14: Quantum circuit after adding a CNOT gate

$$\begin{bmatrix} 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \end{bmatrix}$$

Now we can see we have entangled the qubits!

Let's measure the output of our circuit.

```
# QuantumCircuit.measure([q1,q2,..qn], [c1,c2,..,cn])
# where q1,q2,..qn is a list of qubits and
# c1,c2,..,cn is a list of classical bits
# It maps q1 to c1, q2 to c2, etc.
circuit.measure([0,1],[0,1])

# Draw the circuit
circuit.draw(output.'mpl')
```

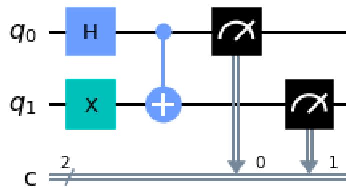


Figure 3.15: Quantum circuit after adding measurement

Now that we have it all set up, we can run our quantum circuit on a simulated quantum machine. We run multiple simulations to see the distribution of the final measurements.

Right now, we will use Qiskit's default quantum machine simulator, the Aer Simulator. It has different properties you can specify when setting it up, but for now we will use the default. Later, we will expand on using these properties, using other simulators, and running circuits on real IBM quantum machines.

Let's now create a simulator, run some trials, and then print out the statistics of the measurements we take for each trial.

```
# AerSimulator() returns the default Aer simulator
sim = Aersimulator()

# AerSimulator.run(c, shots = n) creates and returns a job for the simulator
# where c represents a QuantumCircuit and n is the number of simulations to run
job = sim.run(circuit, shots=1000)

# job.result() returns the result object for the given simulation job
result = job.result()

# results.get_counts() returns the number of simulations that resulted
# in each measurement
counts = result.get_counts()

# Stating the variable prints out the value of counts
counts
```

If you run this cell it will report the number of '10' measurements and the number of '01' measurements. It should be around a 50/50 split. However, if you run the cell a couple more times, you can see that this distribution changes slightly. Our entanglement is working!

Throughout the rest of the report, we will build and run circuits using Qiskit to implement each of the algorithms we discuss.

Chapter 4

Early Algorithms

Many quantum algorithms have been developed, each demonstrating some advantage over their classical counterparts. Some, such as Shor's algorithm for factoring integers, were developed before a physical quantum computer was implemented. However, as quantum computing is mostly linear algebra, mathematicians, physicists, and computer scientists alike can work to develop algorithms that demonstrate the quantum advantage and emphasize the need for quantum computer literacy.

4.1 The Hidden Subgroup Problem

Some of the first applications of quantum algorithms found speedups by theoretically being able to solve the hidden subgroup problem (HSP) faster than a classical computer. The first four algorithms covered—Deutsch-Jozsa, Bernstein-Vazirani, Simon, and Shor—solve a variation of this problem.

Hidden Subgroup Problem

Let G be a group. Let $H \leq G$ be a subgroup of G. Let X be a set and let $f : G \to X$, where, for all $g_1, g_2 \in G$, $f(g_1) = f(g_2)$ if and only if $g_1 H = g_2 H$. By querying f, find a generating set for H.

For this general description of the problem, G could be an infinite group, it could lack commutativity of elements, or both. This lack of structure causes the problem to be difficult to approach. There are no known algorithms for solving this problem classically, or by quantum means, in full generality. However when G holds certain properties, we can find a classical and quantum algorithm to solve the hidden subgroup problem. For the rest of this report, we will assume G is finite and abelian.

4.1.1 Problem Reductions

At first glance, the hidden subgroup problem does not seem very useful. Its usefulness becomes more apparent when we consider problem reductions. A *reduction* is a transformation from one problem into another in such a way that solving the second allows a solution for the first to be found using that transformation.

Finding the period of an element in a multiplicative group modulo n is reducible to the hidden abelian subgroup problem. The function that hides the subgroup is *periodic*, that is, for all g in G, $f(g) = f(g + r)$ and the subgroup hidden is the group generated by r. Solving this problem is central to Shor's algorithm in Chapter 5.

The discrete logarithm problem is also reducible to the hidden abelian subgroup problem. This problem is covered in §5.7. Not covered in this text are reductions from the graph isomorphism problem and shortest vector problem. These problems are not abelian; instead, they rely on symmetric groups and dihedral groups, respectively.

In 2003, Kuperberg [18] created a sieving algorithm for dihedral groups, D_N, utilizing quantum computing. His algorithm achieves a runtime of $2^{\mathcal{O}\left(\sqrt{\log N}\right)}$. This is an improvement over a classical query algorithm, which would require $\mathcal{O}(\sqrt{N})$ queries. At first glance, the runtime of the quantum algorithm does not appear to be an improvement, but as an exercise, plot out both runtimes in a graphing calculator. Kuperberg improved the space requirements with the help of Regev in 2011 [19]. However, the reduction of the shortest vector problem to the HSP does not yield an improvement over classical algorithms such as Lenstra-Lenstra-Lovász (LLL) and Block Korkine-Zolotarev (BKZ) [20].

4.2 Deutsch-Jozsa

The Deutsch-Jozsa problem [21] was designed to show the usefulness of quantum computing. Thus, its main goal is not to solve a real-world problem, but to demonstrate how a quantum computer might solve a specific problem more efficiently than a classical computer. The Deutsch-Jozsa problem and subsequent algorithm additionally helped separate the complexity classes P and EQP (exact quantum polynomial).

The first iteration of the algorithm was developed by David Deutsch in 1985 [22], although this version only provided a solution for the simplest case. In collaboration with Richard Jozsa, Deutsch was able to improve the algorithm to solve a more general case in 1992 [21]. The now deterministic algorithm was able to take n bits of input and determine whether the function was balanced using 2 queries.

In 1998, Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca developed the algorithm we know today [23]. Though the name Deutsch-Jozsa was kept to pay homage to the groundbreaking work

of the original creators, this new version is able to solve the problem in a single query instead of two.

4.2.1 The Problem

In the Deutsch-Jozsa problem, we are given a function $f : \{0,1\}^n \to \{0,1\}$ and asked to find whether it is constant (i.e., all outputs are 1, or all outputs are 0) or balanced (i.e., the outputs are split 50-50 between 0 and 1). Computing this on a classical computer requires $2^{n-1}+1$ queries of f in the worst case. However, using the Deutsch-Jozsa algorithm, we only need to query f once, using it as a quantum oracle. This oracle maps $|x\rangle|y\rangle$ to $|x\rangle|y \oplus f(x)\rangle$. For an example of the implementation of the oracle, see §4.2.3.

4.2.2 The Algorithm

We start with $n+1$ qubits, with the first n initialized to $|0\rangle$ and the last to $|1\rangle$. We then apply a Hadamard gate to all $n+1$ qubits. Utilizing what we learned in §2.7.1 the resulting state of the quantum register is

$$\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}|x\rangle(|0\rangle-|1\rangle). \tag{4.1}$$

From here, we apply the oracle introduced in the previous subsection. Note that we have two possible results from applying this oracle:

$$|x\rangle|0\rangle \to |x\rangle|0\oplus f(x)\rangle = |x\rangle|f(x)\rangle$$

and

$$|x\rangle|1\rangle \to |x\rangle|1\oplus f(x)\rangle.$$

By linearity, the state (4.1) is mapped to

$$\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}|x\rangle(|f(x)\rangle-|1\oplus f(x)\rangle). \tag{4.2}$$

Note that, as $f(x)$ always outputs either 0 or 1, we can simplify this last qubit further. If $f(x)=0$, then

$$|f(x)\rangle-|1\oplus f(x)\rangle = |0\rangle-|1\rangle.$$

If $f(x)=1$, then

$$|f(x)\rangle-|1\oplus f(x)\rangle = |1\rangle-|0\rangle.$$

We see that the sign is dependent on the outcome of $f(x)$, and thus rewrite as

$$|f(x)\rangle-|1\oplus f(x)\rangle = (-1)^{f(x)}(|0\rangle-|1\rangle).$$

Our state (4.2) can now be written as

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

We now disregard the last qubit, and are left with

$$\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}\left|x\right\rangle.$$

Finally, we apply another Hadamard gate to the first n qubits, bringing our register to

$$\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}\left(\frac{1}{\sqrt{2^n}}\sum_{y=0}^{2^n-1}(-1)^{x\cdot y}\left|y\right\rangle\right). \tag{4.3}$$

After rearranging the summations, we obtain

$$\frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}\right]\left|y\right\rangle.$$

To determine the result of the algorithm, we now measure the first n qubits. Let's look at the behavior of the summation in the cases where $f(x)$ is constant and balanced.

If $f(x)$ is constant, then the term $(-1)^{f(x)}$ is constant. If $f(x)=0$ for all x, then

$$\frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{0}(-1)^{x\cdot y}\right]\left|y\right\rangle=\frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{x\cdot y}\right]\left|y\right\rangle.$$

Let's examine the bitwise dot product mod 2, $x\cdot y$. When $y=0$, $x\cdot y=0$ for all x. So the sign on the inner summation will always be 1. Pulling out the term where of $y=\left|00\dots0\right\rangle$ gives us

$$\frac{1}{2^n}\left(2^n\left|00\dots0\right\rangle+\sum_{y=1}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{x\cdot y}\right]\left|y\right\rangle\right).$$

Now let $y\neq 0$. Then $y\cdot x$ will result in an equal number of zeros and ones when iterated across each x. With an equal number of zeros and ones, the double summation reduces to

$$\frac{1}{2^n}\left(2^n\left|00\dots0\right\rangle+\sum_{y=1}^{2^n-1}2^{n-1}\left|y\right\rangle-2^{n-1}\left|y\right\rangle\right)$$

which then causes the whole summation to reduce to 0. Our state (4.3) is now

$$\frac{1}{2^n}\left(2^n\left|00\dots0\right\rangle+0\right)=\left|00\dots0\right\rangle.$$

So when measuring the first n qubits, we get 0 with a probability of 1.

If $f(x) = 1$ for all x, we have

$$\frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^1(-1)^{x\cdot y}\right]|y\rangle = -\frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{x\cdot y}\right]|y\rangle .$$

The only difference between the first case and this one is that the sign is different. So the whole summation reduces to $-|00\ldots0\rangle$. This also gives a result of 0 with probability 1.

Now we look at the summation when $f(x)$ is balanced:

$$\frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}\right]|y\rangle .$$

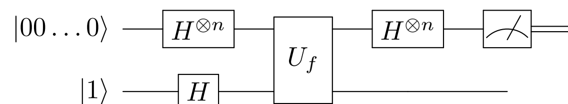
If we draw out the $y = 0$ term again, $(-1)^{y\cdot x} = 1$ for each x, but now since $f(x)$ is balanced, there are now an equal number of 1 and -1 coefficients from $(-1)^{f(x)}$. So extracting $|00\ldots0\rangle$ gives us

$$\frac{1}{2^n}\left(2^{n-1}|00\ldots0\rangle - 2^{n-1}|00\ldots0\rangle + \sum_{y=1}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}\right]|y\rangle\right)$$

and the $|00\ldots0\rangle$ cancels out to have amplitude of 0. While the actual result measured depends on $f(x)$ (and is subject to quantum noise), we are guaranteed that we can never draw $|00\ldots0\rangle$ when $f(x)$ is balanced.

Thus, when we measure the first n qubits, we will receive a register of all zeros if $f(x)$ is constant, and a non-zero result if $f(x)$ is balanced.

We can look at the circuit diagram of this algorithm to view it from another perspective:



Another way to think about the Deutsch-Jozsa algorithm is that the oracle will not alter the input state in a meaningful way if $f(x)$ is constant. Since the Hadamard transform inverts itself, if the unitary oracle does not alter the first n qubits, the value measured is just the initial state, which was zero.

4.2.3 Qiskit

This function will create a Deutsch-Jozsa circuit when given an oracle which applies gates to match a constant or balanced function, and the dimension of the input.

```
# import basic quantum circuits, simulators, and plot tools
from qiskit import QuantumCircuit
from qiskit import Aer
from qiskit.visualization import plot_histogram

def deutsch_jozsa_circuit(oracle, dim):
    """Construct a circuit to solve the Deutsch-Jozsa
    problem when given a function that is promised to either
    be balanced or constant

    :param oracle: A method which applies the unitary oracle
    to the circuit
    :param dim: The dimension of the domain of the oracle
    :return: A qiskit QuantumCircuit object which when run
    will solve the problem
    """
    # Create a circuit with n + 1 qubits, n classical bits,
    # and give it a name
    circuit = QuantumCircuit(dim + 1, dim,
                          name=f'Deutsch-Jozsa on {dim} qubits')

    # Register is in state |00...00> so flip the last bit
    circuit.x(dim)

    # Apply a Hadamard gate to every qubit
    [circuit.h(i) for i in range(dim + 1)]
    circuit.barrier()

    # Apply the oracle
    oracle(circuit,dim)
    circuit.barrier()

    # Apply a Hadamard gate to every qubit except for the
    # last one
    [circuit.h(i) for i in range(dim)]

    # Measure all qubits but the last qubit
    [circuit.measure(i, i) for i in range(dim)]
    return circuit
```

Now all that is needed is to make some oracles. We have 3 cases: all zeros, all ones, or a balanced number of ones and zeros.

The $(n+1)^{\text{st}}$ qubit stores the output of our function, so at the simplest level, all we have to do is set that output based on what the result is. A 0 constant oracle will do nothing, while a 1 constant oracle flips the $(n+1)^{\text{st}}$ qubit to 1. Both of these leave the first n qubits alone, so that the Hadamard gates will undo themselves before measurement and leave us with 0.

```
# 0 constant oracle
def constant_0_oracle(circuit, dim):
    pass

# 1 constant oracle
def constant_1_oracle(circuit, dim):
    circuit.x(dim)
```

Next is the balanced oracle. There are many possibilities to choose from. This simple one is easy to implement, and illustrative. We apply CNOT gates from each qubit as a control to the $(n+1)^{\text{st}}$ qubit. This entangles the state, and results in a non-zero measurement. We can then build and view the Deutsch-Jozsa circuit.

```
# Balanced oracle
def balanced_oracle(circuit, dim):
    [circuit.cx(i, dim) for i in range(dim)]

# View the circuit with a balanced oracle and dimension 4
circuit = deutsch_jozsa_circuit(balanced_oracle, 4)
display(circuit.draw('mpl'))
```

The algorithm produced by this code fragment is shown below. The oracle is applied between the two barriers. We must remember that the implementation of the oracle is hidden to the algorithm.

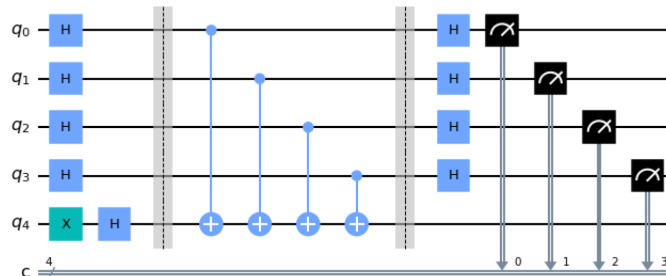


Figure 4.1: The circuit generated by Qiskit for a balanced oracle with $n = 4$.

Now we can run some simulations and view the results.

```
# Run the simulator 1024 times and view the results
aer_sim = Aer.get_backend('aer_simulator')
results = aer_sim.run(circuit, shots=1024).result()
plot_histogram(results.get_counts())
```

The result of the simulations should be depicted in the histogram generated by the above code fragment. For this example, the results should look like the following.

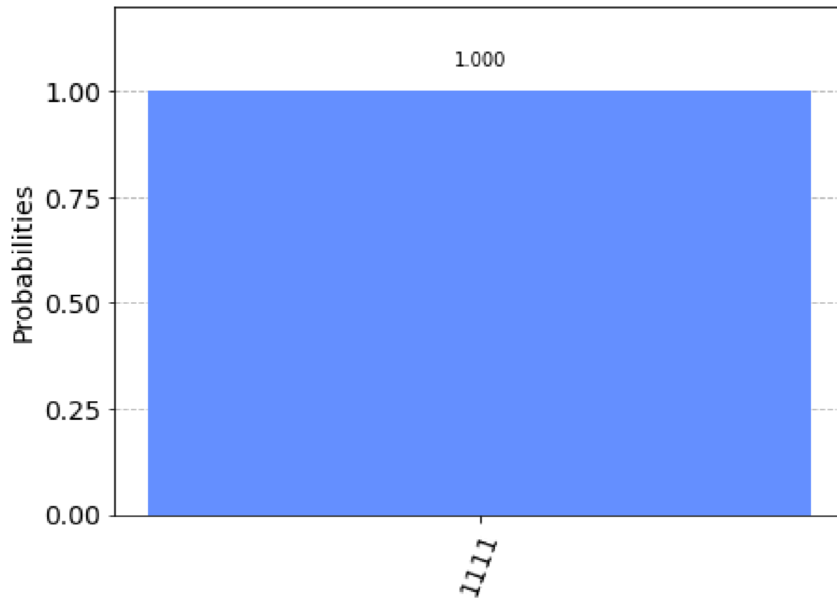


Figure 4.2: The results from 1024 iterations of the Deutsch-Jozsa algorithm for $n = 4$.

From the results and the analysis in the section above, we see that our function was constant. As an exercise, run the code yourself to verify that the algorithm returns the correct results.

4.3 Bernstein-Vazirani

In 1992, Ethan Bernstein and Umesh Vazirani developed their own algorithm to solve a modified version of the Deutsch-Jozsa problem [24]. Rather than classifying functions, this algorithm attempts to find a string encoded within a function. No significant improvements in efficiency or success probability over the original algorithm have been discovered.

4.3.1 The Problem

We are given oracle access to a function $f : \{0,1\}^n \to \{0,1\}$ and are promised that the result is always $f(x) = x \cdot s$, where $(\cdot)$ is the binary dot product modulo 2, and s is a bit-string of length n that is secret to us. Our goal is to determine what this string is.

When viewing the Bernstein-Vazirani problem through the lens of the hidden subgroup problem, the group G is $\mathbb{Z}_{2^n}$, and the hidden subgroup H is $\{h \in G \,|\, h \cdot s = 0\}$.

4.3.2 The Algorithm

On a classical computer we can make n queries to the function where we have only a single bit set to 1 for each query. The result returned gives us the value of the bit in s at that position.

However, with just a single oracle call on a quantum device, we can find the entire bit-string s. We start with $n+1$ qubits, with the first n in state $|0\rangle$ and the last in state $|1\rangle$. We then perform a Hadamard transform on all the qubits:

$$\frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle \left(|0\rangle - |1\rangle\right).$$

We then apply the oracle function, which sends $|x\rangle |y\rangle \to |x\rangle |y \oplus f(x)\rangle$, where $(\oplus)$ is vector addition mod 2. Since we apply this to the entire superposition, we end up with the sign being determined by the result of $f(x)$, similar to Deutsch-Jozsa. We can now disregard the last qubit, leaving us with

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

Then we take a second Hadamard transform. Recalling that $f(x) = x \cdot s$, this gives us

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

The sum $x \cdot s + x \cdot y$ can be rewritten as $x \cdot (s \oplus y)$. So,

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

The vector addition mod 2 of $s \oplus y$ is equal to the zero vector only when $y = s$.

Remark

Consider when y has been chosen not equal to s, that is, let y be given such that $y \neq s$. Then

$$\sum_{x=0}^{2^n-1} (-1)^{x \cdot (s \oplus y)} = \sum_{x=0}^{2^n-1} (-1)^{x \cdot k}$$

where k is a new non-zero vector from the result of $s \oplus y$. Since k is non-zero, over $x = 0 \ldots 2^n - 1$, $x \cdot k$ will equal 0 exactly half the time, and equal 1 exactly half the time. This means that an equal number of 1s and -1s are added together in the summation and

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

Summing over x from 0 to $2^n - 1$, the only vector left is $|y\rangle$ such that $y = s$:

$$\frac{1}{2^n} \sum_{y=0}^{2^n-1} \left(\sum_{x=0}^{2^n-1} (-1)^{x \cdot (s \oplus y)} \right) |y\rangle = |s\rangle .$$

Measuring the state will give us the secret string s.

4.3.3 Qiskit

We can see this algorithm in action using Qiskit. The code snippet below creates a circuit to solve the Bernstein-Vazirani problem for a given binary string s.

```
# import basic quantum circuits, simulators, and plot tools
import matplotlib.pyplot as plt
import numpy as np
from qiskit import Aer, QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.visualization import plot_histogram

def bern_vaz_circuit(s,n):
    """Construct a circuit to solve the Bernstein-Vazirani
    problem when given a hidden binary string.

    :param s: A hidden binary string
    :param n: The length of the hidden binary string
    :return: A qiskit QuantumCircuit object which when run
    will solve the problem
    """
    # Create a circuit with n qubits + 1 ancillary qubit
    # and n classical bits.
    bv_circuit = QuantumCircuit(n+1, n)

    # Initialize the state of the ancillary qubit
    bv_circuit.x(n)

    # Apply a Hadamard gate to each qubit
    for i in range(n):
        bv_circuit.h(i)
    bv_circuit.barrier()

    # Apply the inner-product oracle
    s = s[::-1] # Remember qiskit uses backwards ordering
    for q in range(n):
        if s[q] == '0':
            bv_circuit.i(q)
        else:
            bv_circuit.cx(q, n)
    bv_circuit.barrier()

    # Apply Hadamard gates to the first n qubits
    for i in range(n):
        bv_circuit.h(i)

    # Measure the first n qubits and return the final circuit
    for i in range(n):
        bv_circuit.measure(i, i)
    return bv_circuit
```

Let's run this function using the hidden string $s = 011$ and view the resulting circuit.

```
# Hidden binary string
s = '011'

# Build and view the Bernstein-Vazirani circuit
circuit = bern_vaz_circuit(s,len(s))
display(circuit.draw('mpl'))
```

This should produce the following circuit.

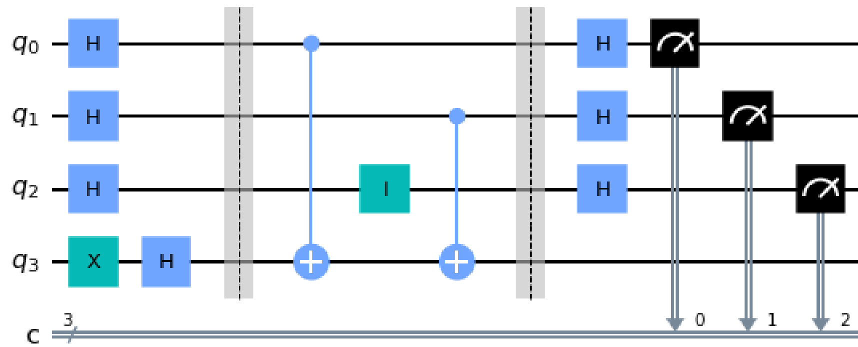


Figure 4.3: The circuit generated by Qiskit for $s = 011$.

```
# Run the simulator 1024 times and view the results
aer_sim = Aer.get_backend('aer_simulator')
results = aer_sim.run(circuit, shots=1024).result()
plot_histogram(results.get_counts())
```

These simulations yield the following results.

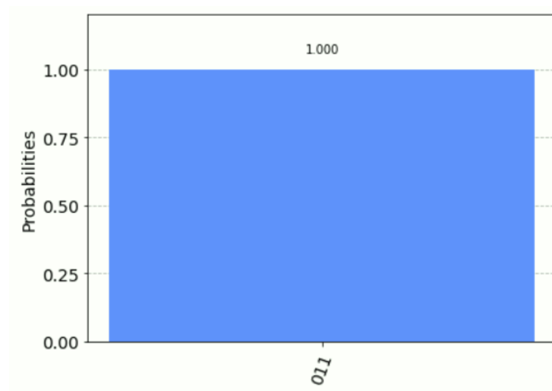


Figure 4.4: The results from 1024 simulations of the Bernstein-Vazirani algorithm.

Our hidden bit string was 011 and all 1024 simulations produced 011 as well. As an exercise, run the code above to make sure you get the same answer. Then try your own bit strings and explore the results.

4.4 Simon

Simon's problem [25] is another problem designed to show the efficiency of quantum computing over classical computing, first proposed by Daniel Simon in 1994, and was later the inspiration for Shor's algorithm [26]. Simon's algorithm solves the problem described below exponentially faster than any classical algorithm. Specifically, it uses a linear number of queries, whereas any classical algorithm must use an exponential number of queries. Like the Deutsch-Jozsa problem, Simon's problem has little real-world use, but the algorithm helped demonstrate the advantages quantum computers have over classical ones.

4.4.1 The Problem

In Simon's problem, we are given oracle access to a function $f : \{0,1\}^n \to \{0,1\}^n$ with the promise that for some unknown $s \in \{0,1\}^n$, for all $x, y \in \{0,1\}^n$, $f(x) = f(y)$ if and only if $x \oplus y = s$. Note that this means f is a one-to-one function if $s = 0^n$, since then

$$f(x) = f(y) \implies x = y \oplus 0^n \implies x = y$$

and f is a two-to-one function if $s \neq 0^n$. The goal in this problem is to determine s with as few queries to $f(x)$ as possible. As in the Deutsch-Jozsa problem, the oracle maps $|x\rangle |y\rangle \to |x\rangle |y \oplus f(x)\rangle$.

Simon's problem is a clear illustration of the hidden subgroup problem with $G = \{0,1\}^n$ and $H = \{0_G, s\}$ for some element s in G.

4.4.2 The Algorithm

Simon's algorithm starts with two n-qubit registers in the state $|0^n\rangle \otimes |0^n\rangle$. A Hadamard transform is then applied to the n qubits of the first register, resulting in the state

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

Next, the oracle is applied to this state, giving us

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

We next apply another Hadamard transform to the first register to obtain

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

We can rewrite this state as

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

Finally, a complete measurement is performed, but we are concerned with the bits in the first register only. The probability of measuring a string y is

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

There are now two cases to consider. If $s \neq 0^n$, then there are two possible inputs, x_1 and $x_2 = x_1 \oplus s$, which correspond to each output $z = f(x)$. Letting $A = f(\{0,1\}^n)$ be the image of f, we have

$$\left|\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{x\cdot y}\left|f(x)\right\rangle\right|^2 = \left|\frac{1}{2^n}\sum_{z\in A}((-1)^{x_1\cdot y} + (-1)^{x_2\cdot y})\left|z\right\rangle\right|^2.$$

Since $x_2 = x_1 \oplus s$, we know

$$(-1)^{x_1\cdot y} + (-1)^{x_2\cdot y} = (-1)^{x_1\cdot y} + (-1)^{(x_1\oplus s)\cdot y} = (-1)^{x_1\cdot y}(1 + (-1)^{y\cdot s}).$$

Thus, the probability can be rewritten as

$$\left|\frac{1}{2^n}\sum_{z\in A}(-1)^{x_1\cdot y}(1 + (-1)^{y\cdot s})\left|z\right\rangle\right|^2.$$

If $y \cdot s = 1$, then $1 + (-1)^{y\cdot s} = 1 - 1 = 0$, so this probability is zero. If $y \cdot s = 0$, then $1 + (-1)^{y\cdot s} = 1 + 1 = 2$, so the probability is

$$\left|\frac{1}{2^n}\sum_{z\in A}(-1)^{x_1\cdot y}(2)\left|z\right\rangle\right|^2 = \left|\frac{1}{2^{n-1}}\sum_{z\in A}(-1)^{x_1\cdot y}\left|z\right\rangle\right|^2 = \frac{1}{2^{n-1}}.$$

Thus, we know that if we measure a string y, then it is guaranteed that $y \cdot s = 0$.

Repeating the algorithm n times gives us the system of equations:

$$y_1 \cdot s = 0$$
$$y_2 \cdot s = 0$$
$$\vdots$$
$$y_{n-1} \cdot s = 0$$

If these equations are linearly independent, we can solve the system for s. If they are not linearly independent, we can repeat the algorithm until we have $n-1$ equations which are linearly independent.

If $s = 0^n$, then there is a unique input x corresponding to each output $f(x)$. Since f is one-to-one, the probability that the measurement results in a string y is

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

As $s = 0^n$, we have that $y \cdot s = 0$ for all y. In this case, we eventually obtain a system of n equations which are linearly independent.

In both cases, we have obtained a homogeneous system of $n-1$ or n linearly independent equations. In the first case, this system has two solutions: the trivial solution 0^n and a non-trivial solution s'. We can solve the system for s', and then check whether $f(s') = f(0^n)$. If $f(s') = f(0^n)$, then $s' = s$. If $f(s') \neq f(0^n)$ or we arrive at a system of rank n (hence admitting only the trivial solution), then it follows that $s = 0^n$. Either way, the problem is solved.

4.4.3 Query Complexity

We have proved above that we can find the solution using Simon's algorithm by obtaining a system of at most n linearly independent equations. We can now ask ourselves how we can guarantee $n-1$ randomly generated equations of the form $y \cdot s = 0$ are linearly independent. In order to prove this, we will find the bounds of the probability that any n equations returned by the algorithm are linearly independent.

Suppose we have k linearly independent equations associated with the vectors $y_1, y_2, \ldots, y_k$. These vectors span a subspace $S \subset \mathbb{Z}_2^n$ of size 2^k where S consists of all vectors in the form $a_1 y_1 + a_2 y_2 + \cdots + a_k y_k$ for $a_1, a_2, \ldots, a_k \in \{0, 1\}$.

Now suppose we find a new equation associated with a vector y_{k+1}. This equation will be linearly independent of all the previous equations as long as y_{k+1} lies outside S.

There is a $1 - 2^{k-n}$ probability that y_{k+1} lies outside S. So the probability that any n equations are linearly independent is the product of those probabilities for every k. This can be expressed as

$$Pr = \left(1 - \frac{1}{2^n}\right) \times \left(1 - \frac{1}{2^{n-1}}\right) \times \cdots \times \left(1 - \frac{1}{2^2}\right) \times \left(\frac{1}{2}\right)$$

$$= \prod_{k=1}^{n} \left(1 - \frac{1}{2^k}\right) > \prod_{k=1}^{\infty} \left(1 - \frac{1}{2^k}\right) > \frac{1}{4}.$$

Therefore, we obtain the solution using n linearly independent y vectors with a probability of greater than $\frac{1}{4}$. With any fewer than $n-1$ queries, there will not be a unique non-zero solution. The lower bound for the number of queries required for Simon's algorithm is $\Omega(n)$ [27].

4.4.4 Qiskit

Let's see Simon's algorithm in action using Qiskit. First, we will walk through creating a Simon circuit by building our own oracle that we know works for our example. Then we will take a look at a Qiskit package subsidiary that will automatically generate the appropriate oracle for a given bit string.

First, we need to include the necessary imports, then create a function to build the circuit used in Simon's algorithm. This function below will create a Simon circuit for a given bit string s with length n. We know the oracle we created is appropriate for the given bit string $s = 101$ that we will use for this example.

```
# import basic quantum circuits, simulators, and plot tools
from qiskit import Aer, QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.visualization import plot_histogram

def simon_circuit(n):
    """Construct a circuit to solve Simon's problem for
    a given dimension

    :param n: The dimension of the domain of the oracle
    :return: A qiskit QuantumCircuit object which when
    run will solve the problem
    """
    # Create a circuit with two quantum registers of 'n'  qubits,
    # register q and register r, and one classical register of 'n' bits
    # Building the registers individually, while not
    # required, will make future steps clearer
    q = QuantumRegister(n, 'q')
    r = QuantumRegister(n, 'r')
    c = ClassicalRegister(n)
    circuit = QuantumCircuit(q, r, c)

    # Apply a Hadamard gate to all the qubits in the first register
    circuit.h(q)
    circuit.barrier()

    # Apply Simon's oracle specific to our example
    circuit.cx(q, r)
    circuit.cx(q[0],r[0])
    circuit.cx(q[0],r[2])
    circuit.barrier()

    # Apply a Hadamard gate to all the qubits in the first register again
    circuit.h(q)

    # Measure the qubits of first register and return the completed circuit
    circuit.measure(q, c)

    return circuit
```

Now that we've written this function, let's take a look at the circuit it generates when we input $s = 101$.

```
# Define the starting bitstring and construct the circuit
s = '101'
circuit = simon_circuit(len(s))
display(circuit.draw('mpl'))
```

This should output the circuit below.

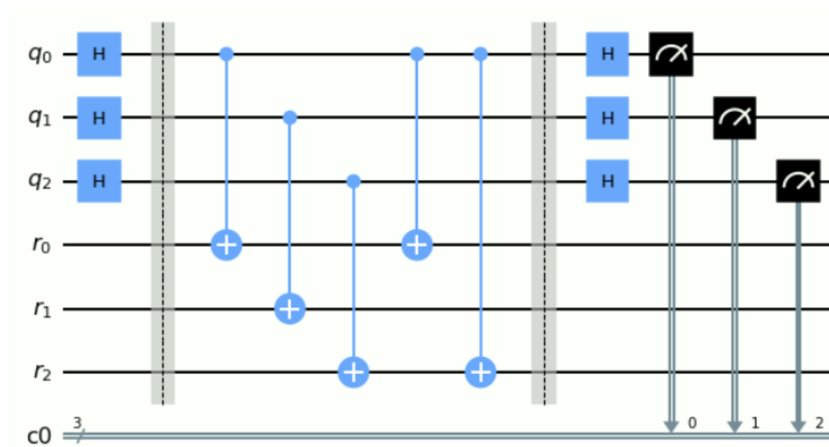


Figure 4.5: The circuit generated for $n = 3$. The oracle is applied between the two barriers.

Now that we have our circuit, we can run some simulations and view the results. Afterwards, we can compare the simulations to the numerical solution.

```
# Run the simulator 100 times and view the results
aer_sim = Aer.get_backend('aer_simulator')
results = aer_sim.run(circuit, shots=1024).result()
plot_histogram(results.get_counts())
```

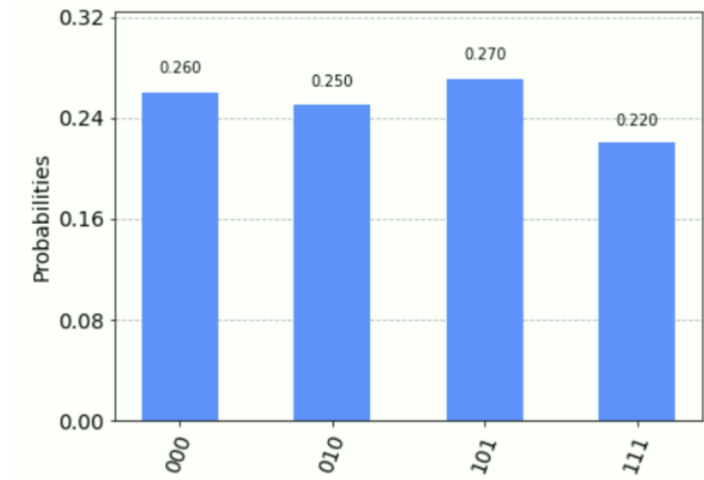


Figure 4.6: Results from running Simon's algorithm 1024 times with $s = 101$.

We can verify these results by calculating the dot product of s and z.

```
# Calculate the dot product of the results
def sdotz(s, z):
    accum = 0
    for i in range(len(s)):
        accum += int(s[i]) * int(z[i])
    return (accum % 2)

for z in results.get_counts():
    print( '{} . {} = {} (mod 2)'.format(s, z, sdotz(s,z)) )
```

For this example, this snippet should produce the following output.

```
101 . 101 = 0 (mod 2)
101 . 111 = 0 (mod 2)
101 . 010 = 0 (mod 2)
101 . 000 = 0 (mod 2)
```

Figure 4.7: Output showing the dot product of the original bit string and the results from Simon's algorithm.

From this calculation, we can verify that the dot products of s and all output values z are 0.

The oracle we used above won't necessarily work for any other inputs. Luckily, the `qiskit_textbook` package has a function which generates Simon's oracle for a given s. In order to install `qiskit_textbook`, activate your conda environment and run the command below.

```
pip install git+https://github.com/qiskit-community/qiskit-textbook.git
#subdirectory=qiskit-textbook-src
```

Once you have installed `qiskit_textbook`, then you can add an additional import and change our `simon_circuit` function.

```python
# Import simon_oracle
from qiskit_textbook.tools import simon_oracle

def qiskit_simon_circuit(s,n):
    """Construct a circuit to solve Simon's problem for a
    given dimension using the Qiskit simon_oracle function.

    :param s: A hidden binary string
    :param n: The dimension of the domain of the oracle
    :return: A qiskit QuantumCircuit object which when run
    will solve the problem
    """

    # Create a circuit with two quantum registers of 'n'
    # qubits, register q and register r, and one classical
    # register of 'n' bits
    circuit = QuantumCircuit(n*2, n)

    # Apply a Hadamard gate to all the qubits in the
    # first register
    circuit.h(range(n))
    circuit.barrier()

    # Apply Simon's oracle using Qiskit's simon_oracle
    circuit += simon_oracle(s)
    circuit.barrier()

    # Apply a Hadamard gate to all the qubits in the
    # first register
    circuit.h(range(n))

    # Measure the qubits of first register and return
    # the completed circuit
    circuit.measure(range(n), range(n))

    return circuit
```

Now we can try inputting another bit string and see how the circuit changes. For example, we can use $s = 111$.

```python
# Define the starting bit string and construct the circuit
s = '111'
circuit = qiskit_simon_circuit(s,len(s))
display(circuit.draw('mpl'))
```

The resulting circuit below is different from the one above! The `simon_oracle` function built a different oracle for the different input value.

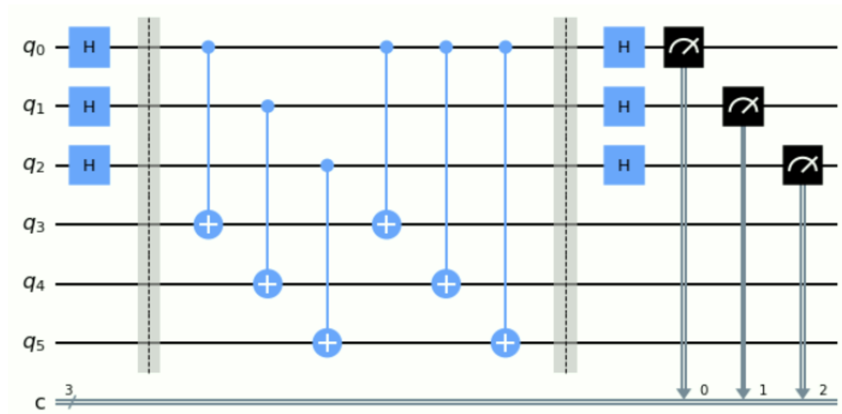


Figure 4.8: The circuit generated for $s = 111$. The oracle is applied between the two barriers.

Now we can run some simulations and view the results.

```python
# Run the simulator 100 times and view the results
aer_sim = Aer.get_backend('aer_simulator')
results = aer_sim.run(circuit, shots=1024).result()
plot_histogram(results.get_counts())
```

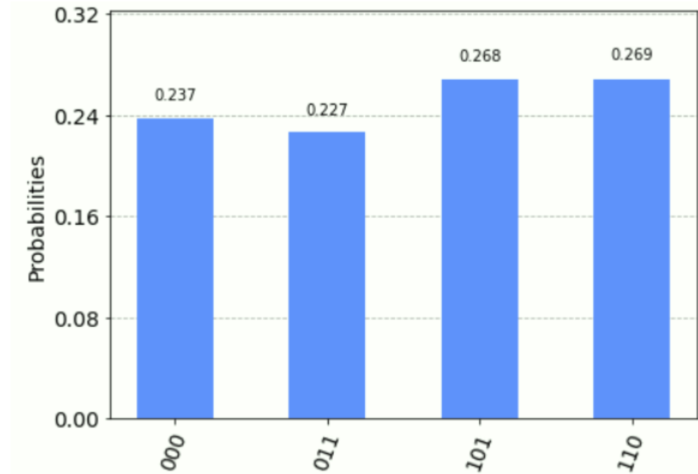


Figure 4.9: Results from running Simon's algorithm 1024 times with $s = 101$.

If we compare the output with the output of our first example, we can see the results are different. We can verify our algorithm worked by calculating the dot product of s and all possible z values using our *sdotx(s,z)* function from before. Looping through all of our solutions will produce the following output.

```
111 . 110 = 0 (mod 2)
111 . 000 = 0 (mod 2)
111 . 011 = 0 (mod 2)
111 . 101 = 0 (mod 2)
```

Figure 4.10: Output showing the dot product of the original bit string and the results from Simon's algorithm.

This gives evidence that our algorithm works. As an exercise, run the above code to verify you get the correct results, then try the algorithm with different s values.

Chapter 5

Shor's Algorithm

Inspired by Simon's algorithm, Peter Shor first developed an algorithm to find the prime factors of an integer in 1994 [26]. Since then, many have offered improvements to Shor's original algorithm (e.g. [28]), but the core has remained largely unchanged. Shor's algorithm is perhaps the most impactful quantum algorithm yet discovered, not because it is immediately implementable or solves a problem of wide industrial applicability, but because of its implied threat to cybersecurity and global commerce. Since the algorithm could hypothetically be employed to break any key for the most widely used public-key encryption schemes—RSA, elliptic curve, and ElGamal—governments worldwide have taken notice and this implication seems to be a common one in justifying further research and investment. To be clear, the threat seems a long way off based on our knowledge of the unclassified implementations. In Chapter 1, all the machines we listed handled fewer than 200 qubits, while a less-than-clever implementation of the factoring algorithm would require many millions of high-fidelity qubits. A recent study [29] by Gidney and Ekerå, however, estimated that one could break an RSA key of 2048 bits in just eight hours with 20 million noisy qubits. It is not out of the question that further efficiencies can be found in the coming years.

5.1 The Problem

Suppose we wish to factor a large integer N into, say, $N = M_1 M_2$ with $1 < M_1 \leq M_2 < N$. A well-established classical approach is to find non-trivial solutions a to $a^2 = 1 \pmod N$ (We will discuss this in §5.5.1). This, in turn, becomes feasible if we can efficiently compute the order of an element in the group $\mathbb{Z}_N^*$: if $x^r = 1$ in $\mathbb{Z}_N$ and r happens to be even, then $a = x^{r/2}$ squares to one. We will show that a randomly chosen element of $\mathbb{Z}_N^*$ has a reasonable chance of having even order. So the problem to attack is efficient computation of the order of an element.

But this can be reduced to the Hidden Abelian Subgroup Problem (HASP) as follows. Suppose $x \in \mathbb{Z}_N$ with $\gcd(x, N) = 1$. Define $F : \mathbb{Z}_{\phi(N)} \to \mathbb{Z}_N^*$ via $F(s) = x^s \bmod N$. Then $F(s) = F(t)$ if and only if $F(t - s) = 1$ and, with $G = \mathbb{Z}_{\phi(N)}$, the subgroup we seek a generator for is

$$H = \{ \ s \ \mid \ x^s = 1 \bmod N \ \} = \langle r \rangle$$

where r is order of x in the group $\mathbb{Z}_N^*$. The cosets of H in G are $s + \langle r \rangle = F^{-1}(F(s))$ so our function F satisfies the conditions for the input to the HASP.

5.2 Discrete Fourier Transform

Before we get to Shor's algorithm, we need to understand the discrete Fourier transform and the quantum Fourier transform. The discrete Fourier transform (DFT) takes a vector x of length N—often N values evenly spaced from 0 to 2π—and maps it to a vector y, defined by

$$y_k = \sum_{n=0}^{N-1} x_n \omega_N^{-kn}$$

where $\omega_N = e^{\frac{2\pi i}{N}}$ is a primitive complex N-th root of unity. The inverse discrete Fourier transform (IDFT) is given by

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} y_k \omega_N^{kn}.$$

Shor's algorithm utilizes the quantum analogue of the inverse discrete Fourier transform.

5.3 Quantum Fourier Transform

The discrete Fourier transform (DFT) converts a finite sequence of evenly-spaced samples of a function into a sequence of equally-spaced samples of another complex valued function, known as the discrete-time Fourier transform (DTFT). The inverse discrete Fourier transform uses the DTFT samples as coefficients of complex sinusoids at the corresponding frequencies. The quantum Fourier transform (QFT) is the quantum application of the inverse discrete Fourier transform. It is applied as a linear transformation to quantum bits.

The QFT takes a quantum state $|x\rangle = \sum_{k=0}^{N-1} x_k \, |k\rangle$ and maps it to a state $\sum_{k=0}^{N-1} y_k \, |k\rangle$, where each y_k is defined by

$$y_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \omega_N^{nk}$$

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

Notice that when $|x\rangle$ is a computational basis state, $x_n = 1$ only when $n = x$ and $x_n = 0$ otherwise, so the QFT can be expressed as

$$QFT_N |x\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{xk} |k\rangle .$$

Thus, the matrix representing this transformation can be expressed as a sum of N^2 rank-one matrices:

$$QFT_N = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \omega_N^{jk} |k\rangle \langle j| .$$

For simplicity, we take $N = 2^\ell$ and we look at how QFT_N acts on a computational basis vector $|j\rangle$:

$$QFT_N |j\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{jk} |k\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi ijk/N} |k\rangle$$

$$= \frac{1}{\sqrt{N}} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} \cdots \sum_{k_\ell=0}^{1} e^{2\pi ijk/N} |k_1\rangle \otimes |k_2\rangle \otimes \cdots \otimes |k_\ell\rangle$$

$$= \frac{1}{\sqrt{N}} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} \cdots \sum_{k_\ell=0}^{1} e^{2\pi ij\left(\frac{k_1}{2}+\frac{k_2}{4}+\cdots+\frac{k_\ell}{2^\ell}\right)} |k_1\rangle \otimes |k_2\rangle \otimes \cdots \otimes |k_\ell\rangle$$

$$= \frac{1}{\sqrt{N}} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} \cdots \sum_{k_\ell=0}^{1} e^{2\pi ij\frac{k_1}{2}} |k_1\rangle \otimes e^{2\pi ij\frac{k_2}{4}} |k_2\rangle \otimes \cdots \otimes e^{2\pi ij\frac{k_\ell}{2^\ell}} |k_\ell\rangle$$

$$= \frac{1}{\sqrt{2}} \left(|0\rangle + e^{\frac{2\pi i}{2}j} |1\rangle\right) \otimes \frac{1}{\sqrt{2}} \left(|0\rangle + e^{\frac{2\pi i}{4}j} |1\rangle\right) \otimes \cdots \otimes \frac{1}{\sqrt{2}} \left(|0\rangle + e^{\frac{2\pi i}{2^\ell}j} |1\rangle\right)$$

where we have written $k = \sum k_N 2^{\ell+1-N}$ so that k/N expands as given.
Now we look at the tth qubit in this register, $1 \leq t \leq \ell$. We want it to be in the state:

$$|\psi_t\rangle = \frac{1}{\sqrt{2}} \left(|0\rangle + e^{\frac{2\pi i}{2^t}j} |1\rangle\right)$$

$$= \frac{1}{\sqrt{2}} \left(|0\rangle + e^{\frac{2\pi i}{2^t}(2^{\ell-1}j_1+2^{\ell-2}j_2+\cdots+j_\ell)} |1\rangle\right)$$

$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta_1} \end{bmatrix}^{j_1} \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta_2} \end{bmatrix}^{j_2} \cdots \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta_\ell} \end{bmatrix}^{j_\ell} (|0\rangle + |1\rangle)$$

where we have expanded $j = j_1 j_2 \cdots j_\ell = 2^{\ell-1}j_1 + \cdots + 2^0 j_\ell$ and the phase is rotated by angles:

$$\theta_1 = \pi 2^{\ell-t}, \theta_2 = \pi 2^{\ell-t-1}, \ldots, \theta_\ell = \pi 2^{1-t}.$$

Notice that we can ignore all rotations by integer multiples of 2π, that is, θ_k when $k < \ell - t + 1$.

We are almost finished. This evolution of the tth qubit, assuming initial state $|0\rangle$, is represented by a Hadamard gate followed by a sequence of $\ell - t$ controlled phase rotation gates $R_{2\pi/2^k}$, abbreviated as R_k, for $k = 2, 3, \ldots, \ell - t + 1$. Note that $R_k := \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}$ can be built from a single CNOT gate and single-qubit gates:

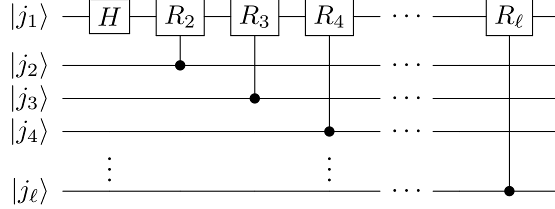


Figure 5.1: The first few gates of the QFT

At the end of the first part of our circuit, the qubit represented by the first wire is in the state

$$|0\rangle + e^{2\pi i(\frac{j_1}{2} + \frac{j_2}{4} + \frac{j_3}{8} + \cdots + \frac{j_\ell}{2^\ell})} |1\rangle ,$$

and, continuing in this fashion, we reach state:

$$\frac{1}{\sqrt{2^\ell}} \left(|0\rangle + e^{2\pi i(\frac{j_1}{2} + \frac{j_2}{4} + \frac{j_3}{8} + \cdots + \frac{j_\ell}{2^\ell})} |1\rangle \right) \otimes \left(|0\rangle + e^{2\pi i(\frac{j_2}{2} + \frac{j_3}{4} + \cdots + \frac{j_\ell}{2^{\ell-1}})} |1\rangle \right) \otimes \cdots \otimes \left(|0\rangle + e^{2\pi i(\frac{j_\ell}{2})} |1\rangle \right).$$

This is almost what we need, except the qubits are in the wrong order. We need to swap the first and last qubits, then the second and second-to-last, etc. which gives us

$$\frac{1}{\sqrt{2^\ell}} \left(|0\rangle + e^{2\pi i(\frac{j_\ell}{2})} |1\rangle \right) \otimes \left(|0\rangle + e^{2\pi i(\frac{j_{\ell-1}}{2} + \frac{j_\ell}{4})} |1\rangle \right) \otimes \cdots \otimes \left(|0\rangle + e^{2\pi i(\frac{j_1}{2} + \frac{j_2}{4} + \frac{j_3}{8} + \cdots + \frac{j_\ell}{2^\ell})} |1\rangle \right).$$

This is the result of applying the QFT as a linear transformation to a state $|j\rangle$.

5.4 Phase Estimation

The last piece we need for Shor's algorithm is quantum phase estimation. Quantum phase estimation is a way of estimating the eigenvalue corresponding to an eigenvector $|\psi\rangle$ of a unitary operator U. If we write this eigenvalue as $e^{2\pi i\theta}$, then the parameter θ is called the *phase*. We can write the eigenvalue in this form because U is a unitary operator, so its eigenvalues must have an absolute value of 1.

On a high level, phase estimation involves putting the first register in superposition using Hadamard gates, applying controlled U gates to the second register, then applying the inverse quantum Fourier transform to the first register and measuring the results. This can be represented by the following circuit:

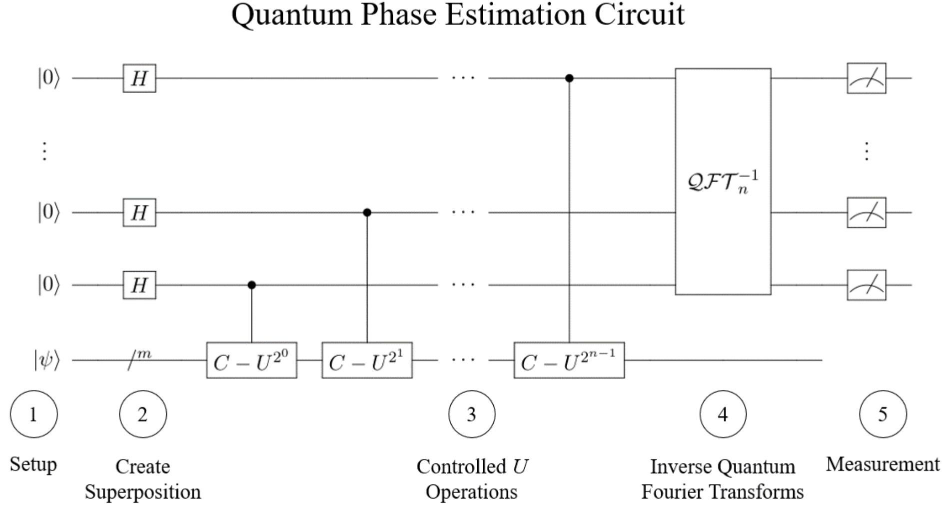


Figure 5.2: Components of a quantum phase estimation circuit

Let U be a unitary operator which operates on m qubits. Let U have eigenvector $|\psi\rangle$ with corresponding eigenvalue $e^{2\pi i\theta}$.

Our circuit uses two registers, the first with n qubits and the second with m qubits. The initial state is $|0^n\rangle|\psi\rangle$. We first apply a Hadamard gate to the first register, resulting in the state:

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

Now we apply controlled U gates to the second register with control qubits in the first register. We use the kth qubit (counting the qubits from right to left starting with 0, as Qiskit does) in the first register as the control for a U^{2^k} gate on the second register. The U^{2^k} gate maps $|\phi\rangle \to e^{2\pi i 2^k\theta}|\psi\rangle$, so the controlled gate maps

$$\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\otimes|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle|\psi\rangle+|1\rangle|\psi\rangle) \to \frac{1}{\sqrt{2}}(|0\rangle|\psi\rangle+e^{2\pi i 2^k\theta}|1\rangle|\psi\rangle) = \frac{1}{\sqrt{2}}(|0\rangle+e^{2\pi i 2^k\theta}|1\rangle)\otimes|\psi\rangle .$$

This stores the phase θ in the control qubit and leaves the second register unchanged. After applying all the controlled U^{2^k} gates, the first register is left in state:

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

Now we apply the inverse quantum Fourier transform to the first register. The inverse QFT maps a computational basis vector $|x\rangle$ to

$$\frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{-2\pi i x k/2^n} |k\rangle \, ,$$

so our state becomes

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} e^{2\pi i x \theta} \left(\frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{-2\pi i x k/2^n} |k\rangle \right) |\psi\rangle = \frac{1}{2^n} \sum_{x=0}^{2^n-1} \sum_{k=0}^{2^n-1} e^{2\pi i x(2^n\theta - k)/2^n} |k\rangle |\psi\rangle \, .$$

Finally, we perform a measurement on the first register. The probability of measuring a state $|k\rangle$ is

$$\left| \frac{1}{2^n} \sum_{x=0}^{2^n-1} e^{2\pi i x(2^n\theta - k)/2^n} \right|^2 .$$

Let $\delta = (2^n\theta - k)/2^n$. If $\delta = 0$ then this probability is 1. Otherwise, this becomes

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

This probability is higher the closer δ is to an integer $a \in \mathbb{Z}$, that is, when $\delta = \theta - k/2^n \approx a$. But θ and $k/2^n$ are both non-negative and less than 1, so it must be that $a = 0$. Thus,

$$\theta - k/2^n \approx 0 \implies \theta \approx k/2^n .$$

This means that k is likely to be close to a n-bit approximation for θ. We can try to determine the phase θ more precisely by using more qubits in the first register.

5.5 The Algorithm

Now that we have the building blocks necessary, we can construct the full algorithm. Phase estimation and the quantum Fourier transform are utilized in the quantum period finding subroutine. Shor's algorithm differs from the previously covered algorithms in that is has fairly sizable classical pre- and post-processing steps.

Shor's Algorithm for Factoring

Input: A non-prime number N

Output: Two factors of N

```
1  Repeat until factors found
2      Pick a number a, 1 < a < N
3      Compute g = gcd(a, N)
4      if g ≠ 1 then
5          return g, N/g
6      end
7
8      Find the order r of a ∈ (Z/NZ)*
9      if 2 | r and a^(r/2) ≢ −1 (mod N) then
10         p = gcd(a^(r/2) − 1, N)
11         q = gcd(a^(r/2) + 1, N)
12         return p, q
13     end
```

5.5.1 Classical Component

The key classical concept behind Shor's algorithm is finding a factor of N by looking at square roots of 1 modulo N. If a is a square root of 1 modulo N, then we have

$$\begin{aligned} a^2 &\equiv 1 \pmod N \\ a^2 - 1 &\equiv 0 \pmod N \\ (a-1)(a+1) &\equiv 0 \pmod N \end{aligned} \tag{5.1}$$

This implies that N divides $(a-1)(a+1)$. If we can find a value for a, we can compute $\gcd(a-1, N)$ to try to obtain a factor of N. There are only two cases in which this will not work: when $\gcd(a-1, N) = N$ and when $\gcd(a-1, N) = 1$.

If $\gcd(a-1, N) = N$, then N divides $(a-1)$, so

$$\begin{aligned} a - 1 &\equiv 0 \pmod N \\ a &\equiv 1 \pmod N \end{aligned}$$

As long as we choose $a \not\equiv 1 \pmod{N}$ we can avoid this case.

If $\gcd(a-1, N) = 1$, $a-1$ and N are coprime, so by the Extended Euclidean Algorithm there exist $u, v \in \mathbb{Z}$ such that

$$(a-1)u + Nv = 1.$$

Multiplying both sides by $a+1$, we find that

$$(a^2-1)u + (a+1)Nv = a+1.$$

We know that N divides $a^2 - 1$ from (5.1), so there exists $m \in \mathbb{Z}$ such that

$$Nmu + N(a+1)v = N(mu + (a+1)v) = a+1.$$

Therefore, N divides $a+1$, so

$$\begin{aligned} a+1 &\equiv 0 \pmod{N} \\ a &\equiv -1 \pmod{N}. \end{aligned}$$

Thus, we also have to avoid $a \equiv -1 \pmod{N}$.

If we avoid these values of a, we are guaranteed

$$1 < \gcd(a-1, N) < N$$

and

$$1 < \gcd(a+1, N) < N.$$

So both are factors of N. By the Chinese Remainder Theorem, an odd integer N with s prime divisors yields 2^s solutions to the equation $x^2 \equiv 1 \pmod{N}$.

We examine the *multiplicative group of integers modulo N* (often denoted $(\mathbb{Z}/N\mathbb{Z})^*$),

$$G = (\{n : \gcd(n, N) = 1, 0 < n < N\}, *)$$

with modular multiplication as its operation and where every element $g \in G$ is coprime to N, that is, $\gcd(g, N) = 1$. The order of this group is exactly $\phi(N)$, where ϕ is Euler's totient function.

Let $a \in G$. The order of a is the smallest integer r such that $a^r = e$ where e is the identity of the group. In $(\mathbb{Z}/N\mathbb{Z})^*$, this would be $a^r \equiv 1 \pmod{N}$, as $e = 1$.

Using the square root of 1 expansion trick above:

$$\begin{aligned} a^r &\equiv 1 \pmod{N} \\ a^r - 1 &\equiv 0 \pmod{N} \\ (a^{r/2}-1)(a^{r/2}+1) &\equiv 0 \pmod{N} \end{aligned}$$

Does this order grant us the desired factors? By definition of r, we cannot have $\gcd(a^{r/2}-1, N) = 1$, since that would imply that r is not the order of a, as $r/2 < r$. So we avoid the case where $\gcd(a^{r/2}-1, N) = 1$.

We must be able to divide r by 2, and we cannot have $a^{r/2} \equiv -1 \pmod{N}$. This is why we have the conditional at line 9 in Shor's algorithm. As long as those conditions are met, we will be given two factors.

Example: Given the number $N = 21$, we have the group

$$G = (\{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}, *)$$

with the operation being multiplication modulo 21.

The order of each element in G can be found easily since G is small:

1	2	4	5	8	10	11	13	16	17	19	20
1	6	3	6	2	6	6	2	3	6	6	2

Would it be bad to draw a number not in G as the randomly selected number a? No, it would be great! Each of these share a common factor with N, so computing the GCD in step 3 of the algorithm would immediately give an answer.

With larger semiprimes, though, the number of coprime elements will greatly outnumber any numbers with a common factor.

Looking at the numbers coprime to 21 and their orders again, we see that an order of 3 appears twice, 6 appears 4 times, and 2 appears 3 times. Since 3 is odd, we cannot take half of it, so it will not be usable in Shor's algorithm. We have to find an element with a different order.

Looking at $a = 5$ with $r = 6$, we have

$$\begin{aligned} 5^3 = 125 &\equiv 20 \pmod{21} \\ &\equiv -1 \pmod{21} \end{aligned}$$

Thus, 5 does not work.

Looking at $a = 8$ with $r = 2$, we have

$$\begin{aligned} 8^1 = 8 &\equiv 8 \pmod{21} \\ \gcd(7, 21) &= 7 \\ \gcd(9, 21) &= 3 \end{aligned}$$

Picking 8 would give us the desired answer. We can now calculate the rest of the possible a values:

$$
\begin{aligned}
10^3 &\equiv 13 \pmod{21}\\
11^3 &\equiv 8 \pmod{21}\\
13^1 &\equiv 13 \pmod{21}\\
17^3 &\equiv -1 \pmod{21}\\
19^3 &\equiv 13 \pmod{21}\\
20^1 &\equiv -1 \pmod{21}
\end{aligned}
$$

There are 5 elements which have a usable order out of 11 possible choices. But any of the elements not coprime to 21 would also have sufficed, so the odds of choosing a random number that would lead to a factor of 21 are $12/19 \approx 63\%$. The probability of not choosing a correct number by the fifth try is approximately $(1-.63)^5$, less than 1%.

We use custom quantum circuits to find the period r of $f(x) = a^x \mod N$ for integer N. The period of f is the smallest non-zero integer r such that $f(x+r) = f(x)$. It follows that $a^r \mod N = 1$. Shor's solution is to use the quantum circuit below to estimate the resulting phase from the circuit shown below. This period finding algorithm is an example of the hidden subgroup problem with $G = \mathbb{Z}_{\phi(N)}$ and $H = \langle r \rangle$.

5.5.2 Quantum Component

We choose Q such that $Q = 2^q$ and $N^2 \leq Q < 2N^2$. This implies that $\frac{Q}{r} > N$, since $r < N$. Next we need input and output registers. The input register needs to hold a superposition of values from 0 to $Q-1$, and therefore has q qubits. This may seem like twice as many qubits as we need, but it guarantees that there are at least N different values of x which map to the same $f(x)$, even when r approaches $\frac{N}{2}$. The output register needs to hold values ranging from 0 to $N-1$, so it has n qubits, where n is the smallest integer such that $N \leq 2^n$.

We first initialize each qubit in both registers to the state $|0\rangle$. We next apply a Hadamard transform to the entire register to obtain the state :

$$|\Phi_0\rangle = \frac{1}{\sqrt{Q}} \sum_{x=0}^{Q-1} |x\rangle |0^n\rangle .$$

Let there be a unitary transformation U_f such that

$$U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle .$$

When applied to the register above, this results in the state

$$|\Phi_1\rangle = U_f |\Phi_0\rangle = U_f \left[\frac{1}{\sqrt{Q}} \sum_{x=0}^{Q-1} |x\rangle |0^n\rangle \right] = \frac{1}{\sqrt{Q}} \sum_{x=0}^{Q-1} |x\rangle |f(x)\rangle .$$

This is still a superposition of Q states, but we have now entangled the q input qubits and n output qubits.

Apply the Quantum Fourier Transform

To find the period r of $f(x) = a^x \mod N$, we need to apply the inverse quantum Fourier transform to the input register. Recall that the quantum Fourier transform applied to a basis state $|x\rangle$ is defined by

$$QFT_N |x\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{xk} |k\rangle$$

where $\omega = e^{\frac{2\pi i}{N}}$ is a primitive N-th root of unity. But here we increase precision from n qubits to $q \approx 2^n$ qubits.

Let's instead choose the primitive Q-th root of unity $\omega = e^{\frac{2\pi i}{Q}}$ and apply the QFT to the first register of our state, $|\Phi_1\rangle$:

$$(QFT_Q \otimes I_n) |\Phi_1\rangle = \frac{1}{Q} \sum_{x=0}^{Q-1} \sum_{y=0}^{Q-1} \omega^{xy} |y\rangle |f(x)\rangle .$$

Reordering this sum, we obtain

$$\frac{1}{Q} \sum_{z=0}^{N-1} \sum_{y=0}^{Q-1} \sum_{\substack{0 \le x < Q \\ f(x)=z}} \omega^{xy} |y\rangle |z\rangle .$$

Perform a Measurement

If we take a measurement, we obtain some outcome y in the input register and some z in the output register. The probability of measuring a given state $|y\rangle |z\rangle$ is

$$\left| \frac{1}{Q} \sum_{\substack{0 \le x < Q \\ f(x)=z}} \omega^{xy} \right|^2 .$$

Let

- r be the period of f,
- x_0 be the smallest value of $x \in \{0, .., Q-1\}$ such that $f(x) = z$,
- $m = \left\lfloor \frac{Q - x_0 - 1}{r} \right\rfloor + 1$ be the number of distinct input values x such that $f(x) = z$,
- b index the values of x running from 0 to $m-1$, so that $0 < x_0 + rb < Q$.

Then the sum above can be rewritten as

$$\left| \frac{1}{Q} \sum_{\substack{0 \le x < Q \\ f(x)=z}} \omega^{xy} \right|^2 = \frac{1}{Q^2} \left| \sum_{b=0}^{m-1} \omega^{(x_0 + rb)y} \right|^2 = \frac{1}{Q^2} \left| \omega^{x_0 y} \right|^2 \left| \sum_{b=0}^{m-1} \omega^{rby} \right|^2 .$$

As $\omega^{x_0 y}$ is unit length, this simplifies to

$$\frac{1}{Q^2}\left|\sum_{b=0}^{m-1}\omega^{rby}\right|^2 = \frac{1}{Q^2}\left|\frac{\omega^{mry}-1}{\omega^{ry}-1}\right|^2 = \frac{1}{Q^2}\left|\frac{e^{2\pi imry/Q}-1}{e^{2\pi iry/Q}-1}\right|^2.$$

We use the fact that $\left|e^{2ix}-1\right|^2 = 4\sin^2(x)$ to further simplify this as

$$\frac{1}{Q^2}\left|\frac{e^{2\pi imry/Q}-1}{e^{2\pi iry/Q}-1}\right|^2 = \frac{1}{Q^2}\frac{\sin^2(\pi mry/Q)}{\sin^2(\pi ry/Q)}.$$

This probability is higher the closer $\frac{ry}{Q}$ is to an integer $k \in \mathbb{Z}$, and rearranging these terms gives us

$$\frac{y}{Q} \approx \frac{k}{r}.$$

We know the values y and Q. Since $Q > N^2$, we propose that there is at most one fraction $\frac{k}{s}$ with $s < N$ satisfying

$$\left|\frac{y}{Q} - \frac{k}{s}\right| \leq \frac{1}{2Q}.$$

To prove this, we argue by contradiction. Suppose there are two fractions $\frac{k_1}{s_1}$ and $\frac{k_2}{s_2}$ that both satisfy the inequality. Then,

$$\left|\frac{k_1}{s_1} - \frac{k_2}{s_2}\right| = \frac{|k_1 s_2 - k_2 s_2|}{s_1 s_2} \geq \frac{1}{s_1 s_2} > \frac{1}{Q}.$$

But $\frac{k_1}{s_1}$ and $\frac{k_2}{s_2}$ are both within $\frac{1}{2Q}$ of $\frac{y}{Q}$, so this is a contradiction. Therefore, there is only one fraction $\frac{k}{s}$ where $s < N$ satisfying this inequality. Thus, we can use the continued fraction expansion of $\frac{y}{Q}$ to find the unique fraction $\frac{k}{s}$.

If $\frac{k}{s}$ is irreducible, then s is very likely to be either the period r or a factor of it. To verify our answer, we can check classically if $a^s \equiv 1 \pmod{N}$. If so, $r = s$ and we are done. If not, we can classically obtain more candidates for r using either multiples of s or other s such that $\frac{k}{s}$ is near $\frac{y}{Q}$. If no candidates satisfy the equality, then we must run the phase estimation subroutine again.

5.5.3 Runtime Analysis

Now that we can find s, we can analyze the runtime of this subroutine. What is the probability that we find a s value that allows us to recover r, and what is the expected number of iterations required to do so?

There are $\phi(r)$ possible values of k relatively prime to r, where ϕ is Euler's totient function. Each of these fractions $\frac{k}{r}$ is close to one fraction $\frac{y}{Q}$ with $\left|\frac{y}{Q} - \frac{k}{r}\right| \leq \frac{1}{2Q}$. There are also r possible values of z for a given y, since r is the period of f. Thus, there are $r\phi(r)$ states $|y\rangle|z\rangle$ which would enable us to obtain r. Each of these states occurs with probability at least $\frac{1}{3r^2}$, so we obtain r with probability at least $\frac{\phi(r)}{3r}$.

To determine the number of iterations required to find r, we will use the geometric distribution. The expected value of an event with success probability p is $\frac{1-p}{p}$. So the expected number of iterations is

$$\frac{1-\frac{\phi(r)}{3r}}{\frac{\phi(r)}{3r}} = \frac{3r-\phi(r)}{\phi(r)} = \frac{3r}{\phi(r)} - 1.$$

5.6 Qiskit

Let's see an example of Shor's algorithm in action. For this example we'll solve the period finding problem for $a = 7$ and $N = 15$. For simplicity we'll define $U|y\rangle = |ay \mod 15\rangle$ without explanation. Just as with phase estimation $|y\rangle \mapsto |a^x y\rangle$ can be achieved by applying a logarithmic number of oracles of the form $|y\rangle \mapsto |a^{2^j} y\rangle$. Then to create U^x, we'll repeat the circuit x times. To start, we need to import some packages and define some helper functions.

```python
# import quantum circuits, simulators, mathematics, and plot tools
import matplotlib.pyplot as plt
import numpy as np
from qiskit import QuantumCircuit, Aer, transpile, assemble
from qiskit.visualization import plot_histogram
from math import gcd
from numpy.random import randint
import pandas as pd
from fractions import Fraction

# Returns a controlled-U gate for a mod 15, repeated power times
def c_amod15(a, power):
    # Controlled multiplication by a mod 15
    if a not in [2,4,7,8,11,13]:
        raise ValueError("'a' must be 2,4,7,8,11 or 13")
    U = QuantumCircuit(4)
    for iteration in range(power):
        if a in [2,13]:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
        if a in [7,8]:
            U.swap(2,3)
            U.swap(1,2)
            U.swap(0,1)
        if a in [4,11]:
            U.swap(1,3)
            U.swap(0,2)
        if a in [7,11,13]:
            for q in range(4):
                U.x(q)
    U = U.to_gate()
    U.name = "%i^%i mod 15" % (a, power)
    c_U = U.control()
    return c_U

# Apply the n-qubit QFTdagger (inverse quantum Fourier transform)
# to the first n qubits in the circuit
def qft_dagger(n):
    qc = QuantumCircuit(n)
    # Don't forget the Swaps!
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cp(-np.pi/float(2**(j-m)), m, j)
        qc.h(j)
    qc.name = "QFT†"
    return qc
```

With these building blocks defined, we can write the following function to produce a circuit to implement Shor's algorithm.

```python
def shor(n):
    """Construct a circuit to solve Shor's problem with an n
    qubit input register

    :param n: The number of qubits in the input register
    :return: A qiskit QuantumCircuit object which when run
    will solve the problem
    """

    # Create QuantumCircuit with an n qubit input register
    # plus 4 qubits for U to act on
    qc = QuantumCircuit(n + 4, n)

    # Initialize input register and output register
    for q in range(n):
        qc.h(q)
    qc.x(n)

    # Apply controlled-U operations
    for q in range(n):
        qc.append(c_amod15(a, 2**q),
                 [q] + [n+i for i in range(4)])

    # Apply inverse QFT
    qc.append(qft_dagger(n), range(n))

    # Measure and return circuit
    qc.measure(range(n), range(n))
    return qc
```

Now that we have built our function, let's see it in action using our example.

```python
# Specify variables
n = 8
a = 7

# Create and view Shor's Algorithm for our example
qc=shor(n)
display(qc.draw('mpl', fold=-1)) # fold=-1 keeps the circuit
# on one line
```

This should draw the following circuit.

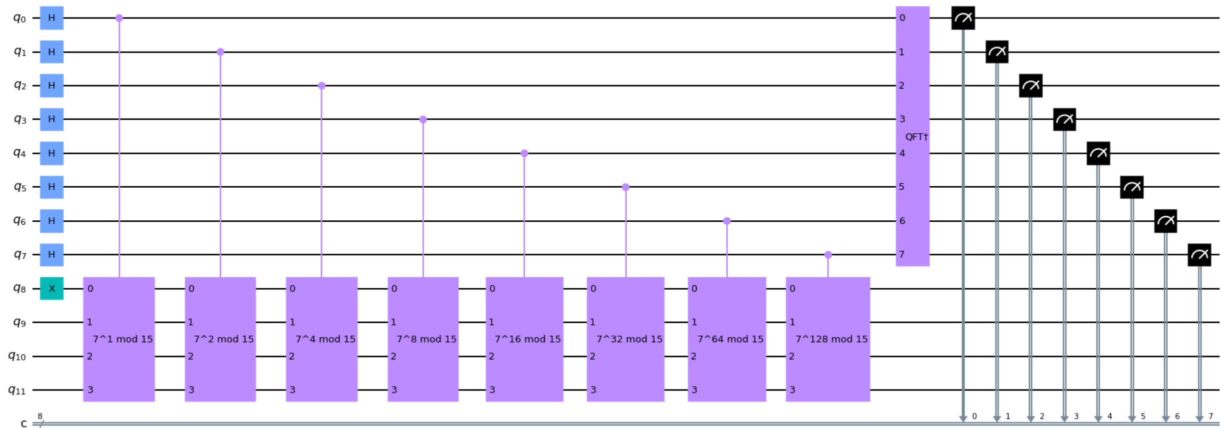


Figure 5.3: Circuit generated to solve the above example using Shor's algorithm.

Now we can run some simulations using this example.

```
# Run the simulator 1024 times and view the results
aer_sim = Aer.get_backend('aer_simulator')
t_qc = transpile(qc, aer_sim)
qobj = assemble(t_qc)
results = aer_sim.run(qobj).result()
plot_histogram(results.get_counts())
```

Running this snippet yields the following results.

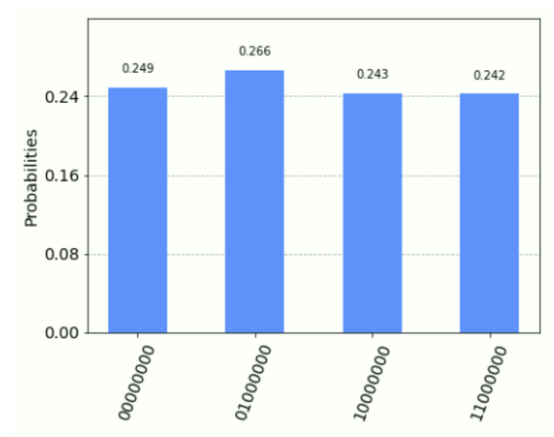


Figure 5.4: Circuit generated to run the quantum subroutine of Shor's algorithm with the above parameters.

Now we can look at these results in terms of phases instead.

```python
rows, measured_phases = [], []
for output in results.get_counts():
    decimal = int(output, 2)  # Convert (base 2) string to
    # decimal
    phase = decimal/(2**n)  # Find corresponding
    # eigenvalue
    measured_phases.append(phase)
    # Add these values to the rows in our table:
    rows.append([f"{output}(bin) = {decimal:>3}(dec)",
                 f"{decimal}/{2**n} = {phase:.2f}"])
# Print the rows in a table
headers=["Register Output", "Phase"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

```
             Register Output           Phase
0  10000000(bin) = 128(dec)  128/256 = 0.50
1  11000000(bin) = 192(dec)  192/256 = 0.75
2  01000000(bin) =  64(dec)   64/256 = 0.25
3  00000000(bin) =   0(dec)    0/256 = 0.00
```

Figure 5.5: Measured eigenvalues of the results of Shor's Algorithm.

Now that we have our measured phases, we can solve for our final result using the continued fraction expansion.

```
   Phase Fraction  Guess for r
0   0.50      1/2            2
1   0.75      3/4            4
2   0.25      1/4            4
3   0.00      0/1            1
```

Figure 5.6: Possible r values from continued fraction expansion on the results of Shor's Algorithm.

We can see that two of these produced the correct result, $r = 4$. This illustrates what we discussed above: some iterations of Shor's Algorithm do not produce the desired results. The way we choose to solve this problem is to repeat the algorithm. As an exercise, repeat the algorithm multiple times and compare the results. Then, try out the circuit for $a = 2, 4, 8, 11$, and 13 and observe the results.

5.7 The Discrete Logarithm Problem

The novelty of Shor's algorithm comes from its ability to solve the hidden abelian subgroup problem (HASP). Any problem reducible to the hidden abelian subgroup problem also can be solved with Shor's algorithm

when proper modifications are made. One such problem is the discrete logarithm problem (DLP).

Discrete Logarithm Problem

Let p be a prime number, and g a generator for $\mathbb{Z}_p^*$. Given $a \in \mathbb{Z}_p^*$, find x such that $g^x \equiv a \pmod{p}$.

Note that the input for this problem, (p, g, a), consists of $3 \log_2 p$ bits. Since p is prime, $\mathbb{Z}_p^*$ is cyclic. Every element in $\mathbb{Z}_p^*$ is uniquely expressed by g to some power k. To reduce this to the HASP, we work with G $= \mathbb{Z}_{p-1} \times \mathbb{Z}_{p-1}$ (coordinatewise addition modulo $p-1$) and consider

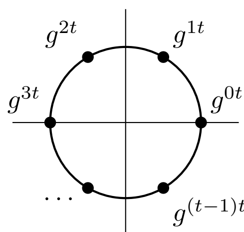
$$f : G \to \mathbb{Z}_p^* \quad \text{via} \quad f : (\alpha, \beta) = g^\alpha a^{-\beta}.$$

Whenever $g^x = a$ and $\alpha = x\beta$, we have $g^{x\beta}a^{-\beta} = (g^x)^\beta a^{-\beta} = a^\beta a^{-\beta} = 1$. The hidden subgroup is generated by $(x, 1)$, and any other pair that produces a matching coset to $(x, 1)$ will retain the property of $\alpha = x\beta$. Finding a generator (α, β) for this hidden subgroup will give an answer to the discrete log problem as $x = \frac{\alpha}{\beta}$.

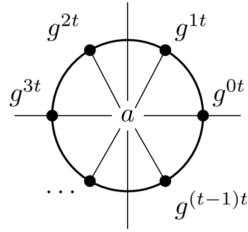
5.7.1 Baby-Step Giant-Step

It's worth noting that the discrete logarithm problem has an elementary exponential time classical solution for finite abelian groups called the baby-step giant-step algorithm [30]. While exhaustive search requires $\mathcal{O}(p) = \mathcal{O}(2^{\log_2 p})$ steps, this approach cuts the work to $\mathcal{O}(\sqrt{p})$ steps, a quadratic speedup but still exponential in the size of the input.

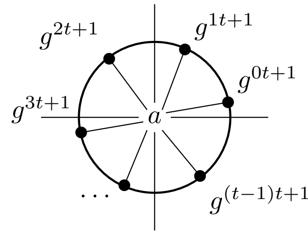
We first compute many powers of g along the unit circle as a part of the giant step. To do this, we define $t := \lceil \sqrt{n}\, \rceil$, where $n = p - 1$ is the order of the group $\mathbb{Z}_p^*$. We compute g^{kt} for integers $0 \leq k < t$:



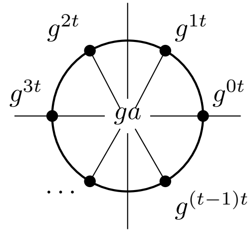
We check to see if a matches any of these values. If it does, we have found a value x such that $g^x \equiv a \pmod{p}$, and we are done:



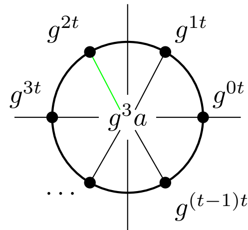
Otherwise, we begin to compute the baby steps. The element a must fall in one of the gaps, so we shift the values we check:



However, instead of recomputing every power g^{kt} along the unit circle, we can instead multiply a by g and then check against the values we already have:



We repeat this process of multiplying our inner value by g until we have a match:



Suppose we find that $g^3a = g^{2t}$, then $a = g^{-3}g^{2t} = g^{2t-3}$. So the value x is $2t-3$. Since $n \leq t^2$, the Division Algorithm gives $x < n$ as $x = tq + r$, with $0 \leq q < t$ and $0 \leq r < t$.

The running time and space complexity of this algorithm is $\mathcal{O}(\sqrt{n})$, where n is the order of the group. There are two other algorithms of interest in specific cases. Pollard's [31] rho algorithm for logarithms can

reduce the space complexity for cyclic groups (which $\mathbb{Z}_p^*$ is). If p is not prime, then the Pohlig–Hellman algorithm [32] can reduce the running complexity. But all of these have worst-case exponential running time $\mathcal{O}(\sqrt{p})$.

It is a worthwhile question to ask why the discrete log problem is considered difficult enough to have a cryptographic scheme designed around it when a polynomial time algorithm exists which solves it. ElGamal encryption suggests key lengths of 1024 bits or larger, so to use the baby-step giant-step algorithm, 2^{512} numbers need to be precomputed and stored. This requirement alone is enough to guarantee security from attacks from an average computer, whereas the key length may require 2048 bits in order to protect against more powerful computing hardware. The baby-step giant-step algorithm halves the security of the length of the key, but as long as it is computationally feasible to use larger length keys, the cryptographic scheme is feasible. In general, cryptographic schemes gauge security by the bit length of their private keys. A key of size n really has a search space of 2^n, so the baby-step giant-step algorithm finds solutions in $2^{n/2}$ time. Shor's algorithm runs in $\mathcal{O}\left(n^2 \log n \log\log n\right)$, which is indeed polynomial in the size, $3n$, of the input.

5.7.2 Overview of Shor's Algorithm for Discrete Logs

Shor's [33] algorithm for discrete logs follows a similar design paradigm to the factoring algorithm with usage of the QFT for solving the hidden abelian subgroup problem, and then classically processing the reduction steps. In 2016 [34] and in 2021 [35] Martin Ekerå made some modifications which improve the success rate to 99% depending on runtime tradeoffs in the quantum or classical step. The original algorithm for a general case of discrete logarithms follows the steps below.

1. First, pick a power of two, $Q = 2^q$, such that Q is close to p, i.e. $p \leq Q \leq 2p$.
2. Initialize three q-qubit registers to zero.
3. Apply Hadamard transformations to the first two registers.
4. Apply a unitary transformation U_f which maps $|x_1\rangle\,|x_2\rangle\,|y\rangle$ to $|x_1\rangle\,|x_2\rangle\,|y \oplus f(x_1, x_2)\rangle$.
5. Apply the inverse QFT to the first two registers.
6. Measure all registers.
7. Process the output classically.

5.7.3 Quantum Component

We begin with three zero registers, and a power of two, $Q = 2^q$, such that $p \leq q \leq 2p$. Q will be used for the dimension of the QFT later and should be a power of two to ensure accuracy with the QFT. Our initial

state is

$$|\Phi_0\rangle = |0\dots0\rangle\,|0\dots0\rangle\,|0\dots0\rangle\,.$$

Next, we apply Hadamard transformations to the first and second register. These registers go from 0 to $p-2$, not $p-1$. Looking back at the formulation of the discrete log problem, we want to solve for x in the equation $g^x \equiv a \pmod{p}$. Shor's algorithm solves the hidden abelian subgroup problem, but we have a way to reduce discrete logarithms to the hidden abelian subgroup problem. We used $\mathbb{Z}_{p-1} \times \mathbb{Z}_{p-1}$ as the additive group modulo p, but each $\mathbb{Z}_{p-1}$ set has $p-1$ elements. Thus,

$$|\Phi_1\rangle = H^{\otimes 2q}\,|\Phi_0\rangle = \frac{1}{p-1}\sum_{x_1=0}^{p-2}\sum_{x_2=0}^{p-2}|x_1\rangle\,|x_2\rangle\,|0\dots0\rangle\,.$$

We use the same unitary transformation trick from our earlier algorithms. We apply a unitary transformation U_f which maps $|x_1\rangle\,|x_2\rangle\,|y\rangle$ to $|x_1\rangle\,|x_2\rangle\,|y \oplus f(x_1,x_2)\rangle$, where $f(x_1,x_2) = g^{x_1}a^{-x_2}$ in $\mathbb{Z}_p$ as above. We see something new here; the function now has two inputs. From our problem reduction, we have a function f which can reduce the DLP to the HASP as $f : (x_1,x_2) \mapsto g^{x_1}a^{-x_2}$. This is exactly the function we will use for our unitary transformation. So,

$$|\Phi_2\rangle = U_f\,|\Phi_1\rangle = \frac{1}{p-1}\sum_{x_1=0}^{p-2}\sum_{x_2=0}^{p-2}|x_1\rangle\,|x_2\rangle\,|g^{x_1}a^{-x_2} \bmod p\rangle\,.$$

Now that the problem has been reduced to the HASP, we apply the inverse quantum Fourier transform as we did in the factoring algorithm. Therefore,

$$|\Phi_3\rangle = QFT_Q\,|\Phi_2\rangle = \frac{1}{(p-1)Q}\sum_{x_1=0}^{p-2}\sum_{x_2=0}^{p-2}\sum_{x_3=0}^{Q-1}\sum_{x_4=0}^{Q-1} e^{\frac{2\pi i}{Q}(x_1x_3+x_2x_4)}\,|x_3\rangle\,|x_4\rangle\,|g^{x_1}a^{-x_2} \bmod p\rangle\,.$$

There are four summations here, but luckily, we are done. All that is left is to measure. The quantum Fourier transform solves the hidden abelian subgroup problem, which returns us a generator for the hidden subgroup. We pass the results of the measurement to the classical portion.

5.7.4 Classical Component

Recall the reduction from the discrete log problem to the hidden abelian subgroup problem. To recover an answer to the discrete log problem, we have a generator (α,β) which has the property that $\alpha = x\beta$. Solving for x should be as simple as dividing α by β. Since (α,β) should be a generator of subgroup H, we expect $\gcd(\beta, p-1) = 1$ and β is invertible in $\mathbb{Z}_{p-1}$. The tuple returned from measurement consists of $(x_3,\, x_4,\, g^{x_1}a^{-x_2})$, and empirically, we know that the quantum Fourier transform gives us exactly such a generator $(x_3,\, x_4)$. Unfortunately, it is not so simple. We must do some processing to the output, as the analysis will show.

We create a new pair (α, β) where

$$\alpha = \frac{x_4}{Q}$$

rounded to the nearest multiple of $\frac{1}{p-1}$ and

$$\beta = \frac{x_3(p-1) - \ell}{Q}, \quad \ell = x_3(p-1) \bmod Q.$$

Now, we are likely to have a pair that will give $x = \frac{\alpha}{\beta}$. There is still a chance we do not get back x though, so it is important to check that $g^{\alpha/\beta} \pmod{p}$ does equal a. In some cases, β is not coprime with $p-1$ and therefore cannot be used as a divisor in $\mathbb{Z}_{p-1}^*$.

5.7.5 Analysis

When we measure $|\Phi_3\rangle$, we interest ourselves in the case where the third register has the form $y \equiv g^k \pmod{p}$ for some integer k. Here we consider all the states where the relation $x_1 - xx_2 \equiv k \pmod{p-1}$ is true. So, our summation will reflect that:

$$\sigma_k = \left| \frac{1}{(p-1)q} \sum_{\substack{x_1, x_2 \\ x_1 - xx_2 \equiv k}} e^{\frac{2\pi i}{q}(x_1 x_3 + x_2 x_4)} \right|^2.$$

This is not a summation of all possible states, only of states that we may consider desirable. It is now up to us to determine what this probability value is. The relation can be reformulated so that x_1 is alone on the left side and modular arithmetic is replaced by integer arithmetic:

$$x_1 = k + xx_2 - (p-1) \left\lfloor \frac{k + xx_2}{p-1} \right\rfloor.$$

Now that we have fully described x_1 with x_2 and k, we can eliminate x_1 from the amplitude:

$$\begin{aligned}
\sigma_k &= \left| \frac{1}{(p-1)Q} \sum_{x_2=0}^{p-2} e^{\frac{2\pi i}{Q}\left(\left(k + xx_2 - (p-1)\left\lfloor \frac{k+xx_2}{p-1} \right\rfloor\right) x_3 + x_2 x_4\right)} \right|^2 \\
&= \left| \frac{1}{(p-1)Q} \sum_{x_2=0}^{p-2} e^{\frac{2\pi i}{Q}\left(kx_3 + xx_2x_3 - x_3(p-1)\left\lfloor \frac{k+xx_2}{p-1} \right\rfloor + x_2 x_4\right)} \right|^2.
\end{aligned}$$

We define a new constant $\ell = x_3(p-1) \bmod Q$ and use this in further equations. A factor of $e^{2\pi i k x_3/Q}$ can be removed from all terms because it has no effect on the probability amplitude:

$$\sigma_k = \left| \frac{1}{(p-1)Q} \sum_{x_2=0}^{p-2} e^{\frac{2\pi i x_2}{Q}\left(xx_3 + x_4 - \frac{x\ell}{p-1}\right)} e^{\frac{2\pi i \ell}{Q}\left(\frac{xx_2}{p-1} - \left\lfloor \frac{k+xx_2}{p-1} \right\rfloor\right)} \right|^2 .$$

By relating the variables x_1, x_2, x_3, and x_4 to another, we have reduced the number of summations with which we concern ourselves with down to one. All that is left is to examine what happens as x_2 goes from 0 to $p-2$. Ideally, we want all the values that occur in the summation to be positive. We know the amplitudes rotate around the unit circle, so if we can show that the rotation is restricted to a maximum of half the unit circle, we can show that all the complex values in the summation have positive real part.

Looking at the left exponential, $xx_3 + x_4 - \frac{x\ell}{p-1}$ is a fixed value, and we want to say that it can be no more than $\frac{1}{2}$. If so, then as x_2 goes from 0 to $p-2$, it never crosses over half of the unit circle and thus remains positive.

Looking at the right exponential, if ℓ is less than $Q/12$, then the exponential falls within the range $\left[1 - e^{\pi i/6}, 1 + e^{\pi i/6}\right]$.

Of all the possible states, the probability a random state satisfies both of these conditions is $\frac{p}{240Q}$ or $\frac{1}{480}$ since $Q \leq 2p$. Unfortunately, calculating this number is out of scope of this text, and interested parties should instead refer to one of the modern versions of Shor's algorithm.

So, one might have to run this algorithm around 480 times in order to receive a candidate for a pair (α, β)! However, this number of runs is fixed for *every* discrete logarithm problem that is input. It does not grow with the input, making this a constant value. It gets worse—the pair may be unusable. Once we get a candidate, we have the relation

$$-\frac{1}{2Q} \leq \frac{x_4}{Q} + x\left(\frac{x_3(p-1) - \ell}{(p-1)Q}\right) \leq \frac{1}{2Q}$$

taken by dividing our conditions of success from the left exponential and dividing by Q. We have to extract x out of this. We know the values p, Q, x_3, x_4, and ℓ. This is why we set $\alpha = \frac{x_4}{Q}$ rounded to the nearest multiple of $\frac{1}{p-1}$ and $\beta = \frac{x_3(p-1)-\ell}{Q}$. If β is not relatively prime to $p-1$, then β is not invertible in the ring $\mathbb{Z}_{p-1}$ and we cannot solve $\alpha = x\beta$ for x in general. The probability that a value β does is not relatively prime to $p-1$ is also out of scope of this text, but we can say that it is about $\frac{1}{2}$; similar to the factoring algorithm for a suitable order value. Shor's algorithm for discrete logs is therefore polynomial, but it comes with a massive constant value cost. Modern versions both reduce this constant and increase the chance of success.

Appendix A

Glossary of Definitions

abelian group An abelian group is a group that has commutativity. That is, for a group $(G, \cdot)$, for all elements $u, v \in G$, $u \cdot v = v \cdot u$. 12, 89, 92

basis A basis B for a vector space V is a minimal spanning set of V that is linearly independent. Each element of V can be written as a unique linear combination of vectors in B. 12

binary dot product Given two bit strings $a = a_1 a_2 \ldots a_n$ and $b = b_1 b_2 \ldots b_n$ where $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ are either 0 or 1, the binary dot product $a \cdot b$ is given by $a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$ mod 2. 27, 52

binary operation A binary operation is a function which takes two inputs—usually from the same set—and gives an output in that set. It is not to be confused with binary numbers. Binary operations may have an additional notational shorthand where the operands are on either side of the operator. For example, given the function $(\cdot) : S \times S \to S$, we can write $(\cdot)(a, b) = c$ as $a \cdot b = c$. 90

complex conjugate The conjugate of a complex number $z = a + bi$ is $\overline{z} = a - bi$, the unique complex number that has the same real part as z and imaginary part with the same magnitude but opposite sign. 11

complex vector space A *complex vector space* is a set V of *vectors* forming an abelian group under vector addition which is closed under multiplication by complex scalars in such a way that the following identities hold for all $|\varphi\rangle, |\psi\rangle \in V$ and all $c, d \in \mathbb{C}$:

- $(c + d) |\varphi\rangle = c |\varphi\rangle + d |\varphi\rangle$
- $c(|\varphi\rangle + |\psi\rangle) = c |\varphi\rangle + c |\psi\rangle$
- $(cd) |\varphi\rangle = c (d |\varphi\rangle)$
- $1 |\varphi\rangle = |\varphi\rangle$

. 12, 90, 92

computational basis The computational basis for the complex vector space $\mathbb{C}^{2^n}$ is defined by $\mathcal{B} = \{|b\rangle \mid b \in \mathbb{Z}_2^n\}$. This is the "standard basis" familiar to students of linear algebra, but with axes labeled by binary n-tuples. 13, 14

continued fraction expansion Any number r can be expressed as a simple continued fraction

$$r = a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \ldots}}$$

where $a_1 \in \mathbb{Z}$ and $a_2, a_3, \ldots$ are positive integers. We can also simplify the form of a continued fraction and write it in coefficient vector notation, $[a_1, a_2, \ldots]$. *Simple* means that the fractions all have 1 as a numerator. A continued fraction may be finite or infinite; particularly, the continued fraction expansion is finite if and only if r is a rational number. To convert a rational number r into a continued fraction, we take the floor of r_i and this becomes the coefficient a_i. Then, we subtract a_i from r_i and take the reciprocal to get r_{i+1}. This process is repeated until it can be continued no more. The recurrence relations that define this process are $a_i := \lfloor r_i \rfloor$ and $r_{i+1} := 1/(r_i - a_i)$. 77

coprime Two numbers are coprime if they share no common factors. This is equivalent to their greatest common divisor being 1. 87, 92

dimension The dimension of a vector space is the cardinality of a basis of that vector space. 13

group A group $(G, \cdot)$ is a set G with a binary operation $(\cdot)$ which satisfies the following properties for all $a, b, c \in G$:

- $a \cdot b \in G$ (*closure*)
- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ (*associativity*)
- There exists an element $e \in G$ such that $a \cdot e = e \cdot a = a$ for all $a \in G$ (*identity*)
- Given a, there exists an element a^{-1} such that $a \cdot a^{-1} = a^{-1} \cdot a = e$ (*inverses*)

A group $(H, \cdot)$ is called a *subgroup* if $H \subseteq G$ and it still respects the properties of being a group under the same operation as G. Notationally, the pair may be omitted when describing a group and it can be referred to by its set. As an example: "Let G be a group and H be a subgroup $H \leq G$." Also notice that the symbol has changed for describing a subgroup. 44, 89

Hermitian inner product For a complex vector space V, a Hermitian inner product is a function $\langle \, , \, \rangle : V \times V \to \mathbb{C}$ which satisfies the following properties for all $u, v, w \in V$ and $\lambda \in \mathbb{C}$:

- $\langle u, v\rangle = \overline{\langle v, u\rangle}$ *(conjugate symmetry)*
- $\langle \lambda u, v\rangle = \lambda\langle u, v\rangle$
- $\langle u + w, v\rangle = \langle u, v\rangle + \langle w, v\rangle$
- $\langle u, u\rangle \geq 0$
- $\langle u, u\rangle = 0$ if and only if $u = 0$

On $\mathbb{C}^n$, this function has the form $\langle z, w\rangle = \sum_{i=1}^{n} z_i \overline{w_i}$. 13

Kronecker product The Kronecker product is an operation on two matrices, denoted by the symbol $\otimes$, which returns a block matrix. If A is an $m \times n$ matrix and B is a $p \times q$ matrix, then the Kronecker product $A \otimes B$ is a $mp \times nq$ block matrix where the (i, j) block is equal to $a_{ij}B$:

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

Note that, unlike matrix multiplication, our dimensions need not agree. The following properties hold for Kronecker products of matrices A, B, C, D, and scalar r:

- $(A \otimes B)(C \otimes D) = (AC \otimes BD)$ (*mixed product property*)
- $(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger$ (*transposition is distributive*)
- $(A + B) \otimes C = A \otimes C + B \otimes C$, $A \otimes (B + C) = A \otimes B + A \otimes C$ (*additivity of linearity*)
- $(rA) \otimes B = A \otimes (rB) = r(A \otimes B)$ (*homogeneity of linearity*)

with the mixed product property requiring matrix dimensions to allow for matrix multiplication to be defined. 14

linear combination A linear combination of a set of vectors is a finite sum of scalar multiples of vectors in that set. Let $|\psi_1\rangle, |\psi_2\rangle, \dots, |\psi_n\rangle$ be a set of vectors and $c_1, c_2, \dots, c_n$ be a set of scalars, then $c_1|\psi_1\rangle + c_2|\psi_2\rangle + \dots + c_n|\psi_n\rangle$ is a linear combination. 12, 91

linearly dependent A set of vectors is linearly dependent if it is not linearly independent. That is, there exists a non-zero set of scalars $c_1, c_2, \dots, c_n$ such that the linear combination $c_1|\psi_1\rangle + c_2|\psi_2\rangle + \dots + c_n|\psi_n\rangle = |0\rangle$ is true. 12

linearly independent A set of vectors $|\psi_1\rangle, |\psi_2\rangle, \dots, |\psi_n\rangle$ is linearly independent if the linear combination $c_1|\psi_1\rangle + c_2|\psi_2\rangle + \dots + c_n|\psi_n\rangle = |0\rangle$ is true only when $c_1 = c_2 = \dots = c_n = 0$. 12, 58, 89, 91

multiplicative group modulo *n* A finite abelian group under the operation multiplication modulo n. The elements of this group consist only of elements coprime to n. The order of this group is then exactly $\phi(n)$ where ϕ is Euler's totient function. The multiplicative group modulo n is denoted by $(\mathbb{Z}/n\mathbb{Z})^*$. 45

oracle An oracle is a "black box" operation, often defined as a function f which takes an input and produces some output without any regard for the process by which the output is obtained. In algorithm analysis, and oracle call is one step. 46, 52, 56

order The order of a group G is the number of elements in G. It is denoted as $|G|$. 92

orthogonal Two vectors v_1 and v_2 are orthogonal if they are perpendicular to each other. More generally, if the inner product of the vectors v_1 and v_2 is 0, then they are orthogonal with respect to that inner product. 13

orthonormal basis An orthonormal basis is a basis in some complex vector space whose vectors are all unit vectors which are orthogonal to one another. 13

superposition A qubit is in superposition if it can be expressed as a linear combination with non-zero constants of two or more basis states. For example, if neither c_1 nor c_2 is zero, the following qubit is in superposition with respect to the computational basis: $|\psi\rangle = c_1 |0\rangle + c_2 |1\rangle$. 13

unit vector A unit vector is a vector which has a magnitude of 1. 13

vector addition mod 2 Given two vectors $a, b \in \mathbb{Z}_2^n$, vector addition mod 2 is elementwise addition with modulo 2 applied, that is, $a \oplus b = [a_1 + b_1 \bmod 2,\ a_2 + b_2 \bmod 2,\ \ldots,\ a_n + b_n \bmod 2]$. This is equivalent to the classical operation XOR on two binary strings. This can also be done with elements of other vector spaces and is called addition over $\mathbb{F}_2$, where $\mathbb{F}_2$ is any field modulo 2. 52

Bibliography

[1] Jack D. Hidary. *Quantum Computing: An Applied Approach*. Springer International Publishing, Cham, 2021.

[2] Isaac L. Chuang, Neil Gershenfeld, and Mark Kubinec. Experimental implementation of fast quantum searching. *Phys. Rev. Lett.*, 80:3408–3411, Apr 1998.

[3] IBM Quantum. https://quantum-computing.ibm.com/, 2021.

[4] Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. Perfect quantum error correcting code. *Phys. Rev. Lett.*, 77:198–201, Jul 1996.

[5] Markus Grassl. Bounds on the minimum distance of linear codes and quantum codes. Online available at http://www.codetables.de, 2007. Accessed on 2022-03-18.

[6] Jerry Chow, Oliver Dial, and Jay Gambetta. IBM quantum breaks the 100-qubit processor barrier. https://research.ibm.com/blog/127-qubit-quantum-processor-eagle, Feb 2021.

[7] Adam Bouland, Bill Fefferman, Chinmay Nirkhe, and Umesh Vazirani. On the complexity and verification of quantum random circuit sampling. *Nature Physics*, 15(2):159–163, Feb 2019.

[8] Yulin Wu, Wan-Su Bao, Sirui Cao, et al. Strong quantum computational advantage using a superconducting quantum processor. *Physical Review Letters*, 127(18), Oct 2021.

[9] Frank Arute, Kunal Arya, Ryan Babbush, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, Oct 2019.

[10] Edwin Pednault, John Gunnels, Dmitri Maslov, and Jay Gambetta. On "Quantum Supremacy". https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/, Oct 2019.

[11] QuTech. Quantum Inspire Starmon-5 fact sheet. https://qutech.nl/wp-content/uploads/2020/04/3.-Technical-Fact-Sheet-Quantum-Inspire-Starmon-5.pdf, Jun 2020.

[12] K. Wright, K. M. Beck, S. Debnath, et al. Benchmarking an 11-qubit quantum computer. *Nature Communications*, 10(1):5464, Nov 2019.

[13] Stephen H. Friedberg, Arnold J. Insel, and Lawrence E. Spence. *Linear Algebra*. Pearson, 2018.

[14] N. David Mermin. *Quantum Computer Science: an Introduction*. Cambridge University Press, Cambridge, 2007.

[15] MD SAJID ANIS, Héctor Abraham, AduOffei, et al. Qiskit: An open-source framework for quantum computing. https://qiskit.org/, 2021.

[16] Download python[screenshot by gwyneth c. ormes], 2021. [Online; accessed March 18, 2021].

[17] Anaconda individual edition[screenshot by gwyneth c. ormes], 2021. [Online; accessed March 18, 2021].

[18] Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM J. Comput.*, 35(1):170–188, jul 2005.

[19] Greg Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In *TQC*, 2013.

[20] Richard Allen, Ratip Emin Berker, Sílvia Casacuberta, and Michael Gul. Quantum and classical algorithms for bounded distance decoding. Cryptology ePrint Archive, Report 2022/195, 2022. https://ia.cr/2022/195.

[21] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439:553 – 558, 1992.

[22] David Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 400:97 – 117, 1985.

[23] Cleve R., Ekert A., Macchiavello C., and Mosca M. Quantum algorithms revisited. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 454:339 – 354, 1998.

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

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

[26] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[27] Guangya Cai and Daowen Qiu. Optimal separation in exact query complexities for simon's problem. *Journal of Computer and System Sciences*, 97:83–93, 2018.

[28] Guoliang Xu, Daowen Qiu, Xiangfu Zou, and Jozef Gruska. Improving the success probability for shor's factorization algorithm. In *Reversibility and Universality*, 2018.

[29] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, apr 2021.

[30] Daniel Shanks. Five number theoretic algorithms. *Proceedings of the Second Manitoba Conference on Numerical Mathematics*, pages 51–70, 1973.

[31] John M. Pollard. Monte Carlo methods for index computation. *Mathematics of Computation*, 32:918–924, 1978.

[32] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over GF(p)and its cryptographic significance (Corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[33] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[34] Martin Ekerå. Modifying Shor's algorithm to compute short discrete logarithms. *IACR Cryptol. ePrint Arch.*, 2016:1128, 2016.

[35] Martin Ekerå. Revisiting Shor's quantum algorithm for computing general discrete logarithms. https://arxiv.org/abs/1905.09084, 2021.