Endterm Report: Quantum Computing, Information and Quantum Technologies

SoS '24

Pranav Malpani
23B1279
Mentor: Anand Narasimhan

July 2024

Abstract

This is an end-term report for **P02 - Quantum Computing, Information and Quantum Technologies**. It aims to highlight my progress in understanding this topic during the span of the past 8 weeks. My prior introduction to this whole topic was limited to a linear algebra and a quantum mechanics course during my second semester.

The material used to study this topic were mainly *Quantum Computation and Quantum Information* by M. Nielsen and I. Chuang (referred to as *the book* in this report), and *The Qiskit Textbook*, maintained by IBM. The Qiskit textbook in its present form is a github repository that contains all the chapters in a `.ipynb` file format.
YouTube videos were also used to understand the key concepts better.

Contents

Chapter 1

Quantum Circuits and Computation

1.1 Controlled Operations Continued

1.1.1 General Controlled Operations in Two Qubits

In the mid-term report, we talked about what the controlled-U gate means, in the case of two qubits. In general, a controlled-Unitary gate is C_U such that:

$$C_U \left|00\right\rangle = \left|00\right\rangle, C_U \left|01\right\rangle = \left|01\right\rangle, C_U \left|10\right\rangle = \left|1\right\rangle (U \left|0\right\rangle), C_U \left|11\right\rangle = \left|1\right\rangle (U \left|1\right\rangle).$$

Now comes the difficult part of constructing such a gate.
First, though, we need to prove some other facts about unitary operations—

$Z - Y$ decomposition for a single qubit says that any unitary operator can be written as a series of 3 rotations about the Z, Y, and Z axes, in order.

$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta)$$

This is exactly the statement of Euler's Rotation Theorem, and it can also easily be confirmed by the construction

$$U = \begin{bmatrix} e^{i(\alpha-\beta/2-\delta/2)} \cos\gamma/2 & e^{i(\alpha-\beta/2+\delta/2)} \cos\gamma/2 \\ e^{i(\alpha+\beta/2-\delta/2)} \cos\gamma/2 & e^{i(\alpha+\beta/2+\delta/2)} \cos\gamma/2 \end{bmatrix}$$

Now, we set $A = R_z(\beta)R_y(\gamma/2)$, $B = R_y(-\gamma/2)R_z(-(\delta+\beta)/2)$ and $C = R_z((\delta-\beta)/2)$, which lends us the identity:

$$ABC = R_z(\beta)R_y(\gamma/2)R_y(-\gamma/2)R_z(-(\delta+\beta)/2)R_z((\delta-\beta)/2) = I$$

Before we move on to the next step, we need to note that

$$XYX = -Y \implies XR_y(\theta)X = R_y(-\theta)$$
$$XZX = -Z \implies XR_z(\theta)X = R_z(-\theta)$$

Using this and the fact that $X^2 = I$, we can see that

$$\begin{aligned} XBX = XR_y(-\gamma/2)R_z(-(\delta+\beta)/2)X &= XR_y(-\gamma/2)XXR_z(-(\delta+\beta)/2)X \\ &= R_y(\gamma/2)R_z((\delta+\beta)/2) \end{aligned}$$

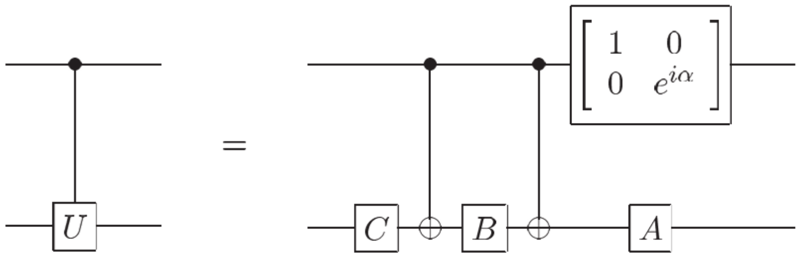
Thus,

$$AXBXC = R_z(\beta)R_y(\gamma)R_z(\delta)$$

and

$$U = e^{i\alpha}AXBXC$$

Finally, we have all the necessary components to construct a controlled-Unitary:



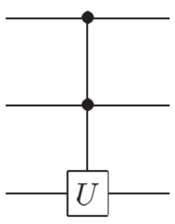
Thus, if the *control* bit is $|1\rangle$, then the controlled-NOT is applied, making the transformation $e^{i\alpha}AXBXC$, and when it is $|0\rangle$, then the controlled-NOT is not applied, making the transformation equivalent to $1 \times ABC = I$.
This is how we construct a general controlled-Unitary.

1.1.2 Multiple-qubit Controlled Operations

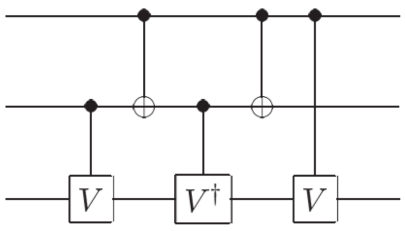
So far we have only talked about controlled operations where one qubit acts as the control, and the other is the target. Now, we shall try to generalize this to any n number of qubits.
Suppose we have $n+1$ qubits, and U is a single qubit operation, and we want to make a multi-qubit operation $C^n(U)$, such that

$$C^n(U)\,|x_0x_1\dots x_{n-1}\rangle\,|x_n\rangle = |x_0x_1\dots x_{n-1}\rangle\, U^{x_0x_1\dots x_{n-1}}\,|x_n\rangle\,,$$

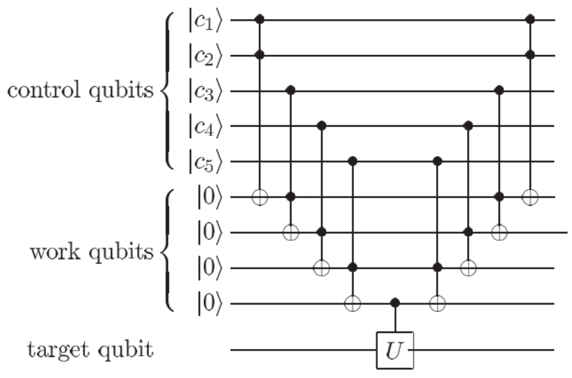
where the superscript on U means that it is only applied when all $x_0, x_1 \dots x_{n-1}$ are 1. Now, let's try to construct a gate by taking the example of $n = 2$. We want a gate U that looks like



We do this by defining a different operator V, such that $V^2 = U$. Now, this gate can be implemented by the following construction:

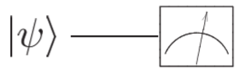


A special example of such a gate is the Toffoli gate, where $U = X$. For an n-qubit implementation, we can extend this idea further:



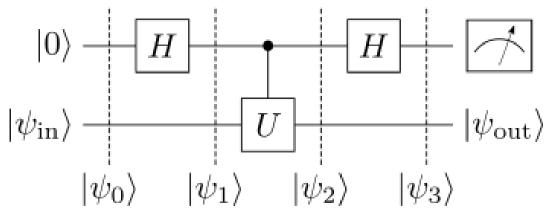
1.2 Measurement

Measurement is an important part and the logical next step of any quantum computation. It is *usually* done at the end of the computation, but the interesting thing is that even if the measurement is involved in the middle of the computation, it can always be moved to the end, with all the classical gates being changed to quantum ones. It is denoted by adding a meter symbol to the qubit.



1.2.1 Measuring an Operator

Let we have an operator U that is both hermitian and unitary, thus it is both an observable and a quantum gate. Now, we wish to measure the observable given by the operator U. We make the following circuit:



Now we know from before that Unitary matrices only have eigenvalues of ± 1. Let's try to solve through this circuit.

$$\begin{aligned}
&\text{At } |\psi_0\rangle, |0\rangle|\psi_{in}\rangle. \\
&\text{At } |\psi_1\rangle, \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|\psi_{in}\rangle \\
&\text{At } |\psi_2\rangle, \frac{1}{\sqrt{2}}C_U(|0\rangle + |1\rangle)|\psi_{in}\rangle = \frac{1}{\sqrt{2}}(|0\rangle|\psi_{in}\rangle + |1\rangle U|\psi_{in}\rangle) \\
&\text{At } |\psi_3\rangle, \frac{1}{2}((|0\rangle + |1\rangle)|\psi_{in}\rangle + (|0\rangle - |1\rangle)U|\psi_{in}\rangle) \\
&= \frac{1}{2}(|0\rangle(I+U)|\psi_{in}\rangle + |1\rangle(I-U)|\psi_{in}\rangle)
\end{aligned}$$

Notice that $(I+U)|\psi_{in}\rangle$ and $(I-U)|\psi_{in}\rangle$ are both eigenvectors of U, as

$$U(I+U)|\psi_{in}\rangle = \left(U+U^2\right)|\psi_{in}\rangle = +1(I+U)|\psi_{in}\rangle$$

$$U(I-U)|\psi_{in}\rangle = \left(U-U^2\right)|\psi_{in}\rangle = -1(I-U)|\psi_{in}\rangle$$

Finally, we get $|\psi_3\rangle$. All that's left to do is to measure the first qubit. If it turns out to be $|0\rangle$, we know that the measurement of the observable is $+1$, since the other qubit has to be $(I+U)|\psi_{in}\rangle$, which is the eigenvector corresponding to the eigenvalue $+1$.

Chapter 2

The Quantum Fourier Transform

The quantum fourier transform is much like the discrete fourier transform we are familiar with. In the usual DFT, the input is a vector of complex numbers, $x_1, x_2, \ldots, x_n$, and the outputs are $y_0, y_1, \ldots, y_n$, where

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi ijk/N}.$$

The QFT is very similar. On an orthonormal basis, $|0\rangle, |1\rangle, \ldots, |N-1\rangle$, it is defined to act as such:

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

Thus, on any arbitrary state,

$$\sum_{j=0}^{N-1} x_j |j\rangle \to \sum_{k=0}^{N-1} y_k |k\rangle.$$

2.1 Tensor Product Representation

Let us take $N = 2^n$, where n is the number of qubits in our register, and our computational basis is $|0\rangle, \ldots, |2^n - 1\rangle$. We write the state $|j\rangle$ as its binary representation, $j = j_1 j_2 \ldots j_n$. More formally, $j = j_1 2^{n-1} + j_2 2^{n-2} + \cdots + j_n 2^0$. Then, the tensor product representation of the QFT means that:

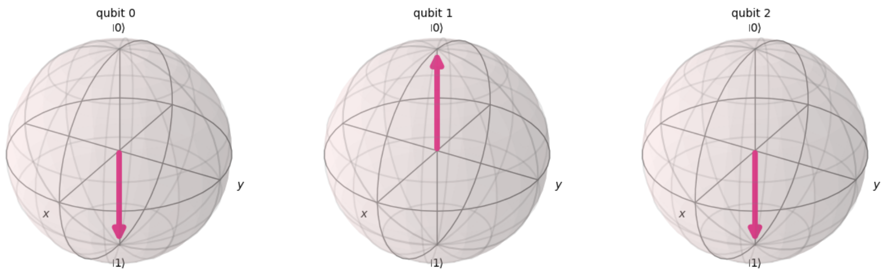
$$
\begin{aligned}
|j\rangle \to \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi i jk/2^n} |k\rangle \\
&= \frac{1}{2^{n/2}} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} e^{2\pi i j\left(\sum_{l=1}^{n} k_l 2^{-l}\right)} |k_1 \dots k_n\rangle \\
&= \frac{1}{2^{n/2}} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} \bigotimes_{l=1}^{n} e^{2\pi i j k_l 2^{-l}} |k_l\rangle \\
&= \frac{1}{2^{n/2}} \bigotimes_{l=1}^{n} \sum_{k_l=0}^{1} e^{2\pi i j k_l 2^{-l}} |k_l\rangle \\
&= \frac{1}{2^{n/2}} \bigotimes_{l=1}^{n} \left[|0\rangle + e^{2\pi i j 2^{-l}} |1\rangle\right],
\end{aligned}
$$

which finally gives $\dfrac{\left(|0\rangle + e^{2\pi i(0.j_n)} |1\rangle\right) \dots \left(|0\rangle + e^{2\pi i(0.j_1 j_2 \dots j_n)} |1\rangle\right)}{2^{n/2}}$,

where $(0.j_n)$, $(0.j_{n-1}j_n)$, etc are the binary forms of the numbers $\frac{j_n}{2}$, $\left(\frac{j_{n-1}}{2} + \frac{j_n}{2^2}\right)$, *et cetera*, since all the whole parts of these numbers do not make a difference. If you didn't follow all that, here's an easier way to look at it.

2.2 Geometrical Intuition

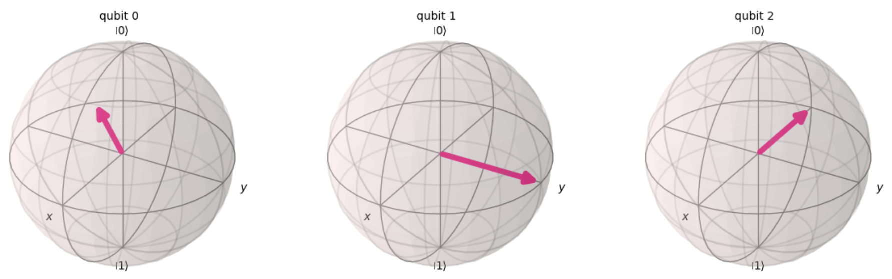
Imagine we have $n = 3$, and the state we want to transform is $|5\rangle$, i.e. $|101\rangle$. In the current state, our qubits look a little something like this



Now, we perform the QFT on this state. We get the statevector:

$$\frac{1}{2^{3/2}} \left(|0\rangle + e^{\frac{5\pi}{1}} |1\rangle\right) \left(|0\rangle + e^{\frac{5\pi}{2}} |1\rangle\right) \left(|0\rangle + e^{\frac{5\pi}{4}} |1\rangle\right),$$

which is equivalent to this figure(except for the third and the first term being swapped, which we'll get to later):



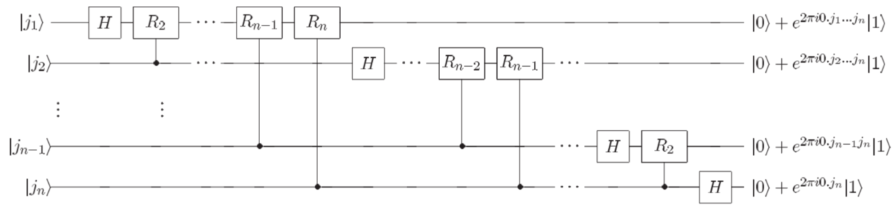
We notice that the number (our initial state), has been *encoded in the phase* of the qubits. The first qubit has a phase of $\frac{5\pi}{4}$, the second has a phase of $\frac{5\pi}{2}$, and the third 5π This is what the Quantum Fourier Transform essentially does.

2.3 Implementation

Now, we shall try to figure out how to implement this through gates. We first define the gate R_k:

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}.$$

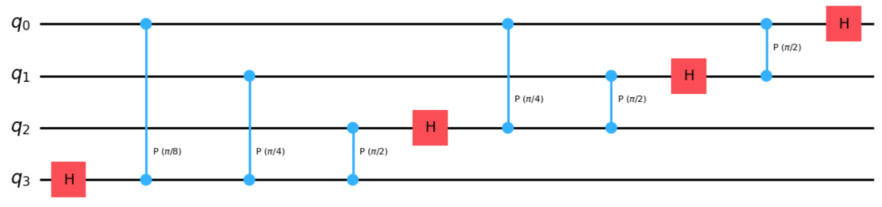
Then, using the circuit described below, we can implement the QFT rather simply. Basically, if a qubit j_k is *set*, the controlled R_k gate is applied, and so the phase $e^{2\pi i/2^k}$ is added to the target qubit.


$|j_1\rangle$
H
R_2
R_{n-1}
R_n
$|0\rangle + e^{2\pi i 0.j_1 \dots j_n}|1\rangle$
$|j_2\rangle$
H
R_{n-2}
R_{n-1}
$|0\rangle + e^{2\pi i 0.j_2 \dots j_n}|1\rangle$
$|j_{n-1}\rangle$
H
R_2
$|0\rangle + e^{2\pi i 0.j_{n-1} j_n}|1\rangle$
$|j_n\rangle$
H
$|0\rangle + e^{2\pi i 0.j_n}|1\rangle$

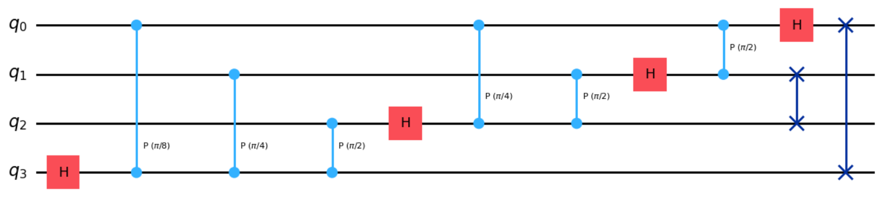
But, you will realize that we are still not at the desired result. To get there, we need to reverse the order of the qubits, which is done by swapping the qubits($|j_1\rangle$ with $|j_n\rangle$, $|j_2\rangle$ with $|j_{n-1}\rangle$, etc.)

2.4 Code

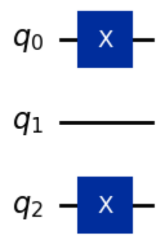
```python
import numpy as np
from numpy import pi
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_bloch_multivector
def qft_rotations(circuit, n):
    if n == 0:
        return circuit
    n -= 1
    circuit.h(n)
    for qubit in range(n):
        circuit.cp(pi/2**(n-qubit), qubit, n)
    qft_rotations(circuit, n)

# Let's see how it looks:
qc = QuantumCircuit(4)
qft_rotations(qc,4)
qc.draw('mpl')
```



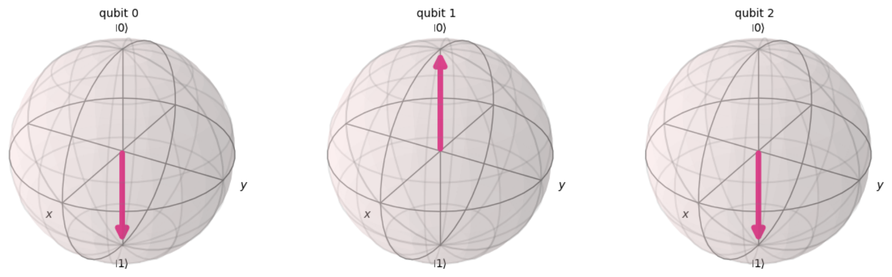
```python
def swap_registers(circuit, n):
    for qubit in range(n//2):
        circuit.swap(qubit, n-qubit-1)
    return circuit

def qft(circuit, n):
    qft_rotations(circuit, n)
    swap_registers(circuit, n)
    return circuit

# Let's see how it looks:
qc = QuantumCircuit(4)
qft(qc,4)
qc.draw('mpl')
```



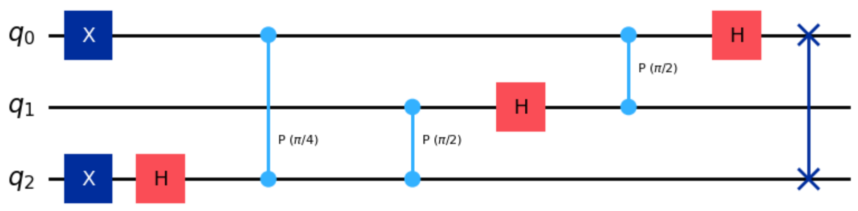
```python
# Create the circuit
qc = QuantumCircuit(3)

# Encode the state 5
qc.x(0)
qc.x(2)
qc.draw('mpl')
```



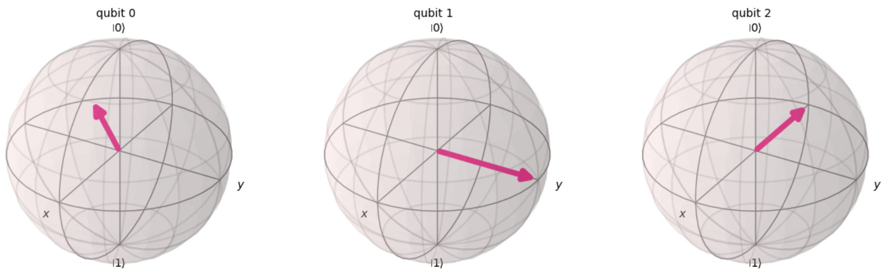
```python
sim = AerSimulator()
qc_init = qc.copy()
qc_init.save_statevector()
statevector = sim.run(qc_init).result().get_statevector()
plot_bloch_multivector(statevector)
```



```python
qft(qc,3)
qc.draw('mpl')
```



```python
qc.save_statevector()
statevector = sim.run(qc).result().get_statevector()
plot_bloch_multivector(statevector)
```


qubit 0
|0⟩
|1⟩
x
y
qubit 1
|0⟩
|1⟩
x
y
qubit 2
|0⟩
|1⟩
x
y

Chapter 3

Shor's Algorithm

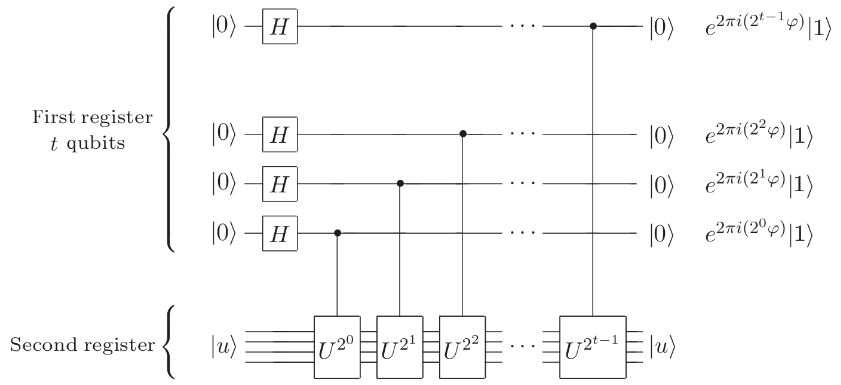
3.1 The Original Trilogy

3.1.1 A New Concept

One of the applications of the Quantum Fourier Transform is *phase estimation.* Suppose we have a unitary U, that has an eigenvector $|v\rangle$, with eigenvalue $e^{2\pi i\phi}$, which is unknown, with $\phi < 1$. We need to estimate ϕ to a certain extent. To compute this, we require certain prerequisites:

- We need to have a mechanism of preparing the state $|u\rangle$.
- We need to have a mechanism of implementing the controlled-U^{2^j} operation

We call these mechanisms *oracles.*



First register
t qubits
Second register
$|0\rangle$
H
$|u\rangle$
U^{2^0}
U^{2^1}
U^{2^2}
$U^{2^{t-1}}$
$e^{2\pi i(2^{t-1}\varphi)}|1\rangle$
$e^{2\pi i(2^2\varphi)}|1\rangle$
$e^{2\pi i(2^1\varphi)}|1\rangle$
$e^{2\pi i(2^0\varphi)}|1\rangle$

The above circuit implements the *state-preparation* stage of this subroutine. The normalization factors of $\frac{1}{\sqrt{2}}$ are ignored for now. Now,

3.1.2 The QFT Strikes Back

We can easily see that applying the Inverse QFT to the first register is very beneficial for us, since it would give us the statevector $|\phi_1 \dots \phi_n\rangle$, which is the binary representation of ϕ. Measuing in the computational basis would give us ϕ.

But what if ϕ cannot be written in n bits?

3.1.3 Return of The Math

There will be an integer in the range 0 to $2^t - 1$ which, when divided by 2^t will give the *best* n-bit binary approximation for ϕ, which is lesser than it. Let this integer be b. We call the difference between ϕ and $\frac{b}{2^t}$ as δ. It is clear that $\delta \leq 2^{-t}$.
If we apply the Inverse QFT to the state after it is *prepared*, we will get:

$$\frac{1}{2^t} \sum_{k,l=0}^{2^t-1} e^{\frac{-2\pi i k l}{2^t}} e^{2\pi i \phi k} |l\rangle.$$

Let

$$\begin{aligned} \alpha_l &= \frac{1}{2^t} \sum_{k=0}^{2^t-1} \left(e^{2\pi i \left(\phi - (b+l)/2^t \right)} \right)^k \\ &= \frac{1}{2^t} \left(\frac{1 - e^{2\pi i \left(2^t \phi - (b+l) \right)}}{1 - e^{2\pi i (\phi - (b+l)\ 2^t)}} \right) \\ &= \frac{1}{2^t} \left(\frac{1 - e^{2\pi i \left(2^t \delta - l \right)}}{1 - e^{2\pi i (\delta - l/2^t)}} \right). \end{aligned}$$

We want to make the probablity of our measurement m not being near b bounded, i.e.

$$p(|m-b| > \epsilon) = \sum_{-2^{t-1} < l \leq -(e+1)} |\alpha_l|^2 + \sum_{e+1 \leq l \leq 2^t - 1} |\alpha_l|^2.$$

But,

$$|\alpha_l| \leq \frac{2}{2^t \left| 1 - e^{2\pi i (\delta - l/2^t)} \right|}.$$

Using the fact that $|1 - exp(i\theta)| \geq 2|\theta|/\pi, -\pi \leq \theta \leq \pi$,

$$\begin{aligned}
|\alpha_l| &\leq \frac{1}{2^{t+1}(\delta - l/2^t)}. \\
\implies p(|m-b| > \epsilon) &\leq \frac{1}{4}\left[\sum_{l=-2^{t-1}+1}^{-(\epsilon+1)} \frac{1}{(l-2^t\delta)^2} + \sum_{l=\epsilon+1}^{2^t-1} \frac{1}{(l-2^t\delta)^2}\right] \\
&\leq \frac{1}{2}\sum_{l=\epsilon}^{2^{t-1}-1} \frac{1}{l^2} \\
&\leq \frac{1}{2}\int_{\epsilon-1}^{2^{t-1}-1} dl \frac{1}{l^2} \\
&\leq \frac{1}{2(\epsilon-1)}.
\end{aligned}$$

Thus, the probability of getting it correct to 2^{-n} accuracy is atleast $1 - 1/2(2^{t-n} - 2)$. Thus, to have a probability of success atleast $1-p$ we choose

$$t = n + \lceil \log\left(2 + \frac{1}{2\epsilon}\right)\rceil.$$

3.2 The Prequel Trilogy

3.2.1 The Quantum Menace

Shor's Algorithm is an algorithm to find a number's prime factorization. It was developed by Peter Shor in 1984. It is one of the few possible applications of quantum computing right now. Classically, our best prime-factorization algoritm works in sub-exponential time, but Shor's Algorithm works in a whopping $O((\log N)^3)$ time. The reason it is popular, though, is that most of our encryption is based upon public-key cryptography, like RSA, which is based on factoring a large number and the fact that it is not possible in polynomial time (classically). So, if Shor's Algorithm is implemented on a large enough scale, it has the potential to destroy all of modern-day encryption.

3.2.2 Attack of the Groans

Let there be two positive integers x and N, with $x < N$, and no common factors. The *order* of x modulo N is defined to be the least positive integer r, such that $x^r = 1(modN)$. This leads us to the following:

$$\begin{aligned}
&x^r = k \cdot N + 1 \\
\implies &x^r - 1 = k \cdot N \\
\implies &\left(x^{\frac{r}{2}} + 1\right)\left(x^{\frac{r}{2}} - 1\right) = k \cdot N
\end{aligned}$$

And we (probably) have two numbers that share a common factor with N. Now, there are two things that can go wrong here (*insert groan*):

- r might not be even.
- one of our guesses might be a factor of k, not N.

In the first case, we can just change the order to $2r$, and then proceed as previously, and in the second case, we run euclid's algorithm.

3.2.3 Revenge of the Qubit

There's probably tons of questions running around in your head now. How do I master the Force? How do we find the order in *better than* polynomial time? Did I leave the lights on in my room again?

Well, the Force will come to you soon enough, you just gotta give it time, but finding the order is a hard problem for a classical computer. Luckily, as it turns out, order-finding can be made equivalent to phase-estimation, with the Unitary operator being

$$U\left|y\right\rangle = \left|xy(modN)\right\rangle$$

For eg, with $x = 3$, $N = 28$.

$$\begin{aligned} U|1\rangle &= |3\rangle \\ U^2|1\rangle &= |9\rangle \\ U^3|1\rangle &= |27\rangle \\ &\vdots \\ U^{(r-1)}|1\rangle &= |19\rangle \\ U^r|1\rangle &= |1\rangle \end{aligned}$$

In this case, $r = 6$. We can check that the states given by

$$\left|u_s\right\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} exp\left[\frac{-2\pi isk}{r}\right]\left|x^k(modN)\right\rangle$$

are eigenstates of U, because

$$U\left|u_s\right\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} exp\left[\frac{-2\pi isk}{r}\right]\left|x^{k+1}(modN)\right\rangle = exp\left[\frac{2\pi is}{r}\right]\left|u_s\right\rangle.$$

Now, the value r is encoded in the state, and thus can be estimated using phase-estimation algorithms. For the oracle to function, we do need to prepare the $|u_s\rangle$ first, for which we will need the value r, which again, we do not know. But luckily, we realize that

$$\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}\left|u_s\right\rangle = \left|1\right\rangle$$

Thus, if we perform phase estimation on the state $|1\rangle$, and make the measurement in the end, we will get one of the states $|u_s\rangle$, with $0 \leq s \leq r-1$, and measure a phase $\frac{s}{r}$, all with equal probability. If the value of s turns out to be 0 after measurement, we can run the circuit again. Then, we can calculate r with high accuracy using the method of continued fractions.

The method of continued fractions is a way of approximating the fractional representation of a rational number. It works something like this: Suppose we have a number 0.312. We can write it as:

$$\begin{aligned} 0.312 &= 0 + \frac{312}{1000} \\ &= 0 + \frac{1}{\frac{125}{39}} = 0 + \frac{1}{3+\frac{8}{39}} \\ &= 0 + \frac{1}{3+\frac{1}{\frac{39}{8}}} = 0 + \frac{1}{3+\frac{1}{4+\frac{7}{8}}} \\ &= 0 + \frac{1}{3+\frac{1}{4+\frac{1}{\frac{8}{7}}}} = 0 + \frac{1}{3+\frac{1}{4+\frac{1}{1+\frac{1}{7}}}}. \end{aligned}$$

Now, we can control the level of precision we want for this number. If we neglect the final $\frac{1}{7}$, we will get $\frac{5}{16}$, et cetera. Thus, we will get the value of r.

And, yes, you did leave them on.

3.3 The Sequel Trilogy

3.3.1 The Code Awakens

```python
import matplotlib.pyplot as plt
import numpy as np
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from math import gcd
from numpy.random import randint
import pandas as pd
from fractions import Fraction
```

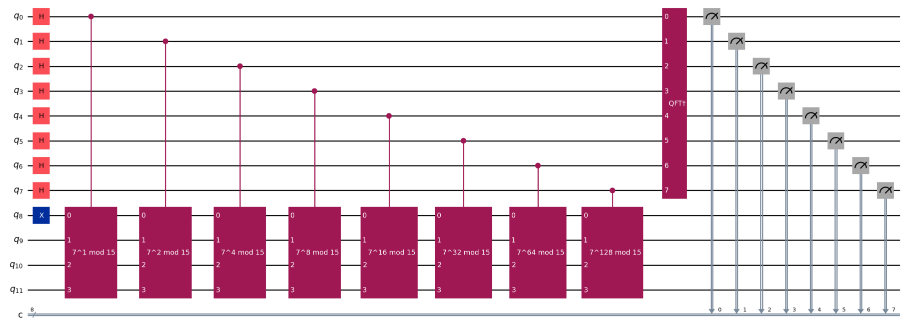
We define a function, an *oracle*, which can do the $|ay(modN)\rangle$ operation for us.

```python
def c_amod15(a, power):
    # Controlled multiplication by a mod 15, number needs to be encoded
      like this.
    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(2,3)
            U.swap(1,2)
            U.swap(0,1)
        if a in [7,8]:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
        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 = f"{a}^{power} mod 15"
    c_U = U.control()
    return c_U
```

```
# Specify variables
N_COUNT = 8  # number of counting qubits
a = 7
```

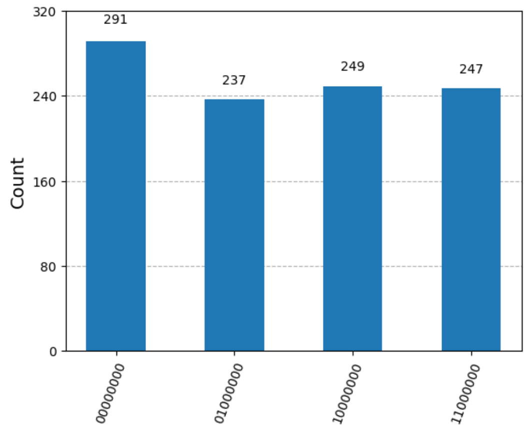
```
def qft_dagger(n):
    """n-qubit QFTdagger the first n qubits in circ"""
    qc = QuantumCircuit(n)
    # Don't forget the Swaps!
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cp(-np.pi/float(2**(j-m)), m, j)
   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

```
# Create QuantumCircuit with N_COUNT counting qubits
# plus 4 qubits for U to act on
qc = QuantumCircuit(N_COUNT + 4, N_COUNT)

# Initialize counting qubits
# in state |+>
for q in range(N_COUNT):
    qc.h(q)

# And auxiliary register in state |1>
qc.x(N_COUNT)

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

# Do inverse-QFT
qc.append(qft_dagger(N_COUNT), range(N_COUNT))

# Measure circuit
qc.measure(range(N_COUNT), range(N_COUNT))
qc.draw('mpl',fold=-1)  # -1 means 'do not fold'
```



3.3.2 The Last Step

```
aer_sim = AerSimulator()
t_qc = transpile(qc, aer_sim)
counts = aer_sim.run(t_qc).result().get_counts()
plot_histogram(counts)
```



```
rows, measured_phases = [], []
for output in counts:
    decimal = int(output, 2)  # Convert (base 2) string to decimal
    phase = decimal/(2**N_COUNT)  # 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_COUNT} = {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  01000000(bin) =  64(dec)    64/256 = 0.25
2  00000000(bin) =   0(dec)     0/256 = 0.00
3  11000000(bin) = 192(dec)   192/256 = 0.75
```

we can use the `fractions` module to turn a float into fraction. We should also probably limit the denominator since our r cannot be bigger than N.

```python
rows = []
for phase in measured_phases:
    frac = Fraction(phase).limit_denominator(15)
    rows.append([phase,
                 f"{frac.numerator}/{frac.denominator}",
                 frac.denominator])
# Print as a table
headers=["Phase", "Fraction", "Guess for r"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

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

We can see that the algorithm gets it right twice, that is, 4. In the rest, we were unlucky, and got 0 for s, or got a number like 2 that was a factor of the actual r. Thus, we need to realize that Shor's Algorithm is a probabilistic process, it gives us the answer sometimes, but on the bright side the probability of getting the right answer is greater than getting a wrong one.
Thus, we need to run this algorithm multiple times as a fail-safe.

```python
def qpe_amod15(a):
    """Performs quantum phase estimation on the operation a*r mod 15.
    Args:
        a (int): This is 'a' in a*r mod 15
    Returns:
        float: Estimate of the phase
    """
    N_COUNT = 8
    qc = QuantumCircuit(4+N_COUNT, N_COUNT)
    for q in range(N_COUNT):
        qc.h(q)     # Initialize counting qubits in state |+>
    qc.x(N_COUNT) # And auxiliary register in state |1>
    for q in range(N_COUNT): # Do controlled-U operations
        qc.append(c_amod15(a, 2**q),
                 [q] + [i+N_COUNT for i in range(4)])
    qc.append(qft_dagger(N_COUNT), range(N_COUNT)) # Do inverse-QFT
    qc.measure(range(N_COUNT), range(N_COUNT))
    # Simulate Results
    aer_sim = AerSimulator()
    # `memory=True` tells the backend to save each measurement in a
    list
    job = aer_sim.run(transpile(qc, aer_sim), shots=1, memory=True)
    readings = job.result().get_memory()
    print("Register Reading: " + readings[0])
    phase = int(readings[0],2)/(2**N_COUNT)
    print(f"Corresponding Phase: {phase}")
    return phase
```

```
from math import gcd
a = 0
N = 15
while gcd(a,N) != 1:
    a = randint(2, 15)
print(f"a: {a}")
FACTOR_FOUND = False
ATTEMPT = 0
while not FACTOR_FOUND:
    ATTEMPT += 1
    print(f"\nATTEMPT {ATTEMPT}:")
    phase = qpe_amod15(a) # Phase = s/r
    frac = Fraction(phase).limit_denominator(N)
    r = frac.denominator
    print(f"Result: r = {r}")
    if phase != 0:
        # Guesses for factors are gcd(x^(r/2) +/-1 , 15)
        guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
        print(f"Guessed Factors: {guesses[0]} and {guesses[1]}")
        for guess in guesses:
            if guess not in [1,N] and (N % guess) == 0:
                # Guess is a factor!
                print(f"*** Non-trivial factor found: {guess} ***")
                FACTOR_FOUND = True
```

```
a: 11

ATTEMPT 1:
Register Reading: 11000000
Corresponding Phase: 0.75
Result: r = 4
Guessed Factors: 3 and 5
*** Non-trivial factor found: 3 ***
*** Non-trivial factor found: 5 ***
```

This is the infamous shor's algorithm, in full.

3.3.3 The Rise of Quantum Computers

So, here we are, having just broken most of modern-day encryption. Much like the sequels, this might be disappointing, but, luckily for us, we can put off worrying about that because at present, quantum computers are not large enough to factor the large numbers typically used in encryption. Right now, 15 and 21 are the peak of what we can factor using shor's algorithm.

Chapter 4

Grover's Search Algorithm

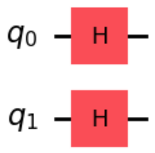
Suppose you have N numbered boxes, in a random order with no structure whatsoever, and we want to find one specific box which has a unique property, say, the number is a root of an equation $f(x) = 0$. We call this item the *marked item*. Using classical computation, on average we can locate the box in $N/2$ tries, with the worst case scenario being N tries. However, on a quantum computer, we can achieve a *quadratic* speed up, using $\sqrt{N}$ steps instead of N.

Grover's Algorithm works on the principle of creating a superposition of all elements, and then amplifying the amplitude of our marked element, using a certain number of oracle calls. We shall discuss the algorithm by dividing it into three parts, and then talk about its geometrical interpretation.

4.1 Init

The first part of Grover's Search is initialization, or state-preparation. We initialize our n(where $2^n = N$) qubits, each to the hadamard stage, so that our initial statevector becomes an equal superposition of all our N possible states.

Suppose we have 2 qubits. We will initialize them like:



This makes our initial statevector

$$\frac{1}{\sqrt{N}} \sum_{r=0}^{N-1} |r\rangle$$

4.2 Oracle

The second step in the grover's algorithm is the oracle calling step. We first define the oracle

$$U_\omega|x\rangle = \begin{cases} |x\rangle & \text{if } x \neq \omega \\ -|x\rangle & \text{if } x = \omega \end{cases}$$

, where ω is our marked element. This oracle will be a diagonal matrix, where the entry that correspond to the marked item will have a negative phase. For example, if we have three qubits and $\omega = 101$, our oracle will have the matrix:

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

4.3 Diffusion

In order to complete the circuit we need to implement the additional reflection $U_s = 2|s\rangle\langle s| - 1$. Since this is a reflection about $|s\rangle$, we want to add a negative phase to every state orthogonal to $|s\rangle$.

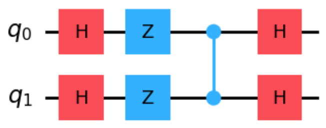
One way we can do this is to use the operation that transforms the state $|s\rangle \to |0\rangle$, which we already know is the Hadamard gate applied to each qubit:

$$H^{\otimes n}|s\rangle = |0\rangle$$

Then we apply a circuit that adds a negative phase to the states orthogonal to $|0\rangle$:

$$U_0 \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle - |11\rangle)$$

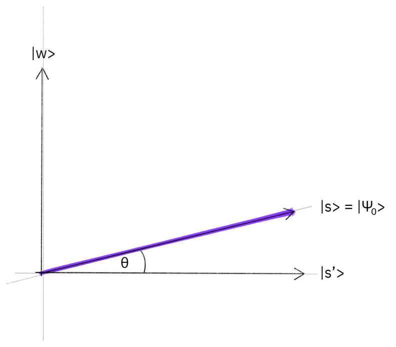
i.e. the signs of each state are flipped except for $|00\rangle$. As can easily be verified, one way of implementing U_0 is the following circuit:

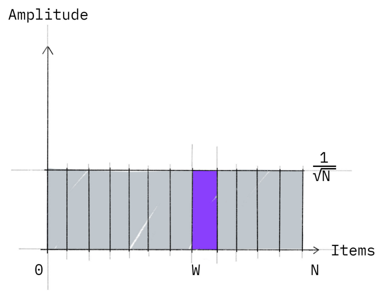


We have now understood all of the components in full, but it might still be beneficial to learn about

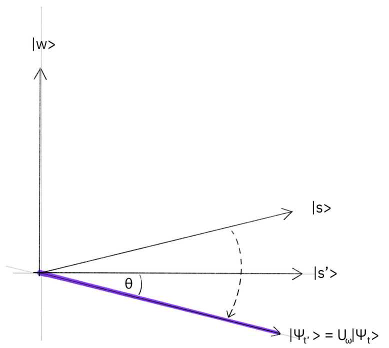
4.4 The Geometric Interpretation

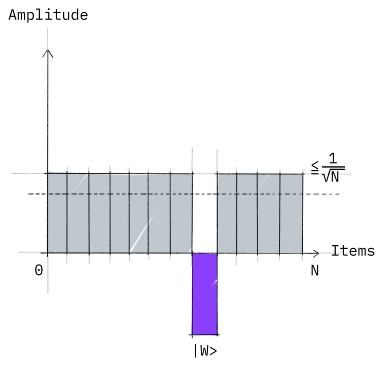
We can define a 2-dimensional state space with the state $|w\rangle$ and the state orthogonal to it, which we can call$|s'\rangle$. Essentially, the init stage is basically done to construct an equal superposition of all elements, something like:



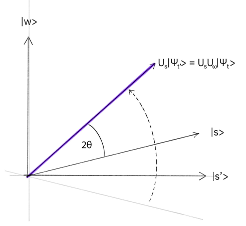


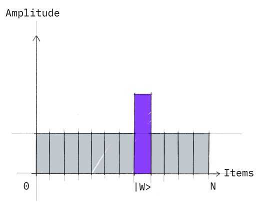
We can easily see that since $\langle w|s\rangle = \frac{1}{\sqrt{N}}, \sin\theta = \frac{1}{\sqrt{N}}$. Now, on applying the oracle U_ω, this happens:





, since the oracle inverts the wanted element, essentially *marking* it. Then, we apply the diffusion subroutine, U_s, which *flips* the vector about the equiprobable state $|s\rangle$.





Thus, we can see that our vector inches closer to the wanted state $|\omega\rangle$. The great part about this geometric representation is that we can calculate the number of times we need to apply the algorithm to reach as close as we can to the wanted state. In this case, we want

$$\begin{aligned}(2t+1)\theta &= \frac{\pi}{2} \\ \implies t &= \frac{\pi}{4\theta} - \frac{1}{2} \\ \implies t &= \lfloor \frac{\pi}{4\arcsin\frac{1}{\sqrt{N}}} \rfloor \\ \implies t &= \lfloor \frac{\pi\sqrt{N}}{4} \rfloor\end{aligned}$$

Thus, this algorithm offers a *quadratic speedup* compared to a regular classical algorithm.

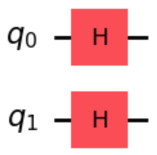
4.5 Code

```python
#initialization
import matplotlib.pyplot as plt
import numpy as np
import math

# importing Qiskit
from qiskit import transpile
from qiskit_aer import AerSimulator
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
# from qiskit.providers.ibmq import least_busy

# import basic plot tools
from qiskit.visualization import plot_histogram
```

```python
n = 2
grover_circuit = QuantumCircuit(n)
```

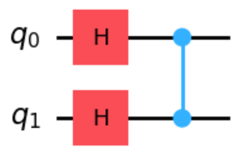
```python
def initialize_s(qc, qubits):
    """Apply a H-gate to 'qubits' in qc"""
    for q in qubits:
        qc.h(q)
    return qc
```

```python
grover_circuit = initialize_s(grover_circuit, [0,1])
grover_circuit.draw('mpl')
```

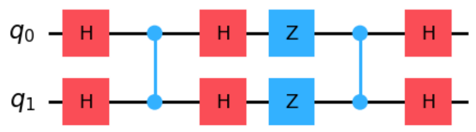


Our marked state is 3, i.e. $|11\rangle$.

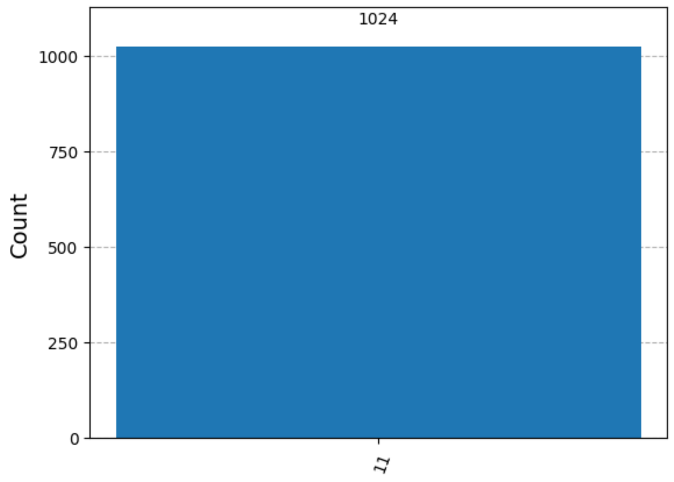
```
grover_circuit.cz(0,1) # Oracle
grover_circuit.draw('mpl')
```



```
# Diffusion operator (U_s)
grover_circuit.h([0,1])
grover_circuit.z([0,1])
grover_circuit.cz(0,1)
grover_circuit.h([0,1])
grover_circuit.draw('mpl')
```



```
grover_circuit.measure_all()

qasm_sim = AerSimulator()
result = qasm_sim.run(grover_circuit).result()
counts = result.get_counts()
plot_histogram(counts)
```



Thus, we get 3 with 100% probability.