Introduction to Coding Quantum Algorithms: A Tutorial Series Using Qiskit

Daniel Koch*, Laura Wessing, Paul M. Alsing
Air Force Research Lab, Information Directorate, Rome, New York

As the field of Quantum Computing continues to grow, so too has the general public's interest in testing some of the publicly available quantum computers. However, many might find learning all of the supplementary information that goes into quantum algorithms to be a daunting task, and become discouraged. This tutorial is a series of lessons, aimed to teach the basics of quantum algorithms to those who may have little to no background in quantum physics and/or minimal knowledge of coding in python. Each lesson covers select physics/coding topics needed for writing quantum algorithms, eventually building up a toolset for tackling more and more challenging quantum algorithms. This tutorial series is designed to provide readers from any background with two services: 1) A concise and thorough understanding of some of the most popular/academically important quantum algorithms. 2) A fluent understanding of how to write code for quantum algorithms, using IBM's publicly available Qiskit.

*corresponding author: daniel.koch.10.ctr@us.af.mil

Code files available upon request.

Table of Contents

Lesson 1 - Intro to QuantumCircuits

Welcome to lesson 1 in this tutorial series. These lessons are designed to supplement existing literature in the field of Quantum Computing and Quantum Information [1–6], with an emphasis in coding quantum algorithms.

This first lesson is designed to introduce you to Qiskit's formalism for running quantum circuits, specifically creating quantum systems using QuantumCircuits and measurements. This lesson is recommended for first time users of Qiskit. If you do not have Qiskit ready for use on your computer, please check out the installation guide:

https://qiskit.org/documentation

https://github.com/Qiskit/qiskit-terra

For those who are already familiar with the basics of Qiskit, I recommend starting with lesson 4, which will cover some additional custom functions necessary before proceeding onto the quantum algorithms.

In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, execute, Aer
import Our_Qiskit_Functions as oq
import numpy as np
import math as m
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

Creating Our First Quantum State

Qiskit is a python language that allows us to create algorithms for a quantum computer. These algorithms tell the quantum computer what kinds of quantum systems to create, and then manipulate them with gates. Compared to classical algorithms, we will find that programming for a quantum computer is quite different, requiring us to face many new limitations posed on us by quantum systems. In turn however, these quantum algorithms allow us to solve problems much faster than any classical approach.

Let's start with the simplest quantum system there is:

$$|\Psi\rangle = |0\rangle$$

This is a quantum system of 1 qubit, in the state $|0\rangle$. Not terribly exciting, but we have to start somewhere! Consider this the "Hello World!" to programing with qubits.

Let's see the code that generates this system, and then dissect its components:

```
from qiskit import QuantumRegister, QuantumCircuit, Aer, execute

q = QuantumRegister(1)
hello_qubit = QuantumCircuit(q)

hello_qubit.iden(q[0])

job = execute(hello_qubit, S_simulator)
result = job.result()
result.get_statevector()
```

```
array([1.+0.j, 0.+0.j])
```

Congrats, you've just created your first quantum system using Qiskit!

$*$ *crickets chirping* $*$

Okay, it's not a very exciting result, but there are already a lot of things going on in this code. Starting with our imports:

```
from qiskit import QuantumRegister, QuantumCircuit, Aer, execute
```

These imports are what allow us to create and see the quantum system we are working with.

QuantumRegister – this is a class that holds our qubits. When we go to perform gate operations on our system, we call on the QuantumRegister's index locations, corresponding to the qubits we are interested in.

QuantumCircuit – this is a class that can be thought of as our "instructions" for the quantum system. As we want to design larger and more complex algorithms, we will *store* operations into $\mathrm{QuantumCircuits}$, which we can then call upon by simulators to run them later.

Aer – this is a class that handles using classical simulator backends. Since we will be doing all of our tutorials via classical simulations, we will be using this class regularly. The actual name for this class is $\mathrm{AerProvider}$, but qiskit just lets us import and use it as Aer.

execute – this is a function that we must import in order to run our quantum algorithms. By itself, a $\mathrm{QuantumCircuit}$ is like a list that hold all our our quantum operations. Therefore, $\mathrm{execute}$ is what will allow us to *run* these instructions.

S_simulator - this is a variable that we have created for the purpose of storing our classical simulator. Essentially, we use the Aer class and call upon a specific backend: 'statevector_simulator'. This backend is what will allow us to view the wavefunction of our quantum system, and is one of two backends that we will use frequently throughout these tutorials.

The goal of this lesson is to become familiar with some of the basics of building and running $\mathrm{QuantumCircuits}$, so don't worry if all of these new terms don't make sense just yet.

Now, let's start with our first three lines of code:

```
q = QuantumRegister(1)
hello_qubit = QuantumCircuit(q)
hello_qubit.iden(q[0])
```

The first line of code is creating a QuantumRegister of 1 qubit, and calling it 'q'. In the next line, we create a $\mathrm{QuantumCircuit}$ called 'hello_qubit', using the quantum register we just created. And lastly, we apply the Identity operator to our single qubit, using the function **iden**, and specifying that we want this Identity operation to be applied to q[0] (We will cover the Identity operator in more detail shortly). The indexing on the $\mathrm{QuantumRegister}$ works the same way as Python ordering, where the first entry is always 0.

These three lines of code are a good template for the basic flow of creating a quantum algorithm in Qiskit: 1) define how many qubits you want 2) store them in a $\mathrm{QuantumRegister}$ 3) create a $\mathrm{QuantumCircuit}$ using all (or just some) of the qubits in your quantum register 4) apply gate operations, measurements, etc.

By default, when we create a $\mathrm{QuantumCircuit}$ of N qubits, all of the qubits start off in the state $|0\rangle$. But, they aren't technically in our system until we apply at least one gate operation to them. Thus, in the example above, in order to create our state $|\Psi\rangle = |0\rangle$, we must apply the Identity gate.

Now onto the remaining lines of code:

```
job = execute(hello_qubit, S_simulator)
result = job.result()
result.get_statevector()
```

In Qiskit, we create QuantumCircuits, but by themselves they do not represent any physical quantum system. They are just a set of instructions, so we must tell Qiskit what we want to do with them, or more specifically, *on what* we want to run them. Our choices for how we can run our quantum circuits come in the form of 'backends'. In our example, we want to run our QuantumCircuit on a classical simulator so that we can see its wavefunction.

Let's now focus solely on the backend that we will be working with: **statevector_simulator**. The following cell of code showcases several features of this backend object:

```
S_simulator = Aer.backends(name='statevector_simulator')[0]

print('simulator: ',S_simulator,'\n')
print('simulator type: ',type(S_simulator),'\n')
print('Aer.backend(name=statevector_simulator): ',Aer.backends(name='statevector_simulator'))
```

```
simulator:  statevector_simulator

simulator type:  <class 'qiskit.providers.aer.backends.statevector_simulator.StatevectorSimulator'>

Aer.backend(name=statevector_simulator):  [<StatevectorSimulator('statevector_simulator') from AerProvider()>]
```

To summarize what is going on here, the single line of code at the top is assigning the class **StatevectorSimulator** to our variable 'S_simulator'. We do this with the function **backends**, which is part of Aer. This StatevectorSimulator class is what is going to allow us to see our wavefunction at the end of our code, simulating the quantum state classically via statevector_simulator. We get this class via the line:

```
Aer.backends(name='statevector_simulator')[0]
```

which returns a class object, as shown above. In essence, all we really need to know is that this first line of code is correctly grabbing the backend we want, and storing it in a variable which we can call upon at any time.

Our last three lines of code then do the rest of the work, converting our QuantumCircuit into a printable wavefunction for us to view. Understanding the full details of this process isn't really necessary for our educational purposes here, but if you are interested, I encourage you to look at the source code. Essentially, the instructions of our QuantumCircuit go through two more classes before finally coming out as a printable wavefunction:

$$\text{execute}(\ \mathit{QuantumCircuit}, \mathit{backend}\ ) \ \rightarrow\ \text{job}\ \rightarrow\ \text{result}\ \rightarrow\ \text{display the results}$$

where the job and results step in our code are the classes:

```
S_simulator = Aer.backends(name='statevector_simulator')[0]

job = execute(hello_qubit, S_simulator)
print('   job = AerJob class: ',type(job))
result = job.result()
print('result = Result class: ',type(result))
```

```
   job = AerJob class:  <class 'qiskit.providers.aer.aerjob.AerJob'>
result = Result class:  <class 'qiskit.result.result.Result'>
```

And **get_statevector** is a function defined in the **Result** class, which prints our wavefunction as an array:

```
result.get_statevector()
```

```
array([1.+0.j, 0.+0.j])
```

If everything just now didn't sink in, don't worry. We've just gotten through all the technically stuff first, for those who might be so inclined as to rummage through Qiskit's code for themselves. If you're not so interested in *how* Qiskit works, and want to learn *how to* get Qiskit to work, don't worry, there's plenty of that left in this tutorial!

Let's Bump Up the Qubits

Returning now to creating quantum systems, so far we've seen how to create a 1-qubit system (pretty exciting, I know). Since we just spent quite a bit of time looking at all of the components in detail, let's see it once again in its entirety:

```
q = QuantumRegister(1)
hello_qubit = QuantumCircuit(q)

hello_qubit.iden(q[0])

job = execute(hello_qubit, S_simulator)
result = job.result()
result.get_statevector()
```

```
array([1.+0.j, 0.+0.j])
```

In this first example, we created a system of a single qubit in the state $|0\rangle$. This was done by simply creating a $\mathrm{QuantumRegister}$ object of 1 qubit, and using it to create a $\mathrm{QuantumCircuit}$ using the Identity operator.

Let's create another simple state, $|\psi\rangle = |000\rangle$, which contains three qubits all in the $|0\rangle$ state:

```
q = QuantumRegister(3)
three_qubits = QuantumCircuit(q)

three_qubits.iden(q[0])
three_qubits.iden(q[1])
three_qubits.iden(q[2])

job = execute(three_qubits, S_simulator)
result = job.result()
result.get_statevector()
```

```
array([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j])
```

Note that in both these examples we are able to use our $\mathrm{S\_simulator}$, since we've already defined it earlier.

Now, in this example we create a $\mathrm{QuantumCircuit}$ of three qubits. Then, since we want each qubit to be in the $|0\rangle$ state, we apply the Identity gate to each one. Using this $\mathrm{QuantumCircuit}$, we create a **job** via $\mathrm{execute}$, create a **result** from that job, and then display the results via our $\mathrm{get\_statevector}$ function. The result is our wavefunction, printed as a length-8 array.

Although there are no labels telling us which states are which in our wavefunction array, we can deduce that the first entry must be the state $|000\rangle$, since it has an amplitude of 1. However, it's not immediately clear as to which entries represent the remaining states. For clarity, the order in which the states are represented above are as follows:

$$\left[\ |000\rangle\ ,\ |100\rangle\ ,\ |010\rangle\ ,\ |110\rangle\ ,\ |001\rangle\ ,\ |101\rangle\ ,\ |011\rangle\ ,\ |111\rangle\ \right]$$

where the order of this qubits is from left to right. Thus, the state $|100\rangle$, where qubit 0 is in the$|1\rangle$ state, can be created as follows:

```
q = QuantumRegister(3)
three_qubits = QuantumCircuit(q)

three_qubits.x(q[0])
three_qubits.iden(q[1])
three_qubits.iden(q[2])

job = execute(three_qubits, S_simulator)
result = job.result()
result.get_statevector()
```

```
array([0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j])
```

Note, the X gate used here flips a qubit's state between 0 and 1 (which we will cover later in this lesson). The array above confirms that the state $|100\rangle$ is indeed located at the index location 1.

Still, these arrays aren't the easiest thing to work with, especially when we start to working with larger systems. Thus, I will offer an alternative here. Rather than working with these statevector arrays, let's import and use a function called **Wavefunction**, from the additional python file accompanying these tutorial lessons: **Our_Qiskit_Functions**.

```
q = QuantumRegister(3)
three_qubits = QuantumCircuit(q)

three_qubits.x(q[0])
three_qubits.iden(q[1])
three_qubits.iden(q[2])

oq.Wavefunction(three_qubits)
```

```
1.0 |100>
```

This custom function will allow us to see the states of our system, written in standard ket notation. In addition, we only need to pass our QuantumCircuit object, which makes our code a bit tidier.

Now, let's be clear about where this function came from. As part of this tutorial series, we will frequently be calling upon functions from the python file Our_Qiskit_Functions.py, which we have included alongside these tutorials and Qiskit. This python file is *not* a part of IBM's Qiskit. It is a python file filled with custom functions designed to coincide with these lessons, for learning purposes.

Our First Superposition State

Now that we can create multiple qubits, we want to *do something* with them. In quantum algorithms, that something is applying gates and making measurements. We've already seen one gate so far, I – the identity operator. The QuantumCircuit class comes with several pre-programed gates for us to use. For a complete list and explanation of all the gates that come standard with QuantumCircuits, see lesson 3 in this tutorial series.

For this intro lesson we will only be using the gates I, X, and H, and we will briefly explain them here:

I (Identity Gate)

As we already saw, this gate acts on a single qubit, and leaves its state unchanged. The matrix for this gate is:

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

While perhaps uninteresting in itself, the I gate is still an essential component for algorithms. We've already seen it used so far to initialize nice simple quantum states, and later we shall see it used in conjunction with other gates, for larger multi-qubit operations.

X (NOT Gate)

This gate is our quantum analog to the NOT gate, which flips a classical bit between 0 and 1. Here, it achieves the same effect with the states $|0\rangle$ and $|1\rangle$. The matrix for this operation is given by:

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

Although it may appear that the X gate is perfect analog to the classical NOT gate, quantum mechanics prevents it from being so. In particular, when we start to create superposition states, we will see that using this gate to flip qubits becomes a bit tricky.

H (Hadamard Gate)

This final gate is going to allow us to create our first superposition state. In particular, the Hadamard gate takes a qubit and splits it into a 50-50 probability distribution between the states $|0\rangle$ and $|1\rangle$. Mathematically, it looks like this:

$$H\,|\,0\rangle \;=\; \frac{1}{\sqrt{2}}\big(\,|\,0\rangle \;+\; |\,1\rangle\,\big)$$

$$H\,|\,1\rangle \;=\; \frac{1}{\sqrt{2}}\big(\,|\,0\rangle \;-\; |\,1\rangle\,\big)$$

which is accomplished by the following matrix:

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Let's see an example:

```
q = QuantumRegister(1)
H_circuit = QuantumCircuit(q)

H_circuit.h(q[0])

oq.Wavefunction( H_circuit )
```

```
0.70711 |0>    0.70711 |1>
```

Sure enough, our qubit is in a superposition state! Our qubit has a 50% chance of being in the state $|0\rangle$ or $|1\rangle$.

Note: The numbers attached to the states here are the system's amplitudes, not probabilities. When working with quantum states, probabilities are always the *observables* that we see, but the amplitudes are the inner workings that really matter. Here, each state has an amplitude of $\frac{1}{\sqrt{2}}$, which when squared, tells us that each state has a probability of $\frac{1}{2}$.

Now let's try making a superposition state of 2 qubits:

```
q = QuantumRegister(2)
H_circuit = QuantumCircuit(q)

H_circuit.h(q[0])
H_circuit.h(q[1])

oq.Wavefunction( H_circuit )
```

```
0.5 |00>    0.5 |10>    0.5 |01>    0.5 |11>
```

The wavefunction printed above shows an equal superposition of four states. These four states come from the following mathematical state:

$$H\,|0\rangle \otimes H\,|0\rangle$$

which is the tensor product of two seperate quantum states (one for each qubit in our system). A more common way of writing this is:

$$(\,H\,|0\rangle\,) \cdot (\,H\,|0\rangle\,)$$

$$= \frac{1}{\sqrt{2}}(\,|0\rangle\,+\,|1\rangle\,) \cdot \frac{1}{\sqrt{2}}(\,|0\rangle\,+\,|1\rangle\,)$$

$$= \frac{1}{2}(\,|0\rangle\,+\,|1\rangle\,) \cdot (\,|0\rangle\,+\,|1\rangle\,)$$

$$= \frac{1}{2}(\,|0\rangle|0\rangle\,+\,|0\rangle|1\rangle\,+\,|1\rangle|0\rangle\,+\,|1\rangle|1\rangle\,)$$

which is typically written using the standard shorthand:

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

And voila! We have our superposition state resulting from two qubits, each with a Hadamard gate applied to them. Recall that a single H gate put our qubit into a 50/50 state between $|0\rangle$ and $|1\rangle$. Now, having two qubits undergo this gate, both of them in this 50/50 state, we get a combined system where any of the four individual combinations has a 25% probability.

As a side note, you may have noticed in these examples that I initialized our qubits with the Hadamard gates:

H_circuit.h(q[0])

Remember, when we assign a new qubit to the QuantumRegister, it starts off in the state $|0\rangle$ by default. However, if we want that qubit to really be a part of our QuantumCircuit, we must apply at least one gate operation to it. But, our first operator on a new qubit does not need to be the Identity operator. We can just assume that the state of the qubit is $|0\rangle$, and skip right to the next operation. The only time it is necessary to initialize a qubit with iden is when we want to specifically start it out in the state $|0\rangle$.

Consider the following example where we would like only one of the qubits to start off in a superposition:

```
q = QuantumRegister(2)
H_circuit = QuantumCircuit(q)

H_circuit.h(q[0])
H_circuit.iden(q[1])

oq.Wavefunction( H_circuit )
```

```
0.70711 |00>    0.70711 |10>
```

As shown above, qubit 0 is initialized in a mixed state, while qubit 1 remains in the state $|0\rangle$. We can see this by the fact that qubit 1's value is 0 in both states, while one state has qubit 0 in $|0\rangle$, and the other in $|1\rangle$.

If you still want to initialize qubits with iden before applying gates, go for it! Understanding code is only as easy as you make, so feel free to add steps for clarity.

Making a Measurement

Now comes the final step for creating quantum algorithms – measuring the quantum states that we create. To do this, Qiskit has a convenient way for us to measure and record the results of our quantum system.

Let's see an example in action and then backtrack to understand each component:

```
from qiskit import ClassicalRegister
M_simulator = Aer.backends(name='qasm_simulator')[0]

q = QuantumRegister(1)
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)

qc.h(q[0])
qc.measure(q,c)

job = execute(qc, M_simulator)
result = job.result()
result.get_counts(qc)
```

```
{'1': 522, '0': 502}
```

This code should look very similar to our earlier examples, but with a few key differences. Let's start with the first three lines of code, which set up our quantum system:

```
q = QuantumRegister(1)
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
```

In the first line, we are creating a QuantumRegister and calling it 'q', same as we've done before. But in the next line, we are creating something new, a **ClassicalRegister**, which we imported from qiskit at the top of the code. A ClassicalRegister is a class, very similar to our QuantumRegister. Whereas the quantum register stores qubits, the classical register stores classical bits (0's and 1's). In our example, we create a ClassicalRegister and call it 'c', which we assign to it 1 bit.

And then lastly, we create our QuantumCircuit and call it 'qc', only this time, we pass both the QuantumRegister and ClassicalRegister as arguments. Thus, now both the quantum and classical registers are a part of our quantum circuit.

Next we have the gate operators that we apply to our qubits:

```
qc.h(q[0])
qc.measure(q,c)
```

In the first line, we initialize qubit 0 with a Hadamard gate, creating a system in the following state:

$$\frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$$

So far nothing new. Then, in the next line we add our measurement, using the function **measure**. This function actually calls upon the class **Measure**, which handles adding the measurement instruction. measure takes two arguments, both the quantum and classical registers.

Note that by using the measure function, we are adding an additional instruction to our QuantumCircuit. That is to say, we aren't *actually* making a measurement with this line of code. Remember, a QuantumCircuit object is just a list of instructions, which aren't actually carried out until we run it on some simulator. Speaking of backends, because we now have a measurement instruction, the backend that we need to call upon is:

```
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

Before, we used the statevector_simulator because we were interested in viewing our wavefunction. Now, we don't care about the wavefunction, so we will instead use **qasm_simulator**, which will allow us to simulate measurements on our quantum state. Note that both of these simulators are still classical, and do not call upon any real quantum devices. We won't go through the full details of the Aer.backend function again here, but we are essentially calling upon qasm_simulator in the exact same way as before, so please see the earlier example if you are unsure as to what this line of code is doing.

The last three lines of code should look familiar as well:

```
job = execute(qc, M_simulator)
result = job.result()
result.get_counts(qc)
```

Again we are using execute to run our simulation, this time on **M_simulator** because we are interesting in measurement results. This this returns to us a job object, from which we extract our result.

The last line of code is the function **get_counts**, which can be thought of as the analogous function to get_statevector from before. Since we are using a simulator designed for measurements, this is the function that returns these measurements to us, much like how we got our wavefunction array. When we call upon this function, we get a dictionary-type object returned to us, which contains 1024 simulated measurements. The entries of the dictionary are the measurement results, and their values are the number of time that each state was measured:

```
q = QuantumRegister(1)
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)

qc.h(q[0])
qc.measure(q,c)

M = execute(qc, M_simulator).result().get_counts(qc)
print('Disctionary entry "0": ',M['0'])
print('Disctionary entry "1": ',M['1'])
```

```
Disctionary entry "0":  497
Disctionary entry "1":  527
```

Alrighty, now let's talk about the interaction between the two registers, via measure. In Qiskit, if we pass the entire quantum and classical registers as arguments to measure, the function will by default make a total measurement on the system, and store each qubit's measurement results to the corresponding index in the ClassicalRegister:

```
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q,c)

qc.h(q[0])
qc.h(q[1])
qc.measure(q,c)

M = execute(qc, M_simulator).result().get_counts(qc)
print(M)
```

```
{'10': 246, '11': 262, '00': 236, '01': 280}
```

If instead we want to only make a partial measurement, say on qubit 0 only, we can do so by specifying the quantum and classical indices:

```
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q,c)

qc.h(q[0])
qc.h(q[1])
qc.measure(q[0],c[0])

M = execute(qc, M_simulator).result().get_counts(qc)
print(M)
```

```
{'01': 524, '00': 500}
```

In the example above, measure(q[0], c[1]) can be understood as "make a measurement on qubit 0, and store the result in the ClassicalRegister 'c' index 0". The result printed shows that the measurement on a single qubit was successful, after which our system is still left in a superposition of two states (because qubit 1 also had a Hadamard gate).

But, maybe you noticed that one thing is off in this example... the states are backwards! (well, sort of) To show what we mean, take a look at the example below, which creates the state $|\psi\rangle = |01\rangle$:

```
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q,c)

qc.iden(q[0])
qc.x(q[1])
qc.measure(q,c)

M = execute(qc, M_simulator).result().get_counts(qc)
print(M)
```

```
{'10': 1024}
```

And now compare this to how we would print the wavefunction:

```
q = QuantumRegister(2)
qc = QuantumCircuit(q)

qc.iden(q[0])
qc.x(q[1])

oq.Wavefunction( qc )
```

```
1.0 |01>
```

In both examples, we prepare our system in the exact same way, but the results get printed in reverse. Neither of them are wrong, we just need to understand what is happening with our classical register. In classical computing, a bit string is usually defined such that the LSB (least significant bit) is the right-most index. For example, the number 11 in binary would be 1011, which can be read as ' 8(1) + 4(0) + 2(1) + 1(1) ', where the bits representing the smallest numbers start from the right.

In Qiskit, the ClassicalRegister works the same way. For example, if we have four qubits, the measurement results would be stores as [qubit 3, qubit 2, qubit 1, qubit 0]. Thus, when we put into our code measure(q[0], c[0]), this would reads as 'measure qubit 0, and store it in the classical register index 0, (the rightmost index)'.

Just like earlier with Wavefunction, we will again be calling upon a custom function, **Measurement** here to correct for this:

```
from Our_Qiskit_Functions import Measurement

q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q,c)

qc.iden(q[0])
qc.h(q[1])
qc.measure(q,c)

oq.Measurement( qc, shots=1024 )
```

```
531|01>    493|00>
```

Looks pretty good, the number of measurement counts corresponding to a state are printed alongside that state (not to be confused with amplitudes). And more importantly, the labeling of the states matches Wavefunction, such that qubit 0 is the leftmost index. Once again, this function is purely cosmetic, but for learning purposes this function comes with several extra tools that we will take advantage of later (see lesson 4).

When using Measurement, we can control the number of times we make simulated measurements by using the argument **shots**:

```
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q,c)

qc.iden(q[0])
qc.h(q[1])
qc.measure(q,c)

oq.Measurement( qc, shots=100 )
```

```
51|01>    49|00>
```

As a final note on measurements, passing both the quantum and classical registers into measure assumes that we intend to store our measurement results in the same index locations as the corresponding qubits. That is to say, when we measure the state of q[1] (qubit 1), we want to store that measurement result in c[1] (ClassicalRegister index 1). Normally, this is what we will always do, but it's worth pointing out that we *can* tell Qiskit to store our measurement results elsewhere.

To do this, all we need to do is specify where in the ClassicalRegister we would like to store the measurement results of each qubit, via the measure function:

```
q = QuantumRegister(3)
c = ClassicalRegister(3)
super0 = QuantumCircuit(q,c)

super0.h(q[0])
super0.iden(q[1])
super0.iden(q[2])

super0.measure(q[0],c[1])
super0.measure(q[1],c[0])
super0.measure(q[2],c[2])

oq.Measurement(super0, shots=100)
```

```
42|010>    58|000>
```

The results of this code show that there were counts in the entries '010' and '000'. So for this example, every time a measurement found qubit 0 in the $|1\rangle$ state, our measure function stores the result in the location c[1]. Simultaneously, we stored the measurement results of qubits 1 and 2 in the locations 0 and 2 respectively. Thus, when we go and check our results, we see all zeros for these qubit indices.

Technically, we know that this state wasn't actually a part of our system though. Qubit 1 was always prepared in the state $|0\rangle$. Thus, when when we read the result '010', we must be careful not to immediately interpret this result as the state $|010\rangle$. In principle, we should always check to see how a code is prepared, and see which qubits get stored in which location indices.

Now, in practice, we will rarely use this process of storing qubit results in various places. For the rest of these tutorials, we will almost always store our measurement results in the corresponding classical index locations, primarily so that our Wavefunction

and Measurement functions will always be in agreement about states. But it's good to know that we *can* do this if we wanted to.

Perfect Coin Algorithm

Now that we know how to create and run QuantumCircuits, and view our result with either Wavefunction or Measurement, we've finished all of our intro topics! In the next lesson, we will go over some more advanced things we can do with QuantumCircuits. But as our final exercise here, we will create a silly quantum algorithm to settle a gambling bet between Alice and Bob:

"Alice and Bob have recently gotten into an argument about the philosophy of picking the correct side of a coin flip. Bob was raised by the moto "Tails Never Fails", while Alice was taught "Tail Always Fails". Alice suggests that they solve their disagreement with a series of coin flips, but Bob doesn't trust any coin that Alice owns, and vice versa for Alice. Thus, they agree to use a qubit as their coin. The loser of the bet must clean the other person's lab equipment for a month!"

Below is a function that will simulate one 'Quantum Coin' flip, using a single qubit in a superposition state:

```
def Quantum_Coin_Flip(flips):
    '''
    Simulates a perfect coin, measuring heads or tails, using a qubit
    '''
    q = QuantumRegister(1)
    c = ClassicalRegister(1)
    perfect_coin = QuantumCircuit(q,c)

    perfect_coin.h(q[0])
    perfect_coin.measure(q,c)

    M = execute(perfect_coin, M_simulator, shots=flips).result().get_counts(perfect_coin)
    heads = M['0']
    tails = M['1']
    return heads,tails
```

This function incorporates all of the topics we've seen thus far. In total, it achieves the following steps:

1) Creates QuantumRegister and ClassicalRegister objects of one qubit

2) Creates a QuantumCircuit using these registers

3) Puts the qubit in a 50/50 superposition state and makes a measurement

4) Simulates the QuantumCircuit the desired number of times

5) Extracts the measurement result

6) Returns two variables: 'heads' and 'tails', which hold integer values

Let's now try out this function, using 100 coin tosses:

```
Heads,Tails = Quantum_Coin_Flip(100)

if(Heads >  Tails):
    print('Looks like Bob has some cleaning to do!')
if(Heads <  Tails):
    print('Tough luck Alice, Tails never Fails!')
if(Heads == Tails):
    print('Stupid quantum coin... Im going home')
print('  ')
print('Final Score -- Alice: ',Heads,' Bob: ',Tails)
```

```
Tough luck Alice, Tails never Fails!

Final Score -- Alice:  47  Bob:  53
```

This concludes our introduction lesson to Qiskit! I hope that the examples in this lesson provide a good starting point for using Qiskit. Before moving on to the next tutorial, I strongly encourage you to write some simple code of your own, and test out all of the various functions and classes we studied here.

Lesson 2 - Creating More Complex QuantumCircuits

In this lesson, we will continue to cover some of the common tools provided by Qiskit for writing quantum algorithms. For a review on the basics of using QuantumRegisters, QuantumCircuits, etc. please check out lesson 1 in this tutorial series:

Lesson 1 - Intro to QuantumCircuits

In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
from qiskit.tools.visualization import circuit_drawer
import Our_Qiskit_Functions as oq
import numpy as np
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

Observing / Editing QuantumCircuits

In lesson 1 we've already covered the basics of how to create quantum systems, view their wavefunctions, and view simulated measurements. Now, we are going to cover some more advanced topics in Qiskit, which will improve our coding abilities / provide us with some new tools. We will begin this tutorial with some functions that will be very handy for viewing / debugging our codes.

qasm

Recall from lesson 1 that a QuantumCircuit is essentially a list of instructions, meant to be run on some backend. If at any point we would like to see the contents of our QuantumCircuit, Qiskit comes with a nice function that allows us to print a QuantumCircuit to console, via **qasm**:

```
q = QuantumRegister(3)
c = ClassicalRegister(3)
super0 = QuantumCircuit(q,c)

super0.h(q[0])
super0.iden(q[1])
super0.iden(q[2])
super0.measure(q[0],c[0])

print( super0.qasm() )
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q18[3];
creg c9[3];
h q18[0];
id q18[1];
id q18[2];
measure q18[0] -> c9[0];
```

The function qasm, part of the QuantumCircuit class, returns a string that contains all of the instructions stored in a QuantumCircuit. When printed, this string gives us a nice and easy to read visualization our algorithm.

For example, 'qreg q[3]' shows that we have created a quantum register of 3 qubits. 'h q[0]' is the instruction telling the code to apply a Hadamard gate to qubit 0. 'measure q[0] -> c[0]' is the instruction for a measurement on qubit 0, which is then stored in the classical register index 0.

Note, the printed results that you see will certainly look a tad bit different than my examples. Namely, there is a number attached to the q's and c's, something like 'h q7[0]'. If you rerun the code, each time you will notice this number going up by one. We can assume that this is an intended feature of Qiskit, so no two QuantumCircuits or registers get mixed up.

If we want to, we can clean up the default printed template ever so slightly in two ways: 1) we can assign names to our registers and QuantumCircuits, so that we avoid names like 'q7'. 2) If we don't want the top two lines to print every time, we can skip them:

```
q = QuantumRegister(1,name='q')
c = ClassicalRegister(1,name='c')
super0 = QuantumCircuit(q,c,name='qc')

super0.h(q[0])

Inst = super0.qasm()
print(Inst[36:len(Inst)])
```

```
qreg q[1];
creg c[1];
h q[0];
```

By using the argument **name**, we can assign customized names as shown above. If we are designing an algorithm that may have multiple registers or quantum circuits, this will be very important. Also, by starting from character 36 in qasm, we can skip the first two lines, reducing some of the clutter.

Extracting from QuantumCircuits

Now that we know how to use qasm to print our QuantumCircuit's elements in a user-friendly way, let's take a look at some other functions built into the QuantumCircuit class for extracting information:

```
q = QuantumRegister(2,name='q')
c = ClassicalRegister(3,name='c')
two_q = QuantumCircuit(q,c,name='qc')

two_q.h(q[0])
two_q.h(q[1])
two_q.measure(q[0],c[0])

print('_____QuantumCircuit.qasm()_____')
print(two_q.qasm()[36:len(two_q.qasm())])

print('_____QuantumCircuit.data_____')
print(two_q.data)

print('\n_____QuantumCircuit.qregs_____')
print(two_q.qregs)

print('\n_____QuantumCircuit.cregs_____')
print(two_q.cregs)
```

```
_____QuantumCircuit.qasm()_____
qreg q[2];
creg c[3];
h q[0];
h q[1];
measure q[0] -> c[0];

_____QuantumCircuit.data_____
[<qiskit.extensions.standard.h.HGate object at 0x000001A321F93710>, <qiskit.extensions.standard.h.HGate object at 0x000001A321F
93550>, <qiskit.circuit.measure.Measure object at 0x000001A321F934A8>]

_____QuantumCircuit.qregs_____
[QuantumRegister(2, 'q')]

_____QuantumCircuit.cregs_____
[ClassicalRegister(3, 'c')]
```

The code above contains four different functions that can all be used to extract information from a QuantumCircuit. These are very useful for debugging code, and I encourage you to try them all out. To summarize what each function does:

1) **qasm** - function for printing everything about a QuantumCircuit object

2) **data** - returns a list object that contains all of the gate / measurement instructions

3) **qregs** - returns a list containing all of the quantum registers

4) **cregs** - returns a list containing all of the classical registers

Amending QuantumCircuits

Now that we know how to view our $\mathrm{QuantumCircuits}$, let's look at some ways of editing them. Typically, quantum algorithms are very rigid, where all of the steps in the algorithm are deliberate from start to finish. This is partly due to the nature of quantum systems and the way measurements collapse wavefunctions, which in turn means they lack the ability to 'go back' and change based on measurement results.

Nevertheless, supposing we *would* like to edit the instructions in a $\mathrm{QuantumCircuit}$, doing so is quite easy. Let's start by taking a look at (2) above, the data property of $\mathrm{QuantumCircuits}$. This property returns a list object which contains all of the instructions we've given to our $\mathrm{QuantumCircuit}$. And being the list object it is, we can perform any usual python actions with it, including checking / adding / or removing certain instructions:

```
q = QuantumRegister(2,name='q')
c = ClassicalRegister(2,name='c')
qc = QuantumCircuit(q,c,name='qc')

qc.h(q[0])
qc.h(q[1])
qc.measure(q[0],c[0])
print('____Initial____')
print(qc.qasm()[36:len(qc.qasm())])

inst = qc.data[1]

del qc.data[1]
print('____del qc.data[1]____')
print(qc.qasm()[36:len(qc.qasm())])

qc.data.append(inst)
print('____qc.data.append( inst )____')
print(qc.qasm()[36:len(qc.qasm())])

qc.data.insert( 0, inst)
print('____qc.data.insert( 0, inst )____')
print(qc.qasm()[36:len(qc.qasm())])
```

```
____Initial____
qreg q[2];
creg c[2];
h q[0];
h q[1];
measure q[0] -> c[0];

____del qc.data[1]____
qreg q[2];
creg c[2];
h q[0];
measure q[0] -> c[0];

____qc.data.append( inst )____
qreg q[2];
creg c[2];
h q[0];
measure q[0] -> c[0];
h q[1];

____qc.data.insert( 0, inst )____
qreg q[2];
creg c[2];
h q[1];
h q[0];
measure q[0] -> c[0];
h q[1];
```

As we can see, by manipulating the list object **QuantumCircuit.data**, we are changing our QuantumCircuit object as we go. We can add or remove instructions as we see fit, but we must remember that we can only run it once. All of these steps should be thought of as editing instructions, not actually manipulating a quantum system in any way. Regardless of how many edits it undergoes along the way, only the final form of the QuantumCircuit is what matters to the simulator.

Sharing Registers

Now suppose we are working with an algorithm that utilizes multiple QuantumCircuits. Qiskit allows us to have QuantumCircuit objects interact in a variety of ways, so long as we are careful in defining which registers are a part of which circuits. For example, let's start with two QuantumCircuits that have no interaction:

```
q = QuantumRegister(1,name='q')
c = ClassicalRegister(1,name='c')
qc = QuantumCircuit(q,c,name='qc')

q2 = QuantumRegister(1,name='q2')
c2 = ClassicalRegister(1,name='c2')
qc2 = QuantumCircuit(q2,c2,name='qc2')

qc.h(q[0])
qc.measure(q[0],c[0])
qc2.h(q2[0])

print('____qc____')
print(qc.qasm()[36:len(qc.qasm())])
print('____qc2____')
print(qc2.qasm()[36:len(qc2.qasm())])
```

```
____qc____
qreg q[1];
creg c[1];
h q[0];
measure q[0] -> c[0];

____qc2____
qreg q2[1];
creg c2[1];
h q2[0];
```

Nothing too special here. Just two circuits defined parallel to each other. We could choose to pass either of them to a simulator, and their results would be completely independent.

Now, there are a couple ways we can have our two QuantumCircuits interact. For example, we can take measurement results from one circuit (after running it on a simulator) and store them in the other's ClassicalRegister, or have one circuit apply gate operations on the other's qubits. BUT, in order to do these kinds of things, we must give each QuantumCircuit access to each other's registers:

```
q = QuantumRegister(1,name='q')
c = ClassicalRegister(1,name='c')
qc = QuantumCircuit(q,c,name='qc')

q2 = QuantumRegister(1,name='q2')
c2 = ClassicalRegister(1,name='c2')
qc2 = QuantumCircuit(q2,c2,name='qc2')

qc.add_register(c2)
qc2.add_register(q)

qc.h(q[0])
qc2.h(q2[0])
qc.measure(q[0],c2[0])
qc2.h(q[0])

print('____qc____')
print(qc.qasm()[36:len(qc.qasm())])

print('____qc2____')
print(qc2.qasm()[36:len(qc2.qasm())])
```

```
____qc____
qreg q[1];
creg c[1];
creg c2[1];
h q[0];
measure q[0] -> c2[0];

____qc2____
qreg q2[1];
qreg q[1];
creg c2[1];
h q2[0];
h q[0];
```

Take a careful look at these two QuantumCircuits. In 'qc', we make sure to include the ClassicalRegister 'c2', so that we can store our measurement result:

'measure q[0] -> c2[0]'

In 'qc2', we do the same thing for the QuantumRegister 'q', so that we can apply a Hadamard gate:

'h q[0]'.

When we go to print each QuantumCircuit with **qasm**, sure enough, we can see that both QuantumCircuit objects have access to the registers we specified. The function that allows us to do this is **add_register**, which belongs to the QuantumCircuit class, taking a register of either type as an argument.

One way to think about the relationship between registers and circuits, is that QuantumCircuits need permission to use registers. Our quantum and classical registers are the physical quantities where our qubits and classical bits live. Thus, we should avoid thinking of our algorithms as "QuantumCircuit1's registers", but more like "QuantumCircuit1 has access to ___ registers".

Combining QuantumCircuits

Now suppose we have multiple QuantumCircuits, and we want to combine them together, or append the instructions from one to another. One nice way of handling this in Qiskit is by using '+' or '+=':

```
q1 = QuantumRegister(2,name='q1')
c1 = ClassicalRegister(3,name='c1')
qc1 = QuantumCircuit(q1,c1,name='qc1')

q2 = QuantumRegister(2,name='q2')
c2 = ClassicalRegister(3,name='c2')
qc2 = QuantumCircuit(q2,c2,name='qc2')

qc1.h(q1[0])
qc1.iden(q1[1])
qc2.iden(q2[0])
qc2.h(q2[1])

qc3 = qc1 + qc2
print('____ q3 = qc1 + qc2 ____')
print(qc3.qasm()[36:len(qc3.qasm())])

qc1 += qc2
print('____ qc1 += qc2 ____')
print(qc1.qasm()[36:len(qc1.qasm())])
```

```
____ q3 = qc1 + qc2 ____
qreg q1[2];
qreg q2[2];
creg c1[3];
creg c2[3];
h q1[0];
id q1[1];
id q2[0];
h q2[1];

____ qc1 += qc2 ____
qreg q1[2];
qreg q2[2];
creg c1[3];
creg c2[3];
h q1[0];
id q1[1];
id q2[0];
h q2[1];
```

Both cases in this example produce the same final QuantumCircuit, combining 'qc1' and 'qc2'. The difference between them, is that in the first case we store the combination of 'qc1 + qc2' as a new QuantumCircuit, called 'qc3'. In the second case, we actually append all of the instructions stored in 'qc2', into 'qc1'.

Thus, the **+** functionality combines two QuantumCircuits, to then be stored in whatever variable we choose, leaving both of the original circuits unchanged. Conversely, **+=** appends all of the instructions from the second circuit onto the first, leaving the second QuantumCircuit unaltered.

Also note that $+$ and $+=$ both add the registers as well as the instructions. In both cases, we can see all of the quantum and classical registers are present in the final QuantumCircuits. If we add together multiple QuantumCircuits that share access to same registers, or call upon different ones, these functions will automatically handle things so that the final QuantumCircuit has access to all of the necessary registers.

Visualizing Circuits

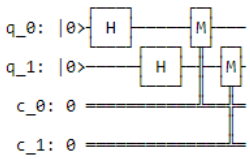
Now for our final topic (and arguably the coolest), using Qiskit's **circuit_drawer** function. This last function is purely cosmetic, having no impact on our quantum system, but allows us to visualize our QuantumCircuit in terms of gates.

Let's try it out on a simple system. We will create the state:

$$\frac{1}{2}\big(\,|00\rangle + |01\rangle + |10\rangle + |11\rangle\,\big)$$

followed my measurements on both qubits:

```
q = QuantumRegister(2,name='q')
c = ClassicalRegister(2,name='c')
qc = QuantumCircuit(q,c,name='qc')

qc.h(q[0])
qc.h(q[1])
qc.measure(q,c)

circuit_drawer(qc)
```



If it is your first time running the circuit_drawer function, you may be prompted for some extra installations before anything pops up. This function calls upon a lot of LaTeX to generate the picture. Once it runs properly, you should see a pretty impressive picture as the output! The picture you see is very similar to the IBM Q Experience.

To summarize, the picture shown above is our QuantumCircuit with all of its instructions displayed in order from left to right. Starting from the leftmost side of the picture, you should see all of the qubits and classical bits that we assigned to our registers. Then, stemming from each type of bit are horizontal lines, which represent the processes each one undergoes from start to finish.

Just like we've written in our code, we can see that the first instructions encountered by each qubit are Hadamard gates, denoted by 'H' inside a square. Then, to the right of each Hadamard gate is a second box, which contains an 'M', representing a measurement and containing a line connecting downward. These connections represent where each measurement result is stored in the classical register. Almost every gate operation / instruction that comes standard with Qiskit comes with an accompanying visualization via circuit_drawer, making it an excellent tool for learning quantum algorithms!

This concludes our second lesson of learning Qiskit! We now have most of the tools we need to start studying some famous quantum algorithms, which begin in lesson 5. The last major component missing from our toolbox is all of the standard gates provided by Qiskit, which we will cover next lesson!

Lesson 3 - Gates Provided by Qiskit

The goal of this lesson is to introduce some of the predefined gates that come standard with the QuantumCircuit class. We will go through each gate, accompanied with a short explanation and working example, and a visualization using Qiskit's circuit_drawing tool.

Before proceeding, please consider reading the previous lessons in this series, which covers all of the basics for working with Qiskit:

Lesson 1 - Intro to QuantumCircuits

Lesson 2 - Creating More Complex QuantumCircuits

In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
from qiskit.tools.visualization import circuit_drawer
import Our_Qiskit_Functions as oq
import math as m
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

Single Qubit Gates

I

The Identity Operator

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

The effect of this gate renders the qubit's state unchanged.

```
q = QuantumRegister(1,name='q')
I_qc = QuantumCircuit(q,name='qc')

I_qc.iden( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(I_qc)

I_qc.iden( q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(I_qc)

circuit_drawer(I_qc)
```

```
  __ Initial __
1.0 |0>

  __ Final __
1.0 |0>

q_0: |0>┤ Id ├┤ Id ├
```

Hadamard (H)

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The effect of this gate is as follows:

$$H\,|0\rangle \;=\; \frac{1}{\sqrt{2}}\big(\,|0\rangle + |1\rangle\,\big)$$

$$H\,|0\rangle \;=\; \frac{1}{\sqrt{2}}\big(\,|0\rangle - |1\rangle\,\big)$$

This gate results in a qubit being in a 50 / 50 superposition of states $|0\rangle$ and $|1\rangle$. While this may seem simple enough, the importance of the Hadamard gate cannot be understated. In the coming lessons, we shall see that the Hadamard gate is largely responsible for the success of many quantum algorithms.

```
q = QuantumRegister(1,name='q')
H_qc = QuantumCircuit(q,name='qc')

H_qc.iden( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(H_qc)

H_qc.h( q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(H_qc)

circuit_drawer(H_qc)
```

```
  __ Initial __
1.0 |0>

  __ Final __
0.70711 |0>    0.70711 |1>

q_0: |0>─┤ Id ├─┤ H ├─
```

Pauli Operators

X

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

The effect of this gate is to flip a qubit's state between $|0\rangle$ and $|1\rangle$. This gate can be thought of the quantum analog to flipping a classical bit (the NOT gate). In systems with many superposition states, this gate will be very useful in isolating particular states for future operations.

```
q = QuantumRegister(1,name='q')
X_qc = QuantumCircuit(q,name='qc')

X_qc.iden( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(X_qc)

X_qc.x( q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(X_qc)

circuit_drawer(X_qc)
```

```
  __ Initial __
1.0 |0>

  __ Final __
1.0 |1>

q_0: |0>┤ Id ├┤ X ├
```

Y

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

The effect of this gate is to flip a qubit's $|0\rangle$ and $|1\rangle$ amplitudes and multiplies by an imaginary number (phase). From a probabilities perspective, this gate has the same effect as the X gate. However, the additional phase makes this gate very useful in creating certain constructive / deconstructive interferences.

```
q = QuantumRegister(1,name='q')
Y_qc = QuantumCircuit(q,name='qc')

Y_qc.iden( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(Y_qc)

Y_qc.y( q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(Y_qc)

circuit_drawer(Y_qc)
```

```
  __ Initial __
1.0 |0>

  __ Final __
1.0j |1>

q_0: |0>┤ Id ├┤ Y ├
```

Z

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

The effect of this gate leaves a qubit's $|0\rangle$ amplitude unchanged, while multiplying by -1 (phase) to a qubit's $|1\rangle$ amplitude. The power of this gate comes from the fact that it only affects the $|1\rangle$ component, which will be frequently used for picking out certain states in the system while leaving others unaltered.

```
q = QuantumRegister(1,name='q')
Z_qc = QuantumCircuit(q,name='qc')

Z_qc.h( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(Z_qc)

Z_qc.z( q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(Z_qc)

circuit_drawer(Z_qc)
```

```
  __ Initial __
0.70711 |0>    0.70711 |1>

  __ Final __
0.70711 |0>    -0.70711 |1>

q_0: |0>┤ H ├┤ Z ├
```

Phase Gates

The following series of gates are all single qubit operations, which multiply a qubit's $|1\rangle$ state component by a phase. Doing so does not change the probability of the system, but is an essential component for algorithms that rely on particular kinds of interference.

PHASE (R_ϕ)

$$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}$$

A gate similar to the Z gate. It leaves a qubit's $|0\rangle$ amplitude unchanged, while multiplying by a phase $e^{i\phi}$ to a qubit's $|1\rangle$ amplitude. In Qiskit, this gate goes by the name 'U_1'. This gate will find many of the same uses as the Z gate, picking out certain states while leaving others unchanged. However, the extra degree of phase is a powerful tool for creating certain interference effects.

```
q = QuantumRegister(1,name='q')
u1_qc = QuantumCircuit(q,name='qc')

u1_qc.h( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(u1_qc)

u1_qc.u1( m.pi/4, q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(u1_qc)

circuit_drawer(u1_qc)
```

```
  __ Initial __
0.70711 |0>    0.70711 |1>

  __ Final __
0.70711 |0>    0.5+0.5j |1>

q_0: |0>┤ H ├┤ U1(0.7854) ├
```

S

$$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

A pre-defined gate for R_ϕ, ϕ=$\frac{\pi}{2}$. It leaves a qubit's $|0\rangle$ amplitude unchanged, while multiplying by i (phase) to a qubit's $|1\rangle$ amplitude.

```
q = QuantumRegister(1,name='q')
S_qc = QuantumCircuit(q,name='qc')

S_qc.h( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(S_qc)

S_qc.s( q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(S_qc)

circuit_drawer(S_qc)
```

```
  __ Initial __
0.70711 |0>    0.70711 |1>

  __ Final __
0.70711 |0>    0.70711j |1>

q_0: |0>─┤ H ├─┤ S ├─
```

T

A pre-defined gate for R_ϕ, ϕ=$\frac{\pi}{4}$

$$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$$

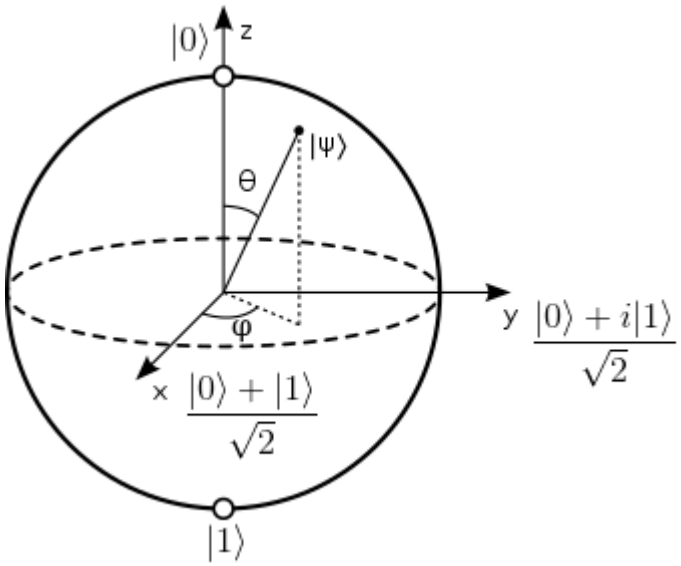
A pre-defined gate for R_ϕ, ϕ=$\frac{\pi}{2}$. It leaves a qubit's $|0\rangle$ amplitude unchanged, while multiplying by i (phase) to a qubit's $|1\rangle$ amplitude.

```
q = QuantumRegister(1,name='q')
T_qc = QuantumCircuit(q,name='qc')

T_qc.h( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(T_qc)

T_qc.t( q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(T_qc)

circuit_drawer(T_qc)
```

```
  __ Initial __
0.70711 |0>    0.70711 |1>

  __ Final __
0.70711 |0>    0.5+0.5j |1>

q_0: |0>─┤ H ├─┤ T ├─
```

Rotation Gates

The follow gates all represent rotations of a state on a Bloch Sphere. A Bloch sphere is a visual representation that maps the state of a qubit to a location on the surface of a sphere, radius = 1. An image of a Bloch sphere and it's axes is given below:



Note that the opposite ends of the x and y axis are:

-x = $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ -y = $\frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$

Thus, opposite axes on a Bloch Sphere represent orthogonal states.

$\mathbf{R}_x(\theta)$

$$\begin{bmatrix} cos(\frac{\theta}{2}) & -i \cdot sin(\frac{\theta}{2}) \\ -i \cdot sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{bmatrix}$$

A rotation gate where the initial and final states can be represented as θ rotation around the x-axis on a Bloch Sphere.

```
q = QuantumRegister(1,name='q')
Rx_qc = QuantumCircuit(q,name='qc')

Rx_qc.iden( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(Rx_qc)

Rx_qc.rx( m.pi/2, q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(Rx_qc)

circuit_drawer(Rx_qc)
```

```
  __ Initial __
1.0 |0>

  __ Final __
0.70711 |0>    -0.70711j |1>

q_0: |0>─| Id |─| Rx(1.5708) |─
```

$\mathbf{R}_y(\theta)$

$$\begin{bmatrix} cos(\frac{\theta}{2}) & -sin(\frac{\theta}{2}) \\ sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{bmatrix}$$

A rotation gate where the initial and final states can be represented as θ rotation around the y-axis on a Bloch Sphere.

```
q = QuantumRegister(1,name='q')
Ry_qc = QuantumCircuit(q,name='qc')

Ry_qc.iden( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(Ry_qc)

Ry_qc.ry( m.pi/2, q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(Ry_qc)

circuit_drawer(Ry_qc)
```

```
  __ Initial __
1.0 |0>

  __ Final __
0.70711 |0>    0.70711 |1>

q_0: |0>┤ Id ├┤ Ry(1.5708) ├
```

$\mathbf{R}_z(\theta)$

$$\begin{bmatrix} e^{\frac{-i\theta}{2}} & 0 \\ 0 & e^{\frac{i\theta}{2}} \end{bmatrix}$$

A rotation gate where the initial and final states can be represented as θ rotation around the z-axis on a Bloch Sphere.

```
q = QuantumRegister(1,name='q')
Rz_qc = QuantumCircuit(q,name='qc')

Rz_qc.h( q[0] )
print('  __ Initial __ ')
oq.Wavefunction(Rz_qc)

Rz_qc.rz( m.pi/2, q[0] )
print('\n  __ Final __ ')
oq.Wavefunction(Rz_qc)

circuit_drawer(Rz_qc)
```

```
  __ Initial __
0.70711 |0>    0.70711 |1>

  __ Final __
0.70711 |0>    0.70711j |1>

q_0: |0>┤ H ├┤ Rz(1.5708) ├
```

Two Qubit Control Gates

All of the following gates act on 2 qubits. In particular, each gate uses a 'target qubit' and a 'control qubit'. The role of the control qubit is to determine whether or not a particular operation is applied to the target qubit. If the control qubit is in the state $|1\rangle$, then the operation is carried out on the target qubit. Conversely, if the control qubit is in the state $|0\rangle$, then the target qubit remains unchanged.

CNOT

The effect of the CNOT gate can be described as follows:

$$\textbf{CNOT} \ |00\rangle \ \rightarrow \ |00\rangle$$

$$\textbf{CNOT} \ |01\rangle \ \rightarrow \ |01\rangle$$

$$\textbf{CNOT} \ |10\rangle \ \rightarrow \ |11\rangle$$

$$\textbf{CNOT} \ |11\rangle \ \rightarrow \ |10\rangle$$

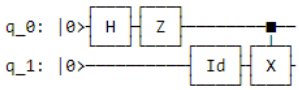
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In this notation, the first qubit is the control and the second qubit is the target. Another way to think of this gate is as a 'control-X' gate, where the state of the control qubit determines whether or not an X gate is applied to the target qubit. In Qiskit, this gate goes by the name 'CX'.

The CNOT gate is perhaps one of the most important tools in our quantum computing arsenal. Since we cannot have a purely 'NOT', the CNOT gate is our closest match. In combination with other gates, it will allow us to construct all manners of multi-qubit operations.

```
q = QuantumRegister(2,name='q')
Cx_qc = QuantumCircuit(q,name='qc')



Cx_qc.h( q[0] )
Cx_qc.z( q[0] )
Cx_qc.iden( q[1] )

print('  __ Initial __ ')
oq.Wavefunction(Cx_qc)

Cx_qc.cx( q[0], q[1] )

print('\n  __ Final __ ')
oq.Wavefunction(Cx_qc)

circuit_drawer(Cx_qc)
```

```
  __ Initial __
0.70711 |00>    -0.70711 |10>

  __ Final __
0.70711 |00>    -0.70711 |11>
```



CZ (Control-Z)

The control-Z gate works similarly to the CNOT gate, only instead of flipping the target qubit (applying an X gate), we apply a Z gate:

$$\textbf{CZ} \ |00\rangle \quad \rightarrow \quad |00\rangle$$

$$\textbf{CZ} \ |01\rangle \quad \rightarrow \quad |01\rangle$$

$$\textbf{CZ} \ |10\rangle \quad \rightarrow \quad |10\rangle$$

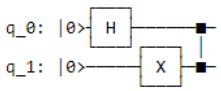
$$\textbf{CZ} \ |11\rangle \quad \rightarrow \quad -|11\rangle$$

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

Recall that a Z gates leaves a qubit in the state $|0\rangle$ untouched, while flipping the sign on a qubit in the state $|1\rangle$. Thus, the CZ gate performs a similar operation, only affecting the state $|11\rangle$, as shown above.

```
q = QuantumRegister(2,name='q')
Cz_qc = QuantumCircuit(q,name='qc')

Cz_qc.h( q[0] )
Cz_qc.x( q[1] )
print('  __ Initial __ ')
oq.Wavefunction(Cz_qc)

Cz_qc.cz( q[0], q[1] )
print('\n  __ Final __ ')
oq.Wavefunction(Cz_qc)

circuit_drawer(Cz_qc)
```

```
  __ Initial __
0.70711 |01>    0.70711 |11>

  __ Final __
0.70711 |01>    -0.70711 |11>
```



Control Phase Gate

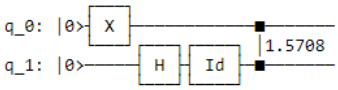
The control-phase gate, also referred to as a CPHASE gate, uses a control qubit to apply a R_ϕ gate to a target qubit. The net effect is similar to that of the control-Z gate, only differing by the phase that gets multiplied to the state $|11\rangle$:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\phi} \end{bmatrix}$$

In qiskit, this gate goes by the name CU_1:

```
q = QuantumRegister(2,name='q')
cu1_qc = QuantumCircuit(q,name='qc')

cu1_qc.x( q[0] )
cu1_qc.h( q[1] )
cu1_qc.iden( q[1] )
print('  __ Initial __ ')
oq.Wavefunction(cu1_qc)

cu1_qc.cu1( m.pi/2, q[0], q[1] )
print('\n  __ Final __ ')
oq.Wavefunction(cu1_qc)

circuit_drawer(cu1_qc)
```

```
  __ Initial __
0.70711 |10>    0.70711 |11>

  __ Final __
0.70711 |10>    0.70711j |11>
```



SWAP

The SWAP gate causes two qubits to trade states.

$$\textbf{SWAP} \ |00\rangle \ \rightarrow \ |00\rangle$$

$$\textbf{SWAP} \ |01\rangle \ \rightarrow \ |10\rangle$$

$$\textbf{SWAP} \ |10\rangle \ \rightarrow \ |01\rangle$$

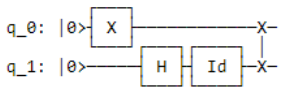
$$\textbf{SWAP} \ |11\rangle \ \rightarrow \ |11\rangle$$

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

A simple way of viewing the effect of this gate is that all of the 0's and 1's in each state switch places. As a result, we can see that the SWAP gate has no effect on the states $|00\rangle$ and $|11\rangle$.

```
q = QuantumRegister(2,name='q')
swap_qc = QuantumCircuit(q,name='qc')

swap_qc.x( q[0] )
swap_qc.h( q[1] )
swap_qc.iden( q[1] )
print('  __ Initial __ ')
oq.Wavefunction(swap_qc)

swap_qc.swap( q[0], q[1] )
print('\n  __ Final __ ')
oq.Wavefunction(swap_qc)

circuit_drawer(swap_qc)
```

```
  __ Initial __
0.70711 |10>    0.70711 |11>

  __ Final __
0.70711 |01>    0.70711 |11>
```



3 Qubit Control Gates

The following two gates take 3 qubits as inputs. They are essentially higher order versions of the CNOT and SWAP gates, adding one extra control qubit to each.

CSWAP

The control-swap gate uses a control qubit to determine whether or not to apply a SWAP gate to two target qubits. If the control qubit is in the state $|1\rangle$, then a SWAP gate is performed. Examples:

$$\textbf{CSWAP}\ |010\rangle \rightarrow |010\rangle$$

$$\textbf{CSWAP}\ |101\rangle \rightarrow |110\rangle$$

$$\textbf{CSWAP}\ |110\rangle \rightarrow |101\rangle$$

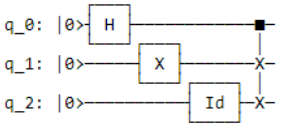
$$\textbf{CSWAP}\ |111\rangle \rightarrow |111\rangle$$

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

This gate is also sometimes referred to as a Fredkin Gate.

```
q = QuantumRegister(3,name='q')
cswap_qc = QuantumCircuit(q,name='qc')

cswap_qc.h( q[0] )
cswap_qc.x( q[1] )
cswap_qc.iden( q[2] )
print('  __ Initial __ ')
oq.Wavefunction(cswap_qc)

cswap_qc.cswap( q[0], q[1], q[2] )
print('\n  __ Final __ ')
oq.Wavefunction(cswap_qc)

circuit_drawer(cswap_qc)
```

```
  __ Initial __
0.70711 |010>    0.70711 |110>

  __ Final __
0.70711 |010>    0.70711 |101>
```



CCNOT

The control-control not gate uses two control qubits to determine if an X gate is applied to a single target qubit. Examples:

$$\textbf{CCNOT} \ |010\rangle \ \rightarrow \ |010\rangle$$

$$\textbf{CCNOT} \ |101\rangle \ \rightarrow \ |101\rangle$$

$$\textbf{CCNOT} \ |110\rangle \ \rightarrow \ |111\rangle$$

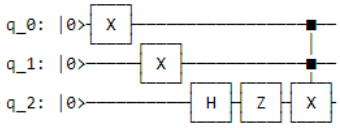
$$\textbf{CCNOT} \ |111\rangle \ \rightarrow \ |110\rangle$$

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

This gate is also sometimes referred to as a Toffoli Gate. Much like the CNOT gate, the effect of this gate is equivalent to an X gate on the states $|110\rangle$ and $|111\rangle$.

```
q = QuantumRegister(3,name='q')
ccnot_qc = QuantumCircuit(q,name='qc')

ccnot_qc.x( q[0] )
ccnot_qc.x( q[1] )
ccnot_qc.h( q[2] )
ccnot_qc.z( q[2] )
print('  __ Initial __ ')
oq.Wavefunction(ccnot_qc)

ccnot_qc.ccx( q[0], q[1], q[2] )
print('\n  __ Final __ ')
oq.Wavefunction(ccnot_qc)

circuit_drawer(ccnot_qc)
```

```
  __ Initial __
0.70711 |110>    -0.70711 |111>

  __ Final __
-0.70711 |110>    0.70711 |111>
```



This concludes all of the quantum gates provided by Qiskit that we will cover here. For the complete list of standard gates, check out:

https://github.com/Qiskit/qiskit-terra/tree/master/qiskit/extensions/standard

Using the gates covered in this lesson, along with all of Qiskit knowledge from lessons 1 & 2, we are now ready to begin coding up some of the most academically important quantum algorithms. But first, I encourage you to take a look at lesson 4, which covers some additional tools not provided by Qiskit, but will be immensely helpful in our education purposes.

Lesson 4 - Our Custom Functions

In this lesson, we will be covering some functions and operators that are not a part of Qiskit, but will be used frequently in the coming lessons. Specifically, we will go through some important functions from the python file Our_Qiskit_Functions.py, which come as a part of this tutorial series (See the appendix). The motivation for using these customs functions will be in the hopes that they make the learning endeavor of future lessons easier.

Before proceeding, please consider reading the previous lessons in this series, which cover all of the Qiskit basics needed for this lesson:

Lesson 1 - Intro to QuantumCircuits

Lesson 2 - Creating More Complex QuantumCircuits

Lesson 3 - Gates Provided by Qiskit

In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
import Our_Qiskit_Functions as oq
import math as m
import numpy as np
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

Throughout all of the coming lessons, the two custom functions that we will be using constantly are **Wavefunction** and **Measurement**. These functions will allow us to view our quantum systems with ease. In particular, these functions will handle many of the tedious steps needed to display wavefunctions and measurement results. In addition, both of these functions come with some optional arguments, which give us more control of how we would like to view our results.

Both of these functions do not amend our code in any way. In essence, they are designed for learning purposes only, providing the user tools for viewing quantum systems in a more understandable way.

Wavefunction

Wavefunction will hands down be the most common function we call upon from Our_Qiskit_Functions. In fact, we've already seen its default use numerous times in lessons 2 and 3. This is because viewing the quantum systems we create is very important if we want to understand what is going on! In the previous lessons, we have only seen the default use of this function, which takes a QuantumCircuit and prints the amplitudes associated with each state:

```
q = QuantumRegister(2,name='q')
qc = QuantumCircuit(q,name='qc')

qc.iden( q[0] )
qc.h( q[1] )
qc.z( q[1] )

print(' ___ statevector ___  ')
print( execute(qc, S_simulator).result().get_statevector() )

print('\n __ Wavefunction __  ')
oq.Wavefunction( qc )
```

```
 ___ statevector ___
[ 0.70710678+0.j  0.        +0.j -0.70710678+0.j  0.        +0.j]

 __ Wavefunction __
0.70711 |00>    -0.70711 |01>
```

Take a look at the cell of code above and notice the differences between the **get_statevector** and Wavefunction. Ignoring the difference in decimal precision, which can be changed for both, the biggest difference is display simplicity and state clarity. At its core, Wavefunction is still calling upon all of the steps shown in the statevector example, but with some extra lines of code to create the displayed states.

In addition to simply displaying wavefunction states, Wavefunction comes with some additional options to improve the viewing capabilities.

Precision

This first optional argument that we will take a look at will control the decimal precision for our amplitudes. This is done using the argument **precision**, which takes an integer:

```
q = QuantumRegister(2,name='q')
qc = QuantumCircuit(q,name='qc')

qc.iden( q[0] )
qc.h( q[1] )
qc.z( q[1] )

print(' __ Wavefunction __  ')
oq.Wavefunction( qc, precision=8 )
```

```
 __ Wavefunction __
0.70710678 |00>    -0.70710678 |01>
```

If the precision argument isn't given, Wavefunction will go to 5 decimals of precision by default.

Column

The next argument is a change in the layout that the states are displayed, but one that can be quite necessary as our quantum systems grow larger. By passing the **column** argument and setting it to True, each state in the wavefunction will be display as its own line. This will be quite handy when we want to focus on a single state amongst the clutter of many.

```
q = QuantumRegister(4,name='q')
qc = QuantumCircuit(q,name='qc')

qc.h( q[0] )
qc.h( q[1] )
qc.z( q[1] )
qc.rx( m.pi/3, q[1] )
qc.h( q[2] )
qc.ry( m.pi/5, q[2] )
qc.h( q[3] )

oq.Wavefunction( qc )

print('\n __ column __  ')
oq.Wavefunction( qc, column=True )
```

```
0.13901+0.08025j |0000>    0.13901+0.08025j |1000>    -0.13901-0.08025j |0100>    -0.13901-0.08025j |1100>    0.27281+0.15751j
|0010>    0.27281+0.15751j |1010>    -0.27281-0.15751j |0110>    -0.27281-0.15751j |1110>    0.13901+0.08025j |0001>    0.13901
+0.08025j |1001>    -0.13901-0.08025j |0101>    -0.13901-0.08025j |1101>    0.27281+0.15751j |0011>    0.27281+0.15751j |1011>
-0.27281-0.15751j |0111>    -0.27281-0.15751j |1111>

 __ column __
0.13901+0.08025j |0000>
0.13901+0.08025j |1000>
-0.13901-0.08025j |0100>
-0.13901-0.08025j |1100>
0.27281+0.15751j |0010>
0.27281+0.15751j |1010>
-0.27281-0.15751j |0110>
-0.27281-0.15751j |1110>
0.13901+0.08025j |0001>
0.13901+0.08025j |1001>
-0.13901-0.08025j |0101>
-0.13901-0.08025j |1101>
0.27281+0.15751j |0011>
0.27281+0.15751j |1011>
-0.27281-0.15751j |0111>
-0.27281-0.15751j |1111>
```

Systems and Show_Systems

In many of the coming algorithms, we will be dealing with ancilla qubits. These qubits make up secondary systems, which serve specific purposes, but ultimately are unimportant in the final measurement. When dealing with these ancilla systems, we may wish to sometimes view their qubits for learning purposes, and other times choose to simply ignore them. As a tool for separating ancilla systems, and choosing whether or not to display them, we will use the arguments **systems** and **show_systems**.

When we pass the argument systems, we must set it equal to a list containing the groupings of qubits. The sum of this list must equal the total number of qubits in the system. For example:

```
q = QuantumRegister(3,name='q')
qc = QuantumCircuit(q,name='qc')

qc.h( q[0] )
qc.iden( q[1] )
qc.h( q[2] )

oq.Wavefunction( qc, systems=[2,1] )
```

```
0.5 |00>|0>    0.5 |10>|0>    0.5 |00>|1>    0.5 |10>|1>
```

In this example, our quantum system is a state of 3 qubits. But perhaps the third qubit is an ancilla, and we would like to monitor is separately from qubits 0 and 1. By passing the argument systems = [2,1], we get the displayed wavefunction above, which puts the ancilla qubit as its own state. Specifically, this argument groups qubits together by their numerical order, according to the length and values of the list. In this example, [2,1] tells the function to display qubits 0 and 1 as a group, followed by qubit 2 as its own group.

Mathematically, we must remember that separating qubits off like this is still the same physical state:

$$|10\rangle\,|1\rangle \quad = \quad |101\rangle$$

Thus, this is once again just a cosmetic change, but a very useful one in lessons to come.

Now, suppose we have a group of ancilla qubits, or even several groups, which we do not want to display when we call upon Wavefunction. For example, we often times deal with ancilla qubits that always remain in the state of all 0's before and after an operation. Thus, viewing all these qubits in the $|0\rangle$ state becomes repetitive, and adds a lot of unnecessary clutter. To avoid this, we can use the show_systems argument to choose which systems we want to view:

```
q = QuantumRegister(6,name='q')
qc = QuantumCircuit(q,name='qc')

qc.h( q[0] )
qc.iden( q[1] )
qc.h( q[2] )

oq.Wavefunction( qc, systems=[2,1,3] )

print('\n __ show_systems __')
oq.Wavefunction( qc, systems=[2,1,3], show_systems=[True,True,False] )
```

```
0.5 |00>|0>|000>    0.5 |10>|0>|000>    0.5 |00>|1>|000>    0.5 |10>|1>|000>

 __ show_systems __
0.5 |00>|0>    0.5 |10>|0>    0.5 |00>|1>    0.5 |10>|1>
```

As shown here, every state in the system carries the same $|000\rangle$ system of ancilla qubits. By passing the argument show_systems, and setting it equal to a list of equal length to systems, containing truth values, we can opt to remove certain systems from our display. Specifically, the index locations of each True or False in the show_systems argument correspond to the same groups in the systems argument. We will use this argument quite frequently in later lessons to avoid any extra clutter on our systems.

Some important points about using systems and show_systems:

1) Both arguments must be lists of equal length, containing only integers and truth values respectively.

2) show_systems will only work if systems is also an argument. Passing show_systems by itself will result in no change to the display of Wavefunction.

3) Using show_systems to remove a system from the display of a wavefunction should only be used when all states in the system have the same ancilla state. Otherwise, the printed wavefunction will show duplicates of the same state:

```
q = QuantumRegister(4,name='q')
qc = QuantumCircuit(q,name='qc')

qc.h( q[0] )
qc.iden( q[1] )
qc.x( q[2] )
qc.h( q[3] )

oq.Wavefunction( qc, systems=[3,1] )
print('  ')
print(' __ show_systems __')
oq.Wavefunction( qc, systems=[3,1], show_systems=[True,False] )
```

```
0.5 |001>|0>    0.5 |101>|0>    0.5 |001>|1>    0.5 |101>|1>

 __ show_systems __
0.5 |001>    0.5 |101>    0.5 |001>    0.5 |101>
```

Compare the two lines above, and note that choosing to not display the ancilla qubit is problematic. Since the ancilla qubit is in a superposition, choosing to not display its results in the wavefunction looks very odd. Mathematically, this odd-looking state is still technically correct in some sense, showing the associated amplitudes with each possible state of the main system (so long as we don't add states together). However, this could lead to some potentially very confusing results, and should be avoided.

Measurement

For instances where we will need to make a measurement on our quantum system, or many, we will call upon the Measurement function. In particular, this function contains within it the execute function, and all of the necessary lines of code to run a simulated measurement on the **qasm_backend**.

However, the one thing that it won't do is add a measurement instruction to the QuantumCircuit. This is a step that we must do manually before calling upon Measurement:

```
q = QuantumRegister(3,name='q')
c = ClassicalRegister(3,name='c')
qc = QuantumCircuit(q,c,name='qc')

qc.h( q[0] )
qc.iden( q[1] )
qc.x( q[2] )
qc.measure(q,c)

oq.Measurement( qc )
```

```
1|001>
```

So long as there is a measurement instruction somewhere in the algorithm, we can pass the QuantumCircuit as the only argument, and we will get back a simulated measurement. By default, this function only performs one measurement.

shots

The most common argument we will pass to the Measurement function will be **shots**, which is the same argument that will be passed along to the execute function. Here, passing shots and setting it equal to an integer will tell the function how many times to measure the system:

```
q = QuantumRegister(3,name='q')
c = ClassicalRegister(3,name='c')
qc = QuantumCircuit(q,c,name='qc')

qc.h( q[0] )
qc.iden( q[1] )
qc.x( q[2] )
qc.measure(q,c)

oq.Measurement( qc, shots=100 )
```

```
49|001>    51|101>
```

Passing the shots argument is useful when we would like to simulate measurements on our system, and gain some insight into the different probabilities of measuring certain states.

print_M

Most of the time we will want to display the measurement results of our qubits, but not always. Perhaps we would like to perform a measurement on our system, store the results, but not necessarily view them. Although the intention behind Measurement is for displaying results, we can opt to display nothing by passing the **print_M** argument, and setting it to False:

```
q = QuantumRegister(3,name='q')
c = ClassicalRegister(3,name='c')
qc = QuantumCircuit(q,c,name='qc')

qc.h( q[0] )
qc.iden( q[1] )
qc.x( q[2] )
qc.measure(q,c)

oq.Measurement( qc, print_M=False )
```

Running this code should result in nothing displayed. By itself, setting print_M to False is sort of a mute step. Nothing is displayed, and nothing is returned to us from the function.

However, often times the motivation behind making a measurement and not viewing the results is to extract these results for some other purpose. For example, in lesson 1 we used the results of a coin flipping algorithm to determine a winner. If this is our intent, then in order to get back measurement results, we need only pass the argument **return_M** as True, which will return a dictionary object to us:

```
q = QuantumRegister(3,name='q')
c = ClassicalRegister(3,name='c')
qc = QuantumCircuit(q,c,name='qc')

qc.h( q[0] )
qc.iden( q[1] )
qc.x( q[2] )
qc.measure(q,c)

M = oq.Measurement( qc, shots=20, print_M=False, return_M=True )
print(M)

```

```
{'001': 10, '101': 10}
```

This dictionary object contains all of the simulated measurement results, for whatever use an algorithm may require. Please note that when using the return_M argument, we must create a variable to store the returned dictionary object. And also, return_M and print_M do not need to be used together. Both arguments are independent of each other, which means we can choose to display results / extract results in any manner we like.

In addition to the arguments already covered, Measurement also takes the column argument as well. Passing the column argument and setting it to True results in the same style of display as shown above.

Higher Order Control Gates

The next two functions that we are about to study will be used very frequently in the coming lessons, but not necessarily always on display. In particular, many of the algorithms we will study in lesson 5 require higher order control gates as a smaller component, typically written into other functions that we will call upon in Our_Qiskit_Functions. As we saw in lesson 3, Qiskit's standard gates only comes with a handful of control gates. In particular, the largest of which use two control qubits (CCNOT). While the gates provided by Qiskit are all that we need for a universal set, it will be helpful to define our own function for constructing these higher order control gates.

The Strategy

The way we are going to construct our higher order control gates is a straightforward strategy, using only CCNOT and CNOT gates. This approach is by no means optimal for many cases, but it is a good foundation for how a higher order control gate *should* function.

The major hurdle to overcome is that we must use the conditions on N qubits in order to invoke a single operation. A single CCNOT gate can only take two control qubits at a time, thus leaving us well short of our goal. However, the trick to this strategy

will be to use the aid of ancilla qubits. In essence, these ancilla qubits will allow us to temporarily store information about our control qubits, and ultimately determine whether or not to apply the operation.

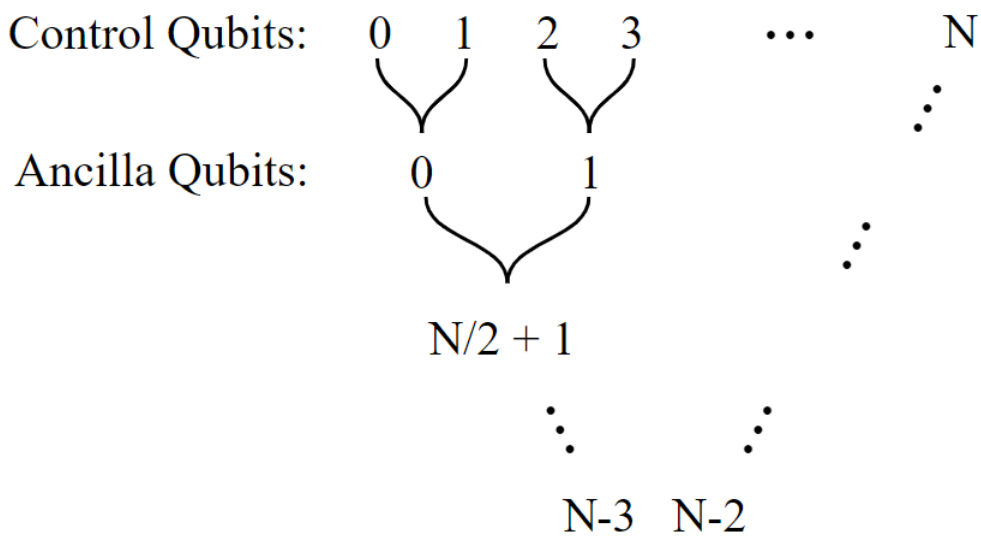
First, let's see the general strategy written in terms of classical code:

```
def CCCC_NOT(q):
    anc = []
    for i in np.arange(2):
        s = 2*i
        if( (q[s]==1) and (q[s+1]==1) ):
            anc.append(1)
        else:
            anc.append(0)
    return anc

state1 = [1,1,1,1]
state2 = [1,1,1,0]

anc1 = CCCC_NOT(state1)
anc2 = CCCC_NOT(state2)

print('state 1: ',state1,'     ancilla 1: ',anc1)
print('state 2: ',state2,'     ancilla 2: ',anc2)
```

```
state 1:  [1, 1, 1, 1]      ancilla 1:  [1, 1]
state 2:  [1, 1, 1, 0]      ancilla 2:  [1, 0]
```

In this example, we can see that 'state1' results in the state of all 1's for the ancilla system, while 'state2' results in a 0 for the second ancilla qubit. If we were to use these two ancilla qubits as the control qubits for a CCNOT, the first system would receive the operation, while the second wouldn't. If we compare this result to our initial states, we would have achieved exactly a 4-qubit control gate: the state $|1111\rangle$ receives the operation while the state $|1110\rangle$ does not.

The key point to this example is the way in which we reduced our 4-qubit problem down to 2. Specifically, we work through our control qubits in groups of 2, putting an ancilla qubit in the state $|1\rangle$ or $|0\rangle$ depending on if the two control qubits are in the state $|11\rangle$ or not. For our quantum code, we are going to invoke this strategy using CCNOT gates to reduce the dimension of our problem by 1 per CCNOT gate, until we eventually arrive at only 2 ancilla qubits:



In this diagram, each layer moving downward holds the information about the preceding layer, where the control qubits are the highest layer (and ultimately the source of the operation). The junctions connecting two qubits in the diagram represent a CCNOT gate, where the resulting state of the ancilla qubit is $|1\rangle$ if both control qubits are in the $|1\rangle$ state, and a $|0\rangle$ if either of the controlling qubits are in the state $|0\rangle$. Using this recursive process, if a single qubit from the highest layer is in the $|0\rangle$ state, it will trickle down all the way to the final layer, which in turn will mean the control-operation does not happen.

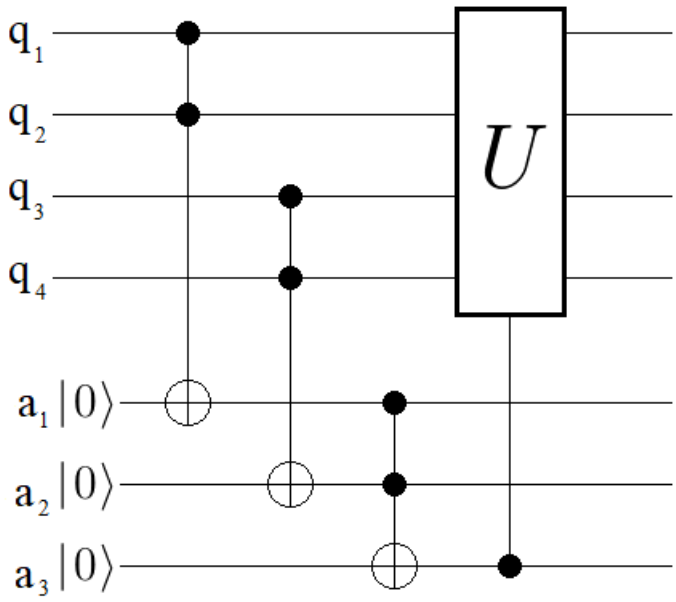
Now, in order for this strategy to work, we need all of the ancilla qubits to be initialized in the state $|0\rangle$. This is because each operation is a CCNOT gate, which is effectively a 2-qubit control X gate. Thus, in order for this CCNOT gate to leave the target

ancilla qubits in the state $|1\rangle$, only when control qubits are in the $|1\rangle$ state, the target qubit must initially be in the $|0\rangle$ state.

In total, in order to condense the information of N control qubits down to two ancilla qubits, we require $N-2$ ancilla qubits. From there, if we wish to combine these two final ancilla qubits down to one, for a single qubit control gate, we will require 1 additional ancilla qubit, bringing our total to $N-1$.

Now, requiring up to $N-1$ ancilla qubits just to perform an operation on N qubits may seem like a steep price to pay. Truthfully, it is. Qubits are not exactly plentiful on current quantum computers, which makes this general strategy a little too resource intensive to be practical in most cases. However, this is a problem to keep in mind, but ignore for the time being. Our goal in these lessons is to learn the basics of quantum algorithms, not solve current research efforts (that's your job afterwards). In the coming lessons we will be using this higher order control strategy frequently, because we have the luxury of using simulated qubits. And, thanks to the argument $\mathrm{show\_systems}$ from earlier, we can effectively ignore as many ancilla qubits as we want, and focus on the important results.

Now then, let's take a look at what the diagram above looks like in terms of a quantum circuit diagram, using $N=4$ as our example:



In this diagram, we are performing the general N-control gate strategy. By using three ancilla qubits, we are able to condense the information of all the states stored on our 4-qubit system, down to a single control ancilla. The operator U in the diagram represents any single control gate we've already seen thus far in lesson 3. As an example, let's implement this diagram in code, replacing U with a control-Z gate:

```
q = QuantumRegister(7,name='q')
qc = QuantumCircuit(q,name='qc')

qc.h( q[0] )
qc.h( q[1] )
qc.h( q[2] )
qc.h( q[3] )
qc.iden( q[4] )
qc.iden( q[5] )
qc.iden( q[6] )

print(' __ Inital State __ ')
oq.Wavefunction(qc, systems=[4,3], column=True)

qc.ccx( q[0], q[1], q[4] )
qc.ccx( q[2], q[3], q[5] )
qc.ccx( q[4], q[5], q[6] )
qc.cz( q[6], q[0] )

print('\n __ After CCCZ __')
oq.Wavefunction(qc, systems=[4,3], column=True)
```

```
 __ Inital State __
0.25 |0000>|000>
0.25 |1000>|000>
0.25 |0100>|000>
0.25 |1100>|000>
0.25 |0010>|000>
0.25 |1010>|000>
0.25 |0110>|000>
0.25 |1110>|000>
0.25 |0001>|000>
0.25 |1001>|000>
0.25 |0101>|000>
0.25 |1101>|000>
0.25 |0011>|000>
0.25 |1011>|000>
0.25 |0111>|000>
0.25 |1111>|000>

 __ After CCCZ __
0.25 |0000>|000>
0.25 |1000>|000>
0.25 |0100>|000>
0.25 |0010>|000>
0.25 |1010>|000>
0.25 |0110>|000>
0.25 |0001>|000>
0.25 |1001>|000>
0.25 |0101>|000>
0.25 |1100>|100>
0.25 |1110>|100>
0.25 |1101>|100>
0.25 |0011>|010>
0.25 |1011>|010>
0.25 |0111>|010>
-0.25 |1111>|111>
```

Success, the code above successfully picks out the $|1111\rangle$ state and applies a control-Z gate. In this code, we've arbitrarily chosen to apply the CZ gate to qubit 0, but any of them will result in the same effect. This is because only the state $|1111\rangle$ will pick up the effect, which correspondingly means any of the qubits in this state are candidates to be the target qubit.

While the coding example above works as intended, there is one detail we've overlooked. Namely, the final state of all our ancilla qubits. Take a look at the results above, and notice which ancilla qubits are in the $|1\rangle$ and $|0\rangle$ states. If there were no further steps in our algorithm, we could in principle leave them as they are currently, since they do not affect a measurement on the main system. But if we wanted to apply any further steps, the fact that all of the states in the system have varying ancilla states is problematic. Specifically, states in our main system will no longer undergo superpositions as we may intend.

The remedy for this problem is that we need to return all of the ancilla qubits back to their original state of all 0's. To do this, we need only apply all of the CCNOT gates in reverse:

```
q = QuantumRegister(7,name='q')
qc = QuantumCircuit(q,name='qc')

qc.h( q[0] )
qc.h( q[1] )
qc.h( q[2] )
qc.h( q[3] )
qc.iden( q[4] )
qc.iden( q[5] )
qc.iden( q[6] )

print(' __ Inital State __ ')
oq.Wavefunction(qc, systems=[4,3])

qc.ccx( q[0], q[1], q[4] )
qc.ccx( q[2], q[3], q[5] )
qc.ccx( q[4], q[5], q[6] )
qc.cz( q[6], q[0] )

print('\n __ After CCCCZ __')
oq.Wavefunction(qc, systems=[4,3])

qc.ccx( q[4], q[5], q[6] )
qc.ccx( q[2], q[3], q[5] )
qc.ccx( q[0], q[1], q[4] )

print('\n __ Reverse All CCNOTS __')
oq.Wavefunction(qc, systems=[4,3], column=True)

```

```
 __ Inital State __
0.25 |0000>|000>    0.25 |1000>|000>    0.25 |0100>|000>    0.25 |1100>|000>    0.25 |0010>|000>    0.25 |1010>|000>    0.25 |0
110>|000>    0.25 |1110>|000>    0.25 |0001>|000>    0.25 |1001>|000>    0.25 |0101>|000>    0.25 |1101>|000>    0.25 |0011>|00
0>    0.25 |1011>|000>    0.25 |0111>|000>    0.25 |1111>|000>

 __ After CCCCZ __
0.25 |0000>|000>    0.25 |1000>|000>    0.25 |0100>|000>    0.25 |0010>|000>    0.25 |1010>|000>    0.25 |0110>|000>    0.25 |0
001>|000>    0.25 |1001>|000>    0.25 |0101>|000>    0.25 |1100>|100>    0.25 |1110>|100>    0.25 |1101>|100>    0.25 |0011>|01
0>    0.25 |1011>|010>    0.25 |0111>|010>    -0.25 |1111>|111>

 __ Reverse All CCNOTS __
0.25 |0000>|000>
0.25 |1000>|000>
0.25 |0100>|000>
0.25 |1100>|000>
0.25 |0010>|000>
0.25 |1010>|000>
0.25 |0110>|000>
0.25 |1110>|000>
0.25 |0001>|000>
0.25 |1001>|000>
0.25 |0101>|000>
0.25 |1101>|000>
0.25 |0011>|000>
0.25 |1011>|000>
0.25 |0111>|000>
-0.25 |1111>|000>
```

The example above is the complete 4-qubit control gate template. In general, this same strategy can be expanded to construct any N-control gate.

For our coming tutorials, we will avoid overcrowding our code with all of these steps every time we wish to use such an N-control gate (which will be a lot). Thus, we will instead call upon the **n_NOT** and **n_Control_U** functions from Our_Qiskit_Functions to condense our code. We will use n_NOT specifically when we want to implement a N-control NOT gate, and n_Control_U for everything else.

n_NOT

The n_NOT function takes the following arguments:

(QuantumCircuit, [control qubits], target qubit, [ancilla qubits])

where the brackets indicate that the argument is a list of the integers corresponding to those particular qubits. For example:

```
q = QuantumRegister(5,name='q')
qc = QuantumCircuit(q,name='qc')

qc.h( q[0] )
qc.h( q[1] )
qc.h( q[2] )
qc.iden( q[3] )
qc.iden( q[4] )

print(' __ Inital State __ ')
oq.Wavefunction(qc, systems=[3,1,1], show_systems=[True,True,False])

oq.n_NOT( qc, [ q[0], q[1], q[2] ], q[3], [q[4]])

print('\n __ n_NOT __')
oq.Wavefunction(qc, systems=[3,1,1], show_systems=[True,True,False])
```

```
 __ Inital State __
0.35355 |000>|0>    0.35355 |100>|0>    0.35355 |010>|0>    0.35355 |110>|0>    0.35355 |001>|0>    0.35355 |101>|0>    0.35355
|011>|0>    0.35355 |111>|0>

 __ n_NOT __
0.35355 |000>|0>    0.35355 |100>|0>    0.35355 |010>|0>    0.35355 |110>|0>    0.35355 |001>|0>    0.35355 |101>|0>    0.35355
|011>|0>    0.35355 |111>|1>
```

In this example we've applied a 3-control NOT gate, where the control qubits are [0, 1, 2], and the target is qubit 3. As shown above, all states initially start with the target qubit in the $|0\rangle$ state. But after we apply our n_NOT function, the state $|111\rangle$ receives an X gate to its target qubit, flipping it to $|1\rangle$. In addition, all of the ancilla qubit are returned to the $|0\rangle$ state (change the last False in each Wavefunction to verify this for yourself).

n_Control_U

The n_Control_U function takes the following arguments:

(QuantumCircuit, [control qubits], [ancilla qubits], [gates])

where **gates** refers to the single-qubit control operations you would like to invoke (can be more than one). Specifically, there are four control-operations supported by this function, with the following formats:

CNOT : ('X', target)

CZ : ('Z', target)

CPHASE : ('PHASE', target, angle)

CSWAP : ('SWAP', target1, target2)

Thus, the gates argument for this function is a list of tuples in the forms shown above. Let's see an example of using a control-Z gate, followed by a control-X:

```
q = QuantumRegister(6,name='q')
qc = QuantumCircuit(q,name='qc')

qc.h( q[0] )
qc.h( q[1] )
qc.h( q[2] )
qc.x( q[3] )
qc.iden( q[4] )
qc.iden( q[5] )

print(' __ Inital State __ ')
oq.Wavefunction(qc, systems=[3,1,2], show_systems=[True,True,False])

oq.n_Control_U( qc, [q[0], q[1], q[2]], [q[4],q[5]] , [('Z',q[3]),('X',q[3])])

print('\n __ n_NOT __')
oq.Wavefunction(qc, systems=[3,1,2], show_systems=[True,True,False])
```

```
 __ Inital State __
0.35355 |000>|1>    0.35355 |100>|1>    0.35355 |010>|1>    0.35355 |110>|1>    0.35355 |001>|1>    0.35355 |101>|1>    0.35355
|011>|1>    0.35355 |111>|1>

 __ n_NOT __
-0.35355 |111>|0>    0.35355 |000>|1>    0.35355 |100>|1>    0.35355 |010>|1>    0.35355 |110>|1>    0.35355 |001>|1>    0.3535
5 |101>|1>    0.35355 |011>|1>
```

Just as intended,this operation first picks out the state $|111\rangle$ and applies a Z gate to the target qubit, followed by an X gate. The result is that the target qubit first picks up a negative phase, and is then flipped to the $|0\rangle$ state.

Since we only need to use all of the CCNOT gates once in order to obtain the control ancilla qubits, we can perform as many control operations as we want before applying all of the CCNOT gates in reverse.

Note that when using n_NOT and n_Control_U, rather than passing a list of qubits for the arguments, we can also pass QuantumRegister objects:

```
q = QuantumRegister(3,name='q')
tgt = QuantumRegister(1,name='t')
anc = QuantumRegister(2,name='a')
qc = QuantumCircuit(q,tgt,anc,name='qc')

qc.h( q[0] )
qc.h( q[1] )
qc.h( q[2] )
qc.x( tgt[0] )
qc.iden( anc[0] )
qc.iden( anc[1] )

print(' __ Inital State __ ')
oq.Wavefunction(qc, systems=[3,1,2], show_systems=[True,True,False])

oq.n_Control_U( qc, q, anc , [('Z',tgt[0]),('X',tgt[0])])

print('\n __ n_NOT __')
oq.Wavefunction(qc, systems=[3,1,2], show_systems=[True,True,False])
```

```
 __ Inital State __
0.35355 |000>|1>    0.35355 |100>|1>    0.35355 |010>|1>    0.35355 |110>|1>    0.35355 |001>|1>    0.35355 |101>|1>    0.35355
|011>|1>    0.35355 |111>|1>

 __ n_NOT __
-0.35355 |111>|0>    0.35355 |000>|1>    0.35355 |100>|1>    0.35355 |010>|1>    0.35355 |110>|1>    0.35355 |001>|1>    0.3535
5 |101>|1>    0.35355 |011>|1>
```

QuantumRegister objects are callable in the same way as a list, when accessing qubits. Thus, when we want to perform higher order control gates with lots of qubits, we can tidy up our code by separating the control qubits / ancilla into separate QuantumRegisters.

This concludes lesson 4, and all of the most relevant functions that we will be using from Our_Qiskit_Functions. There are plenty more functions in this python file, and I encourage you to check them out for yourself. From this point forward, everytime

we need access to new custom functions, we will discuss them in their respective lessons, when we first encounter them.

Lesson 5.1 - Into to Quantum Algorithms (Deutsch)

This tutorial is the first of four, all labeled 'Lesson 5'. The theme of these lessons is to introduce and explain several 'easier' quantum algorithms. These algorithms are all of historical / academic importance, although perhaps not terribly relevant for application purposes. We shall see several threads of commonality between all four algorithms, which make them a good set of algorithms to learn together.

In this tutorial, we will begin by briefly discussing the context for when we want to use quantum algorithms, and what makes a quantum algorithm 'faster'. Then, we will proceed to the main topic: the Deutsch Algorithm.

Before proceeding, please consider reading the previous lessons in this series, which covers all of the Qiskit basics of circuits and measurements needed for this lesson:

Lesson 1 - Intro to QuantumCircuits

Lesson 2 - Creating More Complex QuantumCircuits

Lesson 3 - Gates Provided by Qiskit

Original publication of the algorithm: [7]

In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
from qiskit.tools.visualization import circuit_drawer
import Our_Qiskit_Functions as oq
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

" The Quantum Advantage "

Perhaps you've come across this phrase before - 'The Quantum Advantage' - and you weren't sure what it meant, but it sure sounded cool! The Quantum Advantage is referring to the goal that a quantum computer will be able to outperform a classical computer, for some certain process. Hence, the reason we're here, studying quantum algorithms! Currently, there are already a handful of known *mathematical* cases where a quantum computer *should* provide us with a speedup.

As you may suspect based on my use of italics, the realization of these speedups has yet to happen. We're all pretty sure it *will* happen, but not so sure *when*, or *what* the algorithm will be. Many believe we are certainly very close, especially with bigger and better quantum computers just on the horizon. The Quantum Advantage is going to take a great deal of collaborative effort from physics / engineering / mathematics / computer science / etc. Thus, the goal of these tutorial lessons is to bring us up to speed on everything we need to know in order to start contributing to this effort.

One disclaimer before we get started however, is that the algorithms we are about to work through do not directly translate to real quantum computers. Or in other words, the algorithms we will be writing assume 'perfect' quantum computers. Much later in this tutorial series we will discuss what it means to design algorithms around current quantum computing hardware, and the new challenges that arise. Thus, take these algorithms with a grain of salt, keeping in mind that we are studying them for their academic value, rather than practical purposes. In particular, the math behind some of these algorithms is quite challenging, and simply understanding how each algorithm works is a major milestone. Then, once you've seen all the 'perfect scenario' quantum algorithms, you will be in a much better position to start designing algorithms on real quantum chips.

The Deutsch Algorithm

In terms of simplicity and elegance, there is no better starting point than the Deutsch Algorithm. It's simple to understand, simple to implement, and gets the point across of what it means to outperform a classical algorithm. So let's begin by framing our problem:
Suppose we are given a 'black box function' f. By this we mean that we are given some function f, which we can use, but we don't know its effect. Specifically, f acts on a bit of information, either 0 or 1, and returns an output, also either 0 or 1. Thus, when we feed f the inputs 0 and 1, the function will be describable by two out of the four following possibilities:

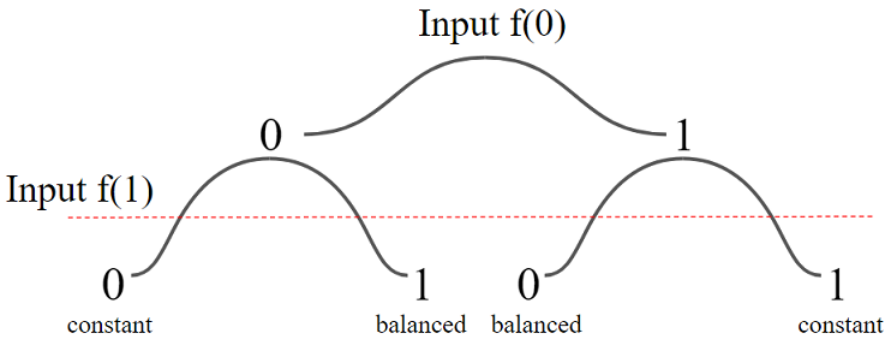
$$f(0) \to 0 \qquad f(0) \to 1$$

$$f(1) \to 0 \qquad f(1) \to 1$$

Based on these possibilities, we can say that f is guaranteed to be either a 'balanced' or 'constant' function. A balanced function means that f's outputs will be half 0's and half 1's, ex: $f(0) \to 1 \quad f(1) \to 0$. A constant function means that the output will be either all 0's or all 1's, ex: $f(0) \to 1 \quad f(1) \to 1$. So then, given this mysterious f, what is the minimum number of uses by which we can determine whether it is a balanced or constant function?

Well, let's take a look at the classical approach. Since we can only work with classical bits, let's say we feed the function a 0, and we get back a 1. We now have one piece of information: $f(0) \to 1$. But based on this one result, can we conclude what will happen for $f(1)$?

The answer is no. The information we got from one call of the function f is insufficient to determine whether f is a balanced or constant function. If we get $f(1) \to 1$, we will conclude that f is constant, while if we get $f(1) \to 0$, we will conclude that it is balanced. Thus, *classically*, we use the black box function f twice in order to determine its nature. If you are still a little unsure, the diagram below represents a flow chart of all the possibilities:



Let's write up a simple code to simulate this problem:

```python
import math as m
import scipy as sci

def blackbox_f():
    '''
    Returns one of four possible f functions
    '''
    def F1(x):
        return 0

    def F2(x):
        return 1

    def F3(x):
        return x%2

    def F4(x):
        return (x+1)%2

    functions = [F1,F2,F3,F4]
    f = functions[ int( m.floor( 4*sci.rand() ) ) ]
    return f

F = blackbox_f()

print('f(0): ',F(0))
print('f(1): ',F(1))

if(F(0) == F(1)):
    print('conclusion: f is constant!')
else:
    print('conclusion: f is balanced!')
```

```
f(0):  1
f(1):  0
conclusion: f is balanced!
```

The cell of code above randomly generates one of the four possible black box functions, tests it with the inputs $f(0)$ and $f(1)$, and concludes whether the function is balanced or constant based on the results. It's kind of a silly example, but it gets the point across.

Now, let's see if we can do any better with our powerful quantum computers! As you might imagine, there's really only one way to be *faster* than the classical approach here. We need to be able to determine if f is constant or balanced in only one function call.

When we move out of the realm of classical computing, and into quantum computing, what we gain are qubits over bits. Thus, we are going to give the function f a qubit as an input. However, we assume that f is a classical function, meaning that we can't actually feed it a qubit in the same way we can feed it a regular bit. There's a couple valid reasons why f shouldn't be able to handle a qubit, but the best is perhaps a simple mathematical argument. Consider what would happen if we sent in a qubit in a superposition state between $|0\rangle$ and $|1\rangle$, for a constant f:

$$f\Big(\ \frac{1}{\sqrt{2}}\big(\,|\,0\rangle + |\,1\rangle\,\big)\ \Big) \rightarrow \frac{1}{\sqrt{2}}\big(\,|\,1\rangle + |\,1\rangle\,\big) = \frac{2}{\sqrt{2}}|\,1\rangle \quad \text{Not Unitary!}$$

Remember that quantum systems always must be unitary (it's not our rule, blame physics!). Thus, f is a strictly classical function that only operates on classical bits.

So then, in order to use a quantum computer, we must side-step the problem of using f. To do this, we will define a quantum operation g, which will incorporate our classical function f in such a way that we still have a unitary operator. For the same reason shown above, there's just no way of creating a unitary operator that incorporates f *and* acts on 1 qubit. Thus, the best we can do is an operator that acts on two qubits:

$$g\,|\,q_1\rangle\,|\,q_2\rangle \quad \longrightarrow \quad |\,q_1\rangle\,|\,q_2 \oplus f(\,q_1)\,\rangle$$

where the symbol $\oplus$ means addition modulo 2: $\quad 0 \oplus 1 = 1 \qquad 1 \oplus 1 = 0$ (basically if a number adds up to 2, it becomes a 0). Let's see a quick example:

$$f(\,0,1) \rightarrow (\,0,1)$$

$$g\ \frac{1}{\sqrt{2}}\big(\,|\,10\rangle + |\,01\rangle\big) \quad\longrightarrow\quad \frac{1}{\sqrt{2}}\big(\,|\,11\rangle + |\,01\rangle\big)$$

More specifically:

$$g\ |\,10\rangle \quad = \quad g\ |\,1\rangle\,|\,0\rangle \quad\longrightarrow\quad |\,1\rangle\,|\,0 \oplus f(1)\,\rangle \quad = \quad |\,11\rangle$$

Now, you may be wondering where the heck this addition modulo 2 came from. This is part of the trick that comes with solving classical problems via quantum algorithms. Sometimes we need to introduce new ways of approaching the problem. Here, we are able to incorporate f into our quantum operation by using $\oplus$, which will guarantee everything stays unitary.

Most importantly however, by using this g, we can see that based on what kind of function f is, we get different final states:

$f(\,0,1) \rightarrow (\,0,1)$

$g(\,|\,00\rangle\,) \rightarrow |\,00\rangle$

$g(\,|\,01\rangle\,) \rightarrow |\,01\rangle$

$g(\,|\,10\rangle\,) \rightarrow |\,11\rangle$

$g(\,|\,11\rangle\,) \rightarrow |\,10\rangle$

$f(\,0,1) \rightarrow (\,1,0)$

$g(\,|\,00\rangle\,) \rightarrow |\,01\rangle$

$g(\,|\,01\rangle\,) \rightarrow |\,00\rangle$

$g(\,|\,10\rangle\,) \rightarrow |\,10\rangle$

$g(\,|\,11\rangle\,) \rightarrow |\,11\rangle$

$f(\,0,1) \rightarrow 0$

$g(\,|\,00\rangle\,) \rightarrow |\,00\rangle$

$g(\,|\,01\rangle\,) \rightarrow |\,01\rangle$

$g(\,|\,10\rangle\,) \rightarrow |\,10\rangle$

$g(\,|\,11\rangle\,) \rightarrow |\,11\rangle$

$f(\,0,1) \rightarrow 1$

$g(\,|\,00\rangle\,) \rightarrow |\,01\rangle$

$g(\,|\,01\rangle\,) \rightarrow |\,00\rangle$

$g(\,|\,10\rangle\,) \rightarrow |\,11\rangle$

$g(\,|\,11\rangle\,) \rightarrow |\,10\rangle$

Just like how our classical f can map the bits (0,1) to one of four possibilities, our g operator can map our two qubit states to one of four final states. Also like the classical case, if we use only one state as an input, we cannot determine whether f is a balanced or constant function. For example, using the state $|00\rangle$ as an input will give us one of two results: $|00\rangle$ or $|01\rangle$. Based on which result we get, we've eliminated two out of the four categories above, but still are left with with two possibilities, one balanced and one constant.

Now that we have our g function mathematically defined, it's time to create it in our code. In the $\mathrm{Our\_Qiskit\_Functions}$ file, our blackbox g has already been created for us (you're welcome). We will go into the specifics of this g later in this tutorial, but for now let's just see it in action. Run the cell of code below a few times, and verify the effect that g is having on our initial state:

```
q = QuantumRegister(2,name='q')
test_g = QuantumCircuit(q,name='qc')

test_g.h( q[0] )
test_g.x( q[1] )
test_g.cz( q[0], q[1] )
test_g.x( q[1] )

print(' ___ Initial State ___ ')
oq.Wavefunction(test_g)

f = oq.Blackbox_g_D(test_g,q)

print('\n ___ After Blackbox ___ ')
oq.Wavefunction(test_g)
```

```
 ___ Initial State ___
0.70711 |00>    -0.70711 |10>

 ___ After Blackbox ___
0.70711 |00>    -0.70711 |11>
```

As mentioned before, if we want to to do better than the classical case, we need to be able to determine whether f is constant or balanced, with only one call of g. As shown above, using only one state as an input does not get the job done. So then, we will send in a superposition of states, since that is the big advantage to using qubits over bits!

Now, let's use the state in our code example to demonstrate what we can do with our final state, say for the case $f(0,1) \rightarrow (1,0)$:

$$g\Big(\frac{1}{\sqrt{2}}\big(|00\rangle - |10\rangle \big) \Big) \quad \rightarrow \quad \frac{1}{\sqrt{2}}\big(|01\rangle - |10\rangle \big)$$

Compare the result we have here, with the four possibilities above. By sending in this superposition state, our output states corresponds to exactly one of the possible f's (we did it!). Thus, if we could read out the entire final state, we would be done. But alas, we can't see wavefunctions, only measurements:

possibility 1

$$g\Big(\frac{1}{\sqrt{2}}\big(|00\rangle + |10\rangle \big) \Big) \quad \rightarrow \quad |01\rangle$$

possibility 2

$$g\Big(\frac{1}{\sqrt{2}}\big(|00\rangle + |10\rangle \big) \Big) \quad \rightarrow \quad |10\rangle$$

Two things are problematic here: 1) Individually, neither measurement result is conclusive as to what kind of function f is. 2) Even if one of the measurements *could* tell us about f, there's only a 50% chance we get that measurement result.

Never fear, for there is a correct input state still to come. This example was just meant to demonstrate the potential that qubits and superposition states have to offer. The trick is that we need to be thinking of what kind of final wavefunction we will get *and* the information we can extract from a measurement on that state. So, without further ado, let's take a look at the input state that is going to solve our problem:

$$|\psi\rangle_{in} = \frac{1}{2}\big(|00\rangle - |01\rangle + |10\rangle - |11\rangle \big)$$

Which is obtainable by the following sequence of gates:

```
q = QuantumRegister(2,name='q')
deutsch_qc = QuantumCircuit(q,name='qc')

deutsch_qc.h( q[0] )
deutsch_qc.x( q[1] )
deutsch_qc.h( q[1] )

oq.Wavefunction(deutsch_qc)

circuit_drawer(deutsch_qc)
```

```
0.5 |00>    0.5 |10>    -0.5 |01>    -0.5 |11>

q_0: |0>─┤ H ├──────────
q_1: |0>──────┤ X ├─┤ H ├─
```

Alright, let's see what happens when we apply our gate g to this input state:

$$f(0,1) \rightarrow (0,1) \qquad\qquad f(0,1) \rightarrow (1,0)$$

$$g\,|\,\psi\rangle_{in} \;\rightarrow\; \frac{1}{2}\big(\,|\,00\rangle - |\,01\rangle - |\,10\rangle + |\,11\rangle\,\big) \qquad g\,|\,\psi\rangle_{in} \;\rightarrow\; \frac{1}{2}\big(\,-|\,00\rangle + |\,01\rangle + |\,01\rangle - |\,11\rangle\,\big)$$

$$f(0,1) \rightarrow 0 \qquad\qquad f(0,1) \rightarrow 1$$

$$g\,|\,\psi\rangle_{in} \;\rightarrow\; \frac{1}{2}\big(\,|\,00\rangle - |\,01\rangle + |\,10\rangle - |\,11\rangle\,\big) \qquad g\,|\,\psi\rangle_{in} \;\rightarrow\; \frac{1}{2}\big(\,-|\,00\rangle + |\,01\rangle - |\,01\rangle + |\,11\rangle\,\big)$$

Now, based on the four results above, we can start to see an interesting result emerging: the output states for both cases where f is balanced are equal, up to a phase difference. And the same holds true for both output states when f is constant. However, as we noted before, we can only see this when looking at the wavefunctions, but a measurement result will not give us this same information. In fact, all four states will produce the same measurement probabilities.

Sooo, let's do one more thing: apply Hadamard gates to both qubits. Now, I will setup the algebra below, but skip most of the steps. I encourage you to go through one of the calculations for yourself. If you plan to follow along through the rest of the lesson 5 tutorials, I *strongly* recommend working through the algebra steps, as we will be using Hadamard transformations like this one *a lot*:

$$H\,\frac{1}{2}\big(\,|\,00\rangle - |\,01\rangle + |\,10\rangle - |\,11\rangle\,\big) \;=\; \frac{1}{2}\Big(\,\frac{1}{\sqrt{2}}\big(\,|\,0\rangle + |\,1\rangle\,\big)\cdot\frac{1}{\sqrt{2}}\big(\,|\,0\rangle + |\,1\rangle\,\big) \;-\; \cdots\Big)$$

$$= |11\rangle$$

Doing all the algebra for the four possible f functions, we get the following final states:

$$f(0,1) \rightarrow (0,1) \qquad\qquad f(0,1) \rightarrow (1,0)$$

$$|11\rangle \qquad\qquad -|11\rangle$$

$$f(0,1) \rightarrow 0 \qquad\qquad f(0,1) \rightarrow 1$$

$$|01\rangle \qquad\qquad -|01\rangle$$

Let's confirm this with our code:

```
q = QuantumRegister(2,name='q')
deutsch_qc = QuantumCircuit(q,name='qc')

deutsch_qc.h( q[0] )
deutsch_qc.x( q[1] )
deutsch_qc.h( q[1] )

print(' ___ Initial State ___ ')
oq.Wavefunction(deutsch_qc)

f = oq.Blackbox_g_D(deutsch_qc,q)

print('\n ___ After Blackbox ___ ')
oq.Wavefunction(deutsch_qc)

deutsch_qc.h( q[0] )
deutsch_qc.h( q[1] )

print('\n ___ After H^2 ___ ')
oq.Wavefunction(deutsch_qc)
```

```
 ___ Initial State ___
0.5 |00>    0.5 |10>    -0.5 |01>    -0.5 |11>

 ___ After Blackbox ___
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>

 ___ After H^2 ___
1.0 |11>
```

So then, how do we extract our information from these finals states is the final question. If we take a look at all four possibilities, we can see that qubit 1 is always in the $|1\rangle$ state, so that's no good to us. However, when we make a measurement on qubit 0, we will get one of two possibilities. If we measure a $|1\rangle$, we can conclude that f is a balanced function, and if we measure a $|0\rangle$, we can conclude that f is constant. Thus, we have successfully identified what kind of function f is, with only one function call!

We have now completed the Deutsch Algorithm, in all its glory. In order to determine if a black box function f is constant or balanced, we do the following:

$$\text{prepare } |01\rangle \quad \rightarrow \quad H^2 \, |01\rangle \quad \rightarrow \quad g \, |\psi_{in}\rangle \quad \rightarrow \quad H^2 \, |\psi_{out}\rangle \quad \rightarrow \quad \text{measure qubit } 0$$

And as shown above, the measurement result on qubit 0 will perfectly determine f's nature for us. In fact, since we never bother to check qubit 1, we can actually get the same results by only applying a single Hadamard gate on qubit 0, after g. This will result in qubit 0 becoming either $|0\rangle$ or $|1\rangle$, while leaving qubit 1 still in a superposition. This is just a slight optimization.

Since the steps to solving the Deutsch Algorithm are always the same, we can create a function that will always apply the steps for us. And, there is already one waiting for us in Our_Qiskit_Functions, called **Deutsch**:

```
q = QuantumRegister(2,name='q')
c =  ClassicalRegister(2,name='c')
deutsch_qc = QuantumCircuit(q,c,name='qc')

deutsch_qc.iden( q[0] )
deutsch_qc.x( q[1] )

f = oq.Deutsch(deutsch_qc,q)
deutsch_qc.measure(q,c)
#---------------------------
Qubit0_M = list(execute(deutsch_qc, M_simulator,shots=1).result().get_counts(deutsch_qc).keys())[0][1]
if(Qubit0_M=='0'):
    print('Measured state |0>        therefore f is constant!')
else:
    print('Measured state |1>        therefore f is balanced!')

print('  ')
print('    hidden f:   ',f)

```

```
Measured state |0>        therefore f is constant!

    hidden f:    f(0,1) -> 0
```

Running this cell of code a couple times should convince you that we have indeed solved our blackbox f problem using the Deutsch Algorithm. Our conclusion about f is always 100% correct, and we can even check the f function to prove it!

Further Analysis of the Deutsch Algorithm

The last code example is the full Deutsch Algorithm, but our work isn't finished yet. In the next three algorithms, we are going to be encountering similar tricks over and over, and they're all related to the Hadamard gate. Specifically, the 'Hadamard Transformation', which just means we apply H gates to all of the qubits in the system. In this final section, we are going to cover why this transformation works, and also briefly on how we constructed our blackbox g operator.

The Hadamard Transformation

Now, if you followed through with the algebra steps we left out previously, then you may have noticed the underlying pattern. To begin, let's imagine for a second that we exclude the g step of our algorithm. Following our state's wavefunction through each step, we get:

$$|01\rangle \quad -H^2 \rightarrow \quad \frac{1}{2}\big(|00\rangle - |01\rangle + |10\rangle - |11\rangle\big) \quad -H^2 \rightarrow \quad |01\rangle$$

Now, if f happens to be constant, this is essentially the whole process. The effect of g will either leave the state completely unchanged ($f \rightarrow 0$), or apply an overall phase -1 ($f \rightarrow 1$). Thus, in both cases we can see that the second H^2 operation maps us back to either $|01\rangle$ or $-|01\rangle$. Note how two applications of H^2 takes us back to our original state.

By contrast, if f is balanced, the net effect of g appears in the form of moving the two negative signs around. Or more specifically, moving the negative signs onto a different pair of states:

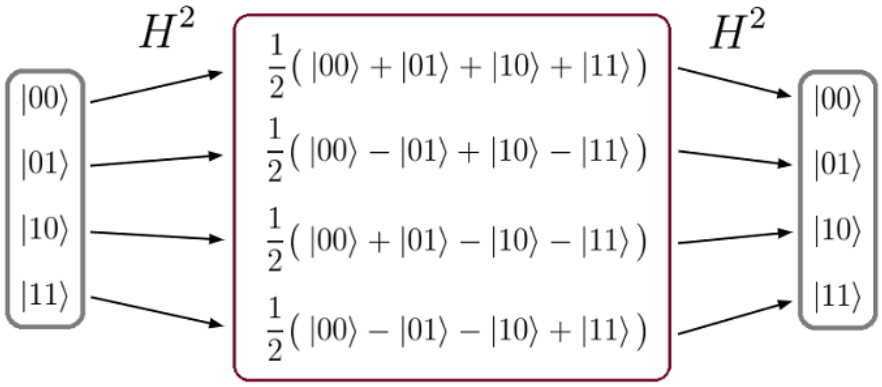
$$f(0,1) \rightarrow (0,1)$$

$$\frac{1}{2}\big(\,|00\rangle - |01\rangle + |10\rangle - |11\rangle\,\big) \quad -g \rightarrow \quad \frac{1}{2}\big(\,|00\rangle - |01\rangle - |10\rangle + |11\rangle\,\big)$$

$$f(0,1) \rightarrow (1,0)$$

$$\frac{1}{2}\big(\,|00\rangle - |01\rangle + |10\rangle + |11\rangle\,\big) \quad -g \rightarrow \quad \frac{1}{2}\big(\,-|00\rangle + |01\rangle + |10\rangle - |11\rangle\,\big)$$

For the top example, we can see that the states $|10\rangle$ and $|11\rangle$ switch amplitudes. And for the second case, we have states $|00\rangle$ and $|01\rangle$ switch. The net effect is that in both balanced cases the states $|00\rangle$ & $|11\rangle$ always have the same sign, as well as $|01\rangle$ & $|10\rangle$. By contrast, for both constant cases, our 'paired' states that always have the same sign are $|00\rangle$ & $|10\rangle$, and $|01\rangle$ & $|11\rangle$. Keep this in mind, as next we are going to show the full Hadamard transformation map on two qubits:



As shown above, the Hadamard Transformation maps each of the four possible two qubit states to a unique superposition state, AND, all of these states are orthogonal (easy to check for yourself). So then, take a look at the mapping above, and compare it to the effect of g we pointed out earlier. You should find that the final states we obtain after g correspond to the ones that map back to $|01\rangle$ for a constant f, and $|11\rangle$ for a balanced f.

Now that we've seen how the H^2 transformation works, let's take a look at the steps of our algorithm again. Run the cell of code below a few times, and confirm for yourself that g always puts our state in one of the superposition states that will get mapped to either $|01\rangle$ or $|11\rangle$ (and don't forget about an overall phase of -1):

```
q = QuantumRegister(2,name='q')
deutsch_qc = QuantumCircuit(q,name='qc')

deutsch_qc.h( q[0] )
deutsch_qc.x( q[1] )
deutsch_qc.h( q[1] )

print(' ___ Initial State ___ ')
oq.Wavefunction(deutsch_qc)

f = oq.Blackbox_g_D(deutsch_qc,q)
print(f)
print('\n ___ After Blackbox ___ ')
oq.Wavefunction(deutsch_qc)

deutsch_qc.h( q[0] )
deutsch_qc.h( q[1] )

print('\n ___ After H^2 ___ ')
oq.Wavefunction(deutsch_qc)
```

```
 ___ Initial State ___
0.5 |00>    0.5 |10>    -0.5 |01>    -0.5 |11>
f(0,1) -> (0,1)

 ___ After Blackbox ___
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>

 ___ After H^2 ___
1.0 |11>
```

Constructing g

We are going to be seeing the Hadamard transformation *a lot* in the coming tutorials, so the section above is a sufficient first exposure. If it didn't full sink in, don't worry. Each lesson we will see the Hadamard transformation used for a slightly different

trick, and in time you will come to appreciate why it's such a powerful tool.

In this final section, we will briefly cover how we constructed the blackbox function g, as it is also insightful into the thought process behind constructing quantum operations from a gate level perspective. To begin, let's see the four possible g operators in matrix form, followed by their gate instructions:

$$f(0,1) \longrightarrow (0,1) \qquad\qquad f(1,0) \longrightarrow (0,1)$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad\qquad \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$f(0,1) \longrightarrow 0 \qquad\qquad f(1,0) \longrightarrow 1$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad\qquad \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

where these matrices are operating on the basis:

$$\begin{bmatrix} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{bmatrix}$$

Take a look at these four matrices and see if you recognize any of them (hint: two of them are gates we studied in lesson 3). For starters, the matrix corresponding to $f(0,1) \rightarrow 0$ is just the Identity matrix on two qubits: I^2. And also, the matrix corresponding to $f(0,1) \rightarrow (0,1)$ is just a CNOT gate! Scroll *all* the way back up to where we first outlined the full effect of g on each possible input state, and confirm for yourself that the operations that describe $f(0,1) \rightarrow 0$ and $f(0,1) \rightarrow (0,1)$ are indeed just I^2 and CNOT.

Now, the case for $f(0,1) \rightarrow 0$ is the easiest to understand. Since everything maps to 0, and our g operator is addition modulo 2: $|\, q_1 \oplus 0\rangle$, every state remains unchanged. For the remaining other three matrices, they are all categorizable by a single pattern: which inputs get mapped to 1. Specifically, if either 0 or 1 gets mapped to 1, we find a corresponding 2×2 matrix located along the diagonal: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. And conversely, if a particular input gets mapped to 0, we find a: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Both of these matrices should be recognizable, as they are just the X and I single qubit gates.

So what is this telling us about g? Well, we are now seeing why we chose to incorporate f into our g operator via $\oplus$ (addition modulo 2). Loosely speaking, adding $\oplus 1$ to a qubit state is equivalent to applying an X, and adding $\oplus\, 0$ is equivalent to doing nothing (which is what an I gate does). But remember that g only affects our second qubit, which means none of these matrices should ever change the state of qubit 0. Or another way of saying that is, the only transformations that are allowed are: $|\, 00\rangle \leftrightarrow |01\rangle$ and $|10\rangle \leftrightarrow |11\rangle$.

So then, how can we deduce what g will look like, based on which inputs map to 0 and 1? Well, for the balanced cases where the input get mapped to one of each output, ask yourself what kind of gate operation flips the states on one qubit, contingent on on the state of the other qubit. That's a CNOT! And for the constant case $f(0,1) \rightarrow 1$, what kind of gate operation flips the states on a particular qubit, regardless of all other qubit states. An X gate!

Starting with the $f(0,1) \rightarrow 1$ case, you may be looking at the matrix representation above and thinking, "that doesn't look like a regular X gate". True, because it's a single qubit X gate applied to a 2-qubit system. Remember, g needs to operate on

the *whole* system, so all matrix representations must be 4×4. Thus, to see what a single qubit operation looks like on a 2-qubit system, we need to take the tensor product: $I \otimes X$. This operation leaves qubit 0 unchanged thanks to the Identity gate, and applies our X gate to qubit 1. If you're new to 'outer product' matrix multiplication, here's a step-by-step walk through:

$$I \otimes X$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 0 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ 0 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix}$$

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

For a reference: https://en.wikipedia.org/wiki/Tensor_product

Luckily, matrix representations are not a necessary ingredient for creating quantum algorithms, only gates (although sometimes matrix representations are very insightful). Thus, when we go to write our code, we need only use a single X gate on qubit 1, and not $I \otimes X$.

Lastly, let's talk about how to construct the case for $f(0,1) \rightarrow (0,1)$. If we look at its matrix representation, it kind of looks like CNOT gate, only backwards. In fact, its effect is exactly like a 'backwards' CNOT gate:

$|00\rangle \rightarrow |01\rangle$ $\quad$ $|01\rangle \rightarrow |00\rangle$ $\quad$ $|10\rangle \rightarrow |10\rangle$ $\quad$ $|11\rangle \rightarrow |11\rangle$.

In essence, it functions like a CNOT gate, where if the control qubit is in the state $|0\rangle$, an X gate is applied to the target.

Since we don't have a gate operation that has this exact effect, we'll have to build one! And to do it, we will essentially borrow a CNOT gate, and 'trick' it into applying an X gate when qubit 0 is in the $|0\rangle$ state. To do this, we will use an X gate on qubit 0 first, then apply a CNOT, and lastly flip qubit 0 back with another X gate:

```
q = QuantumRegister(2,name='q')
zero_CNOT = QuantumCircuit(q,name='qc')

zero_CNOT.h( q[0] )
zero_CNOT.x( q[1] )
zero_CNOT.h( q[1] )

print('\n ___ Initial ___ ')
oq.Wavefunction(zero_CNOT)

zero_CNOT.x( q[0] )

print('\n ___ X ___ ')
oq.Wavefunction(zero_CNOT)

zero_CNOT.cx( q[0],q[1] )

print('\n ___ CNOT ___ ')
oq.Wavefunction(zero_CNOT)

zero_CNOT.x( q[0] )

print('\n ___ X ___ ')
oq.Wavefunction(zero_CNOT)

circuit_drawer(zero_CNOT)
```

```
 ___ Initial ___
0.5 |00>    0.5 |10>    -0.5 |01>    -0.5 |11>

 ___ X ___
0.5 |00>    0.5 |10>    -0.5 |01>    -0.5 |11>

 ___ CNOT ___
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>

 ___ X ___
-0.5 |00>    0.5 |10>    0.5 |01>    -0.5 |11>

q_0: |0>─┤ H ├──────────┤ X ├──■──┤ X ├─
q_1: |0>──────┤ X ├─┤ H ├──────┤ X ├────
```

Compare the first and last wavefunctions, and confirm for yourself that we have indeed achieved the desired operation. By applying X gates before and after the CNOT gate, we are able to effectively use the state $|0\rangle$ as the control. We will see this trick used in future lessons, as it is a very common way for getting a 'control operation' on any specific state.

We are now officially done with Deutsch Algorithm! Although it's just a one-step process, the Deutsch Algorithm is an important first hurdle in terms of understanding how quantum algorithms can outperform classical counterparts, and the subtle math tricks involved. As you've probably already guessed, there's a lot that goes into even the simplest of quantum algorithms! But have no fear, many of the topics covered in this section will be seen again in future algorithms.

This concludes lesson 5.1! As we move through several algorithms over the next couple lessons, you may find that what these quantum algorithm achieves at face value isn't terribly complicated. For example, "determine if f is constant or balanced in one step". But understanding *why* and *how* these quantum algorithms work is much more challenging. This is why our primary focus for these lesson 5 tutorials will be on explaining their inner workings, rather than racing straight to a final code that works.

Lesson 5.2 - Deutsch-Jozsa & Bernstein-Vazirani Algorithms

This tutorial continues the series of in-depth guides to some of the most popular quantum algorithms, all labeled 'Lesson 5'. In this tutorial, we will cover the Deutsch-Jozsa and Bernstein-Vazirani Algorithms, which are problems very closely related to the Deutsch Algorithm. Both algorithms use a Hadamard Transformation as the core to their success, and are in fact solved with the same circuit.

For any reminders / refreshers on Qiskit notation and basics, check out lessons 1 - 4. Also, please consider reading Lesson 5.1 - Intro to Quantum Algorithms (Deutsch), which covers many topics that we will be skipping over for this lesson.

Original publications of the algorithms: [8] & [9]

In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
import Our_Qiskit_Functions as oq
import numpy as np
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

The Deutsch-Jozsa Algorithm

In the first part to this tutorial, we will be studying the Deutsch-Jozsa Algorithm, which is very closely related to the Deutsch Algorithm. The difference in this new problem is that instead of a function f which maps a single bit of 0 or 1 as either a constant or balanced function, we now have an f which maps an *entire* string of bits to either 0 or 1. More specifically, let's say we have a string of bits labeled $\mathrm{x}_0, \mathrm{x}_1, \ldots, \mathrm{x}_n$, where each x_i is either a 0 or a 1. When we pass our string of bits through the function f, it returns a single value of 0 or 1:

$$f(\{x_0, x_1, x_2, ...\}) \quad \rightarrow \quad 0 \text{ or } 1$$

For a sting of n bits, there are a total of 2^n possible combinations. But, just like our previous problem, we are promised that f is either a constant or balanced function. Here, a constant f returns all 0's or 1's for any input, while a balanced f returns 0's and 1's for exactly half of all inputs. For example:

$f_{constant}$		$f_{balanced}$	
$\{0,0,0\} \rightarrow 0$	$\{1,0,0\} \rightarrow 0$	$\{0,0,0\} \rightarrow 0$	$\{1,0,0\} \rightarrow 1$
$\{0,0,1\} \rightarrow 0$	$\{1,0,1\} \rightarrow 0$	$\{0,0,1\} \rightarrow 1$	$\{1,0,1\} \rightarrow 1$
$\{0,1,0\} \rightarrow 0$	$\{1,1,0\} \rightarrow 0$	$\{0,1,0\} \rightarrow 0$	$\{1,1,0\} \rightarrow 0$
$\{0,1,1\} \rightarrow 0$	$\{1,1,1\} \rightarrow 0$	$\{0,1,1\} \rightarrow 1$	$\{1,1,1\} \rightarrow 0$

As shown above, our balanced function returns 0's and 1's for exactly half of the inputs (4 for the case of $n = 3$). And, the ruleset governing which inputs return 0's and 1's is completely independent of the individual bits. That is to say, if our f is truly a randomized balanced function, that takes n bit strings as input, then there are $\frac{2^n!}{2 \cdot 2^{n-1}!}$ possible variations! Thus, there is no

way to conclude any information about f by studying trends with individual bits. In fact, if we instead rewrite our strings of bits as base 10 numbers, then our f function looks like the following:

$f_{constant}$		$f_{balanced}$	
$\{1\} \to 0$	$\{5\} \to 0$	$\{1\} \to 0$	$\{5\} \to 1$
$\{2\} \to 0$	$\{6\} \to 0$	$\{2\} \to 1$	$\{6\} \to 1$
$\{3\} \to 0$	$\{7\} \to 0$	$\{3\} \to 0$	$\{7\} \to 0$
$\{4\} \to 0$	$\{8\} \to 0$	$\{4\} \to 1$	$\{8\} \to 0$

Written this way, it should be more clear that individual bits are meaningless to f. Only the string as a whole determines how f operates on the input. Hopefully the setup for our new problem is clear, and how it varies from our previous one in lesson 5.1. Now then, the question we want to pose to both a classical and quantum computer is: 'how fast can you determine if f is constant or balanced?'

Classical

From the classical perspective, we must check each possible string of bits one by one until we can make a conclusion. In the best case scenario, we get different outputs on our first two inputs, example: 0,0,0,.. $\to$ 0 and 1,0,0,... $\to$ 1. In fact, if we ever get two different outputs, then we can 100% conclude our f is balanced. Conversely, if we continue to see the same output for each input we try, for example:

$$\{0,0,0,0\} \to 1$$

$$\{0,0,0,1\} \to 1$$

$$\{0,0,1,0\} \to 1$$

$$\{0,0,1,1\} \to 1$$

.

.

then we must check exactly 2^{n-1} + 1 combinations in order to conclude that f is constant, which is one more than half the total. To see why we need so many, consider an f where we checked 8 out of 16 total combinations, getting all 0's, only to then find that the 9^{th} input returns a 1. Probabilistically, this is a very unlikely event. In fact, if we get the same result continually in succession, we can express the probability that our f is constant as a function of k inputs as:

$$\mathrm{P}_{constant}(k) \;=\; 1 - \frac{1}{2^{k-1}} \qquad\qquad \text{for } k \leq 2^{n-1}$$

Perhaps more realistically, we can opt to truncate our classical algorithm early, say if we are over X% confident. But if we want the full 100%, then we are stuck checking 2^{n-1} + 1 entries.

Quantum

For our quantum computer, we will solve this problem with 100% confidence after only one function call of f. As we shall see, we achieve this result nearly the exact same way as the Deutsch Algorithm, with only a slight twist.

To show how similar the flow of this algorithm is to the previous one, let's write the complete Deutsch and Deutsch-Jozsa Algorithms side by side:

$$\text{Deutsch:} \quad \text{prepare} \,|\,01\rangle \;\rightarrow\; H^2\,|\,01\rangle \;\rightarrow\; g\,|\,\psi\rangle \;\rightarrow\; H^2\,|\,\psi\rangle \;\rightarrow\; \text{measure qubit 0}$$

$$\text{Deutsch-Jozsa:} \quad \text{prepare} \,|\,0\rangle^{\otimes n}|1\rangle \;\rightarrow\; H^{n+1}|\,0\rangle^{\otimes n}|\,1\rangle \;\rightarrow\; g\,|\,\psi\rangle \;\rightarrow\; H^{n+1}|\,\psi\rangle \;\rightarrow\; \text{measure qubits}^{\otimes n}$$

where once again g is the unitary function that contains our mystery function f.

Comparing these two algorithms, they're nearly identical. The only difference here is that instead of using a single qubit in the state $|0\rangle$, we go through all of the steps with $|0\rangle^{\otimes n}$. Recall that in the Deutsch Algorithm we used a single qubit paired with an ancilla qubit to solve our problem ('ancilla' refers to a qubit(s) that is used but doesn't matter in the final measurement). Here, for our new f that takes an n bit string, we use n qubits paired again with just a single ancilla.

Also like before, we will embed our function f into a unitary operator g, via addition modulo 2 $\oplus$. Let X_i represent some string x_0, x_1, ..., then our g operator acts as follows:

Classical		Quantum
$f\,(\,X_i\,) \rightarrow 0$ or 1	$\Longleftrightarrow$	$g\,\lvert\,X_i\rangle\,\lvert\,\alpha\rangle \rightarrow \lvert\,X_i\rangle\,\lvert\,\alpha \oplus f\,(\,X_i\,)\rangle$

where the state $|X_i\rangle$ refers to the state of the n individual qubits:

$$|\,X_i\rangle \;=\; |\,x_1x_2x_3\cdot\cdot\rangle \;=\; |\,x_1\rangle \otimes |\,x_2\rangle \otimes |\,x_3\rangle\cdot\cdot$$

For example, suppose we had a particular f with the following: $\quad f(010\,) \;\rightarrow\; 1 \quad f(011\,) \;\rightarrow\; 0.$

The corresponding g operator would then:

$$g\,|\,010\rangle\,|\,\alpha\rangle \quad\longrightarrow\quad |\,010\rangle\,|\,\alpha \oplus 0\rangle \quad=\quad |\,010\rangle\,|\,\alpha\rangle$$

$$g\,|\,011\rangle\,|\,\alpha\rangle \quad\longrightarrow\quad |\,011\rangle\,|\,\alpha \oplus 1\rangle \quad=\quad |\,011\rangle\,X\,|\,\alpha\rangle$$

Note that we are using the fact that addition modulo 2 is equivalent to an X gate in this example, a result we showed in lesson 5.1. Thus, we can view the net effect of our g operator as picking out states at random and applying X gates to their ancilla state. For the constant cases, we will have either all or none of the states in the system recieve X gates to their ancilla, while for the balanced cases, exactly half of the states will receive the operation.

Just like the Deutsch Algorithm, a key component is the state $|\alpha\rangle$. As we've laid out our algorithm above, we initialize our ancilla qubit in the $|1\rangle$ state, and then apply a H to it, causing it to be in the state $|-x\rangle$ before our g operator. This means that the effect from the g operation will be as follows:

$$f(\,X_i\,) \;\rightarrow\; 0 \qquad\qquad f(\,X_j\,) \;\rightarrow\; 1$$

$$g\,|\,X_i\rangle\,|-x\rangle \quad\longrightarrow\quad |\,X_i\rangle\,|-x\rangle \qquad\qquad g\,|\,X_j\rangle\,|-x\rangle \quad\longrightarrow\quad -|\,X_j\rangle\,|-x\rangle$$

where this result comes from what happens when we apply an X gate to the state $|-x\rangle$:

$$X\,|-x\rangle = -|-x\rangle$$

Proving this result is a nice exercise that I recommend doing at least once. Write out $|-x\rangle$ in the $\{\,|0\rangle, |1\rangle\,\}$ basis, and it's only a couple line proof.
If you followed along with the Deutsch Algorithm in 5.1, we used this exact trick. Here, we are using this technique again, only on larger qubit states. Let's see this effect in an example, by importing the function **Blackbox_g_DJ** from Our_Qiskit_Functions:

```
q      = QuantumRegister(3,name='q')
anc    = QuantumRegister(1,name='anc')
DJ_qc = QuantumCircuit(q,anc,name='qc')


DJ_qc.h( q[0] )
DJ_qc.h( q[1] )
DJ_qc.h( q[2] )
DJ_qc.x( anc[0] )

print('___ Before g ___')
oq.Wavefunction( DJ_qc, systems=[3,1], show_systems=[True,False] )

DJ_qc.h( anc[0] )
f = oq.Blackbox_g_DJ( 3, DJ_qc, q, anc )
if( f[0]=='constant' ):
    A = 1
else:
    A = 2

DJ_qc.h( anc[0] )

print('\n___ After g ___')
oq.Wavefunction( DJ_qc, systems=[3,A], show_systems=[True,False] )


print('\nf type: ',f[0])
if(len(f)>1):
    print('states mapped to 1: ',f[1:len(f)])
```

```
___ Before g ___
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

___ After g ___
0.35355 |000>    -0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    -0.35355 |101>    -0.35355 |011>    -0.35
355 |111>

f type:  blanaced
states mapped to 1:  ['|111>', '|101>', '|100>', '|011>']
```

We're purposely skipping some of the details regarding the function Blackbox_g_DJ here, in favor of just seeing the important result from g (for example, there's an extra ancilla qubit here that we will explain later).

The code above performs the Deutsch-Jozsa Algorithm up to the step where we apply our blackbox g. Run the cell of code a couple times until you come across a case where f is balanced. When you do, you should notice that exactly half of the states in the system pick up a negative sign. And, these states exactly match up with the states that get mapped to 1 by the embedded f, printed at the bottom.

Next then, our algorithm calls for another Hadamard Transformation of our system, followed by a measurement:

```
q     = QuantumRegister(3,name='q')
anc   = QuantumRegister(1,name='anc')
c     = ClassicalRegister(3,name='c')
DJ_qc = QuantumCircuit(q,anc,c,name='qc')

DJ_qc.h( q[0] )
DJ_qc.h( q[1] )
DJ_qc.h( q[2] )
DJ_qc.x( anc[0] )

print('___ Before g ___')
oq.Wavefunction( DJ_qc, systems=[3,1], show_systems=[True,False] )

DJ_qc.h( anc[0] )
f = oq.Blackbox_g_DJ( 3, DJ_qc, q, anc )
if( f[0]=='constant' ):
    A = 1
else:
    A = 2

DJ_qc.h( anc[0] )

print('\n___ After g      f type: ',f[0],' ___')
oq.Wavefunction( DJ_qc, systems=[3,A], show_systems=[True,False] )

DJ_qc.h( q[0] )
DJ_qc.h( q[1] )
DJ_qc.h( q[2] )

print('\n___ After H^3 ___')
oq.Wavefunction( DJ_qc, systems=[3,A], show_systems=[True,False] )

DJ_qc.measure(q,c)

print('\n___ Measured State ___')
oq.Measurement( DJ_qc, shots=1 )
```

```
___ Before g ___
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

___ After g      f type:  constant  ___
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

___ After H^3 ___
1.0 |000>

___ Measured State ___
1|000>
```

Take a moment to carefully check each step in our 'DJ_qc' printed above, and you should find that all of our steps are in agreement with the algorithm steps we outlined earlier:

$$\text{prepare } |0\rangle^{\otimes n}|1\rangle \quad \rightarrow \quad H^{n+1}|0\rangle^{\otimes n}|1\rangle \quad \rightarrow \quad g\,|\psi\rangle \quad \rightarrow \quad H^{n+1}|\psi\rangle \quad \rightarrow \quad \text{measure qubits}^{\otimes n}$$

The only thing slightly out of order is that we apply the H gate to the ancilla qubit a little early. But, you can verify for yourself that it is applied after g, so it is in agreement with the outlined steps.

Now for the final piece, what to do with our measured state. Recall that the solution to the Deutsch Algorithm problem was based on whether we found qubit 0 in the state $|0\rangle$ or $|1\rangle$:

$$|0\rangle \leftrightarrow f_{constant} \qquad\qquad |1\rangle \leftrightarrow f_{balanced}$$

Here, we will make the same conclusion about f based on the measured state of our n qubits:

$$|000...\rangle \leftrightarrow f_{constant} \qquad\qquad \text{any qubit in state } \; |1\rangle \leftrightarrow f_{balanced}$$

If we measure our n qubit system and we find that any of our qubits are in the $|1\rangle$ state, we can conclude that f is balanced. Conversely, if we find that all of the qubits are in the state $|0\rangle$, we can conclude that f is constant. We can make both of these

conclusions with 100% certainty. With this explanation now in hand, I encourage you to return to the cell of code above, and confirm these results.

And that's the full Deutsch-Jozsa Algorithm! In the next section, we will go back over multiple parts that we skipped over, for a deeper understanding as to why it works.

Deeper Look at the Deutsch-Jozsa

At a quick glance, the three most important keys to the success of the Deutsch-Jozsa algorithm are as follows:

1) the ancilla qubit: $\quad |1\rangle \; - H \rightarrow \; |-x\rangle$

2) Hadamard transformation: $\quad H^{n+1} \;\; (stuff) \;\; H^{n+1}$

3) effect of $\oplus$ in g: $\quad g\,|\,X_i\rangle\,|\,\alpha\rangle \;\equiv\; |\,X_i\rangle\,X\,|\,\alpha\rangle$

It's the combination of these three components that really makes the algorithm tick. We've already seen in the previous section the interplay between keys (1) and (3), and how they result in the flipping of signs on certain states. Now, we're going to focus in particular on the role of the Hadamard Transformation, and why the negative signs make all the difference.

Let's start off with the constant cases, where the claim is that we will always measure the state $|00...0\rangle$. These cases correspond to all states in the system receiving the same action from g, and can be represented as the following matrices:

$$\begin{bmatrix} 1 & 0 & 0 & & & \\ 0 & 1 & 0 & & & \\ 0 & 0 & 1 & & & \\ & & & \cdot & & \\ & & & & \cdot & \\ & & & & & \cdot \end{bmatrix} \equiv \; I^{n+1} \qquad and \qquad \begin{bmatrix} 0 & 1 & & & & \\ 1 & 0 & & & & \\ & & 0 & 1 & & \\ & & 1 & 0 & & \\ & & & & \cdot & \\ & & & & & \cdot \end{bmatrix} \equiv \; I^n \otimes X_{n+1}$$

The first case is just the identity matrix, which makes sense considering that the final state we measure is the exact same as the one we prepare:

$$|0\rangle^{\otimes n}|1\rangle \quad \rightarrow \quad H^{n+1} \quad \rightarrow \quad I^{\otimes n+1} \quad \rightarrow \quad H^{n+1} \quad \rightarrow \quad |0\rangle^{\otimes n}|1\rangle$$

The second case will also result in the same final state, but with a negative sign. To see this, let's work with the case of $n = 2$, plus one ancilla qubit, and consider the effect of the matrix on the states $|000\rangle$ and $|001\rangle$ (where the last qubit is the ancilla):

$$\begin{bmatrix} 0 & 1 & & & \\ 1 & 0 & & & \\ & & \cdot & & \\ & & & \cdot & \\ & & & & \cdot \end{bmatrix} \begin{bmatrix} \alpha_{000} \\ \alpha_{001} \\ \cdot \\ \cdot \\ \cdot \end{bmatrix} = \begin{bmatrix} \alpha_{001} \\ \alpha_{000} \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

The effect of g swaps the amplitudes for the states $|000\rangle$ and $|001\rangle$ (equivalent to an X gate). Now, consider that this g operator is happening between our two Hadamard Transformations. Initially, we only prepare the state $|00\rangle|1\rangle$, but the first H^{n+1}

transformation puts us in an equal superposition of *all* possible states, including $|000\rangle$ and $|001\rangle$. As a result, we observe the following effect from g:

$$\frac{1}{\sqrt{2}}\big(|000\rangle - |001\rangle\big) \quad - g \rightarrow \quad \frac{1}{\sqrt{2}}\big(- |000\rangle + |001\rangle\big)$$

This example only follows the states $|000\rangle$ and $|001$, but if we consider the pattern of the second $f_{constant}$ matrix above, it should be clear that this effect will happen to all of states in our main qubit system:

$$\begin{array}{lllll}
|\,00\rangle\,|-x\rangle & |\,000\rangle - |\,001\rangle & & -|\,000\rangle + |\,001\rangle & -|\,00\rangle\,|-x\rangle \\
|\,01\rangle\,|-x\rangle & |\,010\rangle - |\,011\rangle & & -|\,010\rangle + |\,011\rangle & -|\,01\rangle\,|-x\rangle \\
|\,10\rangle\,|-x\rangle \quad = & |\,100\rangle - |\,101\rangle & -g \rightarrow & -|\,100\rangle + |\,101\rangle \quad = & -|\,10\rangle\,|-x\rangle \\
|\,11\rangle\,|-x\rangle & |\,110\rangle - |\,101\rangle & & -|\,110\rangle + |\,111\rangle & -|\,11\rangle\,|-x\rangle
\end{array}$$

Another way of visualizing what is happening here, is that we are essentially picking up a negative global phase in between our two H^{n+1} transformation. And since a global phase does nothing to our overall system, the state that comes out from the second Hadamard Transformation will be the negative of the state coming into the first:

$$|\,\psi\rangle \quad - H^{n+1} \rightarrow \quad |\,\phi\rangle \;\rightarrow\; -|\,\phi\rangle \quad - H^{n+1} \rightarrow \quad -|\,\psi\rangle$$

And since our initial state for the main qubit system is $|00...0\rangle$, we will get back $-|00...0\rangle$. Thus, we will measure all of the qubits in the $|0\rangle$ state, and conclude that f is constant. In the cell of code below, we follow our quantum system through all the steps of the two constant g cases:

```
q     = QuantumRegister(2,name='q')
anc   = QuantumRegister(1,name='anc')
con1_qc = QuantumCircuit(q,anc,name='qc1')
con2_qc = QuantumCircuit(q,anc,name='qc2')

for i in np.arange(2):
    con1_qc.h( q[int(i)] )
    con2_qc.h( q[int(i)] )
con1_qc.h( anc[0] )
con1_qc.x( anc[0] )
con2_qc.x( anc[0] )
con2_qc.h( anc[0] )

print('___ Before g ___')
oq.Wavefunction( con1_qc)

con2_qc.x( q[0] )
con2_qc.x( q[1] )
con2_qc.x( anc[0] )

print('\n___ After g      f type: balanced ___')
oq.Wavefunction( con1_qc)
print('  ')
oq.Wavefunction( con2_qc)

for i in np.arange(2):
    con1_qc.h( q[int(i)] )
    con2_qc.h( q[int(i)] )
con1_qc.h( anc[0] )
con2_qc.h( anc[0] )

print('\n___ After H^3 ___')
oq.Wavefunction( con1_qc)
print('  ')
oq.Wavefunction( con2_qc)
```

```
___ Before g ___
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

___ After g      f type: balanced ___
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

-0.35355 |000>    -0.35355 |100>    -0.35355 |010>    -0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35
355 |111>

___ After H^3 ___
1.0 |000>

-1.0 |001>
```

As for the balanced cases, their g matrices also have the same $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ structure appearing along the diagonal. However, instead of the full diagonal, only half of the entries will have this X gate structure. That is to say, if we break up the diagonal of a balanced g into 2^n 2×2 blocks, we will find that exactly half of them are $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, and the other half are $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ (an Identity matrix).

What this means for our system, is that if we revisit the the exercise we just did for the states $|000\rangle$ and $|001$ above, we can apply to same logic to half our system. For example:

$$
\begin{array}{ccccccc}
|00\rangle\,|-x\rangle & & |000\rangle - |001\rangle & & -|000\rangle + |001\rangle & & -|00\rangle\,|-x\rangle \\
|01\rangle\,|-x\rangle & & |010\rangle - |011\rangle & & |010\rangle - |011\rangle & & |01\rangle\,|-x\rangle \\
|10\rangle\,|-x\rangle & = & |100\rangle - |101\rangle & -g\rightarrow & |100\rangle - |101\rangle & = & |10\rangle\,|-x\rangle \\
|11\rangle\,|-x\rangle & & |110\rangle - |101\rangle & & -|110\rangle + |111\rangle & & -|11\rangle\,|-x\rangle
\end{array}
$$

which would be the result of the g matrix:

$$\begin{bmatrix} 0 & 1 & & & & & & \\ 1 & 0 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 0 & 1 \\ & & & & & & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha_{000} \\ \alpha_{001} \\ \alpha_{010} \\ \alpha_{011} \\ \alpha_{100} \\ \alpha_{101} \\ \alpha_{110} \\ \alpha_{111} \end{bmatrix}$$

Now, if you take the final state above: $\frac{1}{2}\big(-|\,00\rangle + |\,01\rangle + |\,10\rangle - |\,11\rangle\,\big)\,|-x\rangle$, and apply the second H^{n+1}, you should get -$|11\rangle\,|1\rangle$ as your final state. Which, if we were to make a measurement on, would definitely find at least one qubit in the state $|1\rangle$. Thus, we would conclude that our f is balanced.

But, rather than working through every possible $f_{balanced}$ and verifying that we never get the state $|00\rangle$, all we need to do is notice the trend *why* we will never get it. If we recall why a balanced f in the Deutsch Algorithm would always lead to the state $|11\rangle$, we can apply the same logic here. Specifically, in the Deutsch case, a balanced f would always lead to qubit 0 being in the $|1\rangle$ state because of the second Hadamard Transformation always being applied to either:

$\frac{1}{2}\big(-|\,00\rangle + |\,01\rangle + |\,10\rangle - |\,11\rangle\,\big)$ or $\frac{1}{2}\big(|\,00\rangle - |\,01\rangle - |\,10\rangle + |\,11\rangle\,\big)$, which in turn gives us either $|11\rangle$ or $-|11\rangle$.

So why does that matter. Remember, our condition for concluding if f is constant or balanced is based entirely around measuring the state $|00...0\rangle$. If f is constant, it will be the *only* final state, and if f is balanced, it will *never* be a part of the final system. So then, the question becomes: under what conditions will the second H^n transformation yield only $|00...0\rangle$, or not at all. If you've worked through some of the algebra examples thus far (good for you, gold star!), you may have a hunch as to how the state $|00...0\rangle$ comes out of a Hadamard Transformation. If not, consider all of the individual contributions to the state $|00...0\rangle$ in the example below:

$$H^n\,|\,01101...\rangle \quad \rightarrow \quad \Big(\frac{1}{\sqrt{2}}\Big)^n\big(\,|\,0\rangle + |\,1\rangle\,\big)\big(\,|\,0\rangle - |\,1\rangle\,\big)\big(\,|\,0\rangle - |\,1\rangle\,\big)\cdot\cdot\cdot\cdot$$

No matter how many qubits there are, and no matter which ones are in the state $|0\rangle$ or $|1\rangle$, the state $|00...0\rangle$ will *always* be positive. More specifically, $H|0\rangle$ and $H|1\rangle$ both contribute a positive $|0\rangle$ to the final superposition state. So then, the only way in which we can get a negative sign on the state $|00...0\rangle$ is if the entire state is negative before the Hadamard gate! If there is an overall negative sign before the H^n, then it will carry over to the final state, turning $|00...0\rangle$ negative.

Now then, combine this result with the one just prior: that a balanced f will leave exactly half of the states negative before the second H^{n+1}, and we have our answer. Exactly half of the $|00...0\rangle$ states will come out positive, and the other half will be negative. SO, they cancel out to zero, and our final system will not contain the state $|00...0\rangle$.

No matter to what higher order we go, if f is balanced, this process will *always* cause the state $|00...0\rangle$ to perfectly deconstructively interfere. Thus, we will always measure a state in the system where at least one qubit is in the state $|1\rangle$! And conversely, as we've already shown, the two constant f cases lead to final systems that *only* contain the state $|00...0\rangle$. This is how we can be 100% certain of f based on our measurement result.

```
q     = QuantumRegister(3,name='q')
anc   = QuantumRegister(1,name='anc')
DJ_qc = QuantumCircuit(q,anc,name='qc')


DJ_qc.h( q[0] )
DJ_qc.h( q[1] )
DJ_qc.h( q[2] )
DJ_qc.x( anc[0] )

print('___ Before g ___')
oq.Wavefunction( DJ_qc, systems=[3,1], show_systems=[True,False] )

DJ_qc.h( anc[0] )
f = oq.Blackbox_g_DJ( 3, DJ_qc, q, anc )
if( f[0]=='constant' ):
    A = 1
else:
    A = 2

DJ_qc.h( anc[0] )
print('  ')
print('___ After g ___')
oq.Wavefunction( DJ_qc, systems=[3,A], show_systems=[True,False] )

DJ_qc.h( q[0] )
DJ_qc.h( q[1] )
DJ_qc.h( q[2] )

print('\n___ After H^3 ___')
oq.Wavefunction( DJ_qc, systems=[3,A], show_systems=[True,False] )

print('  ')
print('f type: ',f[0])
if(len(f)>1):
    print(' -States mapped to 1: ',f[1:len(f)])
    print(' -Note that the state |000> is not in our final system!')
```

```
___ Before g ___
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

___ After g ___
-0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    -0.35355 |001>    -0.35355 |101>    0.35355 |011>    -0.35
355 |111>

___ After H^3 ___
-0.5 |010>    -0.5 |110>    0.5 |001>    -0.5 |101>

f type:  blanaced
 -States mapped to 1:  ['|000>', '|111>', '|001>', '|101>']
 -Note that the state |000> is not in our final system!
```

A closer look at the Blackbox_g_DJ

To implement the g operator we've been using thus far, we will follow a trick very similar to the one we used in the Deutsch operator. We've already covered both of the constant cases in the previous section, so here we will focus solely on how to create a balanced g. Specifically, we need to be able to encode the following operation on exactly half of the states in the system:

$$\begin{bmatrix} 0 & 1 & & & \\ 1 & 0 & & & \\ & & \ddots & & \\ & & & & \end{bmatrix} \begin{bmatrix} \alpha_{000} \\ \alpha_{001} \\ \vdots \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} \alpha_{001} \\ \alpha_{000} \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

which for systems larger than $n = 2$, this will require a higher order CNOT operation. For example: $CCCNOT \mid 1110\rangle \rightarrow$

$|\,1111\rangle$.

We've already covered how to implement these higher order CNOT gates in lesson 4, so please refer to that lesson for a more detailed explanation. For our balanced g operators here, all we need to do is pick out half of the states in the system and apply our **n_NOT** gate, using the n qubits as the control, and our ancilla qubit in the $|-x\rangle$ state as the target.

But, since the n_NOT gate only operates on the state of all $|1\rangle$'s, we must use some X gates before and after. Let's see an example of this, where we will use $|010\rangle$ as our control state:

```
q      = QuantumRegister(3,name='q')
trgt   = QuantumRegister(1,name='trgt')
anc    = QuantumRegister(1,name='anc')
qc_010 = QuantumCircuit(q,trgt,anc,name='qc')


qc_010.iden( q[0] )
qc_010.h( q[1] )
qc_010.iden( q[2] )
qc_010.x( trgt[0] )
qc_010.h( trgt[0] )

print(' ___ Initial State ___')
oq.Wavefunction( qc_010, systems=[3,1,1], show_systems=[True,True,False] )

qc_010.x( q[0] )
qc_010.x( q[2] )
oq.n_NOT(qc_010, q, trgt[0], anc)
qc_010.x( q[0] )
qc_010.x( q[2] )

print('  ')
print(' ___ After n_NOT ___')
oq.Wavefunction( qc_010, systems=[3,1,1], show_systems=[True,True,False] )
```

```
 ___ Initial State ___
0.5 |000>|0>    0.5 |010>|0>    -0.5 |000>|1>    -0.5 |010>|1>

 ___ After n_NOT ___
0.5 |000>|0>    -0.5 |010>|0>    -0.5 |000>|1>    0.5 |010>|1>
```

As we can see in this code example, we've achieved the desired effect: $|0100\rangle \rightarrow |0101\rangle$ and $|0101\rangle \rightarrow |0100\rangle$ by using X gates on qubits 0 and 2 before and after the n_NOT operation. These X gates transform our desired control state into the state of all 1's, such that the n_NOT operation will work, and then back to the original state:

$$|\,010\rangle\,|\,0\rangle \quad -X_0 \otimes X_2 \rightarrow \quad |\,111\rangle\,|\,0\rangle \quad -\text{n\_NOT} \rightarrow \quad |\,111\rangle\,|\,1\rangle \quad -X_0 \otimes X_2 \rightarrow \quad |\,010\rangle\,|\,1\rangle$$

And, we must not forget that the n_NOT operation uses an additional $n-2$ ancilla qubits. Thus, in the example above we add an extra qubit to our system in the $|0\rangle$ state before the n_NOT step.

By using this technique, we can effectively pick out any state in the system to be our control, which translates to the operation required of our g matrix:

$$f(\,X_i\,) \;\rightarrow\; 1 \qquad\longleftrightarrow\qquad g\,|\,X_i\rangle\,|-x\rangle \;\rightarrow\; -|\,X_i\rangle\,|-x\rangle$$

With this trick in hand, and some classical code for picking out half the states at random, we have our Deutsch-Jozsa g operator!

Bernstein-Vazirani Algorithm

To quickly recap, we just solved the problem of an unknown f, in which we were told it is either constant or balanced, using only one application of the blackbox followed by a measurement:

$$|000...\rangle \leftrightarrow f_{constant} \qquad\qquad \text{any qubit in state } \; |1\rangle \leftrightarrow f_{balanced}$$

Now, in the second part to this tutorial, we will look at a different problem that can be solved using the same quantum circuit. Specifically, we are given a *new* blackbox function f:

$$f(\,X_i\,) = a \cdot X_i \oplus b$$

where X_i is the same string of bits as before: x_0, x_1, ...

Inside this f, we have two unknown quantities: a and b, where a is a string of bits (same length as X), and b is just a single bit. X_i and a are multiplied together via a standard dot product of vectors, and $\oplus$ still refers to addition modulo 2 here. Let's look at an example:

$$X = \{1,0,1,1\} \qquad a = \{1,0,0,1\} \qquad b = 1$$

$$f\,(\,X\,) = \big(\;1\cdot 1\;+\;0\cdot 0\;+\;1\cdot 0\;+\;1\cdot 1\;\big) \oplus 1 \quad = \quad 2 \oplus 1 \quad = \quad 1$$

So then, the problem which we are going to solve is: how quickly can we determine a with a quantum computer? Determining the constant b can be achieved in one step by passing an X_i of all 0's, by both a classical and quantum computer. Thus, the real challenge is in determining a. Classically, we would have to evaluate f, n times, where n is the length of the bit string a. As we shall see, by using our quantum circuit, we will be able to fully determine a in just one step!

Just like the Deutsch-Jozsa problem, we will solve this new f using the exact same steps:

$$\text{prepare}\;|\,0\rangle^{\otimes n}|1\rangle \quad \rightarrow \quad H^{n+1}|\,0\rangle^{\otimes n}|\,1\rangle \quad \rightarrow \quad g\,|\,\psi\rangle \quad \rightarrow \quad H^{n+1}|\,\psi\rangle \quad \rightarrow \quad \text{measure qubits}^{\otimes n}$$

We also embed our new f into a unitary operator g in the same manner as before:

$$f\,|\,X_i\rangle\,|\,\alpha\rangle \quad \rightarrow \quad |\,X\rangle\,|\,\alpha \oplus f\,(\,X_i\,)\,\rangle$$

Let's see the full example first, and then discuss why it works:

```
q     = QuantumRegister(3,name='q')
anc   = QuantumRegister(1,name='anc')
BV_qc = QuantumCircuit(q,anc,name='qc')

for i in np.arange(3):
    BV_qc.h( q[int(i)] )

print(' ____ Before g ____  ')
oq.Wavefunction( BV_qc,  systems=[3,1], show_systems=[True,False] )

BV_qc.x( anc[0] )
BV_qc.h( anc[0] )
a = oq.Blackbox_g_BV( 3, BV_qc, q, anc )
BV_qc.h( anc[0] )

print('\n ____ After g ____  ')
oq.Wavefunction( BV_qc,  systems=[3,2], show_systems=[True,False] )

for i in np.arange(3):
    BV_qc.h( q[int(i)] )

print('\n ____ After H^3 ____  ')
oq.Wavefunction( BV_qc,  systems=[3,2], show_systems=[True,False] )

print('  ')
print('hidden string a =',a)
```

```
 ____ Before g ____
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

 ____ After g ____
0.35355 |000>    0.35355 |100>    -0.35355 |010>    -0.35355 |110>    -0.35355 |001>    -0.35355 |101>    0.35355 |011>    0.35
355 |111>

 ____ After H^3 ____
1.0 |011>

hidden string a = [0, 1, 1]
```

Run the code above a couple times, and you should find that the final state of our system is always $|a\rangle$. Or in other words, our final system is always guaranteed to be in the state exactly matching the string of bits a. Thus, a measurement on the system will reveal a with 100% accuracy, solving our problem in just one step!

Alrighty, now to explain the magic.

First off, let's show an example of an $f(\mathrm{X})$ on two qubits, and the outputs it produces from each of the four states side by side with the corresponding g operator:

$$a = \{1,0\} \qquad b = 1$$

$$f(\{0,0\}) \rightarrow (0\cdot 1 + 0\cdot 0)\oplus 1 = 1 \quad \leftrightarrow \quad g\,|00\rangle\,|-x\rangle \rightarrow -|00\rangle\,|-x\rangle$$

$$f(\{0,1\}) \rightarrow (0\cdot 1 + 1\cdot 0)\oplus 1 = 1 \quad \leftrightarrow \quad g\,|01\rangle\,|-x\rangle \rightarrow -|01\rangle\,|-x\rangle$$

$$f(\{1,0\}) \rightarrow (1\cdot 1 + 0\cdot 0)\oplus 1 = 0 \quad \leftrightarrow \quad g\,|10\rangle\,|-x\rangle \rightarrow \;\;|10\rangle\,|-x\rangle$$

$$f(\{1,1\}) \rightarrow (1\cdot 1 + 1\cdot 0)\oplus 1 = 0 \quad \leftrightarrow \quad g\,|11\rangle\,|-x\rangle \rightarrow \;\;|11\rangle\,|-x\rangle$$

The thing to note about this pattern is that the effect of f has produced a 1 for exactly half of the states, which will in turn cause a sign flip for half of the states in the system. This result will hold true for all a's, with only one exception: a= 0, 0, 0, ... If a is all 0's, then the effect of f is completely determined by b, which will either flip all of the states, or none.

Causing exactly half of the states to pick up a negative sign ties in with the previous algorithm. Recall that when half of the states in the system undergo a sign flip, the amplitude on the state $|00...0\rangle$ always cancels to zero (deconstructively sums to zero):

$$H^3 \frac{1}{2\sqrt{2}}\big(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |110\rangle - |111\rangle \big) \quad \rightarrow \quad |001\rangle$$

$$H^3 \frac{1}{2\sqrt{2}}\big(|000\rangle + |001\rangle - |010\rangle - |011\rangle + |100\rangle - |101\rangle + |110\rangle - |111\rangle\big) \quad \rightarrow \quad \frac{1}{2}\big(|001\rangle + |010\rangle - |101\rangle + |110\rangle\big)$$

As we can see in these two examples, the state $|000\rangle$ is in neither of the final systems. More importantly however, notice that if a certain combination of states are negative before the H^3, all of the states in the system will deconstructively sum to zero, except for one. But if the combination of states isn't a 'special order', then our final system will be in a superposition state.

So then, what dictates one of the 'special combinations', which will result in a single final state. To answer that question, we can work backwards and apply H^3 to any state and get our answer:

$$|001\rangle \quad \longleftarrow H^3 \longrightarrow \quad \frac{1}{2\sqrt{2}}\big(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |110\rangle - |111\rangle \big)$$

By applying a Hadamard gate to each qubit, and working through the algebra steps, we can find out which combinations of negatives corresponds to any state. Or, we can let our code do it for us:

```
q     = QuantumRegister(3,name='q')
H3_qc = QuantumCircuit(q,name='qc')

state = [1,0,1]
print('Quantum State: ',state)
print('  ')
for i in np.arange( len(state) ):
    if( state[i]==1 ):
        H3_qc.x(  q[int(i)] )
    H3_qc.h( q[int(i)] )

print(' ___ Corresponding H^3 State ___ ')
oq.Wavefunction( H3_qc )
```

```
Quantum State:  [1, 0, 1]

 ___ Corresponding H^3 State ___
0.35355 |000>    -0.35355 |100>    0.35355 |010>    -0.35355 |110>    -0.35355 |001>    0.35355 |101>    -0.35355 |011>    0.35
355 |111>
```

By changing the array 'state' in the example above, we can see what the corresponding state gets mapped to via the Hadamard Transformation.

So then, why does our new blackbox f always guarantee we get the correct negative sign flips that will lead to a single final state? The short answer: because it is consistent. That is to say, this f applies the same rules to every X_i. Remember that for the f in the Deutsch-Jozsa Algorithm, we were only granted it was balanced or constant, but for the cases where it was balanced, we gained no information about the inner workings of which combinations got mapped to 0 or 1.

Here, because the f in this problem is a consistent set of rules: $X_i \cdot a \oplus b$, its effect will apply negative signs in a correspondingly consistent manner. If we think about the effect of the dot product $X_i \cdot a$ in particular, this function essentially picks out how many matches of 1's there are between a and the state. For example, $X_i \cdot a \quad \rightarrow \quad \{1,0,1,1\} \cdot \{1,0,0,1\} = 2$, where 2 is exactly the number of cases where X_i and a have 1's in the same position. Then for our g operator: 2 mod 2 = 0, so ignoring b, this inner product would not result in a sign flip on the state $|1011\rangle$, because $f(\ X_i\ ) = 0$.

If that was a little hard to follow, consider the following two matrices, which show an exact 1-to-1 correlation between states where $X_i \cdot a$ = odd $\quad \longleftrightarrow \quad$ negative signs on states resulting from H^3:

$$X_i = 000\ 001\ 010\ 011\ 100\ 101\ 110\ 111$$

$$a = \begin{pmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{pmatrix} \quad \begin{bmatrix} & & & & & & & \\ o & & o & & o & & & o \\ & o & o & & & o & o & \\ o & o & & & o & o & & \\ & & & o & o & o & o & \\ o & & o & o & & o & & \\ & o & o & o & o & & & \\ o & o & & o & & & & o \end{bmatrix} \quad \longleftrightarrow \quad \begin{bmatrix} & & & & & & & \\ - & & - & & - & & & - \\ & - & - & & & - & - & \\ - & - & & & - & - & & \\ & & & - & - & - & - & \\ - & & - & - & & - & & \\ & - & - & - & - & & & \\ - & - & & - & & & & - \end{bmatrix}$$

The matrix on left shows all of the combinations of $X_i \cdot a$ that result in an odd number, while the matrix on the right shows which states come out negative as a result of $\mathrm{H}^3\,|X_i\rangle$. What this is showing then, is that for all the states where $X_i \cdot a$ results in an odd number, these are the exact states that need to be negative in order for H^3 to map back to a single final state.

Because our f function contains addition modulo 2, it has the following effect based on whether we add an even or odd number:

$$\mathrm{odd} \oplus \{0,1\} = \{1,0\} \qquad\qquad \mathrm{even} \oplus \{0,1\} = \{0,1\}$$

This in turn determines which states pick up a negative signs after g, when used in combination with our $|-x\rangle$ ancilla:

$$|-x\ \oplus\ \mathrm{odd}\,\rangle \ \ = \ \ -|-x\rangle \qquad\qquad |-x\ \oplus\ \mathrm{even}\,\rangle \ \ = \ \ |-x\rangle$$

Thus, we get our 1-to-1 correlation between the states that share an odd number of 1's with a, and the states that will pick up a negative sign before the second H^N mapping. By using the clever trick of setting up our ancilla qubit in the state $|-x\rangle$, the net effect of our blackbox operator g results in negative sign flips on just the right states, as shown above. And the only role that the constant b has then is an overall negative sign:

$$|\,\psi\,\rangle_{final} = (b)^{-1}\,|\,a\rangle$$

which is undetectable by a measurement. And that's the full algorithm!

$*$ *The crowd leaps out of their seats in applause* $*$

We won't go into any further analysis of the g operator here, since it is essentially the same as the Deutsch-Jozsa one. The only difference is on which states receive the $\mathrm{n\_NOT}$ gate. In the Deutsch-Jozsa case, the states were picked at random, while for the Bernstein-Vazirani g, the states are determined by a randomly picked a.

To conclude this tutorial, two cells of code are presented for you to try out, each of which incorporates all of the steps outlined above for the respective two algorithms, written into the functions **Deutsch_Jozsa** and **Bernstein_Vazirani**. In the examples below, change Q to be the number of qubits you would like as the main system:

```
Q = 5                    # Change Q to specify the number of qubits for the main system
#---------------------------------------
#---------------------------------------
q     = QuantumRegister(Q,name='q')
anc   = QuantumRegister(1,name='anc')
c     = ClassicalRegister(Q,name='c' )
DJ_qc = QuantumCircuit(q,anc,c,name='qc')

for j in np.arange(Q):
    DJ_qc.iden( q[int(j)] )
DJ_qc.x( anc[0] )

f = oq.Deutsch_Jozsa(Q, DJ_qc, q, anc)

DJ_qc.measure(q,c)

print(' ___ Measured State ___ ')
M = oq.Measurement(DJ_qc, shots=1, return_M=True)
M = list(list(M.keys())[0] )

#---------------------------------
con = True
for i in np.arange(len(M)):
    if( list(M)[i] == '1'):
        con = False
print('   ')
if(con):
    print('Conclusion:   f is a constant function')
else:
    print('Conclusion:   f is a balanced function')
print('  ')
print('sneak peak:    f is',f[0])
```

```
 ___ Measured State ___
1|11101>

Conclusion:   f is a balanced function

sneak peak:    f is blanaced
```

```
Q = 4                    # Change Q to specify the number of qubits for the main system
#---------------------------------------
#---------------------------------------
q     = QuantumRegister(Q,name='q')
anc   = QuantumRegister(1,name='anc')
c     = ClassicalRegister(Q,name='c' )
BV_qc = QuantumCircuit(q,anc,c,name='qc')

for i in np.arange(Q):
    BV_qc.iden( q[int(i)] )
BV_qc.x( anc[0] )

a = oq.Berstein_Vazirani(Q, BV_qc, q, anc)
BV_qc.measure(q,c)

print(' ___ Measured State ___ ')
oq.Measurement(BV_qc, shots=1)

print('\nsneak peak:    a =',a)
```

```
 ___ Measured State ___
1|0010>

sneak peak:    a = [0, 0, 1, 0]
```

This concludes lesson 5.2! The two algorithms studied in this tutorial are important hurdles towards understanding some of the more complex ones to come. In particular, I encourage you to play around with the code examples, and make sure you fully understand why the Hadamard Transformation allowed us to solve these problems in just one step.

Lesson 5.3 - Simon's Algorithm

In this tutorial, we will cover Simon's Algorithm, which is another 'blackbox' style problem, similar to those we've seen in lessons 5.1 and 5.2. The key difference in solving this new problem, is that it will require multiple measurements as well as a classical computing component.

For any reminders / refreshers on Qiskit notation and basics, check out lessons 1 - 4. Also, please consider reading Lesson 5.1 and 5.2, which cover many of the underlying mathematics that we will see in this lesson.

Original publication of the algorithm: [10]

In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
import Our_Qiskit_Functions as oq
import numpy as np
import math as m
import scipy as sci
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

Simon's Algorithm

Simon's Algorithm will be our final 'blackbox' style algorithm in these lesson 5 tutorials. It will share many of the same tricks as before, but with a unique final answer. Just like lessons 5.1 and 5.2, solving Simon's Algorithm revolves around using Hadamard gates before and after we call upon our blackbox function. The Hadamard Transformation will allow us to apply the effect of the blackbox function to all possible combinations at once, which we use to our advantage.

Quantum Component

Now, let's present Simon's problem: we are given an unknown blackbox function f, which is *guaranteed* to be either one-to-one or two-to-one, where one-to-one and two-to-one functions have the following properties:

$one-to-one$	$two-to-one$
$f(1) \rightarrow 1$	$f(1) \rightarrow 1$
$f(2) \rightarrow 3$	$f(2) \rightarrow 2$
$f(3) \rightarrow 2$	$f(3) \rightarrow 1$
$f(4) \rightarrow 4$	$f(4) \rightarrow 2$

One-to-one functions have exactly one unique output for every input, while two-to-one functions map exactly two inputs to every unique output. In addition, if our f turns out to be two-to-one, we are also guaranteed that there is a 'key bit-string' s which correlates which inputs map to the same output:

$$\text{given } \; x_1, x_2 : \quad f(x_1) = f(x_2)$$

$$\text{guaranteed} : \quad x_1 \oplus x_2 = s$$

So then, given this blackbox f, how quickly can we determine if f is one-to-one or two-to-one? Then, if f turns out to be two-to-one, how quickly can we determine s? As it turns out, both cases boil down to the same problem of finding s, where a key bit-string of $s = \{ 0, 0, 0 \ldots \}$ represents the one-to-one f.

Let's see a quick example of this kind of f:

```
N = 3
s = np.zeros(N)
for i in np.arange(N):
    s[i] = m.floor( 2*sci.rand() )
inputs  = np.zeros(2**N)
outputs = []
for o in np.arange(2**N):
    inputs[o] = ( int(o) )
    outputs.append( int(o) )
f = np.zeros(2**N)
for j in np.arange(2**N):
    out = outputs[int( m.floor( len(outputs)*sci.rand() ) )]
    f[j] = int(out)
    f[ int( oq.From_Binary(oq.Oplus(oq.Binary(j,2**N),s)) ) ] = int(out)
    outputs.remove(out)

print('      s: ',s)
print('  ')
print(' inputs: ',inputs )
print('outputs: ',f)
```

```
      s:  [1. 1. 0.]

 inputs:  [0. 1. 2. 3. 4. 5. 6. 7.]
outputs:  [7. 1. 3. 5. 3. 5. 7. 1.]
```

The cell of code above simulates a blackbox f for a random key bit-string. Run the code a few times, and verify for yourself that all the correlated output states obey: $x_1 \oplus x_2 = s$ (hint, the correlation must be done in binary). Don't worry too much about understanding the lines of code, as the final result is really what we're after: an example of an f function that maps inputs to outputs based on the string s.

Classically, if we want to know what s is for a given f, with 100% certainty, we have to check up to 2^{N-1}+1 (just over half the total) inputs until we find two cases of the same output. Although, probabilistically the average number of inputs will be closer to the order of $O\left(\sqrt{2^N}\right)$. Much like the Deutsch-Jozsa problem, if we get lucky, we could solve the problem with our first two tries. But if we happen to get an f that is one-to-one, or get *really* unlucky with an f that's two-to-one, then we're stuck with the full 2^{N-1}+1.

For our quantum computer, we shall see that our quantum circuit can solve the problem in one step, *some* of the time. Specifically, when we go to measure all the qubits in our second system, there is one measurement result for which we can conclude that f is one-to-one. But if we get any other measurement result, then our work is not quite finished, and we need to do some additional measurements and calculations in order to determine s.

Like our previous quantum algorithms, we will embed our blackbox f into a unitary operator g:

$$g \, | \, X_i \rangle \, | \, \alpha \rangle \quad \longrightarrow \quad | \, X_i \rangle \, | \, \alpha \oplus f \, (\, X_i \,) \, \rangle$$

where α will be the state $|00...0\rangle$.

Let's take a look at an example of a particular f, and its corresponding g operation:

$$s = \{1, 0\}$$

$$\begin{array}{lcl} f(\,00\,) \;\rightarrow\; 11 & & g\,|\,00\rangle\,|\,00\rangle \;\rightarrow\; |\,00\rangle\,|\,11\rangle \\ f(\,01\,) \;\rightarrow\; 10 & & g\,|\,01\rangle\,|\,00\rangle \;\rightarrow\; |\,01\rangle\,|\,10\rangle \\ f(\,10\,) \;\rightarrow\; 11 & \longleftrightarrow & g\,|\,10\rangle\,|\,00\rangle \;\rightarrow\; |\,10\rangle\,|\,11\rangle \\ f(\,11\,) \;\rightarrow\; 10 & & g\,|\,11\rangle\,|\,00\rangle \;\rightarrow\; |\,11\rangle\,|\,10\rangle \end{array}$$

Compare the function f on the left, to the effect of g on the right. The classical version of our f function takes in a string of bits, and outputs a string of bits of equal length. Note that this is different from the Deutsch-Jozsa and Bernstein-Vazirani algorithms we saw in lesson 5.2. The consequence of having an f that outputs a string of bits is that we need to increase the size of our second system. Thus, if our f is a function of an N-bit input, then we need N qubits for our second system.

Just to illustrate this point, let's focus on one particular state:

$$g\;|\;01\rangle\;|\;00\rangle \quad\longrightarrow\quad |\;01\rangle\;|\;f(\,01\,)\,\rangle = |\;01\rangle\;|\;10\rangle$$

Note that the effect of f is not consistent among individual qubits. Only the string as a whole determines which states get mapped to where.

Now, let's use this 2-qubit example to showcase the role of s in this problem. We've already said that s is a string of bits that correlates inputs, such that $x_1 \oplus x_2 = s$. Looking at which input states share the same outputs, we have: $|00\rangle \leftrightarrow |10\rangle$ and $|01\rangle \leftrightarrow |11\rangle$ as our correlated inputs. If we add these states together (modulo 2), we get:

$$\{0,0\} \;\oplus\; \{1,0\} \;=\; \{1,0\}$$

$$\{0,1\} \;\oplus\; \{1,1\} \;=\; \{1,0\}$$

which is indeed the s for this particular f. This string of bits s doesn't provide us any information about the outputs we will get, only the inputs that will share the same output. Which outputs result from correlated input pairs is still completely hidden within f, which means that if we want a complete picture for a given f, more work needs to be done.

Turning now to some code, let's see an example of applying this g operator to a 2-qubit system:

```
q    = QuantumRegister(2,name='q')
anc  = QuantumRegister(2,name='anc')
S_qc = QuantumCircuit(q,anc,name='qc')

S_qc.h( q[0] )
S_qc.h( q[1] )
S_qc.iden( anc[0] )
S_qc.iden( anc[1] )

print(' ____ Initial State ____')
oq.Wavefunction(S_qc, systems=[2,2])

S_qc,s,f = oq.Blackbox_g_S(2, S_qc, q, anc)
print('\ns = ',s)
print('\n ____ After g ____')
oq.Wavefunction(S_qc, systems=[2,2,1], show_systems=[True,True,False])
```

```
 ____ Initial State ____
0.5 |00>|00>    0.5 |10>|00>    0.5 |01>|00>    0.5 |11>|00>

s =  [1. 0.]

 ____ After g ____
0.5 |00>|00>    0.5 |10>|00>    0.5 |01>|10>    0.5 |11>|10>
```

Running the cell of code above should produce the following initial state:

$$\frac{1}{2}\big(\,|\,00\rangle\,|\,00\rangle + |\,01\rangle\,|\,00\rangle + |\,10\rangle\,|\,00\rangle + |\,11\rangle\,|\,00\rangle\,\big)$$

Then, we apply our g matrix, which will result in some final state based on f:

$$\frac{1}{2}\big(\,|\,00\rangle\,|\,f(00)\rangle \;+\; |\,01\rangle\,|\,f(01)\rangle \;+\; |\,10\rangle\,|\,f(10)\rangle \;+\; |\,11\rangle\,|\,f(11)\rangle\,\big)$$

In addition, the hidden key bit-string s is also printed, just so we can verify that the operation is working as intended.

Just like the algorithms in 5.2, we once again need additional ancilla qubits for this g operation, stemming from the fact that there are higher order control-gates within **Blackbox_g_S**. For our goal of understanding Simon's Algorithm, we can ignore these extra ancilla qubits.

I encourage you to run the cell of code above a couple times, to generate different f functions. In the final wavefunction, you should see a result that is similar to our examples earlier, where all four of the initial states in qubits 0 and 1 are still present, and there are always two pairs of states for qubits 2 and 3 (unless you happen upon the case of $s = \{0, 0\}$). The exact final states on qubits 2 and 3 will be different each time, but you should always see two pairs.
Now that we have our unitary operator g, we can write out the full Simon's Algorithm:

$$\text{prepare } |\,0\rangle^N \otimes |\,0\rangle^N \;\longrightarrow\; H^N\,|\,0\rangle^N \otimes |\,0\rangle^N \;\longrightarrow\; g \;\longrightarrow\; H^N\,|\,0\rangle^N \otimes |\,f(x)\,\rangle^N \;\longrightarrow\; \text{measure}$$

1) Prepare both systems in the state of all 0's: $|00...0\rangle$

2) Apply H^N on system 1

3) Apply the unitary operator g

4) Apply H^N on system 1

5) Measure system 1

Using our example code above, let's add in the second Hadamard Transformation and see what we get:

```
q    = QuantumRegister(2,name='q')
anc  = QuantumRegister(2,name='anc')
S_qc = QuantumCircuit(q,anc,name='qc')

S_qc.h( q[0] )
S_qc.h( q[1] )
S_qc.iden( anc[0] )
S_qc.iden( anc[1] )

print(' ____ Initial State ____')
oq.Wavefunction(S_qc, systems=[2,2])

S_qc,s,f = oq.Blackbox_g_S(2, S_qc, q, anc)
print('\ns = ',s)

print('\n ____ After g ____')
oq.Wavefunction(S_qc, systems=[2,2,1], show_systems=[True,True,False])

S_qc.h( q[0] )
S_qc.h( q[1] )

print('\n ____ After H^2 ____')
oq.Wavefunction(S_qc, systems=[2,2,1], show_systems=[True,True,False])
```

```
 ____ Initial State ____
0.5 |00>|00>    0.5 |10>|00>    0.5 |01>|00>    0.5 |11>|00>

s =  [1. 0.]

 ____ After g ____
0.5 |00>|01>    0.5 |10>|01>    0.5 |01>|11>    0.5 |11>|11>

 ____ After H^2 ____
0.5 |00>|01>    0.5 |01>|01>    0.5 |00>|11>    -0.5 |01>|11>
```

That's essentially the full quantum algorithm, so now what? The only thing missing is the final measurement, but seeing the final wavefunction is sufficient here. In the example above, for a non $s = \{0, 0\}$ case, you should find that our main system collapses to two possible states, but neither of them necessarily tells us anything about s.

Unlike the algorithms in lessons 5.1 and 5.2, Simon's Algorithm will require multiple runs. Essentially, each time we run the quantum algorithm, we will extract a new piece of information. Then, with enough pieces, and some luck, we will arrive at our solution. But before going any further into the solution, we need to revisit a property about Hadamard Transformations first. This is now the 3^{rd} lesson in a row where we've used a Hadamard Transformation as the key ingredient of our algorithm (yeah, we're pros at it now!). In order to appreciate what is happening with Simon's Algorithm here, this time we will go over the effects of the Hadamard Transformation using some new math formalism.

To start off, we know that H^N maps the state $|00...0\rangle$ to an even superposition:

$$H^2 \,|\, 00\rangle \quad = \quad \frac{1}{2}\big(\,|\,00\rangle + |\,01\rangle + |\,10\rangle + |\,11\rangle\,\big)$$

But when applied to any other state, H^N will result in some of the states being negative:

$$H^2 \,|\, 01\rangle \quad = \quad \frac{1}{2}\big(\,|\,00\rangle - |\,01\rangle + |\,10\rangle - |\,11\rangle\,\big)$$

The states which become negative are predictable, following a pattern based on which qubits are in the $|1\rangle$ state. We can determine the final state after an H^N transformation as follows:

$$H^N \,|\, x\rangle \quad = \quad \frac{1}{\sqrt{2^N}} \sum_{0}^{2^N-1} (-1)^{x\cdot y} |\, y\rangle$$

where the states $|y\rangle$ are just all of the 2^N possible states. For the case of N = 2, the $|y\rangle$ states are just $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$.

The main reason for introducing this notation is because of the $(-1)^{x \cdot y}$ term, which is necessary for our explanation of Simon's Algorithm. Essentially, this term shows exactly how to pick out which states will be negative, based on the dot product $x \cdot y$. Let's show this using the example above:

$$H \,|\, 01\rangle \;=\; \frac{1}{2}\big(|00\rangle - |01\rangle + |10\rangle - |11\rangle\big) :$$

$$x \quad \cdot \quad y$$

$$\{0,1\} \cdot \{0,0\} = 0 \qquad\qquad (-1)^{x \cdot y} = \;\; 1$$

$$\{0,1\} \cdot \{0,1\} = 1 \qquad \longrightarrow \qquad (-1)^{x \cdot y} = -1$$

$$\{0,1\} \cdot \{1,0\} = 0 \qquad\qquad (-1)^{x \cdot y} = \;\; 1$$

$$\{0,1\} \cdot \{1,1\} = 1 \qquad\qquad (-1)^{x \cdot y} = -1$$

As we can see in this example, the the dot product $(x \cdot y)$ does indeed get all of the correct signs. In general, if this dot product yields an even number, then the state will be positive, and vice versa for an odd number. Note that this result also proves something we pointed out in lesson 5.2, that the effect of the Hadamard Transformation always leaves the state $|00...0\rangle$ positive (a result that led to the interference effects responsible for solving both the Deutsch-Jozsa and Bernstein-Vazirani problems).

Now back to the problem at hand, and the reason why this notation will help us. The key to why we are able to solve Simon's Algorithm faster using a quantum computer comes from the fact that we are guaranteed a key bit-string s in our f. We know that there are exactly two inputs that map to the same output, and all of them are correlated by s. For example:

$$|\,\psi\rangle \;=\; \frac{1}{2}\big(\,|\,00\rangle\,|\,00\rangle \;+\; |\,01\rangle\,|\,00\rangle \;+\; |\,10\rangle\,|\,00\rangle \;+\; |\,11\rangle\,|\,00\rangle\big)$$

$$s \;=\; \{1,1\}$$

$$g\,|\,\psi\rangle \;=\; \frac{1}{2}\big(\,|\,00\rangle\,|\,10\rangle \;+\; |\,01\rangle\,|\,01\rangle \;+\; |\,10\rangle\,|\,01\rangle \;+\; |\,11\rangle\,|\,10\rangle\big)$$

Remember that after we apply g, we have no more interactions with the ancilla system, so we must understand how the mapping of these ancilla qubits affects our main system. To do this, let's rewrite the state above, grouping terms together that share the same ancilla state:

$$|\,\psi\rangle_g = \;\frac{1}{2}\Big(\;\big(\,|\,00\rangle \;+\; |\,11\rangle\,\big)\otimes|\,01\rangle \;+\; \big(\,|\,01\rangle \;+\; |\,10\rangle\,\big)\otimes|\,10\rangle\;\Big)$$

Thus, the effect of g can be thought of as a 'regrouping' of our states. Before g, all of the states in our main system could interfere with each other. But after g, only states that are correlated by s will interfere when we apply the next H^2 gate:

$$H^2\,|\,\psi\rangle_g = \;\frac{1}{2}\Big(\;\big(\,H^2\,|\,00\rangle \;+\; H^2\,|\,11\rangle\,\big)\otimes|\,01\rangle \;+\; \big(\,H^2\,|\,01\rangle \;+\; H^2\,|\,10\rangle\,\big)\otimes|\,10\rangle\;\Big)$$

Now, using our new way of describing the effect of a Hadamard gate, let's see the interference that happens between the states $|00\rangle$ and $|11\rangle$:

$$H^2\,|\,00\rangle \;+\; H^2\,|\,11\rangle \;=\; \sum_y \big((-1)^{00 \cdot y} \;+\; (-1)^{11 \cdot y}\big)|\,y\rangle$$

For each $|y\rangle$, the $|00\rangle$ and $|11$ states are each going to contribute either a 1 or -1, which will lead to some states deconstructively interfering. More specifically, we can show that the states which will deconstructiviley interfere to 0 are related to s:

$$(-1)^{00\cdot y} \; + \; (-1)^{11\cdot y}$$

$$= \quad (-1)^{00\cdot y} \; + \; (-1)^{(\,00\,\oplus\, s\,)\cdot y}$$

$$= \quad (-1)^{00\cdot y} \; + \quad (-1)^{00\cdot y} \cdot \; (-1)^{s\cdot y}$$

$$= \quad (-1)^{00\cdot y} \cdot \big(1 \; + \quad (-1)^{s\cdot y}\big)$$

which will result in all of the $|y\rangle$ to deconstructively interfere when $s \cdot y =$ odd. Thus, we have just shown a correlation between the states that will go to 0, and s. Regardless of the exact mapping of f, the final states of our main system will always be determined by s.

For completeness, we skipped the following steps above:

$$(\,x_1 \oplus x_2\,) \cdot y \;\; = \;\; (\,x_1 \oplus y\,) \cdot (\,x_2 \oplus y\,)$$

$$(-1)^{(\,x_1\,\oplus\, y\,)\,\cdot\,(\,x_2\,\oplus\, y\,)} \;\; = \;\; (-1)^{\,x_1\,\oplus\, y} \cdot (-1)^{\,x_2\,\oplus\, y}$$

which aren't too difficult to verify.

But back to the main point, we now know that the final states of our main system will be entirely determined by s, the string of bits which correlates inputs. Which means, there is a direct link between our final measured state, and our unknown blackbox f.

Classical Solving

Now comes part 2 to Simon's Algorithm, the classical component. If you followed along all of the quantum steps up to this point, the hardest part is over. Once you understand *how* and *why* the quantum component of Simon's Algorithm works, the classical steps are much more straightforward.

The final result of our analysis above is that the final state of our main system consists of only *half* of all possible states. Specifically, the states that survive the second Hadamard Transformation correspond to states where $s \cdot y =$ even (we showed that states where $s \cdot y =$ odd all deconstructively go away).

SO THEN, knowing this, we can use our measurement results to try and figure out s. Specifically, each new measurement result we get gives us another piece of information about s. Once we get enough unique measurement results, we can combine them together in a set of linear equations. For example, let's use our 2-qubit example, and suppose we got the following measurement results:

$$1) \; |\,00\rangle \qquad\qquad 2) \; |\,00\rangle \qquad\qquad 3) \; |\,11\rangle$$

which we can then combine into the set of equations:

$$\{0,0\} \cdot s = 0 \qquad\qquad\qquad\qquad\qquad\qquad (\text{no information})$$

$$\{1,1\} \cdot s = 0 \qquad (\text{modulo } 2) \qquad\qquad \longleftrightarrow \qquad\qquad s_0 \oplus s_1 = 0$$

On our first two trials we measure the state $|00\rangle$, which gives us no information about s (repeat measurement results is a unavoidable problem of our quantum algorithm). But on the third trial, we find the state $|11\rangle$, which does give us some information. There are two possible solutions to the second equation above: s={0,0} or s={1,1}. The first of these two solutions is *always* a solution, and represents a special case (which we will cover next). The second solution is a valid candidate, and since we've already measured all possible states (because we showed that exactly half of all states survive after the second H^2), there's nothing more we can do with our quantum system.

So then, to conclude if our candidate s' = {1,1} is really our s, we test our f classically:

$$f(\,00\,) \;=\; f(\,s'\,) \qquad\qquad \therefore \;\; s = s'$$

$$f(\,00\,) \;\neq\; f(\,s'\,) \qquad\qquad \therefore \;\; s = \{\,0,\,0\,\}$$

By testing our classical f for the cases s' and 0^N, we will arrive at one of two conclusions about s. Following the logic from our set of linear equations, we can narrow down s to only two possibilities: s' and 0^N. By prompting $f(\,s\,)$ and $f(\,0^N\,)$, finding that they both give the same output will conclude that s' is indeed our hidden key bit-string. Conversely, if they yield different outputs, then the only possibility for s is the string of all 0's.

To fully understand this conclusion requires that you understood all of the preceding quantum steps, so it may take a few reads before it full sinks in. Essentially, because of the way we arrive at the two candidates above, we are *guaranteed* that they are the *only* possibilities.

Since we are ultimately turning our quantum results over to a set of linear equations for solving, it is worth noting how many equations we need to solve for s. Our final quantum system will be an even distribution of 2^{N-1} states, which gives us 2^{N-1} equations. However, $\{0, 0, 0, ...\}$ may emerge from one of our measurements, and several others may not be linearly independent. Since f is a function of N bits, we need at most N linearly independent equations for a solution (but as we shall see, we can often get away with fewer).

Thus, there's no guarantee on how long it takes our algorithm to arrive at a set of linearly independent equations in order to solve for s. While this isn't ideal, it's important to understand that not all quantum algorithms are deterministic. The algorithms in 5.1 and 5.2 solve their respective problems with 100% success rates, but they are the exception to the rule. Almost all quantum algorithms that we will study from this point on will come with some inherent probabilities of failure, where 'failures' typically mean we have to run the quantum algorithm again.

For Simon's Algorithm, we are ultimately reliant on the final measurements. First, we are probabilistically halted until we measure up to N independent states, and then hope that they are enough to solve for s. As N gets bigger however, our probability of getting N out of 2^{N-1} possible states goes up. But once we have enough unique measurements, the solving of the linear equations is easy, since we can just let a classical algorithm handle that (much later we shall see that there are even quantum algorithms to handle this portion too!).

Ideally, we would want to perform measurements simultaneously while trying to solve our set of linear equations. This way we perform the fewest number of quantum runs. We will do this later, but for learning purposes now, we will 'overshoot' and run the quantum component of our algorithm more times than needed in order to make sure we get enough unique measurements. After that, we let our custom function **Simons_Solver** classically work through all the possible s' candidates, and return back an answer:

```
q    = QuantumRegister(3,name='q')
c    = ClassicalRegister(3,name='c')
anc  = QuantumRegister(3,name='anc')
S_qc = QuantumCircuit(q,anc,c,name='qc')

for i in np.arange(6):
    if(i < 3):
        S_qc.h( q[int(i)] )
    else:
        S_qc.iden( anc[int(i-3)] )

S_qc,s,f = oq.Blackbox_g_S(3, S_qc, q, anc)

for i in np.arange(3):
    S_qc.h( q[int(i)] )

S_qc.measure(q,c)
#-------------------------------------
run_quantum = True
while( run_quantum ):
    M = oq.Measurement( S_qc, shots=20, return_M=True, print_M=False)
    if( len(list(M.keys())) >= 4 ):
        run_quantum = False
        print('Measurement Results: ',M)
        Equations = []
        for i in np.arange( len(list(M.keys())) ):
            if( list(M.keys())[i] != '000' ):
                Equations.append([ int(list(M.keys())[i][0]), int(list(M.keys())[i][1]), int(list(M.keys())[i][2]) ])
        s_primes = oq.Simons_Solver(Equations,3)
        print('\ncandidate:  ',s_primes)
        print('\n hidden s:  ',s)
```

```
Measurement Results:  {'010': 4, '001': 6, '011': 6, '000': 4}

candidate:   [[1, 0, 0]]

 hidden s:   [1. 0. 0.]
```

Run this example a couple of times and see that our algorithm works! Using our measurement results, the Simons_Solver function takes all of the linear equations obtained through measurements and returns to us a list of possible candidates for s. If we don't provide enough equations, it will return a list of multiple candidates. If our system of equations is sufficient, the function will return a list with a single candidate (s'). If we happen to get an s of all 0's, it will return a lost of either one or no entries (which is why the final step is to always check $f\left(0^N\right) = f\left(s'\right)$).

As we can see in the 'Measurement Results' line above, we run our quantum system far more times than needed. If we want to optimize the process, we can run the Simons_Solver after every unique measurement result, until it returns only a single value:

```
q    = QuantumRegister(3,name='q')
c    = ClassicalRegister(3,name='c')
anc  = QuantumRegister(3,name='anc')
S_qc = QuantumCircuit(q,anc,c,name='qc')

for i in np.arange(6):
    if(i < 3):
        S_qc.h( q[int(i)] )
    else:
        S_qc.iden( anc[int(i-3)] )

S_qc,s,f = oq.Blackbox_g_S(3, S_qc, q, anc)

for i in np.arange(3):
    S_qc.h( q[int(i)] )
S_qc.measure(q,c)
#-------------------------------------
run_quantum = True
Equations = []
Results = []
quantum_runs = 0
while( run_quantum ):
    quantum_runs += 1
    M = oq.Measurement( S_qc, shots=1, return_M=True, print_M=False)
    new_result = True
    for r in np.arange(len(Results)):
        if( list(M.keys())[0] == Results[r]):
            new_result = False
    if(new_result):
        Results.append( list(M.keys())[0] )
        Equations.append([ int(list(M.keys())[0][0]), int(list(M.keys())[0][1]), int(list(M.keys())[0][2]) ])
        s_primes = oq.Simons_Solver(Equations,3)
        if( len(s_primes) ==1 ):
            run_quantum = False

print('\n          candidate:  ',s_primes)
print('\n           hidden s:  ',s)
print('\nunique measurements:  ',Results)
print('\n       quantum runs:  ',quantum_runs)
```

```
          candidate:  [[1, 1, 1]]

           hidden s:  [1. 1. 1.]

unique measurements:  ['011', '000', '101']

       quantum runs:  9
```

Compare the 'unique measurements' line in this example, to the 'Measurement Results' from the previous one. For a 3-qubit system, often times only two unique measurements is sufficient to solve for s. By trying to solve for s after each time we get a unique measurement, we ensure that we don't run our quantum system more times than needed. BUT, keep in mind that every time Simons_Solver doesn't return a single value, we've essentially 'wasted' some computational time (although an argument can be made that these quantum and classical computers work in parallel, and do not bottleneck each other). Thus, for our code example, there's a balance between the number of unique measurements we should acquire before we start solving the linear equations.

This is just one example of how quantum algorithms are not straightforward speedups. Because probability is always involved, we find situations where the exact number of 'steps' isn't constant. For Simon's Algorithm, the 'speed' at which we arrive at our answer is largely determined by how lucky we get at finding unique solutions. At lower problem sizes, this probability actually causes our quantum algorithm to be slower (a very common feature as we shall see). Thus, we need to implement Simon's Algorithm on larger problems if we really want to see a speedup.

To conclude this tutorial, the cell of code below incorporates all of the steps outlined thus far into the function **Simons**. Change Q to be the number of qubits you would like as the main system:

```
Q = 4
#------------------------------
#------------------------------
q    = QuantumRegister(Q,name='q')
c    = ClassicalRegister(Q,name='c')
anc  = QuantumRegister(Q,name='anc')
S_qc = QuantumCircuit(q,anc,c,name='qc')

for i in np.arange(2*Q):
    if(i < Q):
        S_qc.iden( q[int(i)] )
    else:
        S_qc.iden( anc[int(i-Q)] )

S_qc,s  = oq.Simons_Quantum(Q, S_qc, q, c, anc)
sp,r,qr = oq.Simons_Classical(Q, S_qc)

print('\n          candidate:  ',sp)
print('\n           hidden s:  ',s)
print('\nunique measurements:  ',r)
print('\n       quantum runs:  ',qr)
```

```
          candidate:   [[1, 1, 0, 0]]

           hidden s:   [1. 1. 0. 0.]

unique measurements:   ['1111', '0001', '1110', '1101']

       quantum runs:   6
```

This concludes lesson 5.3! The algorithm studied in this lesson is our first taste of what is commonly referred to as a 'hybrid' algorithm, whereby the final solution is reached via a mix of quantum / classical computing. Simon's Algorithm is an excellent introduction to these kinds of algorithms because the final result is still relatively deterministic. That is to say, given enough measurements on the system, we will eventually have enough linear equations to solve for s.

Lesson 5.4 - The Grover Search

In this final lesson 5 tutorial, we will cover the Grover Algorithm. Like the previous algorithms we've studied, at the heart of the Grover Algorithm is a Hadamard Transformation. However, this algorithm does not solve a 'blackbox' problem, making it different from our previous three lessons. Instead, we will be solving a searching problem, whereby we would like to locate one particular state with a measurement, out of 2^N.

Original publication of the algorithm: [11]

In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
import Our_Qiskit_Functions as oq
import numpy as np
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

The Grover Algorithm

The Grover Algorithm, also referred to as a Grover Search, is a quantum algorithm that can be thought of as searching through an unordered list. Imagine you want to look someone up in a directory, which is alphabetically ordered by last name, but you only have their first name. In this scenario, you are stuck going through each entry one at a time, until you eventually happen upon the person you are looking for.

Exhaustively searching through the database represents the classical approach, which requires on average $\frac{N}{2}$ evaluations, which is of the order $O(N)$. By instead using the Grover Algorithm, we can complete this search (with a high success probability) using only $O(\sqrt{N})$ evaluations.

Setting Up the Problem

Our goal is to create a quantum algorithm that will allow us to pick any state we want (within the 2^N space), and then attempt to find that state with a single measurement. As we shall see, we will measure our desired state, which we shall refer to as our 'marked state', with a high success probability. In addition, larger systems will result in higher success probabilities, a nice feature that is unique to the quantum approach!

Like the classical search, our quantum algorithm needs to first reflect the problem of having no *a priori* knowledge of where the marked entry is located. For our quantum algorithm, we can represent this by starting our system in an equal superposition of all states. Thus, the starting point for our code will be to specify the size of our problem, and then create an equal superposition:

```
N = 3
#----------------
q  = QuantumRegister(N,name='q')
qc = QuantumCircuit(q,name='qc')

for i in np.arange(N):
    qc.h( q[int(i)] )
oq.Wavefunction( qc )
```

```
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>
```

In the code above, we specify the size of our problem with the parameter N, creating a quantum system of the size 2^N. N is the number of qubits we will be using, which means we can create significantly large systems with only a minimal amount of

qubits. We prepare our system in an equal superposition of all 2^N states by applying a Hadamard gate to each qubit, creating the following initial state:

$$H^{\otimes N}|000...0\rangle \;=\; \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} |k\rangle \;\equiv\; |\,s\,\rangle$$

Now, let's do some simulated measurements on this state. These measurements represent the classical approach of picking blindly until we happen on our desired state:

```
N = 3
#----------------
q  = QuantumRegister(N,name='q')
c  = ClassicalRegister(N,name='c')
qc = QuantumCircuit(q,c,name='qc')

for i in np.arange(N):
    qc.h( q[int(i)] )
qc.measure(q,c)
oq.Measurement( qc, shots=100)
```

```
11|010>    15|110>    7|111>    18|001>    17|000>    13|100>    7|101>    12|011>
```

Take a look at the measurement counts for each state and verify that all states in the system are equally probable (although it's rare to get a perfectly even distribution). Using a quantum system like this to find a specific state is quite slow, and in fact it's even worse than the classical analog! Consider what the typical method would be if we were to pick states at random classically: suppose we are looking for the state $|000\rangle$, but instead got $|110\rangle$. It would be crazy to put $|110\rangle$ back into the mix and try again. Thus, we would naturally remove is from the problem, thereby improving our odds of finding $|000\rangle$ on the next try.

The main advantage to a classical search is the ability to 'remember' past measurements, and remove them from the problem. By doing so, the classical approach will slowly narrow down the pool of possible entries, until eventually finding the desired one. When using a quantum approach, we can't do this. If we measure the state $|110\rangle$, that's it. Our wavefunction collapses to that state, and we've failed our search. And, when we go to prepare the system the next time, we have no way of removing the state $|110\rangle$ from the system, which means we could get it again!

The difference between the classical and quantum approaches to a search problem are very noteworthy. Since our quantum system has no memory of past measurements, we can only hope to find our desired state with a single attempt. Thus, the goal of the Grover Algorithm will be to boost our chance of measuring the desired state.

Implementing an Oracle

Now that we have our equal superposition of 2^N states, we can begin to construct our Grover Algorithm.

To do this, the first thing we need is an operator U_w, known as an 'oracle.' Simply put, this is an operator that picks out a single state in the system, say $|0101\rangle$, and applies an operation. Specifically, this oracle operator U_w isolates a single state such that it is the *only* state in the system that will then receive the desired operation.

We've worked with similar operators in the past, such as the control gates from lesson 3, and the higher order $\mathrm{n\_NOT}$ gates in lesson 4. In essence, that's exactly what we're going to do here as well. By default, control gates only pick out states where all of the control qubits are in the state $|1\rangle$, for example:

$$CCNOT \;\; |\,100\rangle \;\rightarrow\; |\,100\rangle$$

$$CCNOT \;\; |\,110\rangle \;\rightarrow\; |\,111\rangle$$

For our Grover algorithm, we need our oracle to be able to pick out *any* state, including states with 0's on any qubit. Luckily, we've already seen how to pull off this trick before in our past blackbox functions.

In order to make sure that only our marked state is the control-state, we will perform a series of X gates to *transform* our marked state to the state of all 1's: $|111...1\rangle$. Simultaneously, this transformation will also guarantee that our marked state is the *only* state in the system of all 1's. Thus, when we apply our N-qubit control gate operation, its effect will *only* get applied to our marked state. Then, we will transform all of the states back to the original basis, using the same X gates:

```
q    = QuantumRegister(2,name='q')
G_qc = QuantumCircuit(q,name='qc')

G_qc.h( q[0] )
G_qc.h( q[1] )
G_qc.cz( q[0],q[1] )

print('____ Initial State ____')
oq.Wavefunction(G_qc)

print('\nX_Transformation:   |00>  <-->  |11>')


oq.X_Transformation(G_qc, q, [0,0])
print('\n____ After X(0) + X(1) ____')
oq.Wavefunction(G_qc)
```

```
____ Initial State ____
0.5 |00>    0.5 |10>    0.5 |01>    -0.5 |11>

X_Transformation:   |00>  <-->  |11>

____ After X(0) + X(1) ____
-0.5 |00>    0.5 |10>    0.5 |01>    0.5 |11>
```

In the example above, we transform the state $|00\rangle \rightarrow |11\rangle$ by applying X gates on qubits 0 and 1. We mark the $|11\rangle$ state with a negative phase just for clarity here, so we can track which state it gets transformed to (the state that ends up with the negative sign is the original state that maps to $|11\rangle$).

In this example, we use our custom function **X_Transformation** to perform the correct X gates, specifying the desired state we want to map to the state of all 1's. In general, choosing which X gates to perform is very straightforward, as all we need to do it look at where the 0's are for our marked state. In the example above, our marked state would be $|00\rangle$, which has 0's in the qubit locations 0 and 1, *therefore* we applied the gates $X(0)$ and $X(1)$.

Equally as important as the transformation of the marked state, is the effect of this transformation on the rest of the system. For example, consider the effect of the X Transformation when $|001\rangle$ is the marked state:

$$\text{Applying X(0) + X(1)}$$

$$\begin{aligned}
|\,000\rangle &\rightarrow |\,110\rangle & |\,100\rangle &\rightarrow |\,010\rangle \\
|\,001\rangle &\rightarrow |\,111\rangle\,\star & |\,101\rangle &\rightarrow |\,011\rangle \\
|\,010\rangle &\rightarrow |\,100\rangle & |\,110\rangle &\rightarrow |\,000\rangle \\
|\,011\rangle &\rightarrow |\,101\rangle & |\,111\rangle &\rightarrow |\,001\rangle
\end{aligned}$$

No other state in the system gets mapped to $|111\rangle$, exactly the result we need. If we consider that our marked state is 'unique', in that no other state in the system has the same 0's and 1's, it makes sense that the transformation to $|111\rangle$ is unique as well, mapping all other states elsewhere.

The last step is the transformation back to our original basis. For our Grover Algorithm, transforming our marked state to $|11...1\rangle$ will allow us to apply a higher order control operation, but afterwards, we must transform back in order to search for the marked state in its original form. Lucky for us, the transformation back to our original basis is just as easy. All we need to do is apply the exact same X gates again:

```
q    = QuantumRegister(2,name='q')
G_qc = QuantumCircuit(q,name='qc')
marked = [0,1]

G_qc.h( q[0] )
G_qc.h( q[1] )
G_qc.cz( q[0],q[1] )
G_qc.x( q[0] )

print(' ____ Initial State ____')
oq.Wavefunction(G_qc)

oq.X_Transformation(G_qc, q, marked)

print('\n ____ X(0)  ____')
oq.Wavefunction(G_qc)

oq.X_Transformation(G_qc, q, marked)
print('\n ____ X(0)  ____')

oq.Wavefunction(G_qc)
```

```
 ____ Initial State ____
0.5 |00>    0.5 |10>    -0.5 |01>    0.5 |11>

 ____ X(0)  ____
0.5 |00>    0.5 |10>    0.5 |01>    -0.5 |11>

 ____ X(0)  ____
0.5 |00>    0.5 |10>    -0.5 |01>    0.5 |11>
```

In the example above, we successfully transform back and forth between our marked state and $|11...1\rangle$. Next, we are going to use this transformation to effectively apply a higher order control-Z gate to our marked state.

Sign Flip on $|111...1\rangle$ (The Oracle Function)

The first component to our Grover Algorithm, the oracle U_w, will achieve the effect of an N-control-Z gate, applied to our marked state. That is to say, it will achieve the effect: $|11...1\rangle \rightarrow -|11...1\rangle$. In matrix form, this operator looks like:

$$\begin{bmatrix} 1 & 0 & 0 & . & . & . & & \\ 0 & 1 & 0 & & & & & \\ 0 & 0 & 1 & & & & & \\ . & & & . & & & & \\ . & & & & . & & & \\ . & & & & & . & & \\ & & & & & & 1 & 0 \\ & & & & & & 0 & -1 \end{bmatrix}$$

There are a couple ways to achieve this operation, but we are going to use the most common method, which involves an ancilla qubit in the state $|-x\rangle$. In fact, we've already seen this trick in lessons 5.1 and 5.2.

Essentially, we will be taking advantage of the effect of an X gate on the state $|-x\rangle$:

$$X\,|-x\rangle \;=\; -|-x\rangle$$

Since every state in the system will be coupled to this $|-x\rangle$ state, we must be sure that *only* our marked state receives the X gate operation on the ancilla qubit. For example, suppose $|01\rangle$ was our marked state:

$$
\begin{array}{lcll}
|\,00\rangle\,|-x\rangle & & |\,00\rangle\,|-x\rangle & |\,00\rangle\,|-x\rangle \\
|\,01\rangle\,|-x\rangle & & |\,01\rangle\,X\,|-x\rangle & -\,|\,01\rangle\,|-x\rangle \\
|\,10\rangle\,|-x\rangle & \longrightarrow & |\,10\rangle\,|-x\rangle & = \quad |\,10\rangle\,|-x\rangle \\
|\,11\rangle\,|-x\rangle & & |\,11\rangle\,|-x\rangle & |\,11\rangle\,|-x\rangle
\end{array}
$$

This example above shows the desired effect of our Oracle, essentially causing our marked state to pick up a negative phase. Then, after the negative sign has been applied, we work with our main system only, completing ignoring the ancilla.

To achieve the effect shown above, we will need the combination of our $\mathrm{X\_Transformation}$ function with **n_NOT**, also one of our custom functions. In short, the $\mathrm{n\_NOT}$ function is equivalent to any higher order CNOT gate of our choosing. Thus, we will use it to perform an N^{th} order CNOT operation (N being the number of qubits in our system), with the target qubit being the ancilla.

For a refresher on exactly how our $\mathrm{n\_NOT}$ operation achieved a higher order CNOT gate, please refer to lesson 4.

In total, the flow of our Oracle function will be as follows:

$$
|\,\Psi\rangle_i \otimes |-x\rangle \quad \rightarrow \quad \mathrm{X\_Transformation} \quad \rightarrow \quad \mathrm{n\_NOT} \quad \rightarrow \quad \mathrm{X\_Transformation} \quad \rightarrow \quad |\,\Psi\rangle_f \otimes |-x\rangle
$$

Let's see it in code:

```
q     = QuantumRegister(3,name='q')
anc   = QuantumRegister(1,name='anc')
n_anc = QuantumRegister(1,name='nanc')
G_qc  = QuantumCircuit(q,anc,n_anc,name='qc')
marked = [0,1,0]

G_qc.h( q[0] )
G_qc.h( q[1] )
G_qc.h( q[2] )
G_qc.x( anc[0] )

print('____ Initial State ____')
oq.Wavefunction(G_qc, systems=[3,1,1], show_systems=[True,False,False] )


G_qc.h( anc[0] )
oq.X_Transformation(G_qc, q, marked)
print('\n____ H(q[3]) + X_Transformation  ____')
oq.Wavefunction(G_qc, systems=[3,1,1], show_systems=[True,True,False] )

oq.n_NOT(G_qc, q, anc[0], n_anc)
print('\n____ n_NOT  ____')
oq.Wavefunction(G_qc, systems=[3,1,1], show_systems=[True,True,False] )


oq.X_Transformation(G_qc, q, marked)
G_qc.h( anc[0] )
print('\n____ X_Transformation + H(q[3])  ____')
oq.Wavefunction(G_qc, systems=[3,1,1], show_systems=[True,False,False] )
```

```
____ Initial State ____
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

____ H(q[3]) + X_Transformation  ____
0.25 |000>|0>    0.25 |100>|0>    0.25 |010>|0>    0.25 |110>|0>    0.25 |001>|0>    0.25 |101>|0>    0.25 |011>|0>    0.25 |11
1>|0>    -0.25 |000>|1>    -0.25 |100>|1>    -0.25 |010>|1>    -0.25 |110>|1>    -0.25 |001>|1>    -0.25 |101>|1>    -0.25 |011
>|1>    -0.25 |111>|1>

____ n_NOT  ____
0.25 |000>|0>    0.25 |100>|0>    0.25 |010>|0>    0.25 |110>|0>    0.25 |001>|0>    0.25 |101>|0>    0.25 |011>|0>    -0.25 |1
11>|0>    -0.25 |000>|1>    -0.25 |100>|1>    -0.25 |010>|1>    -0.25 |110>|1>    -0.25 |001>|1>    -0.25 |101>|1>    -0.25 |01
1>|1>    0.25 |111>|1>

____ X_Transformation + H(q[3])  ____
0.35355 |000>    0.35355 |100>    -0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>
```

The example above follows all of the steps for our Oracle operator. Note that in these steps, the Hadamard gates on the ancilla qubit are separated out to better show the negative sign being applied to the marked state. Feel free to change the array 'marked' in this example, and see that it will always pick out the correct state. Also note that calling upon the $\mathrm{n\_NOT}$ function requires the use $N-2$ extra qubits, which we've chosen not to display in our last two wavefunctions.

To avoid clutter, we combine all of the operation steps above into a function called **Grover_Oracle**:

```
q     = QuantumRegister(3,name='q')
anc   = QuantumRegister(1,name='anc')
n_anc = QuantumRegister(1,name='nanc')
G_qc  = QuantumCircuit(q,anc,n_anc,name='qc')
marked = [0,1,0]

G_qc.h( q[0] )
G_qc.h( q[1] )
G_qc.h( q[2] )
G_qc.x( anc[0] )

print('____ Initial State ____')
oq.Wavefunction(G_qc, systems=[3,1,1], show_systems=[True,False,False] )
print('  ')

oq.Grover_Oracle(marked, G_qc, q, anc, n_anc)

print('____ Final State ____')
oq.Wavefunction(G_qc, systems=[3,1,1], show_systems=[True,False,False] )
```

```
____ Initial State ____
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

____ Final State ____
0.35355 |000>    0.35355 |100>    -0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>
```

In this example we can see that Grover_Oracle takes care of all the instructions for us, so long as we prepare the system in the correct initial state:

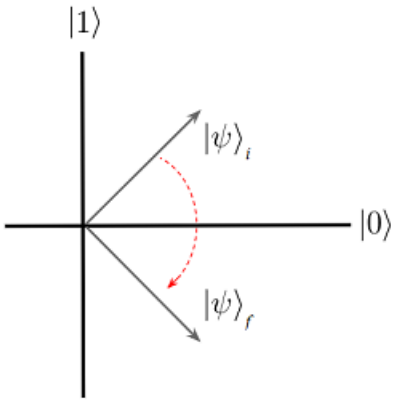
$$|\,00...0\rangle\,|\,1\rangle$$

With the Oracle operator U_w now in hand, we're ready to move on to the second part of Grover's Algorithm, which will require us to revisit the Hadamard transformation one final time.

Reflection About the Average

Like Simon's Algorithm from lesson 5.3, Grover's Algorithm will require multiple runs of our quantum system. The difference here, is that we will not be making measurements after each run. Instead, we will perform multiple 'Grover Iterations', followed by a single measurement at the very end.

In one sentence, we can say that mathematically: "One Grover Iteration is equivalent to a reflection about the average amplitude." (Don't worry, we will make sense of this.) Let's start by talking about a reflection. Geometrically, a reflection involves two components: the object who is being reflected, and the point, line, plane, etc. with which we reflect about. For example, consider the diagram below, which illustrates a reflection of state $|\psi\rangle$ about the state $|0\rangle$:



$$|\psi\rangle_i = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

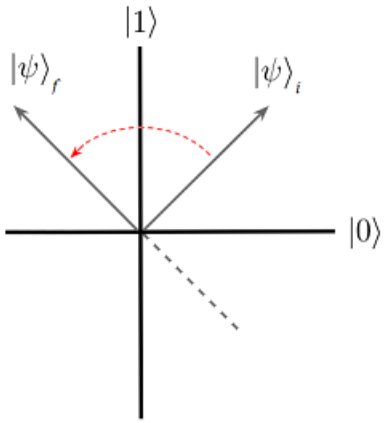
$$|\psi\rangle_f = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

In this example, the object being reflected is the state $|\,\psi\rangle$, and the point of reflection is the state $|0\rangle$. We can see that a 'reflection

about $|0\rangle$' is equivalent to a sign flip on the $|1\rangle$ state. And in general, a reflection about a single state $|\phi\rangle$ leaves a quantum state's $|\phi\rangle$ component unchanged, while flipping the sign on all other components:

$$|\,\psi\rangle_i = \frac{1}{2}\big(\,|\,00\rangle + |\,01\rangle + |\,10\rangle + |\,11\rangle\,\big) \qquad \longrightarrow \qquad \frac{1}{2}\big(\,|\,00\rangle - |\,01\rangle - |\,10\rangle - |\,11\rangle\,\big)$$

However, Grover's Algorithm will require use to perform reflections about a state for arbitrarily large systems, which translates to implementing many of these sign flips. Needless to say, sign flipping every single state besides just one is a bit tedious, and quite costly in terms of gates. Luckily for us, we can achieve the same net effect by taking the reverse route: only flipping the sign on the single state. Consider our first example again, only this time we will flip the sign on the $|0\rangle$ component:



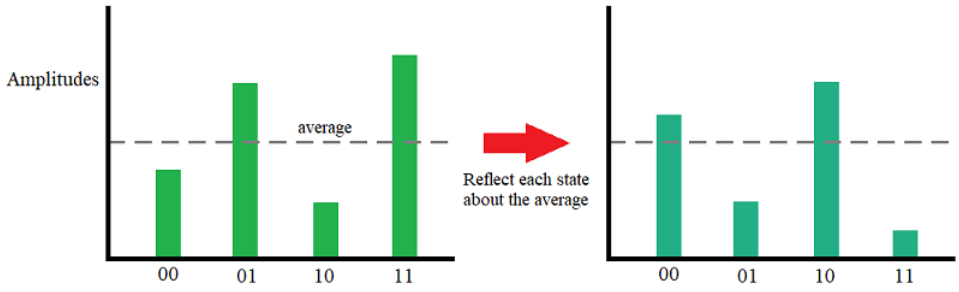
$$|\psi\rangle_i = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

$$|\psi\rangle_f = -\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

In the diagram above, notice how both final states 'align', shown by the dashed line marking where $|\,\psi\rangle_f$ from the first example was. Denoting the final states from the two examples as $|\,\psi\rangle_{1f}$ and $|\,\psi\rangle_{2f}$, we have that $|\,\psi\rangle_{1f} = -\,|\,\psi\rangle_{2f}$. Or more specifically, the two states are parallel, with opposite phase.

The nice thing about this for us, is that a measurement on the system can't tell the difference between $|\psi\rangle_{1f}$ and $|\psi\rangle_{2f}$. Thus, so long as the opposite phase isn't an issue anywhere else in our algorithm, we are free to use either reflection method as we see fit. And for our Grover Algorithm, we are definitely going to use the second method in order to minimize steps.

Now, let's discuss what it means to 'reflect about the average amplitude'. Perhaps the easiest way to understand this initially, is with a diagram:



This diagram illustrates the effect we are going for: we take the average of all the states' amplitudes, and reflect each state's individual amplitudes about that average. We can see that states with amplitudes above the average get reflected below it, and vice versa. In total, the average amplitude for the system is unchanged, even though all of the states have. Mathematically, this is then a unitary operation:

$$\alpha_i \equiv \text{amplitude of each state}$$

$$\alpha_{avg} = \frac{\sum_i^N \alpha_i}{N}$$

$$\sum_i^N \alpha_i^2 = 1 \quad and \quad \sum_i^N (\alpha_{avg} + (\alpha_{avg} - \alpha_i))^2 = 1$$

We won't go through this proof here, but rather provide a simple arithmetic example (which is by no means a proof):

$$2^2 + 3^2 + 5^2 + 9^2 = 119$$

$$\text{reflect around the average: } 4.75$$

$$7.5^2 + 6.5^2 + 4.5^2 + 0.5^2 = 119$$

$\mathbf{U}_s$ - Grover Diffusion Operator

The operator that is going to achieve this reflection about the average will be U_s, often referred to as the Grover Diffusion Operator. We will start by writing out the effect of the operation we want:

$$U\,|\,\psi\rangle \;\; = \;\; |\,\psi\rangle - 2\big(\,|\,\psi\rangle - |\,r\rangle\,\big).$$

where $|\psi\rangle$ is the state of our system, and $|r\rangle$ represents an equal superposition of all states, where each state has an amplitude of α_{avg}:

$$|\,r\rangle \;\; = \;\; \alpha_{avg} \sum_i^N |\,i\rangle$$

This state $|r\rangle$ is most definitely *not* normalized, meaning we can't physically create it, but represents what we want to happen as a result from our operation. Specifically, the operation $|\,\psi\rangle - 2\big(\,|\,\psi\rangle - |\,r\rangle\,\big)$ is written this way in order to understand its two components:

1) take the difference in amplitudes between each state and the average: $\big(\,|\,\psi\rangle - |\,r\rangle\,\big)$

2) double each of these differences, and subtract them from the initial amplitudes: $|\,\psi\rangle - 2\big(\,\cdots\big)$.

For example, suppose we have a system where the amplitude for the state $|01\rangle$ is α_{01} = 0.7, and the average amplitude for the whole system is α_{avg} = 0.45. We want the effect of our operation to do the following:

$$U_s\,|\,01\rangle \;\; \rightarrow \;\; \big(0.7 - 2\,(\,.07 - .45\,)\big)\,|\,01\rangle \;\; = \;\; 0.2\,|\,01\rangle$$

Hopefully this example illustrates what we are going for. We want an operation that uses the difference between each state and the average ($\alpha_i - \alpha_{avg}$), and subtracts double this amount from the initial amplitude. If $\alpha_i - \alpha_{avg}$ is positive, then the final amplitude will be smaller (like state 11 in the diagram above), possibly even negative. Conversely, if $\alpha_i - \alpha_{avg}$ is negative, then the final amplitude will be larger (like state 00 in the diagram), which we shall see happens to our marked state.

Now then, let's see how we can construct this U_s operator. First off, let's do a little rewriting:

$$U_s \,|\, \psi\rangle \quad = \quad |\, \psi\rangle - 2\big(\,|\, \psi\rangle - |\, r\rangle\,\big)$$

$$= \quad 2\,|\, r\rangle - |\, \psi\rangle$$

The second part of this operation should stand out to you, it's just the Identity operator I. Thus, our unitary operator will have the following form:

$$U \quad \equiv \quad \text{something} \; - \; I$$

This *something*, is a operation that when applied to a state $|\, \psi\rangle$, results in the state $2|\, r\rangle$. As mentioned before, $|\, r\rangle$ is a state that is not guaranteed to be normalized, thus we cannot physically create it. However, the combination of $2\,|\, r\rangle - |\, \psi\rangle$ *will* be normalized.

The matrix operation that creates the state $|r\rangle$ is as follows:

$$|\, s\rangle \quad \equiv \quad \frac{1}{\sqrt{N}} \sum_{i}^{N} |\, i\rangle \quad \text{(equal superposition of all states)}$$

$$|\, r\rangle \quad = \quad |\, s\rangle\langle s \,|\, \psi\rangle$$

Thus, we can create $|\, r\rangle$ by using the state $|\, s\rangle$, which is definitely a physically realizable state (Hadamard gates on every qubit). However, $|\, s\rangle\ \langle s\,|$ is not a unitary operator (if it were, it would mean that we could physically create $|\, r\rangle$). Let's quickly show how these two quantities are equal:

1) The inner product $\langle s|\psi\rangle$ results in the following sum of all the amplitudes: $\frac{1}{\sqrt{N}} \sum_i^N \alpha_i$

2) We borrow the remaining $\frac{1}{\sqrt{N}}$ term from the other $|s\rangle$ state, giving us our average amplitude: $\frac{1}{N} \sum_i^N \alpha_i = \alpha_{avg}$.

3) This average amplitude α_{avg} is left multiplying all of the states leftover from $|s\rangle$, leaving us with:

$$|\, s\rangle\langle s \,|\, \psi\rangle \quad = \quad \alpha_{avg}\,|\, 000\rangle \; + \; \alpha_{avg}\,|\, 001\rangle \; + \; ... \quad = \quad |\, r\rangle$$

Thus, we now have a full mathematical description for U_s:

$$U_s = 2\,|\, s\rangle\langle s\,| \; - \; I$$

Implementing $\mathbf{U}_s$ via $\mathbf{H}^N$

Although we just derived a nice compact form for our Grover Diffusion Operator, implementing it into our quantum algorithm is a tad bit more challenging. As we pointed out, the operator as a whole is unitary, but the individual contributions are physically unrealizable. But fear not, there is an impressively simple way of realizing U_s, using a Hadamard Transformation (our favorite).

To start, we must take a slight detour from our algorithm in order to talk about a very important property of the Hadamard Transformation, particularly how it transforms the state of all 0's:

$$H^2\,|\, 00\rangle = \frac{1}{2}\big(\,|\, 00\rangle + |\, 01\rangle + |\, 10\rangle + |\, 11\rangle\,\big) = |\, s\rangle$$

Nothing new here, but we want to take special note of how the Hadamard Transformation is a map between the state of all 0's, and the equal superposition state $|s\rangle$:

$$H^2 \, | \, s \rangle \quad = \quad | \, 00 \rangle \qquad\qquad\qquad H^2 \, | \, 00 \rangle \quad = \quad | \, s \rangle$$

We just saw in our previous discussion that the state $| \, s \rangle$ was exactly what we needed to create U_s, and it's no coincidence that we are seeing it again here via the Hadamard Transformation. This H^N mapping is what is going to allow us to implement the Grover Diffusion Operator.

Although we've seen the Hadamard Transformation at the core of all our previous algorithms, this implementation is a bit different. Previously, we used H^N as a way of simultaneously sampling all possible entries for our blackbox problems. Here, we are using H^N in order to transform our system to a basis where the Grover Diffusion Operator is achievable in one simple operation, and then transforming back. This use of H^N is identical to our use of $\mathrm{X\_Transformation}$, where we transform our system to a different basis in order to use control gates.

Consider this somewhat silly example: Imagine you need to lift a 1 ton brick onto a shelf under Earth's gravity, so you *transform* your problem to the moon where gravity is weaker, do the lift, and then transform back to Earth. That's the spirit of what we're going to achieve with this Hadamard Transformation here in the Grover Algorithm.

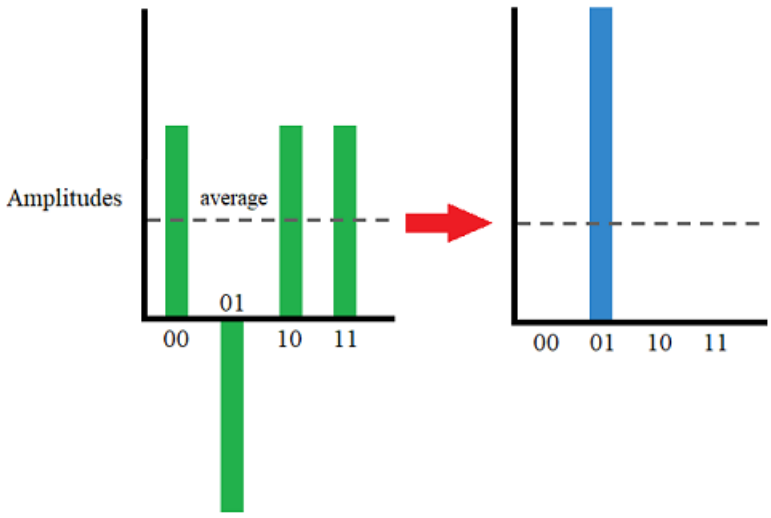
Before any further explanation, it's more powerful to see it in action first:

```
q     = QuantumRegister(2,name='q')
anc   = QuantumRegister(1,name='anc')
n_anc = QuantumRegister(1,name='n_anc')
G_qc  = QuantumCircuit(q,anc,name='qc')
marked = [1,0]

G_qc.h( q[0] )
G_qc.h( q[1] )
G_qc.x( anc[0] )

print('____ Initial State ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

oq.Grover_Oracle(marked, G_qc, q, anc, n_anc)
print('\n____ Grover Oracle: ',marked,' ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

G_qc.h( q[0] )
G_qc.h( q[1] )
oq.Grover_Oracle([0,0], G_qc, q, anc, n_anc)
G_qc.h( q[0] )
G_qc.h( q[1] )

print('\n____ After Grover Diffusion ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])
```

```
____ Initial State ____
0.5 |00>    0.5 |10>    0.5 |01>    0.5 |11>

____ Grover Oracle:  [1, 0]  ____
0.5 |00>    -0.5 |10>    0.5 |01>    0.5 |11>

____ After Grover Diffusion ____
-1.0 |10>
```

And viola! Like magic, we've increased the probability of our marked state, while suppressing all other states. And $N = 2$ (four total states) is a special case, where all non-marked states get suppressed to amplitudes of 0! Feel free to change the marked state in the example above, and see that the Grover Algorithm always makes our marked state dominant.

To see why this happened, let's again draw the amplitudes before and after reflecting about the average:



Because we flipped the sign on our marked state before U_s (via the Oracle Operator), we effectively changed the average amplitude. Then, because the average amplitude is positive and our marked state is negative, the reflection about this new average results in a huge increase in amplitude. Simultaneously, all of our non-marked states have larger amplitudes than the average, so the reflection causes their amplitudes to decrease.

Now, let's run the code above once more, this time observing the state of our system at each point during the Grover Diffusion Operator:

```
marked = [0,1]
q      = QuantumRegister(2,name='q')
anc    = QuantumRegister(1,name='anc')
n_anc  = QuantumRegister(1,name='n_anc')
G_qc   = QuantumCircuit(q,anc,name='qc')


G_qc.h( q[0] )
G_qc.h( q[1] )
G_qc.x( anc[0] )
print('____ Initial State ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

oq.Grover_Oracle(marked, G_qc, q, anc, n_anc)
print('\n____ Grover Oracle: ',marked,' ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

G_qc.h( q[0] )
G_qc.h( q[1] )
print('\n____ H^2 Transformation ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

oq.Grover_Oracle([0,0], G_qc, q, anc, n_anc)
print('\n____ Grover Oracle:  [0, 0]  ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

G_qc.h( q[0] )
G_qc.h( q[1] )
print('\n____ H^2 Transformation ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

```

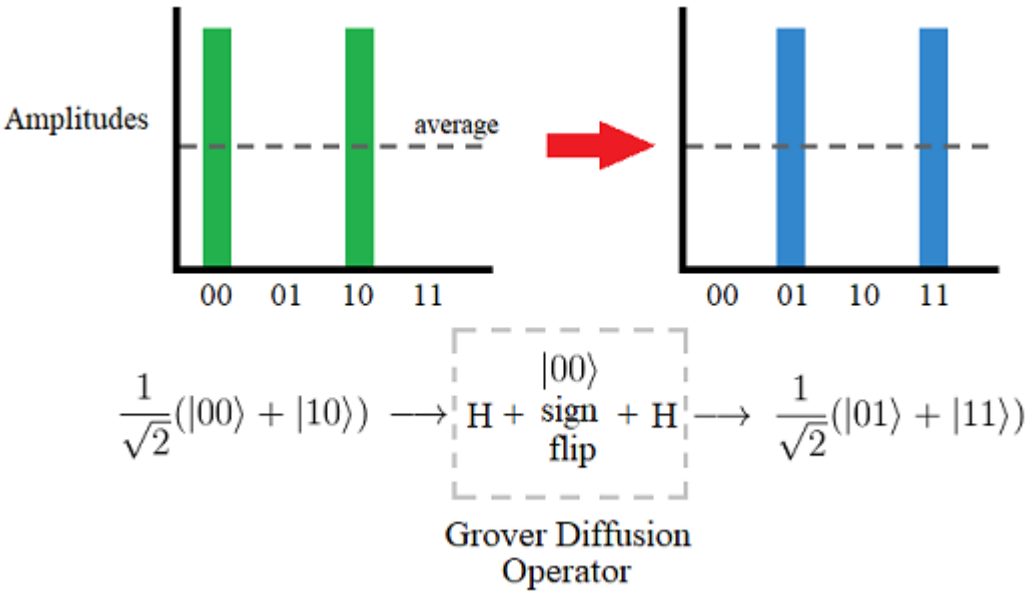
```
____ Initial State ____
0.5 |00>    0.5 |10>    0.5 |01>    0.5 |11>

____ Grover Oracle:  [0, 1]  ____
0.5 |00>    0.5 |10>    -0.5 |01>    0.5 |11>

____ H^2 Transformation ____
0.5 |00>    -0.5 |10>    0.5 |01>    0.5 |11>

____ Grover Oracle:  [0, 0]  ____
-0.5 |00>    -0.5 |10>    0.5 |01>    0.5 |11>

____ H^2 Transformation ____
-1.0 |01>
```

And there it is, the full Grover Algorithm. The key here is that once in the transformed basis (after the first H^N), all we did was *flip the sign on the state* $|00\rangle$, again via our Oracle function. Then, when we transformed back to our original basis (after the second H^N), our main system is in the state $-|01\rangle$. Which means... the sign flip on the state $|00\rangle$ *was* the Grover Diffusion Operator. Well, technically no. It is more accurate to say that as a whole:

$$H^N \; + \; \text{Oracle}(\,|\,00\rangle\,) \; + \; H^N \quad \equiv \quad \text{Grover Diffusion Operator}$$

Remember earlier that we took take special note of the transformation $|\,00...0\rangle \longleftrightarrow H^N \longleftrightarrow |\,s\rangle$. One way of thinking about unitary transformations, is that operations performed in the two bases can look very different, but turn out to be equivalent. Here, we avoid doing some complicated series of operations in our original basis by using a Hadamard Transformation to achieve the same result with ease, and then transform back.

Let's take a look at one more example:




```
q     = QuantumRegister(2,name='q')
anc   = QuantumRegister(1,name='anc')
n_anc = QuantumRegister(1,name='n_anc')
G_qc  = QuantumCircuit(q,anc,name='qc')
marked = [1,0]

G_qc.h( q[0] )
G_qc.iden( q[1] )
G_qc.x( anc[0] )
print('____ Initial State ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])


G_qc.h( q[0] )
G_qc.h( q[1] )
oq.Grover_Oracle([0,0], G_qc, q, anc, n_anc)
G_qc.h( q[1] )
G_qc.h( q[0] )

print('\n____ After Grover ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])
```

```
____ Initial State ____
0.70711 |00>    0.70711 |10>

____ After Grover ____
-0.70711 |01>    -0.70711 |11>
```

In the example above, we start in the state $\frac{1}{\sqrt{2}}\left(\,|\,00\rangle + |\,10\rangle\,\right)$, which has an average amplitude of $\frac{1}{2\sqrt{2}}$. A reflection of each state about this average results in the states $|00\rangle$ and $|10\rangle$ going to zero, and the states $|01\rangle$ and $|11\rangle$ going to $\frac{1}{\sqrt{2}}$.

The code we've written above achieves exactly this, except for one thing. We get the correct final states and amplitudes, but our final states all have negative phases. In fact, you may have already picked up on this in all of our previous examples as well. All of our results are in agreement with the corresponding diagrams, except for their final phases. To understand these results, we must return to the first two diagrams in the "Reflection About an Average" section.

Remember we showed that a reflection about a single state is equivalent to flipping the sign on all other states in the system. For example, a reflection about the state $|10\rangle$:

$$\frac{1}{2}\left(\,|\,00\rangle + |\,01\rangle + |\,10\rangle + |\,11\rangle\,\right) \quad \longrightarrow \quad \frac{1}{2}\left(\,-|\,00\rangle - |\,01\rangle + |\,10\rangle - |\,11\rangle\,\right)$$

But we also showed that we can achieve a parallel state by only flipping the sign on the one state:

$$\frac{1}{2}\big(\,|00\rangle + |01\rangle + |10\rangle + |11\rangle\,\big) \quad \longrightarrow \quad \frac{1}{2}\big(\,|00\rangle + |01\rangle - |10\rangle + |11\rangle\,\big)$$

Now, we must apply this concept to the way we flip the $|00...0\rangle$ state in the transformed basis. In particular, we pointed out that the Hadamard transformation is a map between: $|00...0\rangle \longleftarrow H^N \longrightarrow |s\rangle$. One way to understand this mapping is to say that these states are equivalent, via the H^N transformation. Thus, performing our reflection about $|00...0\rangle$ in the transformed basis is equivalent to a reflection about $|s\rangle$ in our original basis. And, since flipping the sign on $|00...0\rangle$ achieves a state parallel to the reflection, transforming back via H^N will also result in the parallel state to the reflection about the average, explaining where our minus signs are coming from.

Understanding how the mapping of $|00...0\rangle \longleftarrow H^N \longrightarrow |s\rangle$ produces our reflection about the average is really the most important topic in this lesson, and will likely take a little time to fully sink in. As another example, let's suppose we wanted to perform the proper reflection about the average, without picking up the final phase difference on our state. As we showed earlier, to do this we need to to flip the sign on all other states in the system:

```
q     = QuantumRegister(2,name='q')
anc   = QuantumRegister(1,name='anc')
n_anc = QuantumRegister(1,name='n_anc')
G_qc  = QuantumCircuit(q,anc,name='qc')
marked = [1,0]

G_qc.h( q[0] )
G_qc.h( q[1] )
G_qc.x( anc[0] )

print('____ Initial State ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

oq.Grover_Oracle(marked, G_qc, q, anc, n_anc)
print('\n____ Grover Oracle:  |01> ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

G_qc.h( q[0] )
G_qc.h( q[1] )
print('\n____ H^2 Transformation ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

oq.Grover_Oracle([0,1], G_qc, q, anc, n_anc)
oq.Grover_Oracle([1,0], G_qc, q, anc, n_anc)
oq.Grover_Oracle([1,1], G_qc, q, anc, n_anc)
print('\n____ Flipping the sign on:   |01> |10> |11> ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])

G_qc.h( q[0] )
G_qc.h( q[1] )
print('\n____ H^2 Transformation ____')
oq.Wavefunction(G_qc, systems=[2,1], show_systems=[True,False])
```

```
____ Initial State ____
0.5 |00>    0.5 |10>    0.5 |01>    0.5 |11>

____ Grover Oracle:  |01> ____
0.5 |00>    -0.5 |10>    0.5 |01>    0.5 |11>

____ H^2 Transformation ____
0.5 |00>    0.5 |10>    -0.5 |01>    0.5 |11>

____ Flipping the sign on:   |01> |10> |11> ____
0.5 |00>    -0.5 |10>    0.5 |01>    -0.5 |11>

____ H^2 Transformation ____
1.0 |10>
```

In the example above, we have achieved a true reflection about the average, which results in the same final state as predicted by our amplitude diagrams. We can see that the true reflection about $|00\rangle$ comes from flipping the sign on all other states in the system. And when we transform back to our original basis via H^N, sure enough we get the expected result. However, going through all the trouble of flipping extra states just for a final phase isn't really worth the extra cost in quantum steps. Thus, when we go to implement our Grover Algorithm later, we will opt for the more efficient method of only flipping $|00...0\rangle$ in the transformed basis.

A reflection about a single state is much easier to understand at first than a reflection about the average, but we can express them in a similar way. Earlier we showed that a reflection about some state $|X_i\rangle$ will leave $|X_i\rangle$ unchanged, while flipping the sign on all other states. Let's show that this property holds true for our reflection about the average as well, using our 2-qubit example. To do this, we will need to manually separate out the average state from our system (which is $|r\rangle$ from earlier), and flip the sign on everything else:

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

$$|r\rangle = \frac{1}{4}\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big)$$

$$|\psi\rangle_i = |r\rangle + \frac{1}{4}\big(|00\rangle - 3|01\rangle + |10\rangle + |11\rangle\big)$$

$$* \, reflection \;\; about \;\; |r\rangle *$$

$$|\psi\rangle_f = |r\rangle - \frac{1}{4}\big(|00\rangle - 3|01\rangle + |10\rangle + |11\rangle\big)$$

$$|\psi\rangle_f = \big(\tfrac{1}{4} - \tfrac{1}{4}\big)|00\rangle + \big(\tfrac{1}{4} + \tfrac{3}{4}\big)|01\rangle + \big(\tfrac{1}{4} - \tfrac{1}{4}\big)|10\rangle + \big(\tfrac{1}{4} - \tfrac{1}{4}\big)|11\rangle$$

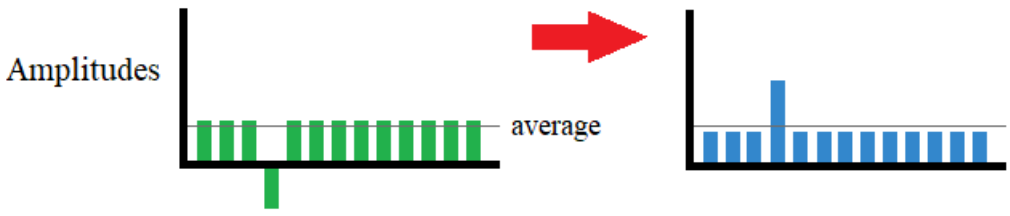
$$|\psi\rangle_f = |01\rangle$$

Here we can see that the average state $|r\rangle$ is unchanged through this reflection, just like our earlier example with the state $|0\rangle$. Although $|r\rangle$ is an unphysical state, hopefully this example helps illuminate what it means to reflect about the average.

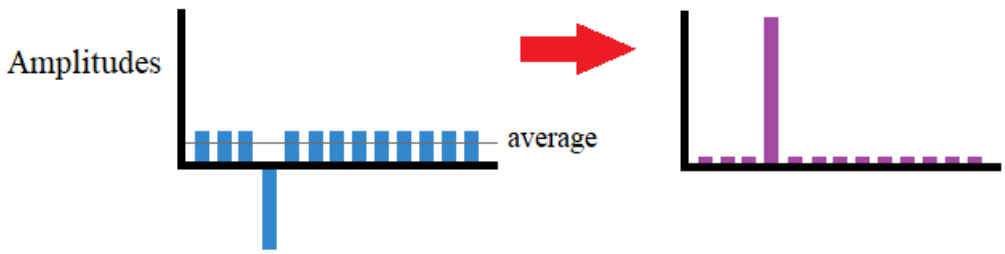
The Full Grover Search

In the coding examples above, we were able to fully pick out our marked state with only one application of our Grover Diffusion Operator. Two qubits is a special case, and in general we will need many more applications in order to make our marked state significantly probable. Specifically, we will need a certain number of Grover Iterations, based on the size of the problem. To remind ourselves, a single Grover Iteration is defined as: 1) Flipping the sign on our marked state via the Oracle Operator 2) Applying the Grover Diffusion Operator.

The reason we will need many Grover Iterations as our problem size gets larger, is because each individual iteration will only boost the probability of our marked state by so much. Consider the diagram below, which shows that a single Grover Iteration is not enough to give our marked state a significant probability:

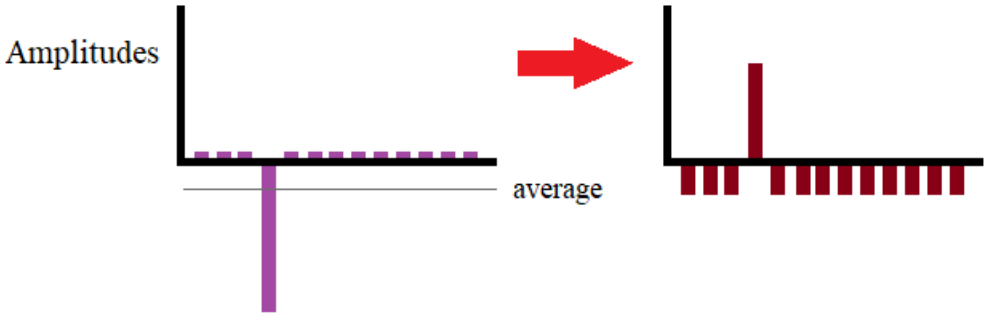


While one step does indeed make our marked state *more* probable, it isn't significant enough to where it is worth making a measurement. And as the size of our problem increases, this first step will be less and less impactful. However, we can simply repeat the process as many times as we need to, until we reach a desirable probability distribution. For example, let's apply one more Grover Iteration to our diagram example:



By applying a second Grover Iteration, we are essentially starting with a state where the amplitude of our marked state is already larger than all the rest. This in turn causes the average amplitude to be smaller, which further decreases all the non-marked states. Thus, after two Grover Iterations, we reach a state where a measurement on the system will find our marked state with a high probability of success.

But, we must point out something very important here. The Grover Iteration is not a magical operation that *always* boosts the amplitude of our marked state. The trick relies on the average amplitude, and at a certain point, the Grover Iteration actually works against us. Let's continue our diagram example with one more iteration to show this negative effect:



Take note of where the average amplitude is located in this third step. Because our marked state's amplitude is so large, it actually weighs the average down below 0 after we flip its sign. It is at this point where the Grover Iteration is working against us. This negative average amplitude causes all of our non-marked states to *increase* in amplitude, which comes at the cost of our marked state.

Even worse yet, try and visualize where the next average amplitude would be after we flip the marked state. Because all of the non-marked states now have negative amplitudes, a fourth Grover Iteration will result in an even lower probability on our marked state, eventually leading to a point where the marked state is the *least* probable state in the system.

Thus, this example has highlighted the final piece to the Grover Algorithm: when to stop. Too many Grover Iterations will make things worse, so we need to never go over the optimal amount. Luckily for us, there is a well known trend that tells us when to stop, for a system of N states:

$$\text{optimal steps:} \quad \approx \quad \frac{\pi}{4}\sqrt{N}$$

There is an 'exact optimal' number of steps for any given N, which may not be exactly $\frac{\pi}{4}\sqrt{N}$. But once N is large enough, applying $\frac{\pi}{4}\sqrt{N}$ Grover Internations will always be nearly optimal. The more problematic cases are for smaller N's, but these aren't really too concerning since using a quantum algorithm for a search on a list of say 4 or 8 entries, is a bit of an overkill. The real merit of this algorithm is for searching on very large lists, where the $\sqrt{N}$ factor is a significant speedup.

Now that we've seen the effect of too many Grover Iternations, let's see it in a coding example. To do this, we will import **Grover_Diffusion** from Our_Qiskit_Functions:

```
q     = QuantumRegister(3,name='q')
anc   = QuantumRegister(1,name='anc')
n_anc = QuantumRegister(1,name='n_anc')
G_qc  = QuantumCircuit(q,anc,n_anc,name='qc')
marked = [1,1,0]

G_qc.h( q[0] )
G_qc.h( q[1] )
G_qc.h( q[2] )
G_qc.x( anc[0] )

print('____ Initial State ____')
oq.Wavefunction(G_qc, systems=[3,1,1], show_systems=[True,False,False])

iterations = 3

for i in np.arange(iterations):
    oq.Grover_Oracle(marked, G_qc, q, anc, n_anc)
    oq.Grover_Diffusion(marked, G_qc, q, anc, n_anc)
    print('\n____ ',int(i+1),' Grover Iteration ____')
    oq.Wavefunction(G_qc, systems=[3,1,1], show_systems=[True,False,False])
```

```
____ Initial State ____
0.35355 |000>    0.35355 |100>    0.35355 |010>    0.35355 |110>    0.35355 |001>    0.35355 |101>    0.35355 |011>    0.35355
|111>

____  1  Grover Iteration ____
-0.17678 |000>    -0.17678 |100>    -0.17678 |010>    -0.88388 |110>    -0.17678 |001>    -0.17678 |101>    -0.17678 |011>    -
0.17678 |111>

____  2  Grover Iteration ____
-0.08839 |000>    -0.08839 |100>    -0.08839 |010>    0.97227 |110>    -0.08839 |001>    -0.08839 |101>    -0.08839 |011>    -
0.08839 |111>

____  3  Grover Iteration ____
0.30936 |000>    0.30936 |100>    0.30936 |010>    -0.57452 |110>    0.30936 |001>    0.30936 |101>    0.30936 |011>    0.30936
|111>
```

Take a look at the amplitudes displayed above. After 1 Grover Iteration, we have a 78% percent chance of measuring our marked state. After the second Grover Iteration, this probability jumps to over 94%! But, if we apply a third iteration, our probability of measuring the marked state plummets to a measly 33% (but is still the highest single state). If we had carried out a fourth iteration, we would find our marked state with a 1% probability, the complete opposite of what we set out to do!

The Grover Algorithm is cyclic in the way it increases / decreases the probability of our marked state. $\frac{\pi}{4}\sqrt{N}$ represents the first peak, which corresponds to half of the cycle. If we perform $\frac{\pi}{2}\sqrt{N}$ iterations, we will find the point where our marked state is *least* probable. But from there, the probabilities will begin to increase again, peaking at $\frac{3\pi}{4}\sqrt{N}$, and so on. But for the purpose of our searching algorithm, we will only ever aim for the first $\frac{\pi}{4}\sqrt{N}$ peak.

The cell of code below is our complete Grover Algorithm, combining all of the steps we've covered thus far into the function **Grover**. Change Q to be any number of qubits you like, corresponding to a system of the size 2^Q, and pick a corresponding marked state of length Q:

```
Q = 4
marked = [0,1,1,0]
#-----------------------------
#-----------------------------
G_qc,q,an1,an2,c = oq.Grover(Q, marked)

oq.Wavefunction(G_qc, systems=[Q,1,Q-2], show_systems=[True,False,False], column=True)
print(' ')
G_qc.measure(q,c)
print('\n ___ Measurement Results ___')
oq.Measurement(G_qc, shots=100)
```

```
0.05078 |0000>
0.05078 |1000>
0.05078 |0100>
0.05078 |1100>
0.05078 |0010>
0.05078 |1010>
-0.98047 |0110>
0.05078 |1110>
0.05078 |0001>
0.05078 |1001>
0.05078 |0101>
0.05078 |1101>
0.05078 |0011>
0.05078 |1011>
0.05078 |0111>
0.05078 |1111>

 ___ Measurement Results ___
98|0110>    1|1110>    1|0011>
```

This concludes lesson 5.4, and our series of introductory quantum algorithms! We have now seen four lessons worth of Hadamard Transformations, and the various problems it can solve. We saved this algorithm for last because of the way in which we used H^N, which is very analogous to the next lesson to come. In general, the use of unitary transformations are at the core of a lot of the most successful quantum algorithms to date.

Lesson 6 - Quantum Fourier Transformation

In this final tutorial, we will cover an important transformation used at the heart of many successful quantum algorithms: the Quantum Fourier Transformation (QFT). Much like how the Hadamard Transformation was the basis for all of the algorithms studied in lessons 5.1 - 5.4, the QFT plays a major role in algorithms like Shor's, Quantum Phase Estimation, Variational Quantum Eigensolver, and many more. At their core, the two transformations share a lot of similarities, both in their effect and usage in quantum algorithms.

Original publication of the algorithm: [12]

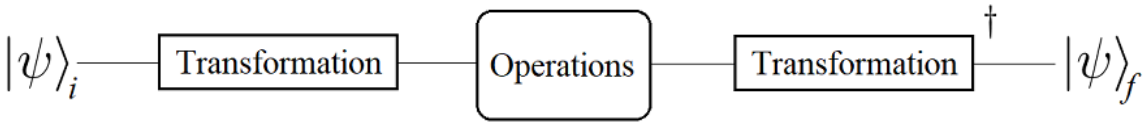
In order to make sure that all cells of code run properly throughout this lesson, please run the following cell of code below:

```
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
import Our_Qiskit_Functions as oq
import math as m
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]
```

Importance of Unitary Transformations

If we think back to lessons 5.1 - 5.4, we should ask ourselves: what was it about the Hadamard Transformation that allowed all of those algorithms to be successful. For the blackbox problems, we would say that it allowed us to work with all possible states at once, thus out performing classical algorithms that are forced to check only one input at a time. And for the Grover Algorithm, the vital role of the Hadamard Transformation was that it allowed us to perform a 'reflection about the average' by transforming to a different basis.

The success of any transformation can always be traced to *the way* it maps states. In particular, by studying the way a certain transformation maps individual states, as well as how it maps combinations of states, we can learn about what types of advantages it can achieved. Or in other words, a transformation provide us with two 'domains' in which to work, where we can use the advantages of each to solve complex problems. Visually, moving to a transformed basis in order to achieve some desired effect looks like:



The operations we perform 'inside' the transformation are dependent on the algorithm, and what type of problem we are trying to solve. Sometimes, we need to perform transformations *within* transformations in order to get a certain effect. For example, the Grover Diffusion Operator from lesson 5.4 is essentially and X Transformation inside of a H Transformation, in order to flip the sign on the state $|00...0\rangle$.

Another important property of transformations are the operators that map back and forth between the bases. For the Hadamard Transformation, the same operator is used for both transformations, but in general this is not always the case. In the figure above, this is represented by the Transformation and $\text{Transformation}^\dagger$ operations. As an example, consider the role of orthogonality when using a Hadamard Transformation:

$$\langle 01 \mid 10 \rangle = 0$$

$$H^2 \mid 01\rangle \quad = \quad \frac{1}{2}\big(\mid 00\rangle - \mid 01\rangle + \mid 10\rangle - \mid 11\rangle \big) \qquad\qquad H^2 \mid 10\rangle \quad = \quad \frac{1}{2}\big(\mid 00\rangle + \mid 01\rangle - \mid 10\rangle - \mid 11\rangle \big)$$

$$\frac{1}{4}\big(\langle\, 00 \mid - \langle 01 \mid + \langle 10 \mid \, - \langle 11 \mid \big) \big(\mid 00\rangle + \mid 01\rangle - \mid 10\rangle - \mid 11\rangle \big) \quad = \quad \frac{1}{4}\big(1 - 1 - 1 + 1\big) \quad = \quad 0$$

Or written in a more elegant way:

$$\begin{aligned} \langle 01 \mid H^{\dagger 2}\, H^2 \mid 10\rangle \quad &= \quad \langle 01 \mid \, (H^\dagger H) \otimes (H^\dagger H) \, \mid 10\rangle \\ &= \quad \langle 01 \mid \, (H^\dagger H) \otimes (H^\dagger H) \, \mid 10\rangle \\ &= \quad \langle 01 \mid \, I \otimes I \, \mid 10\rangle \\ &= \quad \langle 01 \mid 10\rangle \quad = \quad 0 \end{aligned}$$

What's important to note in the second example is the property $H^\dagger H = 1$. This is true of all unitary operators, *but*, not all unitary operators are their own complex conjugate like H^N. That is to say, the Hadamard transformation is special in that $H = H^\dagger$, a property known as being Hermitian, which means that we can apply the same operation to transform back and forth between bases. And since an H^N transformation is essentially N individual 1-qubit Hadamard Transformations in parallel: $H \otimes H \otimes H \cdots$, the net result is that ${H^N}^\dagger = H^N$.

If we have an operation that acts on N qubits, and can be decomposed into N individual Hermitian operators: $O_0 \otimes O_1 \otimes O_2 \cdots$, then the total operator is Hermitian as well. For example:

```
q  = QuantumRegister(4,name='q')
H_qc = QuantumCircuit(q,name='qc')

H_qc.x( q[0] )
H_qc.iden( q[1] )
H_qc.x( q[2] )

print('__ Initial State __')
oq.Wavefunction(H_qc)

H_qc.h( q[0] )
H_qc.x( q[1] )
H_qc.y( q[2] )
H_qc.z( q[3] )

print('\n__ Opertor: H + X + Y + Z __')
oq.Wavefunction(H_qc)

H_qc.h( q[0] )
H_qc.x( q[1] )
H_qc.y( q[2] )
H_qc.z( q[3] )

print('\n__ Opertor: H + X + Y + Z __')
oq.Wavefunction(H_qc)
```

```
__ Initial State __
1.0 |1010>

__ Opertor: H + X + Y + Z __
-0.70711j |0100>    0.70711j |1100>

__ Opertor: H + X + Y + Z __
1.0 |1010>
```

In this example, we make up a 4-qubit operator, which can be decomposed as: $H_0 \otimes X_1 \otimes Y_2 \otimes Z_3$. Each of the individual components is Hermitian, therefore the total operator is Hermitian as well. This is demonstrated by the fact that two applications of this operator return us back to our original state.

However, as we pointed out earlier, not all multi-qubit operations are their own complex conjugate. For example, we've already seen such an operator in lesson 4 when we showed how to construct an N-NOT gate. This is because the N-NOT operation uses a specific ordering of gates. And in linear algebra, the order of operators is not always interchangeable. For example, consider a single qubit operator that can be decomposed as: $X_0 \otimes Z_0$

```
q  = QuantumRegister(1,name='q')
XZ_qc = QuantumCircuit(q,name='qc')

XZ_qc.iden( q[0] )

print('__ Initial State __')
oq.Wavefunction(XZ_qc)

XZ_qc.x( q[0] )
XZ_qc.z( q[0] )

print('\n__ Opertor: XZ __')
oq.Wavefunction(XZ_qc)

XZ_qc.x( q[0] )
XZ_qc.z( q[0] )

print('\n__ Opertor: XZ __')
oq.Wavefunction(XZ_qc)
```

```
__ Initial State __
1.0 |0>

__ Opertor: XZ __
-1.0 |1>

__ Opertor: XZ __
-1.0 |0>
```

As we can see, applying this operator twice does not return us to our original state. Thus, $X_0 \otimes Z_0$ is not a Hermitian operator, *even though* it is made up of Hermitian components. If we define an operation that contains several gates that must act on the

same qubit in a specific order, then chances are it won't be Hermitian. So then, if our algorithm requires us to use such an operator as a transformation, then we will need to find a *different* operator if we want to transform back. Specifically, we will need the complex conjugate of the operator.

Luckily, if we know how to decompose an operation like the one in our example above, then finding the complex conjugate is as simple as reversing the order (with one caveat that we will see later):

```
q  = QuantumRegister(1,name='q')
XZ_qc = QuantumCircuit(q,name='qc')


XZ_qc.iden( q[0] )

print('__ Initial State __')
oq.Wavefunction(XZ_qc)
print(' ')

XZ_qc.x( q[0] )
XZ_qc.z( q[0] )

print('__ Opertor: XZ __')
oq.Wavefunction(XZ_qc)
print(' ')

XZ_qc.z( q[0] )
XZ_qc.x( q[0] )

print('__ Opertor: ZX __')
oq.Wavefunction(XZ_qc)
```

```
__ Initial State __
1.0 |0>

__ Opertor: XZ __
-1.0 |1>

__ Opertor: ZX __
1.0 |0>
```

As you may have guess, the reason we've gone out of our way to discuss non-Hermitian operations is because the transformation we will be studying in this lesson is exactly that. The Quantum Fourier Transformation (QFT), which we will be using as the core of the next couple lessons, is an example where QFT and $QFT^{\dagger}$ are different operations. As we shall see, the relation between these two transformations is very straightforward, and is analogous to the way we constructed our $\mathrm{n\_NOT}$ gate in lesson 4.

Discrete Fourier Transformation

The QFT is essentially the Discrete Fourier Transformation (DFT), but applied to the states of our quantum system. Thus, we will begin with a quick review the DFT. Formally written, the Discrete Fourier Transformation looks like this:

$$X = \{ x_0, \ \ldots, x_k, \ \ldots, x_{N-1} \}$$

$$\tilde{X} = \{ \tilde{x}_0, \ \ldots, \tilde{x}_k, \ \ldots, \tilde{x}_{N-1} \}$$

$$\tilde{x}_k \ = \ \sum_{j=0}^{N-1} x_j \cdot e^{2\pi i \frac{k \cdot j}{N}}$$

Where the DFT maps all of the numbers in X to $\tilde{X}$, and $e^{\pm i\theta} \ = \ cos(\theta) \ \pm \ i \, sin(\theta)$.

$$X \quad - DFT \rightarrow \quad \tilde{X}$$

The DFT is defined by the sum above, which shows that each output value $\tilde{x}_k$, receives a contribution from each input value x_k. Specifically, each input value is multiplied by a complex number of the form $e^{i\theta}$, which are then all summed together. The value of each θ is determined by the multiplication of $k \cdot j$. Let's see a quick example:

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

$$\begin{aligned} \tilde{x}_1 &= \sum_{j=0}^{3} x_j \cdot e^{2i\pi \frac{1 \cdot j}{4}} \\ &= 1 \cdot e^0 - 1 \cdot e^{\frac{i\pi}{2}} - 1 \cdot e^{i\pi} + 1 \cdot e^{\frac{3i\pi}{2}} \\ &= 1 - i + 1 - i \\ &= 2 - 2i \end{aligned}$$

and the full transformation:

$$X = \begin{bmatrix} 1 & -1 & -1 & 1 \end{bmatrix} \qquad \longrightarrow \qquad \tilde{X} = \begin{bmatrix} 0 & 2-2i & 0 & 2+2i \end{bmatrix}$$

These $e^{i\theta}$ terms are derived from the concept of taking the roots of -1, which we will not cover here. I encourage you to work through all of the example above, as you will want to really develop a good feel for these transformations if you plan to continue onto the lesson 7 algorithms. For our goal of understanding the QFT, we will only be taking from the DFT what we need.

In particular, let's see what this DFT looks like in a matrix representation:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2-2i \\ 0 \\ 2+2i \end{bmatrix}$$

where the values in the matrix above can all be expressed in terms of:

$$\omega \equiv e^{\frac{2i\pi}{N}} \qquad \text{DFT matrix:} \quad F_4 = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}$$

The powers on all of the ω's come from the the products of $k \cdot j$, and N refers to the total number of values being transformed ($N = 4$ for our example):

$$\begin{aligned} k \cdot j: \quad & 0 \cdot 1 \quad & 1 \cdot 2 \quad & 3 \cdot 1 \\ \omega^{k \cdot j}: \quad & \omega^0 \quad & \omega^2 \quad & \omega^3 \\ = \quad & e^0 \quad & e^{i\pi} \quad & e^{\frac{3i\pi}{2}} \\ = \quad & 1 \quad & -1 \quad & -i \end{aligned}$$

We could go on and on about the things one can do with DFT, but we will end our discussion here. I encourage you to read other references about the Discrete Fourier Transformation, and the various things it can be used for. Doing so will help you get a deeper understanding for why the QFT is so powerful.

Quantum Fourier Transformation

We now have a formal definition for the Discrete Fourier Transformation, so how do we make it quantum? Well, we've already shown how to represent the DFT as a matrix, so our task is to implement it as an operator. Since we are dealing with quantum systems, we will naturally gravitate towards transformations of the size 2^N.

Let's use a 2-qubit example so illustrate how the DFT will look on a quantum system:

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

$$F_4\,|\psi\rangle \;=\; \frac{1}{2}\big(\;(1-i)\,|10\rangle \;+\; (1+i)\,|11\rangle\;\big)$$

This example is the quantum version of our $X \;\rightarrow\; \tilde{X}$ transformation from earlier. Our initial state corresponds to X, and our final state is $\tilde{X}$. Verifying that is operation is indeed unitary is simple enough, which means that F_4 is a legitimate quantum operation. And in general, any DFT matrix is guaranteed to be unitary.

For clarity, the vector representing the state of our system is in the following order:

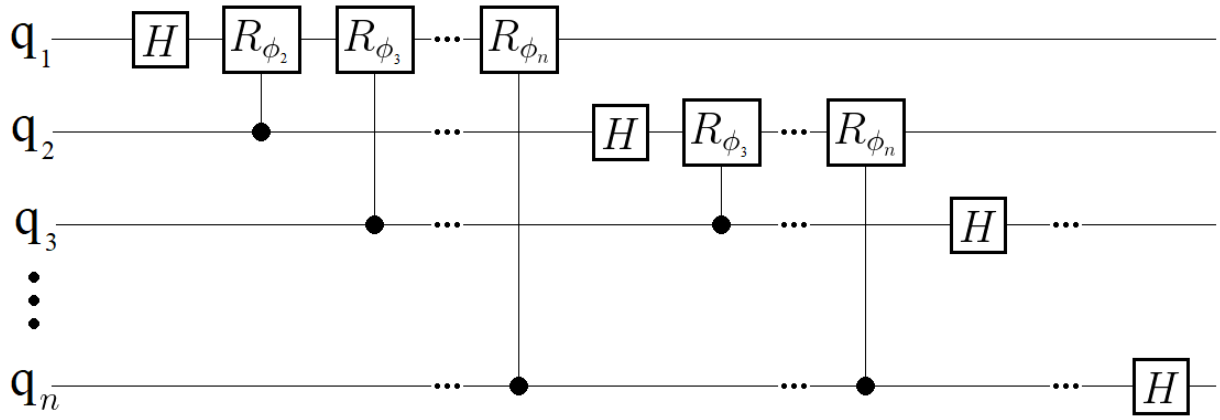
$$\begin{bmatrix} |00\rangle \\ |10\rangle \\ |01\rangle \\ |11\rangle \end{bmatrix}$$

Implementing a QFT

At this point, we have the structure for generating our QFT matrices, and the corresponding vector representations of our states. From the mathematical perspective, we have the full picture for the QFT. However, as we've already seen with past algorithms, simply writing it down doesn't do it justice. And, if we want to actually run a QFT in our quantum algorithms, we need a way of translating the mathematical picture into gates.

The way in which we are going to achieve our QFT's is quite elegant, and by no means obvious at first. As it turns out, the only gates we need in order to construct a 2^N QFT are H and R_ϕ: our trusty Hadamard gate along with some control-phase gates. Even better yet, we will not require any additional ancilla qubits.

Below is the general template for how to construct a QFT circuit on N qubits, acting on a 2^N space of states:



where

$$\phi_m = e^{\frac{2i\pi}{2^m}}$$

At first glance, this circuit may look a bit complex, but it is actually quite straightforward. Each qubit in the system undergoes the same process: a Hadamard gate followed by a series of control-phase gates. The number of R_ϕ gates that a qubit experiences is determined by its index. The first qubit in the system receives $N-1$, while the last qubit doesn't receive any. In addition, the phase of each R_ϕ is determined by which qubit acts as the control, as shown by the equation above (note that we typically start our first qubit as 0, but here we are starting with 1).

Now, it isn't immediately obvious why the circuit above works, but we're going to first test it out with a coding example:

```
q  = QuantumRegister(2,name='q')
F_qc = QuantumCircuit(q,name='qc')

F_qc.x( q[0] )
F_qc.h( q[0] )
F_qc.x( q[1] )
F_qc.h( q[1] )

print('__ Initial State __')
oq.Wavefunction(F_qc)

F_qc.h( q[0] )
F_qc.cu1( m.pi/2,q[1],q[0] )
F_qc.h( q[1] )

print('\n__ After QFT __')
oq.Wavefunction(F_qc)
```

```
__ Initial State __
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>

__ After QFT __
0.5-0.5j |10>    0.5+0.5j |11>
```

Try and match the pattern in the template above, with the steps we've implemented in this code example.:

$$1) \quad H_0 \qquad\qquad 2) \quad R_{\frac{\pi}{2}\,10} \qquad\qquad 2) H_1$$

where the 10 subscript on the control-phase gate represents qubit 1 is the control, and qubit 0 is the target. Confirm for yourself that these are indeed the steps written into our coding example, and that they match the QFT template.

Next, we will do one more example, this time with 3 qubits:

$$QFT\,|\,001\rangle \;=\; \tfrac{1}{4}\Big(\,\sqrt{2}\,|\,000\rangle \;-\; \sqrt{2}\,|\,001\rangle \;+\; (1+i)\,|\,010\rangle \;-\; (1+i)\,|\,011\rangle$$
$$+\,(1+i)\,|\,100\rangle \;-\; (1+i)\,|\,101\rangle \;+\; \sqrt{2}i\,|\,110\rangle \;-\; \sqrt{2}i\,|\,111\rangle\,\Big)$$

$*$ For an extra exercise, try deriving this result by writing out the full 8×8 matrix for a 3-qubit transformation via our definitions earlier.

Now to implement this transformation in code:

```
q  = QuantumRegister(3,name='q')
F_qc = QuantumCircuit(q,name='qc')

F_qc.iden( q[0] )
F_qc.iden( q[1] )
F_qc.x( q[2] )

print('__ Initial State __')
oq.Wavefunction(F_qc)

#-------------------------   qubit 0
F_qc.h( q[0] )
F_qc.cu1( m.pi/2,q[1],q[0] )
F_qc.cu1( m.pi/4,q[2],q[0] )
#-------------------------   qubit 1
F_qc.h( q[1] )
F_qc.cu1( m.pi/4,q[2],q[1] )
#-------------------------   qubit 2
F_qc.h( q[2] )

print('__ After QFT __')
oq.Wavefunction(F_qc)
```

```
__ Initial State __
1.0 |001>
__ After QFT __
0.35355 |000>    0.25+0.25j |100>    0.25+0.25j |010>    0.35355j |110>    -0.35355 |001>    -0.25-0.25j |101>    -0.25-0.25j |
011>    -0.35355j |111>
```

In this example, we've broken up the QFT instructions into three sections, where each section incorporates all of the operations being applied to one of the three qubits. Just like in the QFT template shown above, the number of operations decreases by 1 per qubit, where the last qubit only receives a single H.

Ultimately, writing out all the steps for a QFT is a tedious task, so just like the $\mathrm{n\_NOT}$ function, we will use the function **QFT** from $\mathrm{Our\_Qiskit\_Functions}$ instead:

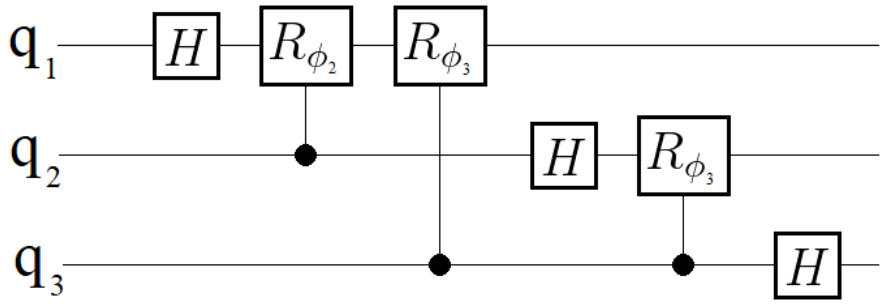
```
from Our_Qiskit_Functions import QFT

q  = QuantumRegister(3,name='q')
F_qc = QuantumCircuit(q,name='qc')

F_qc.iden( q[0] )
F_qc.iden( q[1] )
F_qc.x( q[2] )

print('__ Initial State __')
oq.Wavefunction(F_qc)
print(' ')

oq.QFT(F_qc,q,3)

print('__ After QFT __')
oq.Wavefunction(F_qc)
```

```
__ Initial State __
1.0 |001>

__ After QFT __
0.35355 |000>    0.25+0.25j |100>    0.25+0.25j |010>    0.35355j |110>    -0.35355 |001>    -0.25-0.25j |101>    -0.25-0.25j |
011>    -0.35355j |111>
```

Why the QFT Circuit Works

Now that we have shown that we *can* implement a QFT, let's talk about why it works. If you followed along the derivation of the DFT matrix at the beginning of this lesson, then the way we are achieving these operations may seem surprisingly simple. For example, take a look at all of the complexity happening in the 2-qubit QFT matrix from earlier, and then note that we achieve all of this with only 2 H's and one R_ϕ.

To make sense of how these gates are achieving all the desired phases, we will work through a 3-qubit example:



In particular, let's start with q_1, and see what its final state will look like at the end. We want to be general here, so we will say that our qubit starts off in the state $|q_1\rangle$, where q_1 is either a 0 or 1. Following along with all of the operations that q_1 receives:

$$H: \ \frac{1}{\sqrt{2}}\left(\,|\,0\rangle \ + \ e^{q_1 \cdot i\pi}\,|\,1\rangle\,\right)$$

$$R_{\phi_2}: \ \frac{1}{\sqrt{2}}\left(\,|\,0\rangle \ + \ e^{q_1 \cdot i\pi} \cdot e^{q_2 \cdot \frac{i\pi}{2}}\,|\,1\rangle\,\right)$$

$$R_{\phi_3}: \ \frac{1}{\sqrt{2}}\left(\,|\,0\rangle \ + \ e^{q_1 \cdot i\pi} \cdot e^{q_2 \cdot \frac{i\pi}{2}} \cdot e^{q_3 \cdot \frac{i\pi}{4}}\,|\,1\rangle\,\right)$$

First, take a look at how we've chosen to write the effect of our Hadamard gate on q_1: $\frac{1}{\sqrt{2}}\left(\,|\,0\rangle \ + \ e^{q_1 \cdot i\pi}\,|\,1\rangle\,\right)$. Typically we would write this with something like $(-1)^{q_1}$, where the state of q_1 determines whether or not the the Hadamard gate results in a positive or negative $|1\rangle$ state. Here however, we've chosen to express -1 as $e^{i\pi}$, in order to be consistent with the other gate effects.

Next are the control-phase gates, which produce a similar effect to that of the Hadamard gate at first glance, but have an important difference. Remember that control-phase gates only apply an effect when both the target and control qubits are in the $|1\rangle$ state. This is why a H gate is necessary before any of the R_ϕ's, to ensure that q_1 is in a superposition state of both $|0\rangle$ and $|1\rangle$. Then, effect of the R_ϕ gate applies an additional phase to the $|1\rangle$ component of q_1.

However, because this is a control gate, and we must take into account that q_2 and q_3 may not be in the $|1\rangle$ state, so there is an extra term multiplying each of the added phases, for example: $e^{q_2 \cdot \frac{i\pi}{2}}$. We can understand this extra term as our condition that q_2 is in the $|1\rangle$ state. If it is, then 1 times the rest of the power will leave it unchanged. But if q_2 is in the $|0\rangle$ state, then we will get e^0, which is just a multiplication of q_1 by 1, meaning that no phase is applied to q_1's $|1\rangle$ component.

This pattern continues for each qubit, all the way down to the last. Each qubit receives a number of phases added to their $|1\rangle$ component, which will then all be multiplied together in the final state:

$$|\,\psi\rangle_f \ = \ \left(\,q_{1f}\,\right) \otimes \left(\,q_{2f}\,\right) \otimes \left(\,q_{3f}\,\right)$$

$$= \ \frac{1}{2\sqrt{2}}\left(\,|\,0\rangle \ + \ e^{q_1 \cdot i\pi}\,|\,1\rangle\,\right) \otimes \left(\,|\,0\rangle \ + \ e^{q_1 \cdot i\pi} \cdot e^{q_2 \cdot \frac{i\pi}{2}}\,|\,1\rangle\,\right) \otimes \left(\,|\,0\rangle \ + \ e^{q_1 \cdot i\pi} \cdot e^{q_2 \cdot \frac{i\pi}{2}} \cdot e^{q_3 \cdot \frac{i\pi}{4}}\,|\,1\rangle\,\right)$$

This is how we are able to achieve all of the various phases shown in the QFT matrices from earlier. Multiplying the states and phases of each qubit together results in our normal 2^N states, where each state will be a unique combination of phases, contributed by the $|0\rangle$'s and $|1\rangle$'s that make up the state. The math is still a little cumbersome, even for just three qubits, but hopefully this illustrates the idea behind why we are able to achieve a QFT with this quantum circuit.
As a final optional exercise, I would encourage you to prove for yourself that mathematically our circuit representation is equal to our matrix representation:

$$\text{show that} \qquad H_1 \, R_\phi \, H_0 \, |\, \psi\rangle \quad = \quad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & i & -i \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -i & i \end{bmatrix} \begin{bmatrix} |\, 00\rangle \\ |\, 01\rangle \\ |\, 10\rangle \\ |\, 11\rangle \end{bmatrix} \qquad \phi = \frac{\pi}{2}$$

$$\text{hint: don't forget to represent } H_0 \text{ and } H_1 \text{ as 4x4 matrices!} \quad \longrightarrow \quad H_0 \equiv H_0 \otimes I_1$$

Inverse QFT

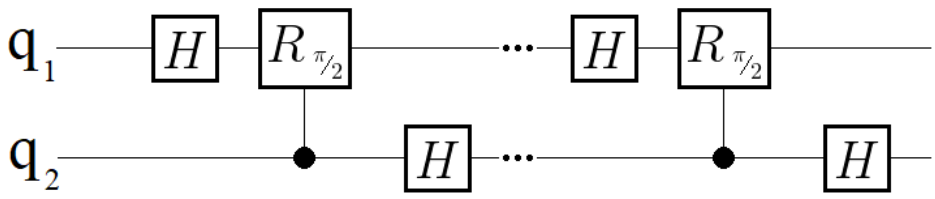
Now that we have a way of transforming our system via a QFT, and hopefully a better intuition as to why it works, next we need to be able to transform back. As we mentioned earlier, the power of using transformations in quantum algorithms relies on being able to transform back and forth between bases. And as we've also mentioned already, our QFT transformation is not Hermitian, so the same construction of gates will not transform us back.

Just to verify this, let's try to use our QFT function twice:

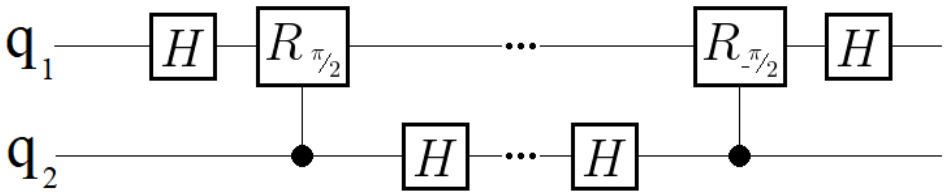
```
q   = QuantumRegister(2,name='q')
F_qc = QuantumCircuit(q,name='qc')

F_qc.x( q[0] )
F_qc.h( q[0] )
F_qc.x( q[1] )
F_qc.h( q[1] )

print('__ Initial State __')
oq.Wavefunction(F_qc)

oq.QFT(F_qc,q,2)

print('\n__ First QFT __')
oq.Wavefunction(F_qc)

oq.QFT(F_qc,q,2)

print('\n__ Second QFT __')
oq.Wavefunction(F_qc)
```

```
__ Initial State __
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>

__ First QFT __
0.5-0.5j |10>    0.5+0.5j |11>

__ Second QFT __
0.5 |00>    -0.5j |01>    -0.5+0.5j |11>
```

Sure enough, we do not return to our original state. From our quantum computing perspective, we can understand why the QFT doesn't transform us back to our original state if we look at two QFTs in a row:



What should jump out at you is the apparent lack of symmetry here. Recall our example earlier of the gate XZ, and that the correct inverse transformation was to change the order: ZX. Here, if we want to implement the inverse of our QFT, we will need to invoke the same strategy of reversing the order of all the gates. In essence, imagine placing a mirror after our QFT, and the reflection will be our inverse QFT, with one slight change:



The slight change here is that our second R_ϕ has the opposite sign of our first. Conceptually, this should make sense: if our original transformation applies a phase θ, then our inverse should apply the opposite phase, -θ. As we pointed out earlier, the inverse of a transformation needs to be the complex conjugate of the original, which is why we need negative phases on all of the θ's. All together, our inverse QFT must be the *exact* reverse ordering our QFT, with all opposite phases on the R_ϕ gates:

```
q  = QuantumRegister(2,name='q')
F_qc = QuantumCircuit(q,name='qc')

F_qc.x( q[0] )
F_qc.h( q[0] )
F_qc.x( q[1] )
F_qc.h( q[1] )

print('__ Initial State __')
oq.Wavefunction(F_qc)
print(' ')

F_qc.h( q[0] )
F_qc.cu1( m.pi/2,q[1],q[0] )
F_qc.h( q[1] )

print('__ QFT __')
oq.Wavefunction(F_qc)
print('  ')

F_qc.h( q[1] )
F_qc.cu1( -m.pi/2,q[1],q[0] )
F_qc.h( q[0] )

print('__ Inverse QFT __')
oq.Wavefunction(F_qc)
```

```
__ Initial State __
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>

__ QFT __
0.5-0.5j |10>    0.5+0.5j |11>

__ Inverse QFT __
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>
```

Sure enough, we recover our original state, which means that we performed the correct inverse transformation. And like our QFT function, we can use **QFT_dgr** from Our_Qiskit_Functions to implement our inverse QFT:

```
q  = QuantumRegister(2,name='q')
F_qc = QuantumCircuit(q,name='qc')

F_qc.x( q[0] )
F_qc.h( q[0] )
F_qc.x( q[1] )
F_qc.h( q[1] )

print('__ Initial State __')
oq.Wavefunction(F_qc)
print(' ')

oq.QFT(F_qc,q,2)

print('__ QFT __')
oq.Wavefunction(F_qc)
print('  ')

oq.QFT_dgr(F_qc,q,2)

print('__ Inverse QFT __')
oq.Wavefunction(F_qc)
```

```
__ Initial State __
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>

__ QFT __
0.5-0.5j |10>    0.5+0.5j |11>

__ Inverse QFT __
0.5 |00>    -0.5 |10>    -0.5 |01>    0.5 |11>
```

Now that we have QFT and QFT_dgr, we are finished covering the basics of the Quantum Fourier Transformation. In the next couple lessons, we will be using these QFT's as the basis for some very important algorithms. If you would like to proceed to those lessons now, this is a sufficient concluding spot in the tutorial. The next and final section is an aside about the QFT, comparing some of its properties to the Hadamard transformation.

Aside: Comparing QFT and H Transformations

Now that we have built up an understanding of how to use a QFT, let's discuss its similarities with the Hadamard Transformation. First off, if you remove all of the R_ϕ gates from the QFT template, you're left with just a Hadamard Transformation. And in fact, our last qubit in the system only receives a single H. What this means, is that we can think of the QFT as a 'more complex' version of a Hadamard Transformation in some sense, where the extra bit of complexity is the additional phases. To see this, let's compare the 4×4 unitary matrices for the QFT and Hadamard Transformation on two qubits:

$$QFT \qquad\qquad H$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \qquad\qquad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

The two transformations are nearly identical, except for the extra presences of a couple i's in the QFT. These i's represent the extra complexity of the QFT for the 2-qubit case. And when we look at larger transformations, we will see more and more unique amplitudes accompanying states in the system.

However, regardless of size, one property that both the Hadamard Transformation and QFT share is the way they map the state of all 0's:

$$|\,00...0\rangle \qquad \longleftrightarrow \qquad \frac{1}{\sqrt{2^N}}\big(\,|\,00...0\rangle \;+\; \cdots \;+\; |\,11...1\rangle\,\big)$$

Both transformations map the state of all 0's to an equal superposition, where all the states have the same positive phase. For H^N, we've shown that this result comes from the fact that $H|0\rangle$ and $H|1\rangle$ both produce a state where the $|0\rangle$ component is positive. Similarly, if we return to our earlier example where we broke down all of the gate operations for the 3-qubit QFT, we get the exact same result. Because each qubit initially receives a H gate followed by all control-gates, the $|0\rangle$ component for every qubit will always be positive. Simultaneously, since we are dealing with the state $|00...0\rangle$, none of the R_ϕ are applying any phases.

This mapping was the core ingredient for the the Grover Algorithm. Specifically, we used this mapping of $|00...0\rangle$ as our way of achieving a reflection about the average. Thus, since our QFT also has this mapping property, we should be able to perform the Grover Algorithm using a QFT in place of the H^N transformations:

```
q    = QuantumRegister(2,name='q')
anc  = QuantumRegister(1,name='anc')
FG_qc = QuantumCircuit(q,anc,name='qc')
marked = [1,0]

FG_qc.iden( q[0])
FG_qc.iden( q[1] )
FG_qc.x( anc[0] )

print('marked state: ',marked)
print('  ')

oq.QFT(FG_qc,q,2)
print('____ Initial State (QFT) ____')
oq.Wavefunction(FG_qc, systems=[2,1], show_systems=[True,False])
print('  ')

oq.X_Transformation(FG_qc, q, marked)
FG_qc.h( anc[0] )
FG_qc.ccx( q[0], q[1], anc[0] )
oq.X_Transformation(FG_qc, q, marked)

FG_qc.h( anc[0] )
print('____ Flip the Marked State ____')
oq.Wavefunction(FG_qc, systems=[2,1], show_systems=[True,False])
print('  ')


oq.QFT(FG_qc,q,2)
print('____ QFT ____')
oq.Wavefunction(FG_qc, systems=[2,1], show_systems=[True,False])
print('  ')
FG_qc.h( anc[0] )


oq.X_Transformation(FG_qc, q, [0,0])
FG_qc.ccx( q[0], q[1], anc[0] )
FG_qc.h( anc[0] )
oq.X_Transformation(FG_qc, q, [0,0])

print('____ Flip the |00> state ____')
oq.Wavefunction(FG_qc, systems=[2,1], show_systems=[True,False])
print('  ')

oq.QFT_dgr(FG_qc,q,2)
print('____ QFT_dgr  ____')
oq.Wavefunction(FG_qc, systems=[2,1], show_systems=[True,False])
```

```
marked state:  [1, 0]

____ Initial State (QFT) ____
0.5 |00>    0.5 |10>    0.5 |01>    0.5 |11>

____ Flip the Marked State ____
0.5 |00>    -0.5 |10>    0.5 |01>    0.5 |11>

____ QFT ____
0.5 |00>    0.5 |10>    -0.5 |01>    0.5 |11>

____ Flip the |00> state ____
-0.5 |00>    0.5 |10>    -0.5 |01>    0.5 |11>

____ QFT_dgr  ____
-1.0 |10>
```

Success! By using the QFT and inverse QFT, we are able to perform a Grover Search for a marked state. For an explanation of the Grover Algorithm, please refer to lesson 5.4. Note that in this coding example there are a lot of added steps, in order to display all of the individual steps nicely.

Hopefully this example gives you an idea of just how similar the QFT and Hadamard transformation are at their core. But, the reason we will be able to use the QFT to accomplish some more complex algorithms, comes from the fact that the states it maps to contain more phase differences. Or another way of saying that is, the QFT allows us to create 'more orthogonal' states (not literally), where the extra phases will prove very useful.

This concludes lesson 6 and all of the tutorials in this series! Understanding the QFT is a bit tricky at first, so don't worry if everything in this tutorial doesn't feel second nature yet. Just like all of the practice we got with the Hadamard Transformation in lessons 5.1 - 5.4, you will need to see the QFT in action a few times to truly understand and appreciate its role in quantum algorithms

---

This concludes all of the lessons in this tutorial series, but there is still much to learn about quantum algorithms! If you are looking to continue your learning endeavors, I encourage you to take a look at the following more advanced algorithms:

-Quantum Phase Estimation

-Shor's Algorithms

-Variational Quantum Eigensolver (VQE)

-Quantum Approximate Optimization Algorithm (QAOA)

In addition to these, there are many many more quantum algorithms available at The Quantum Algorithm Zoo:

http://quantumalgorithmzoo.org/

References

[1] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, Cambridge, Cambridge University Press (2000)
[2] P. Kaye, R. Laflamme, M. Mosca, *An Introduction to Quantum Computing*, Oxford, Oxford University Press (2007)
[3] G. Chen et al., *Quantum Computing Devices: Principles, Designs, and Analysis*, Boca Raton, Taylor & Francis Group (2007)
[4] M. Nakahara and T. Ohmi, *Quantum Computing: From Linear Algebra to Physical Realizations*, Boca Raton, Taylor & Francis Group (2008)
[5] N. Yanofsky and M. Mannucci, *Quantum Computing for Computer Scientists* New York, Cambridge University Press (2008)
[6] J. Bergou and M. Hillery, *Introduction to the Theory of Quantum Information Processing* New York, Springer (2013)
[7] D. Deutsch, Proceedings of the Royal Society, London Series A **400**, 97 (1985)
[8] D. Deutsch and R. Jozsa, Proceedings: Mathematical and Physical Sciences **439**, 1907 (1992)
[9] E. Bernstein and U. Vazirani, Proceedings of the 25th Annual ACM Symposium on Theory of Computing (1993)
[10] D. Simon, Proceedings of the 35th Annual IEEE Symposium on the Foundations of Computer Science (1994)
[11] L. K. Grover Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (1996)
[12] D. Coppersmith, arXiv: 0201067 (1994)

Appendix

Below are all of the functions contained within the accompanying python file Our_Qiskit_Functions.py. These functions were written to be compatible with Qiskit v0.7.

```python
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit, Aer, execute
import numpy as np
import math as m
import scipy as sci
S_simulator = Aer.backends(name='statevector_simulator')[0]
M_simulator = Aer.backends(name='qasm_simulator')[0]


#==================================================
#-----------     Displaying Results     -----------
#==================================================

def Wavefunction( obj , *args, **kwargs):
    '''
    Displays the wavefunction of the quantum system
    '''
    if(type(obj) == QuantumCircuit  ):
        statevec = execute( obj, S_simulator, shots=1 ).result().get_statevector()
    if(type(obj) == np.ndarray):
        statevec = obj
    sys = False
    NL = False
    dec = 5
    if 'precision' in kwargs:
        dec = int( kwargs['precision'] )
    if 'column' in kwargs:
        NL = kwargs['column']
    if 'systems' in kwargs:
        systems = kwargs['systems']
        sys = True
        last_sys = int(len(systems)-1)
        show_systems = []
        for s_chk in np.arange(len(systems)):
            if( type(systems[s_chk])!=int ):
                raise Exception('systems must be an array of all integers')
        if 'show_systems' in kwargs:
            show_systems = kwargs['show_systems']
            if( len(systems)!=len(show_systems) ):
                raise Exception('systems and show_systems need to be arrays of equal length')
            for ls in np.arange(len(show_systems)):
                if((show_systems[ls]!=True)and(show_systems[ls]!=False)):
                    raise Exception('show_systems must be an array of Truth Values')
                if(show_systems[ls]==True):
                    last_sys = int(ls)
        else:
            for ss in np.arange(len(systems)):
                show_systems.append(True)
    wavefunction = ''
    qubits = int(m.log(len(statevec),2))
    for i in np.arange( int(len(statevec)) ):
        value = round(statevec[i].real, dec) + round(statevec[i].imag, dec) * 1j
        if( (value.real!=0) or (value.imag!=0) ):
            state = list(Binary2(int(i),int(2**qubits)))
            state_str = ''
            if( sys == True ):                                  #Systems and Show_Systems
                k = 0
                for s in np.arange(len(systems)):
                    if(show_systems[s]==True):
                        if(int(s)!=last_sys):
                            state.insert( int(k+systems[s]),'>|' )
                            k = int(k+systems[s]+1)
                        else:
                            k = int(k+systems[s])
                    else:
                        for s2 in np.arange(systems[s]):
                            del state[int(k)]
            for j in np.arange(len(state)):
                if(type(state[j])!=str):
                    state_str = state_str+str(int(state[j]))
```

```
                else:
                    state_str = state_str+state[j]
            if( (value.real!=0) and (value.imag!=0) ):
                if( value.imag > 0):
                    wavefunction = wavefunction +str(value.real)+'+'+str(value.imag)+'j |'+state_str+'>    '
                else:
                    wavefunction = wavefunction +str(value.real)+''+str(value.imag)+'j |'+state_str+'>    '
            if( (value.real!=0) and (value.imag==0) ):
                wavefunction = wavefunction +str(value.real)+' |'+state_str+'>    '
            if( (value.real==0) and (value.imag!=0) ):
                wavefunction = wavefunction +str(value.imag)+'j |'+state_str+'>    '
            if(NL):
                wavefunction = wavefunction + '\n'
    print(wavefunction)


def Measurement(quantumcircuit, *args, **kwargs):
    '''
    Displays the measurement results of a quantum circuit
    '''
    p_M = True
    S=1
    ret = False
    NL = False
    if 'shots' in kwargs:
        S = int(kwargs['shots'])
    if 'return_M' in kwargs:
        ret = kwargs['return_M']
    if 'print_M' in kwargs:
        p_M = kwargs['print_M']
    if 'column' in kwargs:
        NL = kwargs['column']
    M1 = execute(quantumcircuit, M_simulator, shots=S).result().get_counts(quantumcircuit)
    M2 = {}
    k1 = list(M1.keys())
    v1 = list(M1.values())
    for k in np.arange(len(k1)):
        key_list = list(k1[k])
        new_key = ''
        for j in np.arange(len(key_list)):
            new_key = new_key+key_list[len(key_list)-(j+1)]
        M2[new_key] = v1[k]
    if(p_M):
        k2 = list(M2.keys())
        v2 = list(M2.values())
        measurements = ''
        for i in np.arange( len(k2) ):
            m_str = str(v2[i])+'|'
            for j in np.arange(len(k2[i])):
                if( k2[i][j] == '0' ):
                    m_str = m_str+'0'
                if( k2[i][j] == '1' ):
                    m_str = m_str+'1'
                if( k2[i][j] == ' ' ):
                    m_str = m_str+'>|'
            m_str = m_str+'>    '
            if(NL):
                m_str = m_str + '\n'
            measurements = measurements + m_str
        print(measurements)
    if(ret):
        return M2

#================================================
#-----------     Math Operations     -----------
#================================================

def Oplus(bit1,bit2):
    '''
    Adds two bits of 0's and 1's (modulo 2)
    '''
    bit = np.zeros(len(bit1))
    for i in np.arange( len(bit) ):
        if( (bit1[i]+bit2[i])%2==0 ):
            bit[i] = 0
        else:
            bit[i] = 1
    return bit

def Binary(number,total):
    '''
    Converts a number to binary, right to left LSB
    '''
    qubits = int(m.log(total,2))
    N = number
    b_num = np.zeros(qubits)
    for i in np.arange(qubits):
        if( N/((2)**(qubits-i-1)) >= 1 ):
            b_num[i] = 1
```

```
    for j in np.arange(len(b_num)):
        B.append(int(b_num[j]))
    return B


def From_Binary(string):
    '''
    Converts a binary number to base 10, right to left LSB
    '''
    num = 0
    for i in np.arange(len(string)):
        num = num + string[int(0-(i+1))] * 2**(i)
    return num



#=========================================
#-----------     Custom Gates     -----------
#=========================================
def X_Transformation(qc, qreg, state):
    '''
    Tranforms the state of the system, applying X gates according to 0's in the vector 'state'
    '''
    for j in np.arange(len(state)):
        if( int(state[j])==0 ):
            qc.x( qreg[int(j)] )



def n_NOT(qc, control, target, anc):
    '''
    performs an n-NOT gate
    '''
    n = len(control)
    instructions = []
    active_ancilla = []
    q_unused = []
    q = 0
    a = 0
    while( (n > 0) or (len(q_unused)!=0) or (len(active_ancilla)!=0) ):
        if( n > 0 ):
            if( (n-2) >= 0 ):
                instructions.append( [control[q], control[q+1], anc[a]] )
                active_ancilla.append(a)
                a = a + 1
                q = q + 2
                n = n - 2
            if( (n-2) == -1 ):
                q_unused.append( q )
                n = n - 1
        elif( len(q_unused) != 0 ):
            if(len(active_ancilla)!=1):
                instructions.append( [control[q], anc[active_ancilla[0]], anc[a]] )
                del active_ancilla[0]
                del q_unused[0]
                active_ancilla.append(a)
                a = a + 1
            else:
                instructions.append( [control[q], anc[active_ancilla[0]], target] )
                del active_ancilla[0]
                del q_unused[0]
        elif( len(active_ancilla)!=0 ):
            if( len(active_ancilla) > 2 ):
                instructions.append( [anc[active_ancilla[0]], anc[active_ancilla[1]], anc[a]]  )
                active_ancilla.append(a)
                del active_ancilla[0]
                del active_ancilla[0]
                a =  a + 1
            elif( len(active_ancilla)==2):
                instructions.append([anc[active_ancilla[0]], anc[active_ancilla[1]], target])
                del active_ancilla[0]
                del active_ancilla[0]
    for i in np.arange( len(instructions) ):
        qc.ccx( instructions[i][0], instructions[i][1], instructions[i][2] )
    del instructions[-1]
    for i in np.arange( len(instructions) ):
        qc.ccx( instructions[0-(i+1)][0], instructions[0-(i+1)][1], instructions[0-(i+1)][2] )

def Control_Instruction( qc, vec ):
    '''
    Ammends the proper quantum circuit instruction based on the input 'vec'
    Used for the function 'n_Control_U
    '''
    if( vec[0] == 'X' ):
        qc.cx( vec[1], vec[2] )
    if( vec[0] == 'Z' ):
        qc.cz( vec[1], vec[2] )
    if( vec[0] == 'PHASE' ):
        qc.cu1( vec[2], vec[1], vec[3] )
```

```
252
253
254 def n_Control_U(qc, control, anc, gates):
255     '''
256     Performs a list of single control gates, as an n-control operation
257     '''
258     if( len(gates)!=0 ):
259         instructions = []
260         active_ancilla = []
261         q_unused = []
262         n = len(control)
263         q = 0
264         a = 0
265         while( (n > 0) or (len(q_unused)!=0) or (len(active_ancilla)!=0) ):
266             if( n > 0 ):
267                 if( (n-2) >= 0 ):
268                     instructions.append( [control[q], control[q+1], anc[a]] )
269                     active_ancilla.append(a)
270                     a = a + 1
271                     q = q + 2
272                     n = n - 2
273                 if( (n-2) == -1 ):
274                     q_unused.append( q )
275                     n = n - 1
276             elif( len(q_unused) != 0 ):
277                 if(len(active_ancilla)>1):
278                     instructions.append( [control[q], anc[active_ancilla[0]], anc[a]] )
279                     del active_ancilla[0]
280                     del q_unused[0]
281                     active_ancilla.append(a)
282                     a = a + 1
283                 else:
284                     instructions.append( [control[q], anc[active_ancilla[0]], anc[a]] )
285                     del active_ancilla[0]
286                     del q_unused[0]
287                     c_a = anc[a]
288             elif( len(active_ancilla)!=0 ):
289                 if( len(active_ancilla) > 2 ):
290                     instructions.append( [anc[active_ancilla[0]], anc[active_ancilla[1]], anc[a]]  )
291                     active_ancilla.append(a)
292                     del active_ancilla[0]
293                     del active_ancilla[0]
294                     a =  a + 1
295                 elif( len(active_ancilla)==2):
296                     instructions.append([anc[active_ancilla[0]], anc[active_ancilla[1]], anc[a]] )
297                     del active_ancilla[0]
298                     del active_ancilla[0]
299                     c_a = anc[a]
300                 elif( len(active_ancilla)==1):
301                     c_a = anc[active_ancilla[0]]
302                     del active_ancilla[0]
303         for i in np.arange( len(instructions) ):
304             qc.ccx( instructions[i][0], instructions[i][1], instructions[i][2] )
305         for j in np.arange(len(gates)):
306             control_vec = [ gates[j][0], c_a ]
307             for k in np.arange( 1, len(gates[j])):
308                 control_vec.append( gates[j][k] )
309             Control_Instruction( qc, control_vec )
310         for i in np.arange( len(instructions) ):
311             qc.ccx( instructions[0-(i+1)][0], instructions[0-(i+1)][1], instructions[0-(i+1)][2] )
312
313
314 #=========================================
315 #-----------     Lesson 5.1     -----------
316 #=========================================
317
318
319 def Blackbox_g_D(qc,qreg):
320     '''
321     Generates a random blackbox unitary operator g, based on a balanced or constant f
322     '''
323     f_type = ['f(0,1) -> (0,1)','f(0,1) -> (1,0)','f(0,1) -> 0','f(0,1) -> 1']
324     r = int( m.floor( 4*sci.rand() ) )
325     if(r==0):
326         qc.cx( qreg[0], qreg[1] )
327     if(r==1):
328         qc.x( qreg[0] )
329         qc.cx( qreg[0], qreg[1] )
330         qc.x( qreg[0] )
331     if(r==2):
332         qc.iden( qreg[0] )
333         qc.iden( qreg[1] )
334     if(r==3):
335         qc.x( qreg[1] )
336     return f_type[r]
337
338 def Deutsch(qc,qreg):
339     '''
340     Apply the Deutsch Algorithm to a QuantumCircuit
```

```
    qc.h( qreg[0] )
    qc.h( qreg[1] )
    return f

#========================================
#----------     Lesson 5.2     ----------
#========================================


def Blackbox_g_DJ(Q,qc,qreg,an1):
    '''
    Generates a random blackbox unitary operator g, based on a balanced or constant f
    '''
    f_type = ['constant','blanaced']
    f = []
    r = int( m.floor( 2**Q * sci.rand() ) )
    control = []
    for i in np.arange(Q):
        control.append( qreg[int(i)] )
    if(r==0):
        for i in np.arange(Q):
            qc.iden( qreg[int(i)] )
        f.append(f_type[0])
    if(r==1):
        qc.x( qreg[int(Q-1)] )
        f.append(f_type[0])
    if(r>=2):
        an2 = QuantumRegister(int(Q-2),name='nn_anc')
        QC = QuantumCircuit(an2)
        qc += QC
        f.append(f_type[1])
        S = []
        for s in np.arange(2**Q):
            S.append( int(s) )
        for k in np.arange(2**(Q-1)):
            S_num = S[int(m.floor(len(S)*sci.rand()))]
            state = Binary( S_num ,2**Q )
            S.remove(S_num)
            f_string = '|'
            for j in np.arange(len(state)):
                f_string = f_string+str(int(state[j]))
                if( int(state[j])==0 ):
                    qc.x( qreg[int(j)] )
            n_NOT( qc, control, an1[0], an2 )
            for j in np.arange(len(state)):
                if( int(state[j])==0 ):
                    qc.x( qreg[int(j)] )
            f.append(f_string+'>')
    return f


def Deutsch_Jozsa(Q,qc,qreg,an1):
    '''
    Takes in the initial state, adds all of the instructions for the Deutsch-Jozsa Algorithm
    '''
    for i in np.arange(Q):
        qc.h( qreg[int(i)] )
    qc.h( an1[0] )
    f = Blackbox_g_DJ(Q, qc, qreg, an1)
    for i in np.arange(Q):
        qc.h( qreg[int(i)] )
    qc.h( an1[0] )
    return f

def Blackbox_g_BV(Q,qc,qreg,an1):
    '''
    Generates a random blackbox unitary operator g, based on a balanced or constant f
    '''
    a = Binary( int( m.floor( 2**Q * sci.rand() )), 2**Q)
    control = []
    for i in np.arange(Q):
        control.append( qreg[int(i)] )
    an2 = QuantumRegister(int(Q-2),name='nn_anc')
    QC = QuantumCircuit(an2)
    qc += QC
    for s in np.arange(2**Q):
        state = Binary(int(s),2**Q)
        dp = np.vdot( a, state )
        if( dp%2 == 1 ):
            for j in np.arange(len(state)):
                if( int(state[j])==0 ):
                    qc.x( qreg[int(j)] )
            n_NOT( qc, control, an1[0], an2 )
            for j in np.arange(len(state)):
                if( int(state[j])==0 ):
                    qc.x( qreg[int(j)] )
    return a

```

```
def Berstein_Vazirani(Q,qc,qreg,an1):
    '''
    Takes in the initial state, adds all of the instructions for the Deutsch-Jozsa Algorithm
    '''
    for i in np.arange(Q):
        qc.h( qreg[int(i)] )
    qc.h( an1[0] )
    a = Blackbox_g_BV(Q, qc, qreg, an1)
    for i in np.arange(Q):
        qc.h( qreg[int(i)] )
    qc.h( an1[0] )
    return a

#=======================================
#----------     Lesson 5.3     ----------
#=======================================

def Blackbox_g_S(Q, qc, q, anc1):
    '''
    Apples the blackbox opertor g, for Simon's Algorithm
    '''
    anc2 = QuantumRegister(int(Q-1),name='nU_anc')
    QC = QuantumCircuit(anc2)
    qc += QC
    s = np.zeros(Q)
    for i in np.arange(Q):
        s[i] = m.floor( 2*sci.rand() )
    outputs = []
    for o in np.arange(2**Q):
        outputs.append( int(o) )
    f = np.zeros(2**Q)
    for j in np.arange(2**Q):
        out = outputs[int( m.floor( len(outputs)*sci.rand() ) )]
        f[j] = int(out)
        f[ int( From_Binary(Oplus(Binary(j,2**Q),s)) ) ] = int(out)
        outputs.remove(out)
    output_states = []
    for k in np.arange(2**Q):
        output_states.append( Binary(f[k],2**Q) )
    for a in np.arange(2**Q):
        c_ops = []
        for b in np.arange(Q):
            if( output_states[a][b] == 1 ):
                c_ops.append( [ 'X', anc1[int(b)] ] )
        X_Transformation( qc, q, Binary(a,2**Q) )
        n_Control_U( qc, q, anc2, c_ops)
        X_Transformation( qc, q, Binary(a,2**Q) )
    return qc, s, f

def Simons_Quantum(Q, qc, q, c, anc1):
    '''
    Takes in the initial state, adds all of the instructions for Simon's Algorithm
    '''
    for i in np.arange(Q):
        qc.h( q[int(i)] )
    qc,s,f = Blackbox_g_S(Q, qc, q, anc1)
    for i in np.arange(Q):
        qc.h( q[int(i)] )
    qc.measure(q,c)
    return qc, s

def Simons_Classical(Q, qc):
    '''
    Takes the circuit for Simon's Algorithm and solves for s
    '''
    run_quantum = True
    Equations = []
    Results = []
    quantum_runs = 0
    while( run_quantum ):
        quantum_runs += 1
        M = Measurement( qc, shots=1, return_M=True, print_M=False)
        new_result = True
        for r in np.arange(len(Results)):
            if( list(M.keys())[0] == Results[r]):
                new_result = False
        if(new_result):
            Results.append( list(M.keys())[0] )
            eq = []
            for e in np.arange(Q):
                eq.append( int(list(M.keys())[0][e]) )
            Equations.append( eq )
            s_primes = Simons_Solver(Equations,Q)
            if( len(s_primes) ==1 ):
                run_quantum = False
    return s_primes,Results,quantum_runs

```

```
def Simons_Solver(E,N):
    '''
    Returns an array of s_prime candidates
    '''
    s_primes = []
    for s in np.arange(1,2**N):
        sp = Binary2( int(s), 2**N )
        candidate = True
        for e in np.arange( len(E) ):
            value = 0
            for i in np.arange( N ):
                value = value + sp[i]*E[e][i]
            if(value%2==1):
                candidate=False
        if(candidate):
            s_primes.append(sp)
    return s_primes


#=========================================
#----------     Lesson 5.4     -----------
#=========================================


def Grover_Oracle(mark, qc, q, an1, an2):
    '''
    picks out the marked state and applies a negative phase
    '''
    qc.h( an1[0] )
    X_Transformation(qc, q, mark)
    if( len(mark) > 2 ):
        n_NOT( qc, q, an1[0], an2 )
    if( len(mark) == 2 ):
        qc.ccx( q[0], q[1], an1[0] )
    X_Transformation(qc, q, mark)
    qc.h( an1[0] )

def Grover_Diffusion(mark, qc, q, an1, an2):
    '''
    ammends the instructions for a Grover Diffusion Operation to the QuantumCircuit
    '''
    zeros_state = []
    for i in np.arange( len(mark) ):
        zeros_state.append( 0 )
        qc.h( q[int(i)] )
    Grover_Oracle(zeros_state, qc, q, an1, an2)
    for j in np.arange( len(mark) ):
        qc.h( q[int(j)] )

def Grover(Q, marked):
    '''
    Ammends all the instructions for a Grover Search
    '''
    q   = QuantumRegister(Q,name='q')
    an1 = QuantumRegister(1,name='anc')
    an2 = QuantumRegister(Q-2,name='nanc')
    c   = ClassicalRegister(Q,name='c')
    qc  = QuantumCircuit(q,an1,an2,c,name='qc')
    for j in np.arange(Q):
        qc.h( q[int(j)] )
    qc.x( an1[0] )
    iterations = round( m.pi/4 * 2**(Q/2.0) )
    for i in np.arange( iterations ):
        Grover_Oracle(marked, qc, q, an1, an2)
        Grover_Diffusion(marked, qc, q, an1, an2)
    return qc, q, an1, an2, c

```

```
#=======================================
#----------     Lesson 6    ------------
#=======================================


def QFT(qc,q,qubits):
    '''
    Assigns all the gate operations for a Quantum Fourier Transformation
    '''
    R_phis = [0]
    for i in np.arange(2,int(qubits+1)):
        R_phis.append( 2/(2**(i)) * m.pi )
    for j in np.arange(int(qubits)):
        qc.h( q[int(j)] )
        for k in np.arange(j+1,int(qubits)):
            qc.cu1( R_phis[k], q[int(k)], q[int(j)] )

def QFT_dgr(qc, q, qubits):
    '''
    Assigns all the gate operations for an inverse Quantum Fourier Transformation
    '''
    R_phis = [0]
    for i in np.arange(2,int(qubits+1)):
        R_phis.append( -2/(2**(i)) * m.pi )
    for j in np.arange(int(qubits)):
        for k in np.arange( int(j) ):
            qc.cu1(R_phis[int(qubits-(k+1))], q[int(qubits-(k+1))], q[int(qubits-(j+1))] )
        qc.h( q[int(qubits-(j+1))] )
```