Ismail Akram
PHYS 315 Quantum Computing
Prof. Seth Cottrell

To You, One Year from Now

Contents

Ismail Akram
PHYS 315 Quantum Computing
Prof. Seth Cottrell

Abstract

Dear Ismail of the past, I know you're fearful of your future and struggling with self-doubt. Know that you're on a journey. You'll reach a point in your college career that you're comprehending subjects beyond your current comprehension. That's what a journey is, it's okay to not understand every minutia. Your daydreaming imagination and love for Mathematics led you to pursue Computer Science. There you delved further into Machine Learning, Neural Networking, Artificial Intelligence, and even Quantum Computing! Through that delving, you'll experience challenges along the way, to the point of wanting to quit and relinquish this journey. You'll feel inadequate the further you go and realize that there is so much more to learn. But you need to understand that not knowing something isn't a sign of inadequacy, it's a chance for you to learn and overcome that journey and grow. By the time you take Quantum Computing, you'll be on the verge of a breakdown, and that's okay. What matters is that you get back up and recover.

The first lecture slide discussed a "sobering observation" about the implications of the "science of the small". In that, Quantum mechanical laws seems to be universal, that there's nothing special about photons. "Does the photon really go through both slits?" To borrow a term from Thomas Kuhn, I understood this to be a new possible paradigm shift in understanding our universe's physical mechanisms.

For my project, I chose the Deutsch-Jozsa problem because it is the first known quantum algorithm to be exponentially faster any known classical algorithm. It's best to start from the humble beginning.

Deutsch-Jozsa Algorithm

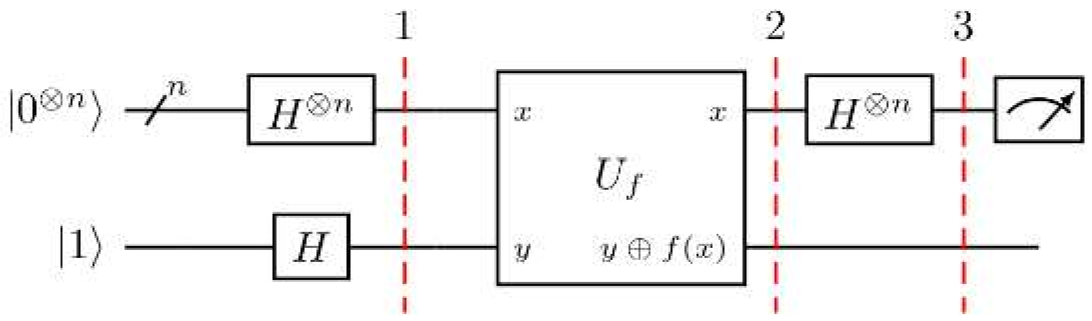


Figure 1. Circuit for Deutsch-Jozsa algorithm

Ismail Akram
PHYS 315 Quantum Computing
Prof. Seth Cottrell

Introduction

The Deutsch-Jozsa algorithm was the first known quantum algorithm that performed exponentially faster than a classical algorithm. It is an n-bit extension of the single Deutsch problem. Despite it being more of a “proof-of-concept” rather than practical, it demonstrated the advantages of using a quantum computer as a computational tool for a specific problem.

We start by assuming we have a black box function $f : \{0,1\} \rightarrow \{0,1\}$. Black box meaning that we can only give inputs and observe this function’s outputs.

Function f takes as inputs, a string of bits, and returns either a $0\ or\ 1$:

$$f(\{x_0, x_1, x_2, \ldots\}) \rightarrow 0\ or\ 1, where\ x_n\ is\ 0\ or\ 1$$

The question behind Deutsch-Jozsa is “Is f balanced or constant?”, balanced as in $f = 0$ exactly as often as $f = 1$. In other words, a constant function returns all 0’s or all 1’s for any input, whereas a balanced function returns 0’s for exactly half of all inputs and 1’s otherwise. In the classical approach, we need to evaluate f for both its possible inputs, 0 and 1.

If $f(0) = f(1)$, then it’s balanced

If $f(0) \neq f(1)$, then it’s constant

We can do this in one evaluation, by enacting $f(x)$ through a unitary operation U_f

$$U_f \ |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$$, where $\oplus$ is modulo 2

The Classical Approach

Classically, in the best case, we would need two queries to the oracle to determine if the hidden Boolean function is balanced.

e.g. if we get both:

$f(0,0,0,\dots) \rightarrow 0$ and $f(1,0,0,\dots) \rightarrow 1$, then f is balanced (two different outputs).

However, in the worse case scenario, if we continue to see the same output for each input, we'd need to check exactly half of all possible inputs plus one to ascertain that $f(x)$ is indeed constant. Since the total number of possible inputs is 2^n, we'd need $2^n + 1$ test inputs.

e.g. in a 4-bit string, if we observe that 8/16 possible combinations net 0's, it's possible that the 9th could return a 1 (probabilistically unlikely, but *still* possible), which would mean $f(x)$ is balanced. We can express this as:

$$P_{constant}(k) = 1 - \frac{1}{2^k - 1}\ for\ 1 < k \leq 2^{n-1}$$

Realistically, by this expression, we can truncate our classical algorithm early at a particular threshold, but for 100% confidence, we need to check for $2^{n-1} + 1$ inputs.

The Quantum Approach

Using a quantum computer, we can solve this problem through one call to function $f(x)$ with 100% confidence, provided we have f implemented as a quantum oracle, mapping:

$$U_f\ |x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$, where $\oplus$ is modulo 2

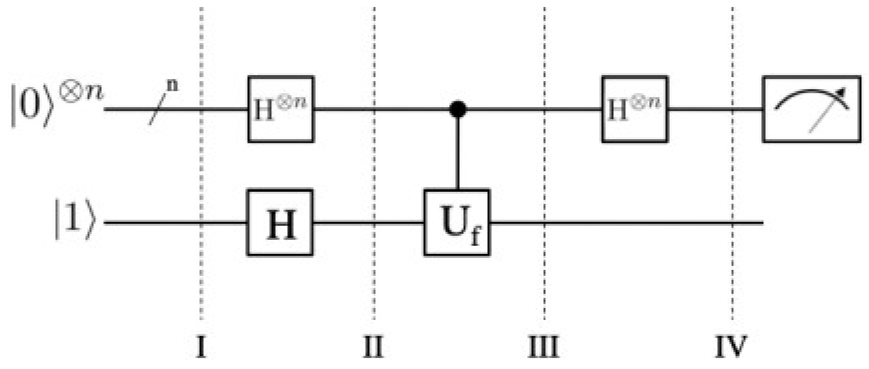


Figure 2. Generic circuit for n-bit Deutsch-Jozsa algorithm, note the hash mark on the top wire.

$$|0\rangle^{\otimes n} \equiv \underbrace{|0\rangle|0\rangle\dots|0\rangle}_{n} = |00\dots0\rangle \qquad H^{\otimes n} \equiv \underbrace{H \otimes H \otimes \dots \otimes H}_{n}$$

Ismail Akram
PHYS 315 Quantum Computing
Prof. Seth Cottrell

Initial state:

We prepare two quantum registers:
first: n-bit qubit initialized to $|0\rangle$,
second: 1-bit qubit initialized to $|1\rangle$

$$I: |\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle$$

Applying the Hadamard operator to each qubit, we get:

$$II: |\psi_1\rangle = \frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^{n-1}} |\mathrm{x}\rangle(|0\rangle - |1\rangle)$$

Then apply the unitary operation U_f:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^{n-1}} |x\rangle(|f(x)\rangle - |1 \oplus f(x)\rangle)$$

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

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

Then apply the last Hadamard gate:

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

Where $x \cdot y \equiv x_1y_1 \oplus x_2y_2 \oplus \ldots \oplus x_ny_n$

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

Notice that when measuring the first qubit, the probability of measuring $|0\rangle^{\otimes n} = \left|\frac{1}{2^n}\sum_{x=0}^{2n-1}(-1)^{f(x)}\right|^2$, which results in 1 if $f(x)$ is constant and 0 is $f(x)$ is balanced.

Explanation, why does it work?

Constance Oracle

When constant, the oracle has no effect on the input qubits. The quantum states before and after oracle query are the same. Since a H-gate is its own inverse, we can get the initial quantum state of $|00\ldots\ 0\rangle$ in the first register after the two H applications.

$$H^{\otimes n}\begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}=\frac{1}{\sqrt{2^n}}\begin{bmatrix}1\\1\\1\\\vdots\\1\end{bmatrix}\xrightarrow{after\ U_f} H^{\otimes n}\frac{1}{\sqrt{2^n}}\begin{bmatrix}1\\1\\1\\\vdots\\1\end{bmatrix}=\begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}$$

Balanced Oracle

After the first H-gate application, our input register is an equal superposition of all the states in the computational basis. So, in a balanced oracle, phase kickback adds a negative phase to exactly half of these states:

$$U_f\frac{1}{\sqrt{2^n}}\begin{bmatrix}1\\1\\1\\\vdots\\1\end{bmatrix}=\frac{1}{\sqrt{2^n}}\begin{bmatrix}-1\\1\\-1\\\vdots\\1\end{bmatrix}$$

The quantum states before and after oracle query are orthogonal! So, when applying the H-gate, we should get a quantum state that is orthogonal to $|00\ldots\ 0\rangle$. Meaning, we shouldn't measure the all-zero state.

Worked example

Consider the 2-bit balanced function $f(x_0,x_1)=x_0\oplus x_1$ such that:

$$f(0,0)=0$$

$$f(0,1)=1$$

$$f(1,0)=1$$

$$f(1,1)=0$$

Corresponding phase oracle:

$$U_f|x_1,x_0\rangle=(-1)^{f(x_1,x_0)}|x\rangle$$

Example state (first register of 2 qubits register to $|00\rangle$ and the second register qubit to $|1\rangle$):

$$|\psi_0\rangle=|00\rangle_{01}\otimes|1\rangle_2$$

Apply Hadamard on all qubits:

$$|\psi_1\rangle=\frac{1}{2}(|00\rangle+|01\rangle+\ |10\rangle+\ |11\rangle_{01}\otimes\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)_2$$

Apply unitary oracle function U_f:

$$|\psi_2\rangle = \frac{1}{2\sqrt{2}} * \dots$$

$$[(|00\rangle_{01} \otimes |0 \oplus 0 \oplus 0\rangle - |1 \oplus 0 \oplus 0\rangle_2$$

$$+ [(|01\rangle_{01} \otimes |0 \oplus 0 \oplus 1\rangle - |1 \oplus 0 \oplus 1\rangle_2$$

$$+[(|10\rangle_{01} \otimes |0 \oplus 1 \oplus 0\rangle - |1 \oplus 1 \oplus 0\rangle_2$$

$$+[(|11\rangle_{01} \otimes |0 \oplus 1 \oplus 1\rangle - |1 \oplus 1 \oplus 1\rangle_2]$$

$$= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)_0 \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)_1 \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)_2$$

Apply Hadamard to first register:

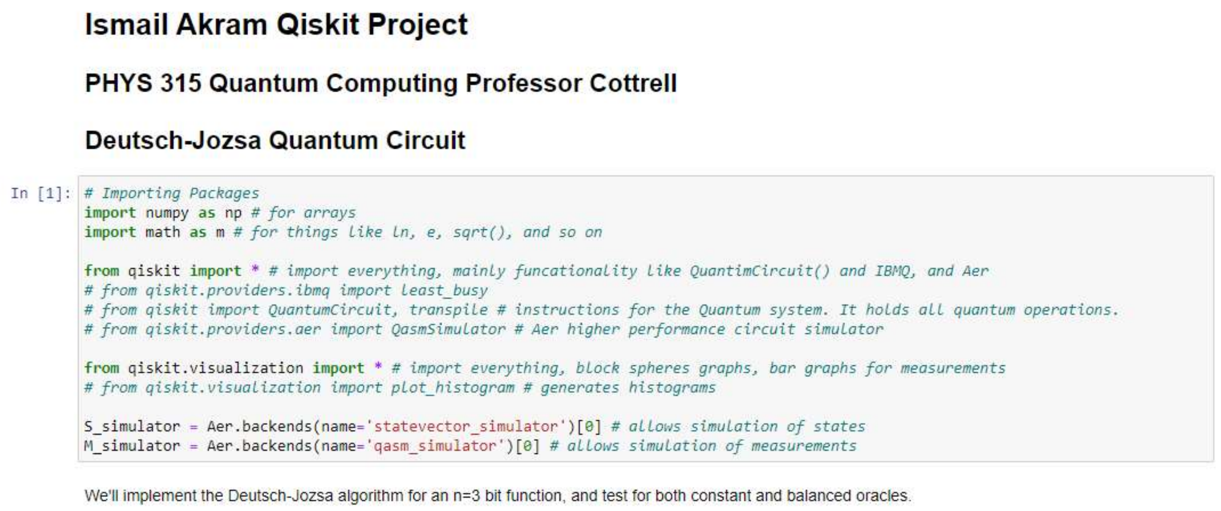
$$|\psi_3\rangle = |1\rangle_0 \otimes |1\rangle_1 \otimes (|0\rangle - |1\rangle)_2$$

Measuring the first two qubits, we get 11 (non-zero), indicating a balanced function.

Qiskit Implementation (using Jupyter Notebook)

The following are screen grabs from my Jupyter notebook (also included in submission):

Importing packages



Ismail Akram Qiskit Project

PHYS 315 Quantum Computing Professor Cottrell

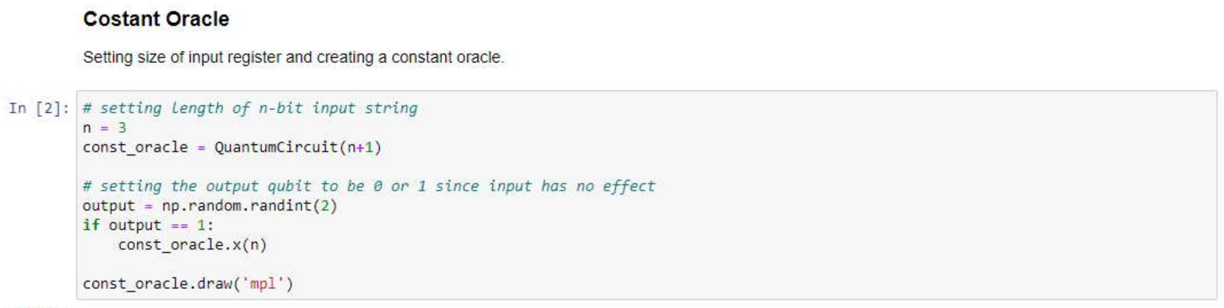
Deutsch-Jozsa Quantum Circuit

In [1]:

```
# Importing Packages
import numpy as np # for arrays
import math as m # for things like ln, e, sqrt(), and so on

from qiskit import * # import everything, mainly funcationality like QuantimCircuit() and IBMQ, and Aer
# from qiskit.providers.ibmq import least_busy
# from qiskit import QuantumCircuit, transpile # instructions for the Quantum system. It holds all quantum operations.
# from qiskit.providers.aer import QasmSimulator # Aer higher performance circuit simulator

from qiskit.visualization import * # import everything, block spheres graphs, bar graphs for measurements
# from qiskit.visualization import plot_histogram # generates histograms

S_simulator = Aer.backends(name='statevector_simulator')[0] # allows simulation of states
M_simulator = Aer.backends(name='qasm_simulator')[0] # allows simulation of measurements
```

We'll implement the Deutsch-Jozsa algorithm for an n=3 bit function, and test for both constant and balanced oracles.

Constance Oracle



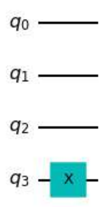
Costant Oracle

Setting size of input register and creating a constant oracle.

In [2]:

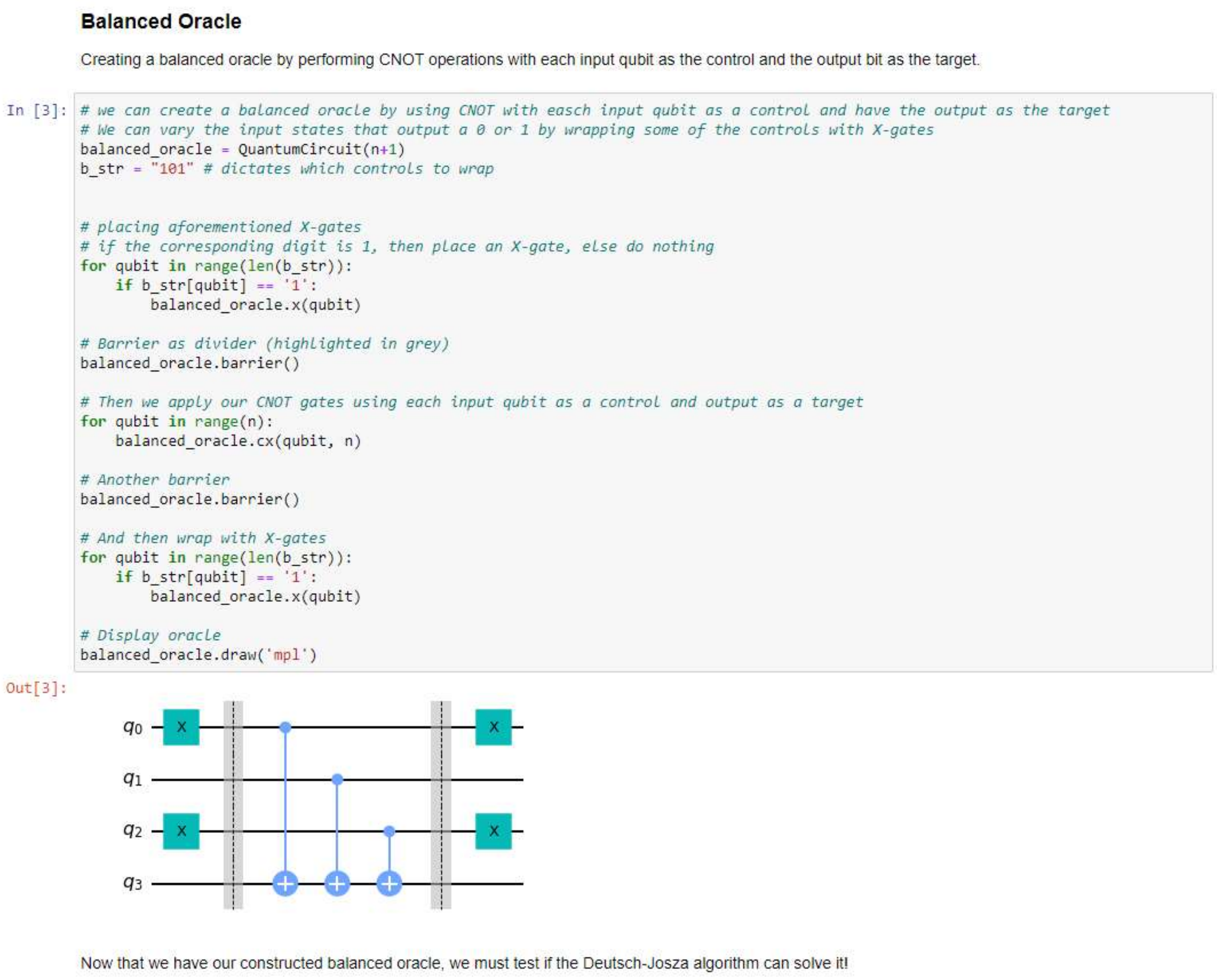
```
# setting length of n-bit input string
n = 3
const_oracle = QuantumCircuit(n+1)

# setting the output qubit to be 0 or 1 since input has no effect
output = np.random.randint(2)
if output == 1:
    const_oracle.x(n)

const_oracle.draw('mpl')
```

Out[2]:



Ismail Akram
PHYS 315 Quantum Computing
Prof. Seth Cottrell

Balanced Oracle



Balanced Oracle

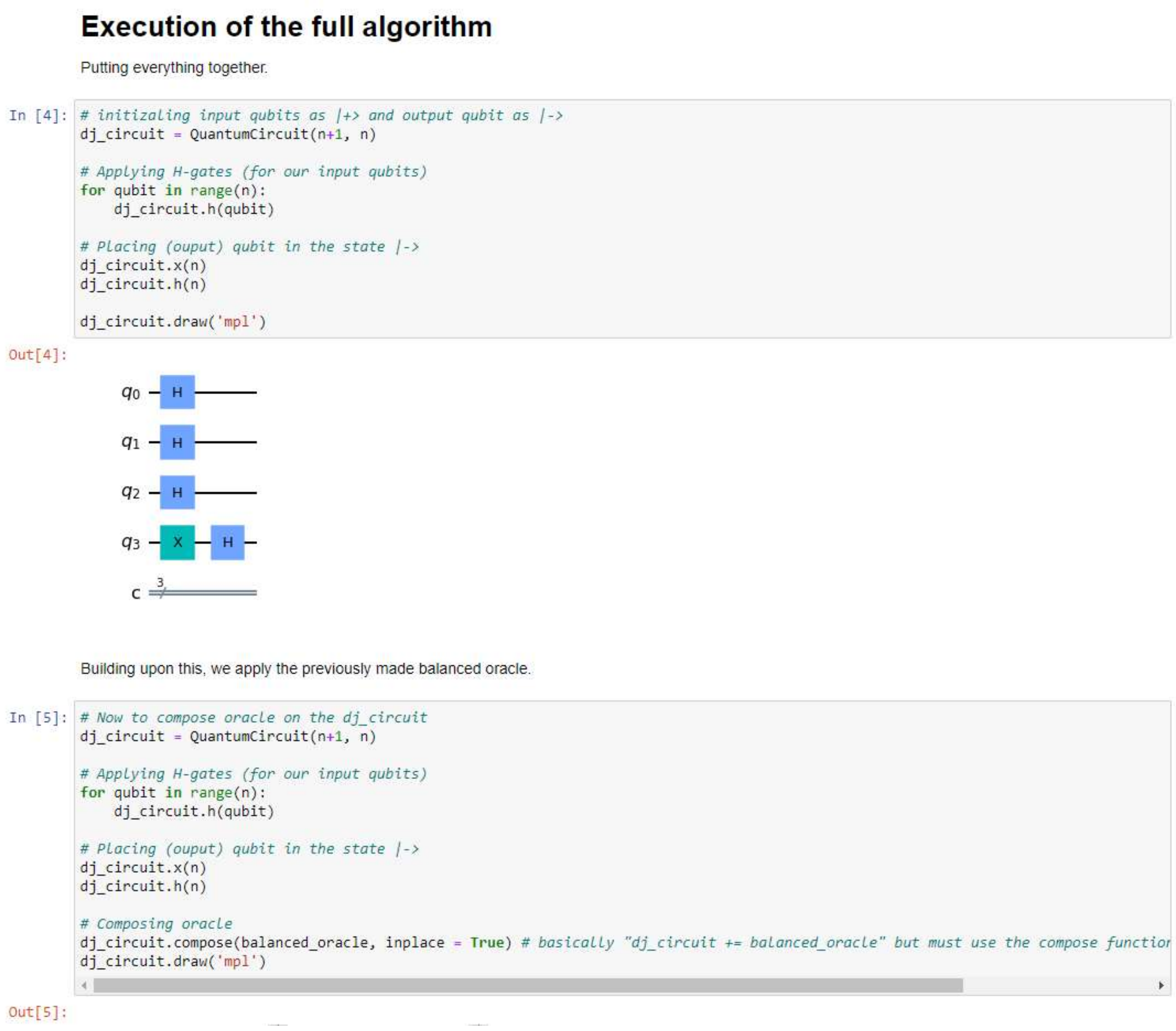
Creating a balanced oracle by performing CNOT operations with each input qubit as the control and the output bit as the target.

In [3]:
```
# we can create a balanced oracle by using CNOT with easch input qubit as a control and have the output as the target
# We can vary the input states that output a 0 or 1 by wrapping some of the controls with X-gates
balanced_oracle = QuantumCircuit(n+1)
b_str = "101" # dictates which controls to wrap


# placing aforementioned X-gates
# if the corresponding digit is 1, then place an X-gate, else do nothing
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)

# Barrier as divider (highlighted in grey)
balanced_oracle.barrier()

# Then we apply our CNOT gates using each input qubit as a control and output as a target
for qubit in range(n):
    balanced_oracle.cx(qubit, n)

# Another barrier
balanced_oracle.barrier()

# And then wrap with X-gates
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)

# Display oracle
balanced_oracle.draw('mpl')
```

Out[3]:

Now that we have our constructed balanced oracle, we must test if the Deutsch-Josza algorithm can solve it!

Execution of the Full Algorithm

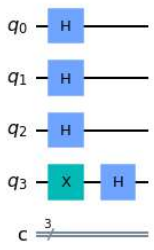


Execution of the full algorithm

Putting everything together.

In [4]:
```
# initizaling input qubits as |+> and output qubit as |->
dj_circuit = QuantumCircuit(n+1, n)

# Applying H-gates (for our input qubits)
for qubit in range(n):
    dj_circuit.h(qubit)

# Placing (ouput) qubit in the state |->
dj_circuit.x(n)
dj_circuit.h(n)

dj_circuit.draw('mpl')
```

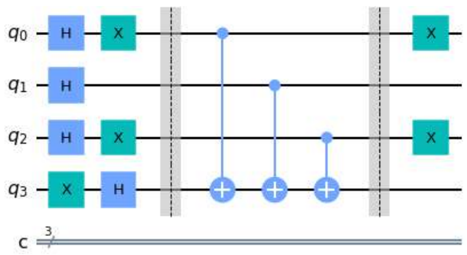
Out[4]:



Building upon this, we apply the previously made balanced oracle.

In [5]:
```
# Now to compose oracle on the dj_circuit
dj_circuit = QuantumCircuit(n+1, n)

# Applying H-gates (for our input qubits)
for qubit in range(n):
    dj_circuit.h(qubit)

# Placing (ouput) qubit in the state |->
dj_circuit.x(n)
dj_circuit.h(n)

# Composing oracle
dj_circuit.compose(balanced_oracle, inplace = True) # basically "dj_circuit += balanced_oracle" but must use the compose functio
dj_circuit.draw('mpl')
```

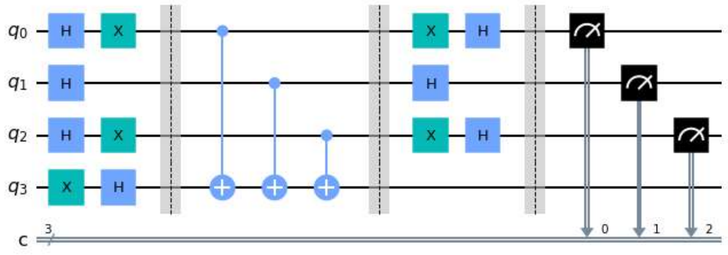
Out[5]:



Perfomring H-gates on our n=3 input qubit, and then measuring.

In [6]:
```
# Fianlly, adding Hadamard and then measuring
dj_circuit = QuantumCircuit(n+1, n)

# Applying H-gates (for our input qubits)
for qubit in range(n):
    dj_circuit.h(qubit)

# Placing (ouput) qubit in the state |->
dj_circuit.x(n)
dj_circuit.h(n)

# Add oracle
dj_circuit.compose(balanced_oracle, inplace = True) # basically dj_circuit += balanced_oracle but must use the compose function c

# Repeating H-gates
for qubit in range(n):
    dj_circuit.h(qubit)
dj_circuit.barrier()

# Measuring
for i in range(n):
    dj_circuit.measure(i, i)

dj_circuit.draw('mpl')
```

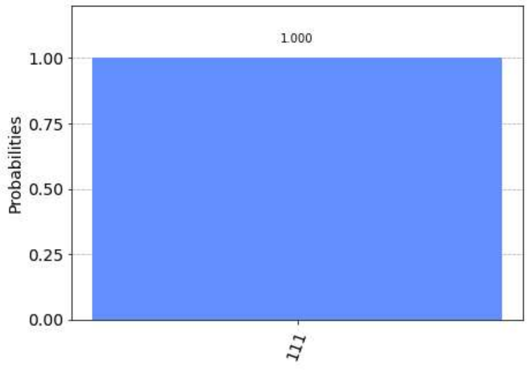
Out[6]:



Analyzing output

In [7]:
```
# Using local simulator
aer_sim = Aer.get_backend('aer_simulator')
qobj = assemble(dj_circuit, aer_sim)
results = aer_sim.run(qobj).result()
answer = results.get_counts()

plot_histogram(answer)
```

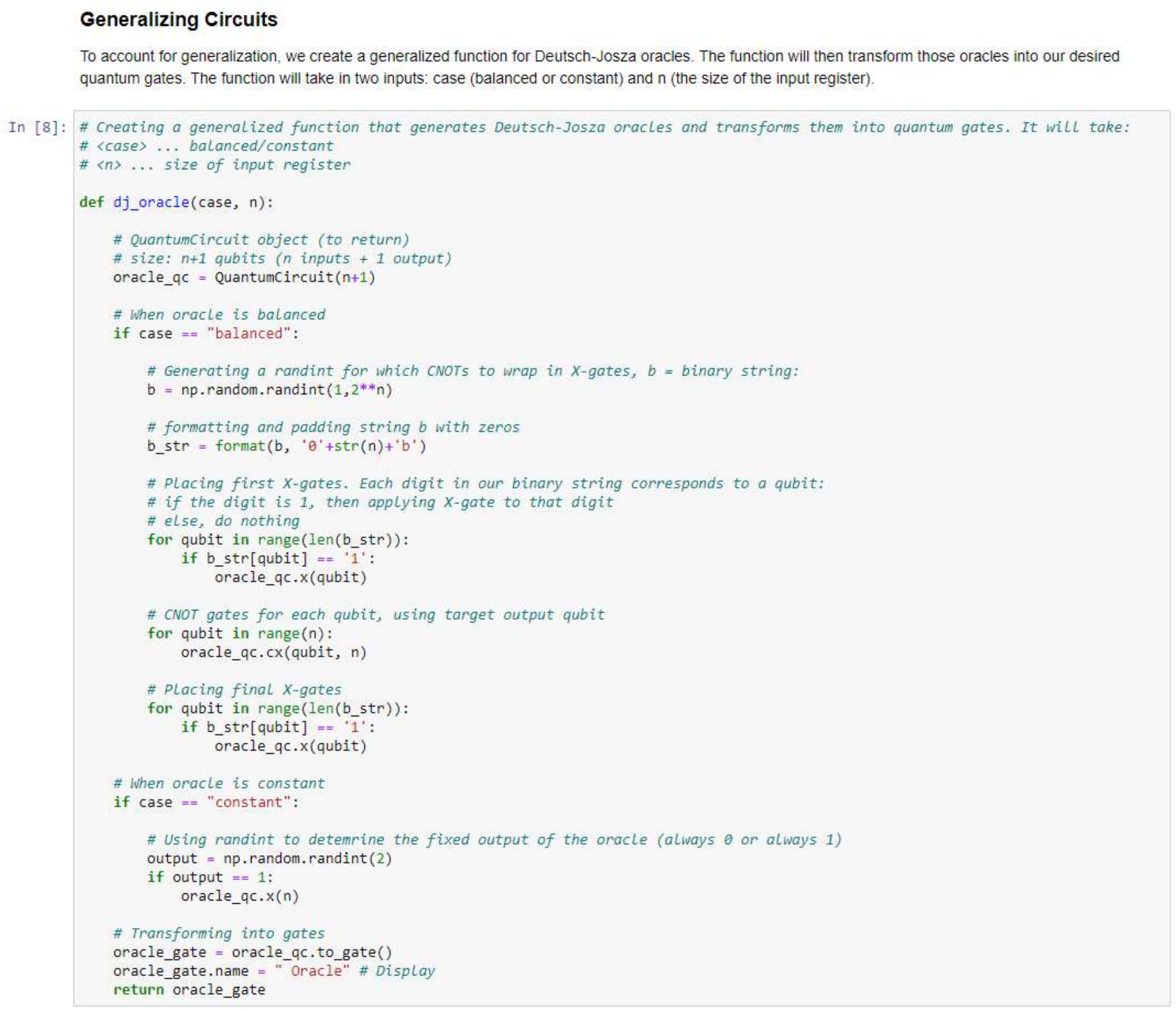
Out[7]:



Conclusion: we have a 0% chance of measuring 000. Which means the function is indeed balanced!

Ismail Akram
PHYS 315 Quantum Computing
Prof. Seth Cottrell

Generalizing Circuits



Generalizing Circuits

To account for generalization, we create a generalized function for Deutsch-Josza oracles. The function will then transform those oracles into our desired quantum gates. The function will take in two inputs: case (balanced or constant) and n (the size of the input register).

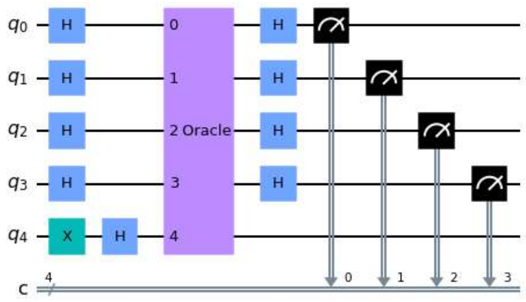
In [8]:
```python
# Creating a generalized function that generates Deutsch-Josza oracles and transforms them into quantum gates. It will take:
# <case> ... balanced/constant
# <n> ... size of input register

def dj_oracle(case, n):

    # QuantumCircuit object (to return)
    # size: n+1 qubits (n inputs + 1 output)
    oracle_qc = QuantumCircuit(n+1)

    # When oracle is balanced
    if case == "balanced":

        # Generating a randint for which CNOTs to wrap in X-gates, b = binary string:
        b = np.random.randint(1,2**n)

        # formatting and padding string b with zeros
        b_str = format(b, '0'+str(n)+'b')

        # Placing first X-gates. Each digit in our binary string corresponds to a qubit:
        # if the digit is 1, then applying X-gate to that digit
        # else, do nothing
        for qubit in range(len(b_str)):
            if b_str[qubit] == '1':
                oracle_qc.x(qubit)

        # CNOT gates for each qubit, using target output qubit
        for qubit in range(n):
            oracle_qc.cx(qubit, n)

        # Placing final X-gates
        for qubit in range(len(b_str)):
            if b_str[qubit] == '1':
                oracle_qc.x(qubit)

    # When oracle is constant
    if case == "constant":

        # Using randint to detemrine the fixed output of the oracle (always 0 or always 1)
        output = np.random.randint(2)
        if output == 1:
            oracle_qc.x(n)

    # Transforming into gates
    oracle_gate = oracle_qc.to_gate()
    oracle_gate.name = " Oracle" # Display
    return oracle_gate
```

Function that takes in the above function and applies Deutsch-Josza algorithm.

```
In [9]: # Function that takes in above oracle gate as input and applys Deutsch-Josza algorithm

        def dj_algorithm(oracle, n):
            dj_circuit = QuantumCircuit(n+1, n)

            # Output qubit
            dj_circuit.x(n)
            dj_circuit.h(n)

            # Input register
            for qubit in range(n):
                dj_circuit.h(qubit)

            # Appending aforementioned oracle gate on our circuit:
            dj_circuit.append(oracle, range(n+1))

            # Performing H-gates (again) and measuring:
            for qubit in range(n):
                dj_circuit.h(qubit)

            for i in range(n):
                dj_circuit.measure(i, i)

            return dj_circuit
```

```
In [10]: # Testing
         n = 4
         oracle_gate = dj_oracle('balanced', n)
         dj_circuit = dj_algorithm(oracle_gate, n)
         dj_circuit.draw('mpl')
```

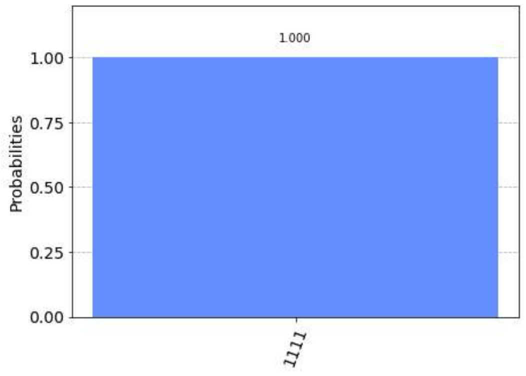
Out[10]:



Now let's see the results of executing this circuit.

In [11]:
```
# Plotting
transpiled_dj_circuit = transpile(dj_circuit, aer_sim)
qobj = assemble(transpiled_dj_circuit)
results = aer_sim.run(qobj).result()
answer = results.get_counts()
plot_histogram(answer)
```

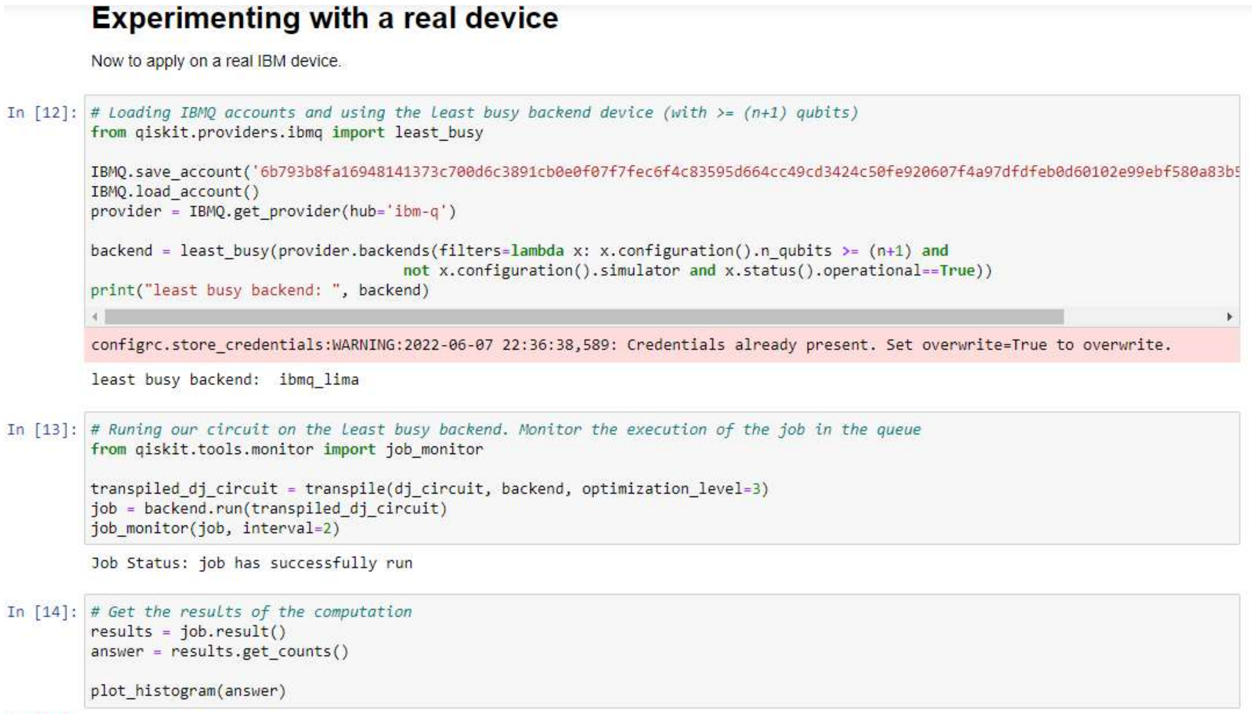
Out[11]:



Conclusion: and like before, we have a 0% chance of measuring 0000. Function is balanced and works!

Ismail Akram
PHYS 315 Quantum Computing
Prof. Seth Cottrell

Experiment with a Real Device



Experimenting with a real device

Now to apply on a real IBM device.

```
In [12]: # Loading IBMQ accounts and using the least busy backend device (with >= (n+1) qubits)
         from qiskit.providers.ibmq import least_busy

         IBMQ.save_account('6b793b8fa16948141373c700d6c3891cb0e0f07f7fec6f4c83595d664cc49cd3424c50fe920607f4a97dfdfeb0d60102e99ebf580a83b5
         IBMQ.load_account()
         provider = IBMQ.get_provider(hub='ibm-q')

         backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= (n+1) and
                                                not x.configuration().simulator and x.status().operational==True))
         print("least busy backend: ", backend)
```

```
configrc.store_credentials:WARNING:2022-06-07 22:36:38,589: Credentials already present. Set overwrite=True to overwrite.
```

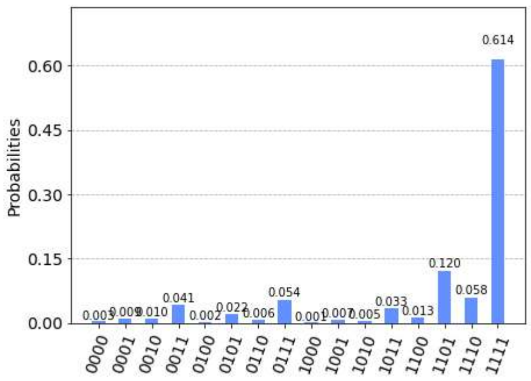
```
least busy backend:  ibmq_lima
```

```
In [13]: # Runing our circuit on the least busy backend. Monitor the execution of the job in the queue
         from qiskit.tools.monitor import job_monitor

         transpiled_dj_circuit = transpile(dj_circuit, backend, optimization_level=3)
         job = backend.run(transpiled_dj_circuit)
         job_monitor(job, interval=2)
```

```
Job Status: job has successfully run
```

```
In [14]: # Get the results of the computation
         results = job.result()
         answer = results.get_counts()

         plot_histogram(answer)
```

Out[14]:



Conclusion: interestingly, this time the most likely result is 1111. We don't have a single block histogram like before. This is due to the other results being errors in the quantum computation (i.e. theory vs reality of execution).

Even so, we know this function is balanced.

Ismail Akram
PHYS 315 Quantum Computing
Prof. Seth Cottrell

From You, One Year Ago

A sobering observation

To Ismail of the future, you can't travel into the past. Let go of wanting to do things differently. Make peace with where you are now and stop comparing yourself to others and a that romanticized alternative version of yourself. Know that the journey you went through in your formative years shaped who you are today. You now have a much better idea of what you want to pursue in your career. The you of the past would've been amazed by what you're studying now. That you plan on delving into Artificial Intelligence and Quantum Computing. And you have your whole journey ahead of you. So, keep learning.

Quantum Computing taught me to think about how we understand our universe from a new perspective. This humble science of the small has grand implications and I'll keep studying it and beyond. Thank you Professor Cottrell for your patience.

References

1. Lectures notes (namely Lecture 4)
2. Qiskit tutorial by Abdullah Amer
3. https://qiskit.org/
4. https://qiskit.org/documentation/intro_tutorial1.html
5. https://qiskit.org/textbook/ch-algorithms/deutsch-jozsa.html#4.-Qiskit-Implementation-

Uploaded Project link on Google Colab (Github):

https://github.com/IsmailAkram/IsmailAkram-Qiskit_Project_Ismail_Akram/blob/main/Qiskit_Project_Ismail_Akram.ipynb