# CSCI 4140
# Quantum Fourier Transform

Mark Green

Faculty of Science

Ontario Tech

# Introduction

- This video lecture presents the Qiskit implementation of the Quantum Fourier Transform (QFT)
- Recall what the QFT does:
  - When the input incrementally counts, the QFT converts these counts into angles
  - The demo in the Qiskit textbook demonstrates this nicely
- The inverse QFT goes in the opposite direction, it takes angles into a linear space

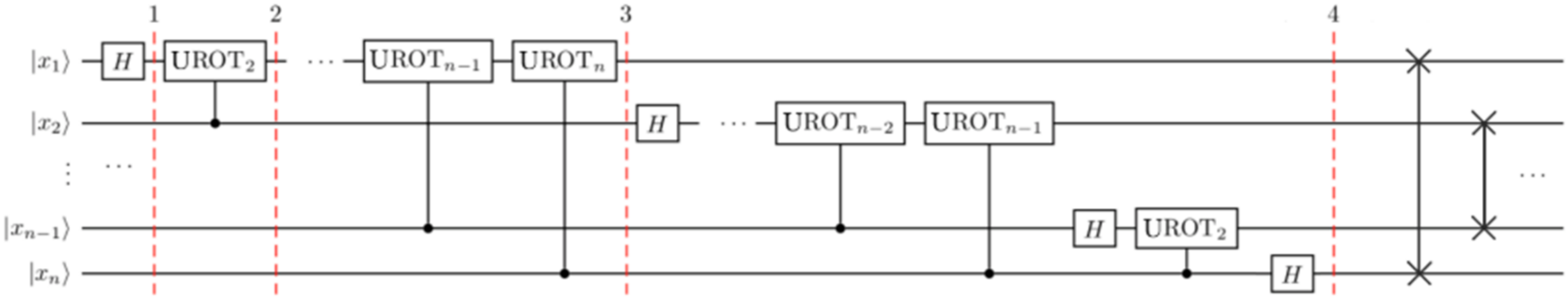
# Introduction

- In the lecture we showed that the QFT algorithm computes the following:

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^{n-1}}x\right)|1\rangle\right] \otimes \ldots$$
$$\otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^1}x\right)|1\rangle\right]$$

- Note that the coefficients in front of the |1> are rotations
- A number of swaps are required at the end to put the bits in the right order for the QFT
- A high level version of the algorithm is on the next slide

# Introduction



# Introduction

- The controlled UROT gate is an important part of this algorithm, it is defined in the following way:

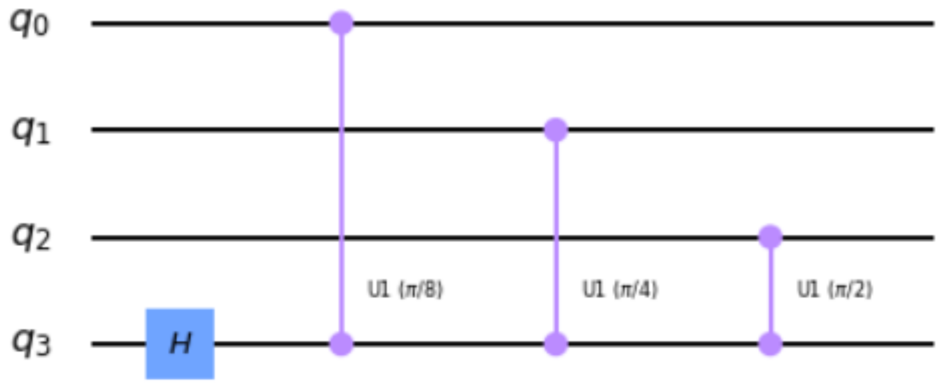
$$CROT_k = \begin{bmatrix} I & 0 \\ 0 & UROT_k \end{bmatrix}$$

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

- Note that the UROT gate is essentially a C1 gate with $\theta = {}^{2\pi}/_{2^k}$

# Implementation

- Since the QFT is generally useful it is worth while building a general procedure that will construct it, with the number of bit as a parameter
- We will build this function up one step at a time, starting with the rotation of a single bit



# Implementation

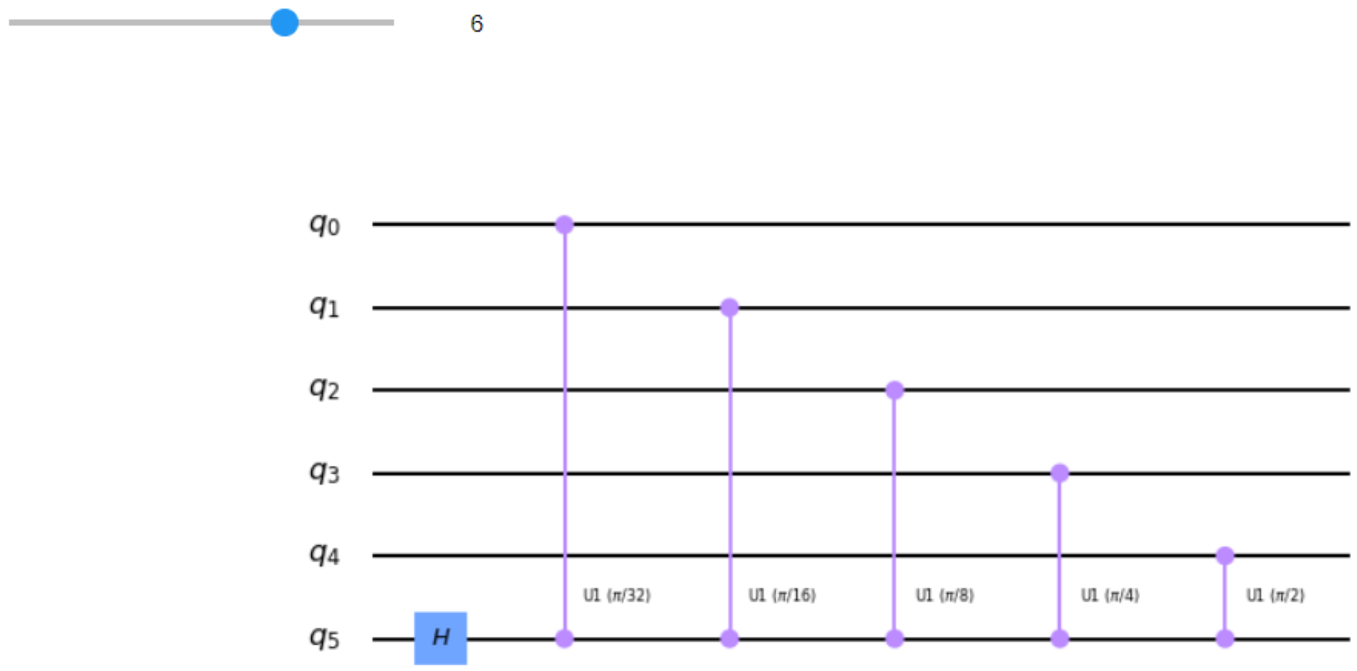
```
def qft_rotations(circuit, n):
    if n == 0: # Exit function if circuit is empty
        return circuit
    n -= 1 # Indexes start from 0
    circuit.h(n) # Apply the H-gate to the most significant qubit
    for qubit in range(n):
        # For each less significant qubit, we need to do a
        # smaller-angled controlled rotation:
        circuit.cu1(pi/2**(n-qubit), qubit, n)
```

```
qc = QuantumCircuit(4)
qft_rotations(qc,4)
qc.draw('mpl')
```

# Implementation

- An interesting way of viewing these circuits is to use the scalable_circuit function from the qiskit textbook:

```
from qiskit_textbook.widgets import scalable_circuit
scalable_circuit(qft_rotations)
```



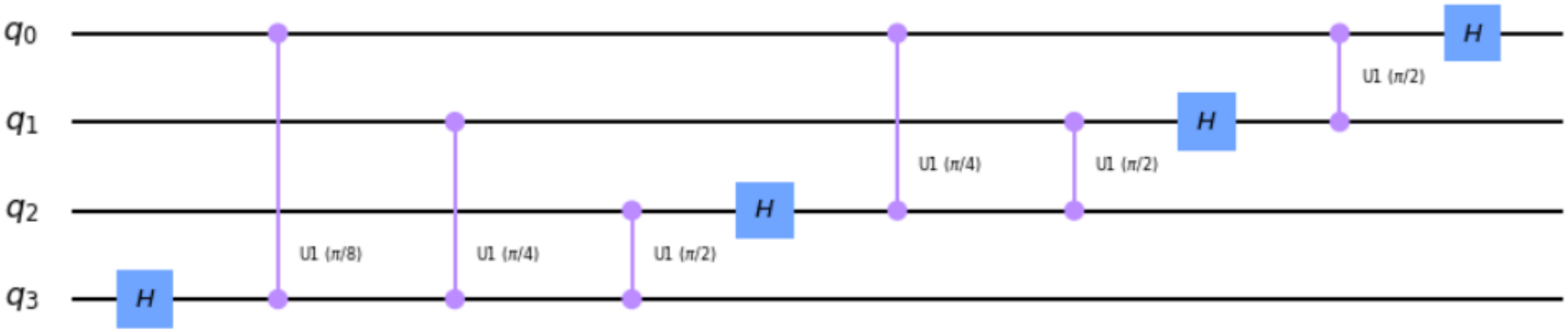
# Implementation

- This only gives us one bit, we need all the bits, the easiest way to do this is to call qft_rotations() recursively, it's already set up for that, we just need to add one more line of code

```
def qft_rotations(circuit, n):
    """Performs qft on the first n qubits in circuit (without swaps)"""
    if n == 0:
        return circuit
    n -= 1
    circuit.h(n)
    for qubit in range(n):
        circuit.cu1(pi/2**(n-qubit), qubit, n)
    # At the end of our function, we call the same function again on
    # the next qubits (we reduced n by one earlier in the function)
    qft_rotations(circuit, n)

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

# Implementation



# Implementation

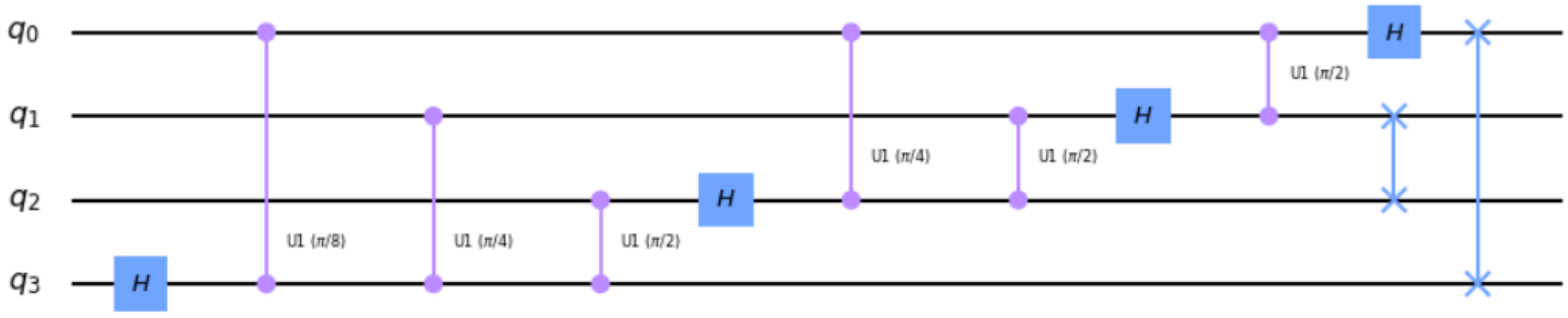
- Now all we need to do is add the swaps, another function, and put it all together

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

def qft(circuit, n):
    """QFT on the first n qubits in circuit"""
    qft_rotations(circuit, n)
    swap_registers(circuit, n)
    return circuit

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

# Implementation

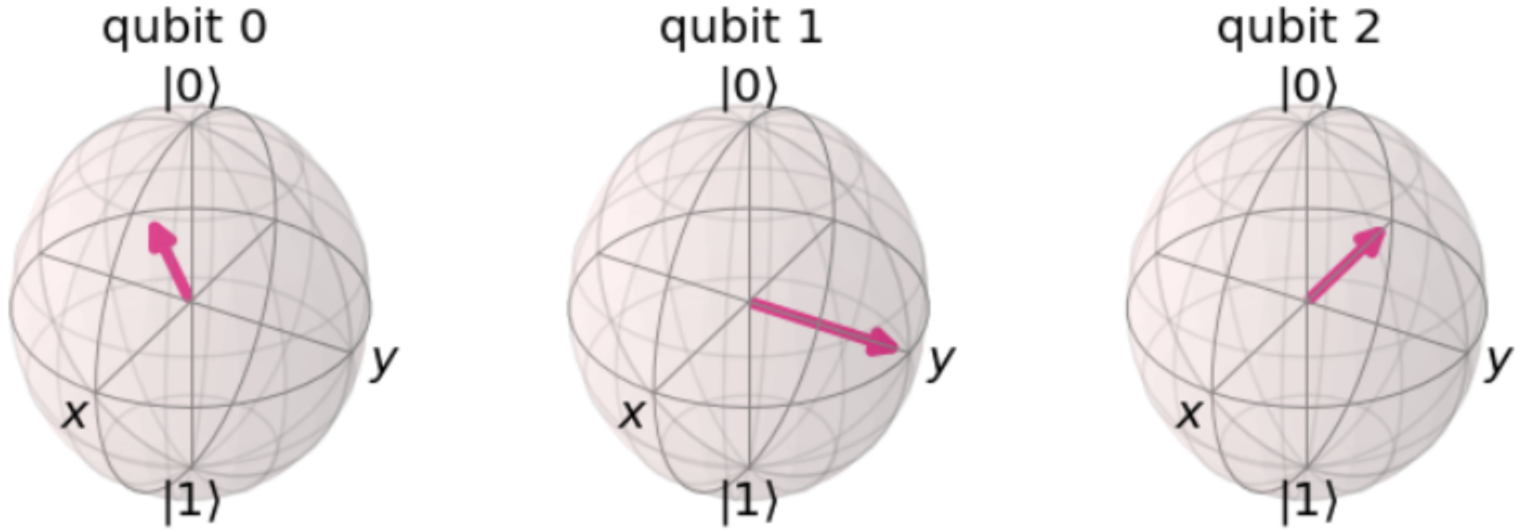


# Example

- To see how this all works we will take the QFT of 5
- The code for this is:

```
qc=QuantumCircuit(3)
# encode the number 5
qc.x(0)
qc.x(2)
# perform QFT
qft(qc,3)
# simulate to get the results
backend = Aer.get_backend("statevector_simulator")
statevector = execute(qc, backend=backend).result().get_statevector()
plot_bloch_multivector(statevector)
```

# Example



# Inverse QFT

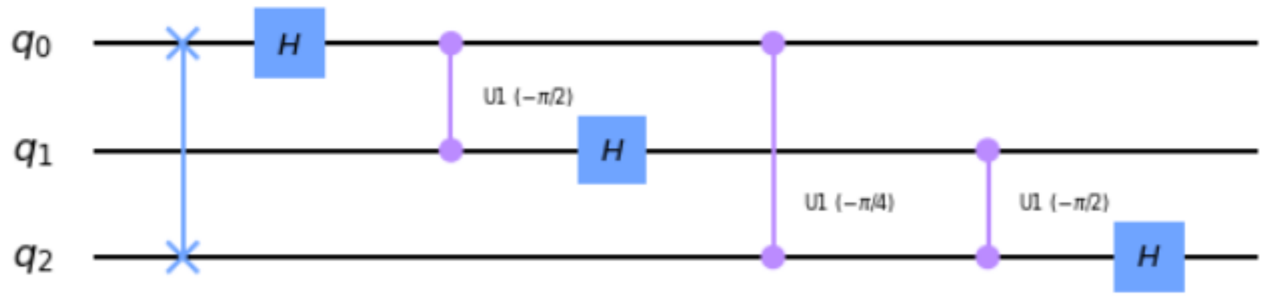
- It turns out that is the right result
- We also need the inverse QFT so lets add that to our collection of tools
- Since the inverse QFT reverses the effect of the QFT all we need to do is reverse the gate order

```python
def qft_dagger(circ, n):
    """n-qubit QFTdagger the first n qubits in circ"""
    # Don't forget the Swaps!
    for qubit in range(n//2):
        circ.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            circ.cu1(-pi/float(2**(j-m)), m, j)
        circ.h(j)
```

# Inverse QFT

- Note that in this case we aren't using recursion, we could write our qft the same way

```
qc=QuantumCircuit(3)
qft_dagger(qc,3)
qc.draw('mpl')
```

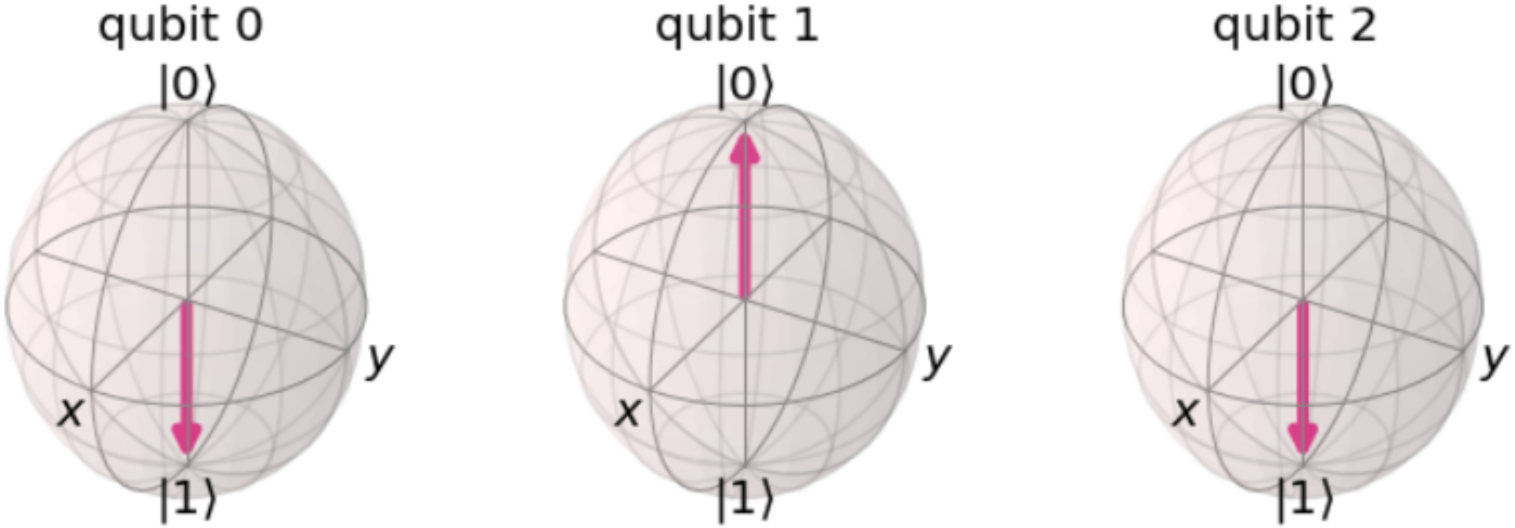


# Inverse QFT

- Now add the inverse to our example and see what the result is

```
qc=QuantumCircuit(3)
# encode the number 5
qc.x(0)
qc.x(2)
# perform QFT
qft(qc,3)
# now apply the inverse QFT
qft_dagger(qc,3)
# simulate to get the results
backend = Aer.get_backend("statevector_simulator")
statevector = execute(qc, backend=backend).result().get_statevector()
plot_bloch_multivector(statevector)
```

# Inverse QFT

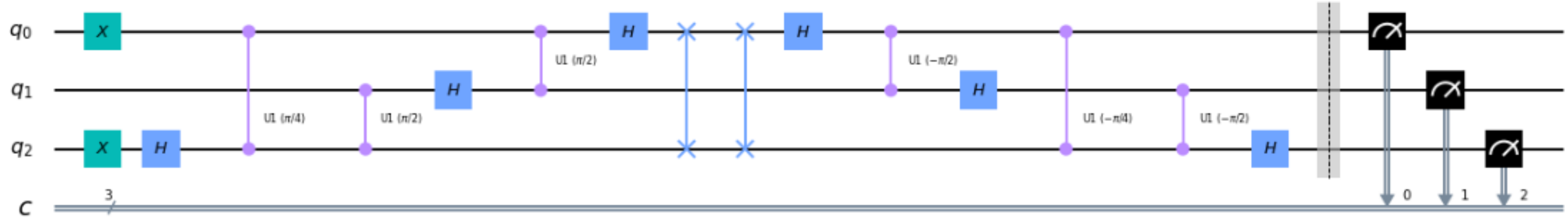


# Real Quantum Computer

- This is nice in theory, but let's run it on a real quantum computer
- We need to make a slight change to the circuit to add measurements

```
qc=QuantumCircuit(3,3)
# encode the number 5
qc.x(0)
qc.x(2)
# perform QFT
qft(qc,3)
# now apply the inverse QFT
qft_dagger(qc,3)
qc.barrier()
qc.measure([0,1,2], [0,1,2])
# simulate to get the results
#backend = Aer.get_backend("statevector_simulator")
#statevector = execute(qc, backend=backend).result().get_statevector()
#plot_bloch_multivector(statevector)
qc.draw()
```

# Real Quantum Computer




```
nqubits=3
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= nqubits
                                       and not x.configuration().simulator
                                       and x.status().operational==True))
print("least busy backend: ", backend)
```

```
least busy backend:  ibmqx2
```

# Real Quantum Computer

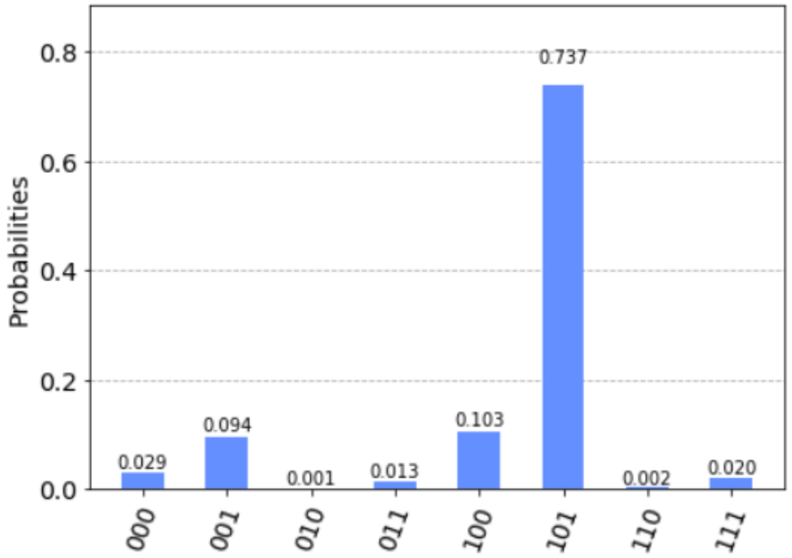
- The code on the previous slide shows how to retrieve the least busy quantum computer
- The next slide shows a run on this quantum computer
- The job was in the queue for several minutes and it took about 10 seconds to run it
- The correct answer 101 was obtained over 70% of the time, which is typical for existing quantum computers
- The correct answer is the most likely, so it is the one that would be selected

# Real Quantum Computer

```
shots = 2048
job = execute(qc, backend=backend, shots=shots, optimization_level=3)
job_monitor(job)
```

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

```
counts = job.result().get_counts()
plot_histogram(counts)
```



# Summary

- Reviewed the QFT
- Shows how the QFT and its inverse can be implemented
- Have two new tools for quantum computing, packaged as functions that we can use in other algorithms
- Show how our circuits can be run on a real quantum computer