1. Installation

In [5]:

```
import numpy as np
from numpy import pi

from qiskit import QuantumCircuit, transpile, assemble, Aer, IBMQ
from qiskit.providers.ibmq import least_busy
from qiskit.tools.monitor import job_monitor
from qiskit.visualization import plot_histogram, plot_bloch_multivector
```

2. Circuit

In [6]:

```
qc = QuantumCircuit(3)

#Qiskit's least significant bit has the lowest index (0), thus the circuit will be mirrored through the horizontal
qc.h(2)

#Next, we want to turn this an extra quarter turn if qubit 1 is in the state |1>

qc.cp(pi/2, 1, 2)   # CROT from qubit 1 to qubit 2

#And another eighth turn if the least significant qubit (0) is |1>

qc.cp(pi/4, 0, 2)   # CROT from qubit 2 to qubit 0
```

Out[6]:

```
<qiskit.circuit.instructionset.InstructionSet at 0x24874a6a7f0>
```

With that qubit taken care of, we can now ignore it and repeat the process, using the same logic for qubits 0 and 1

In [7]:

```
qc.h(1)
qc.cp(pi/2, 0, 1)  # CROT from qubit 0 to qubit 1
qc.h(0)
```

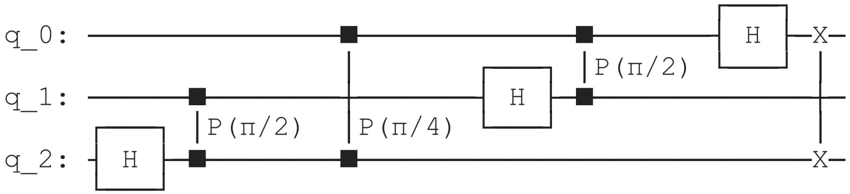
Out[7]:

```
<qiskit.circuit.instructionset.InstructionSet at 0x24820919ac0>
```

Finally we must swap the qubits 0 and 2 to complete the QFT:

In [8]:
```
qc.swap(0,2)
qc.draw()
```

Out[8]:



3. General Circuit

QFT

In [11]:
```
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.cp(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)
```

Swap

In [14]:
```
def swap_registers(circuit, n):
    for qubit in range(n//2):
        circuit.swap(qubit, n-qubit-1)
    return circuit
```

General circuit

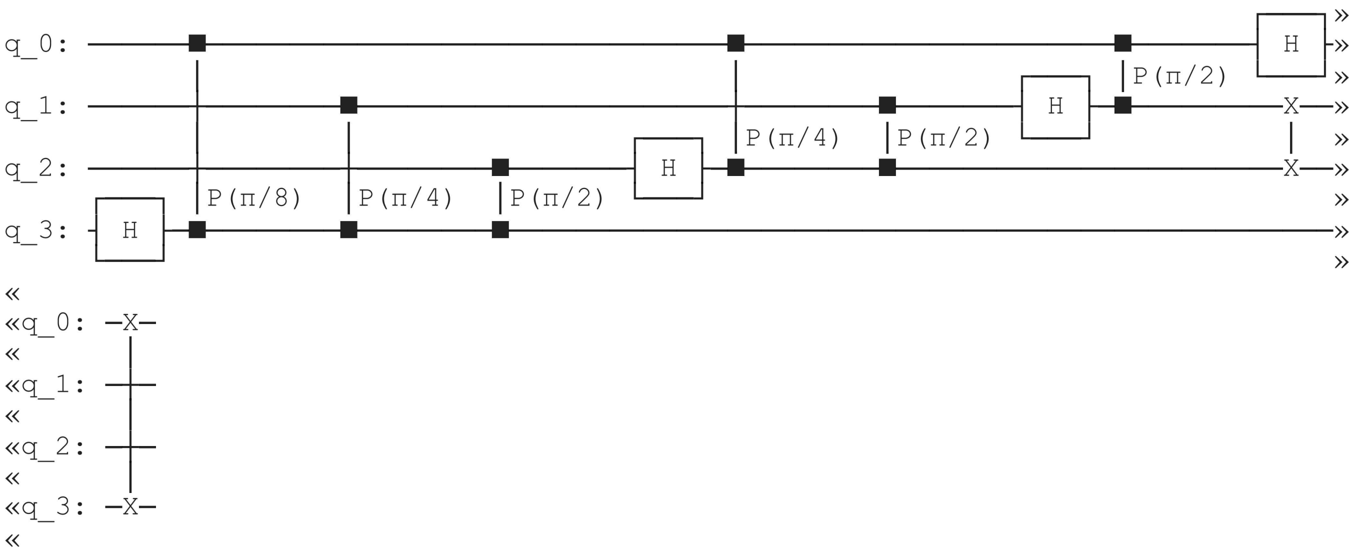
In [15]:

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

In [17]:

```python
qc1 = QuantumCircuit(4)
qft(qc1,4)
qc1.draw()
```

Out[17]:



4. Verifying Circuit (Simulator)

We must first encode a number in the computational basis, here we are selecting '5' which in binary is '101'

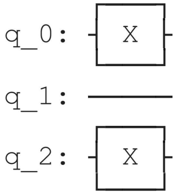
In [19]:

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

# Encode the state 5
qc.x(0)        # since we need '1' at first qubit and at last qubit
```

```
qc.x(2)
qc.draw()
```

Out[19]:



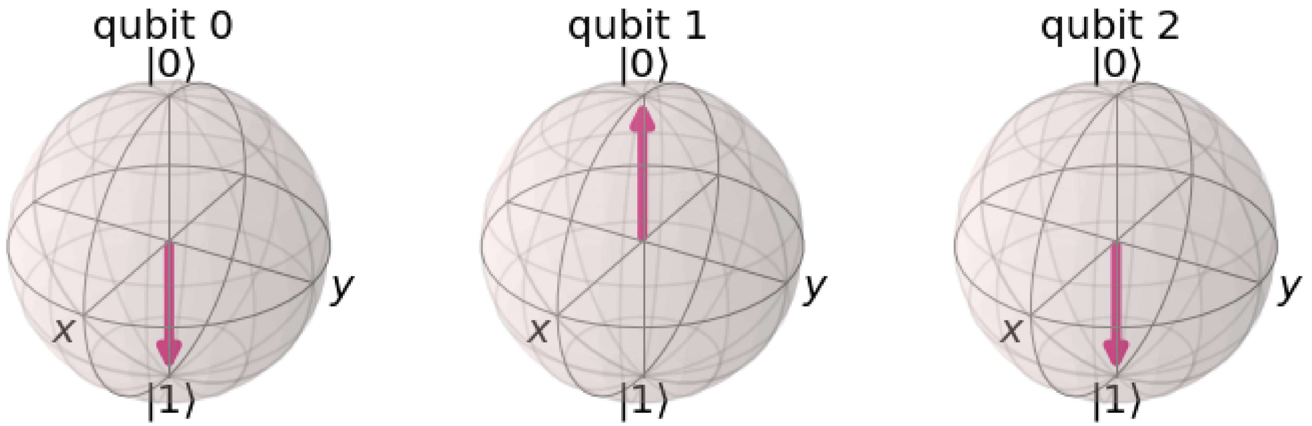
In [31]:

```
# And let's check the qubit's states using the aer simulator:

sim = Aer.get_backend("aer_simulator")
qc_init = qc.copy()     # making a copy so that we can work on the original one
qc_init.save_statevector()
statevector = sim.run(qc_init).result().get_statevector()
plot_bloch_multivector(statevector)

# we can see the state below as '101'
```

Out[31]:

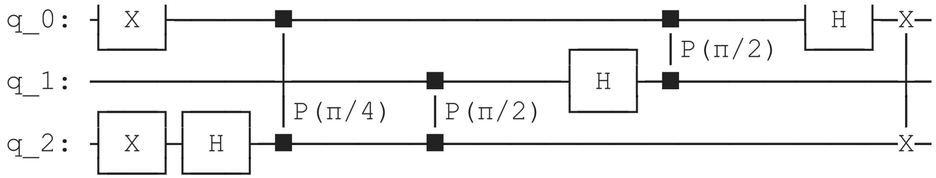


Finally, let's use our QFT function and view the final state of our qubits:

In [22]:

```
qft(qc,3)
qc.draw()
```

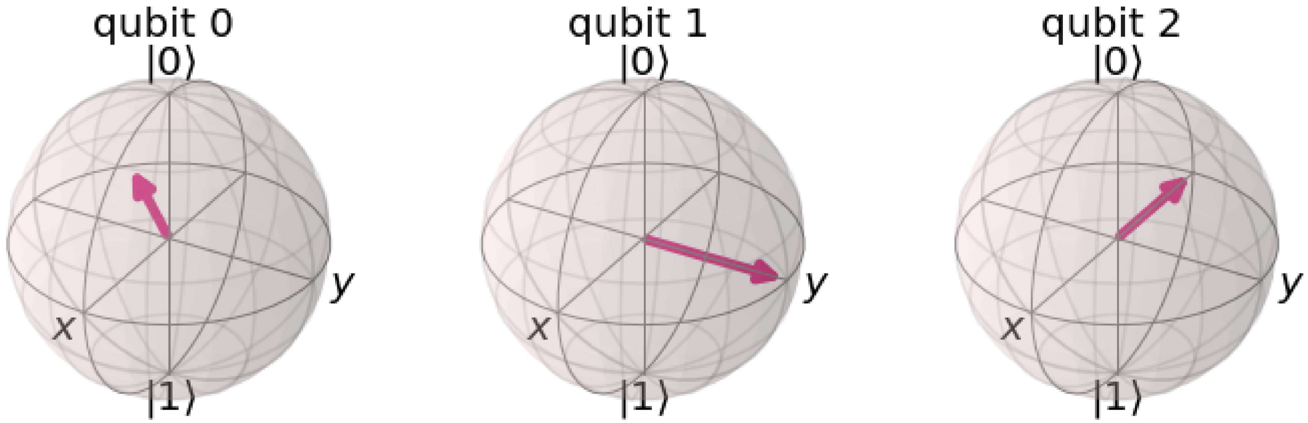
Out[22]:



In [23]:

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

Out[23]:



We can see out QFT function has worked correctly. Compared the state $|\tilde{0}\rangle = |{+}{+}{+}\rangle$, Qubit 0 has been rotated by $\frac{5}{8}$ of a full turn, qubit 1 by $\frac{10}{8}$ full turns (equivalent to $\frac{1}{4}$ of a full turn), and qubit 2 by $\frac{20}{8}$ full turns (equivalent to $\frac{1}{2}$ of a full turn).

5. Real Device

If we want to demonstrate and investigate the QFT working on real hardware, we can instead create the state |5>, run the QFT in reverse, and verify the output is the state |5> as expected

Inverse QFT Operation

In [24]:
```python
def inverse_qft(circuit, n):
    """Does the inverse QFT on the first n qubits in circuit"""
    # First we create a QFT circuit of the correct size:
    qft_circ = qft(QuantumCircuit(n), n)
    # Then we take the inverse of this circuit
    invqft_circ = qft_circ.inverse()
    # And add it to the first n qubits in our existing circuit
    circuit.append(invqft_circ, circuit.qubits[:n])
    return circuit.decompose() # .decompose() allows us to see the individual gates
```

Putting the qubit in the state |5>

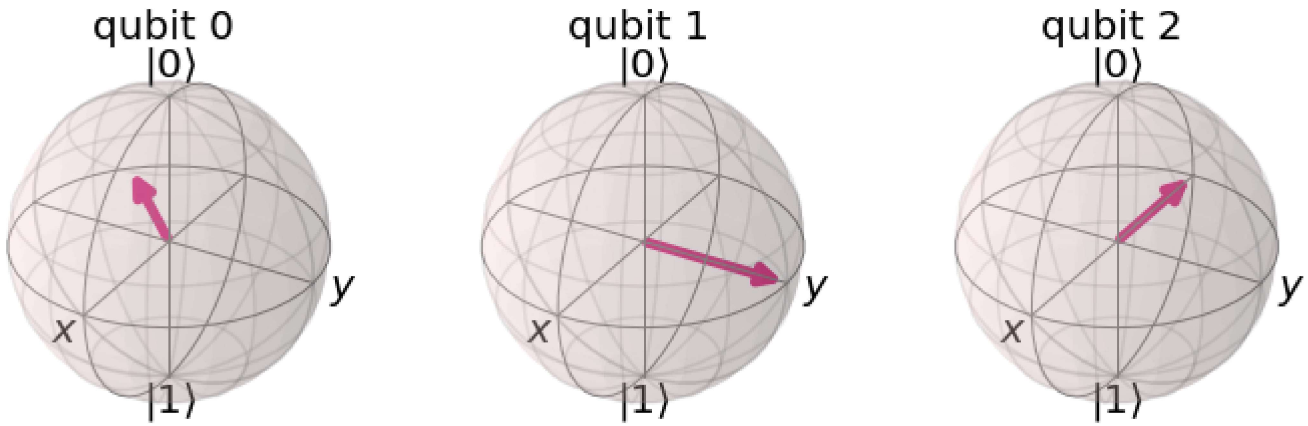
In [25]:
```python
nqubits = 3
number = 5
qc = QuantumCircuit(nqubits)
for qubit in range(nqubits):
    qc.h(qubit)
qc.p(number*pi/4,0)
qc.p(number*pi/2,1)
qc.p(number*pi,2)

qc.draw()
```

Out[25]:
```
     ┌───┐┌─────────┐
q_0: ┤ H ├┤ P(5π/4) ├
     ├───┤├─────────┤
q_1: ┤ H ├┤ P(5π/2) ├
     ├───┤└┬───────┬┘
q_2: ┤ H ├─┤ P(5π) ├─
     └───┘ └───────┘
```

In [26]:
```python
qc_init = qc.copy()
qc_init.save_statevector()
sim = Aer.get_backend("aer_simulator")
statevector = sim.run(qc_init).result().get_statevector()
plot_bloch_multivector(statevector)
```

Out[26]:



Apply Inverse QFT

In [27]:

```
qc = inverse_qft(qc, nqubits)
qc.measure_all()
qc.draw()
```

Out[27]:

```
         ┌─────────┐ ┌──────────────┐    ┌───┐                                      »
  q_0: ─┤ U2(0,π) ├─┤ U(0,0,5π/4) ├─X──┤ H ├─■─────────────────────────────■─────»
         ├─────────┤ ├──────────────┤ │  └───┘ │P(-π/2) ┌───┐                │      »
  q_1: ─┤ U2(0,π) ├─┤ U(0,0,5π/2) ├─┼────────■────────┤ H ├─■──────────────┼─────»
         ├─────────┤ └┬────────────┬┘ │                 └───┘ │P(-π/2)      │P(-π/4) »
  q_2: ─┤ U2(0,π) ├──┤ U(0,0,5π) ├──X──────────────────────────■──────────────■─────»
         └─────────┘  └────────────┘                                               »
meas: 3/═══════════════════════════════════════════════════════════════════════════»
                                                                                    »
«                 ░ ┌─┐
«   q_0: ─────────░─┤M├──────
«                 ░ └╥┘┌─┐
«   q_1: ─────────░──╫─┤M├───
«        ┌───┐   ░  ║ └╥┘┌─┐
«   q_2: ┤ H ├───░──╫──╫─┤M├
«        └───┘   ░  ║  ║ └╥┘
«meas: 3/═══════════╩══╩══╩═
«                   0  1  2
```

Real Device

In [28]:

```
# Load our saved IBMQ accounts and get the least busy backend device with less than or equal to nqubits
```

```
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:  ibmq_bogota
```

In [29]:

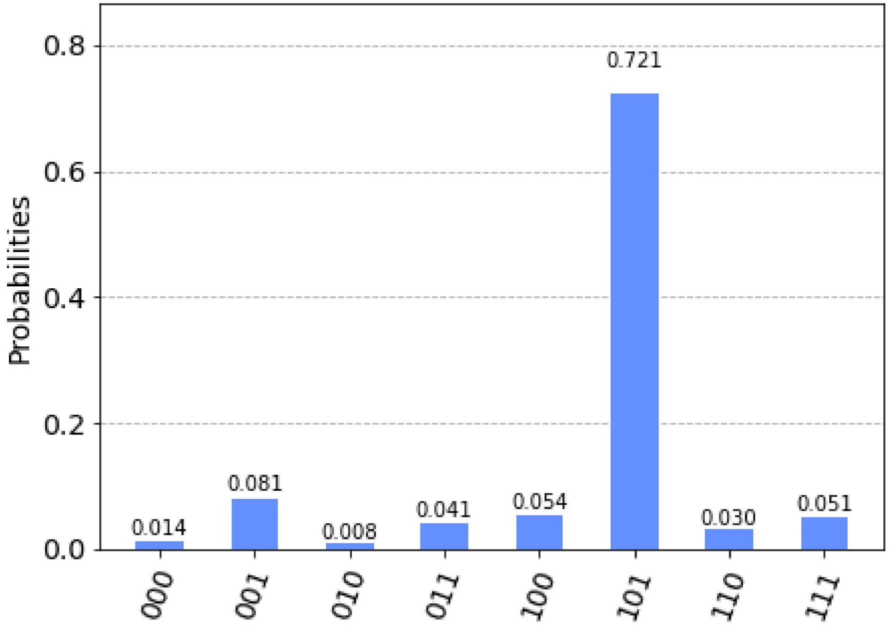
```
shots = 2048
transpiled_qc = transpile(qc, backend, optimization_level=3)
job = backend.run(transpiled_qc, shots=shots)
job_monitor(job)
```

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

In [30]:

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

Out[30]:



We (hopefully) see that the highest probability outcome is 101

In [ ]: