Vivek Vittal Biragoni

211AI041

Assignment 5

QUANTUM WALK SEARCH ALGORITHM

A quantum algorithm called quantum walk search can be used to look for a specific item in an unsorted database. It includes searching the database faster than classical algorithms by using a quantum version of a classical random walk. A quantum oracle is a mysterious device that applies a particular operation to a quantum state. When using a quantum walk search, the quantum oracle is employed to ascertain whether the object being sought is present in the database.

Grover's algorithm is a quantum algorithm that uses fewer queries than a conventional algorithm to search an unsorted database. The programme searches for the object more quickly by conducting a sequence of operations on a quantum superposition of all potential search states.

The Quantum Walk Search problem can be resolved by using Grover's algorithm and the quantum oracle in the following ways:

--> Construct a quantum superposition including each potential search state.

--> Use the quantum oracle to check the database to see if the item being searched for is there. The oracle flips the sign of the associated search state if the item is present in the database.

--> Use Grover's technique to increase the search state's amplitude so that it more closely matches the object being sought after.

--> Repeat steps 2 and 3 up till a high probability search state is located.

--> To get the search result, measure the final state.

In conclusion, Quantum Walk Search efficiently searches an unsorted database by combining Grover's method and the quantum oracle. Comparing the approach to traditional search algorithms, a quadratic speedup may be possible.

In [14]:
```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile
from qiskit.tools.jupyter import *
from qiskit.visualization import *
```

```
from ibm_quantum_widgets import *
from qiskit_aer import AerSimulator

# qiskit-ibmq-provider has been deprecated.
# Please see the Migration Guides in https://ibm.biz/provider_migration_guide for m
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Session, O

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

# Invoke a primitive inside a session. For more details see https://qiskit.org/docu
# with Session(backend=service.backend("ibmq_qasm_simulator")):
#     result = Sampler().run(circuits).result()
```

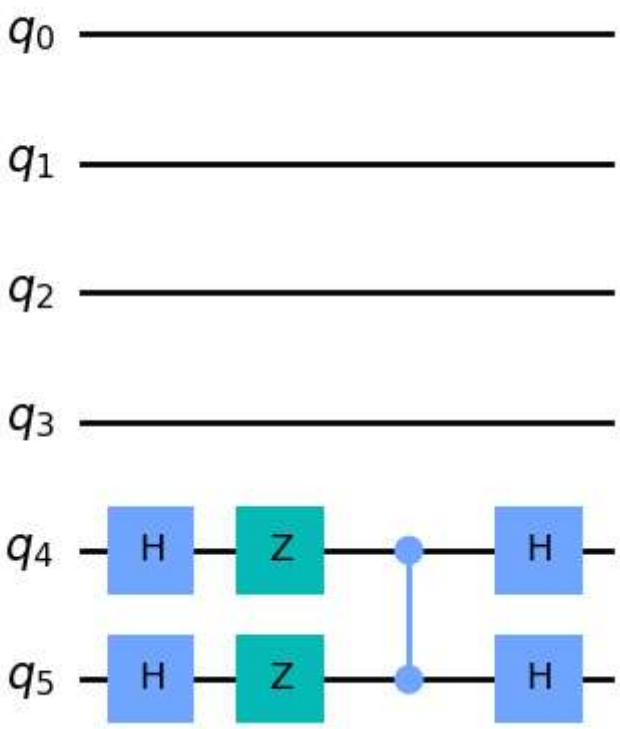
In [15]:
```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, execute, Aer, IBMQ, QuantumRegister, ClassicalRe
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from qiskit.circuit.library import QFT
from numpy import pi
from qiskit.quantum_info import Statevector
from matplotlib import pyplot as plt
import numpy as np
# Loading your IBM Q account(s)
provider = IBMQ.load_account()
```

```
ibmqfactory.load_account:WARNING:2023-04-26 17:00:29,027: Credentials are already
in use. The existing account in the session will be replaced.
```

A Grover coin, which serves as the diffuser in Grover's algorithm, will be represented by two of the circuit's six qubits, giving it a total of six qubits. We put this into practise first.

In [16]:
```
one_step_circuit = QuantumCircuit(6, name=' ONE STEP')
# Coin operator
one_step_circuit.h([4,5])
one_step_circuit.z([4,5])
one_step_circuit.cz(4,5)
one_step_circuit.h([4,5])
one_step_circuit.draw()
```

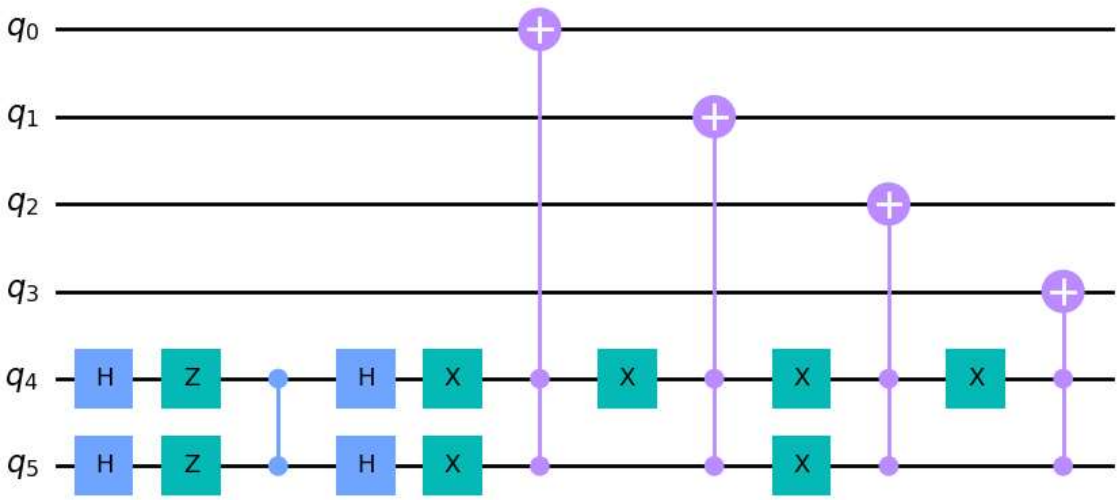
Out[16]:



Let's use the shift operator now. We are aware that the walker can only move to nodes that are close to one another and that these nodes can only differ by a single bit. Applying a NOT gate to one of the node qubits causes the walker to be moved in accordance with the flip of the coin. We shift the walker to the state where the first node qubit differs if the coin is in that state. The walker goes to the state where the second and third qubits, respectively, differ if the coin is or. Finally, we flip the fourth qubit if the Grover coin is. After the Grover coin, we implement this using CCNOT- and NOT gates. They make up one step of a four-dimensional hypercube quantum walk when taken together.

In [17]:
```
# Shift operator function for 4d-hypercube
def shift_operator(circuit):
    for i in range(0,4):
        circuit.x(4)
        if i%2==0:
            circuit.x(5)
        circuit.ccx(4,5,i)

shift_operator(one_step_circuit)

one_step_gate = one_step_circuit.to_instruction()
one_step_circuit.draw()
```

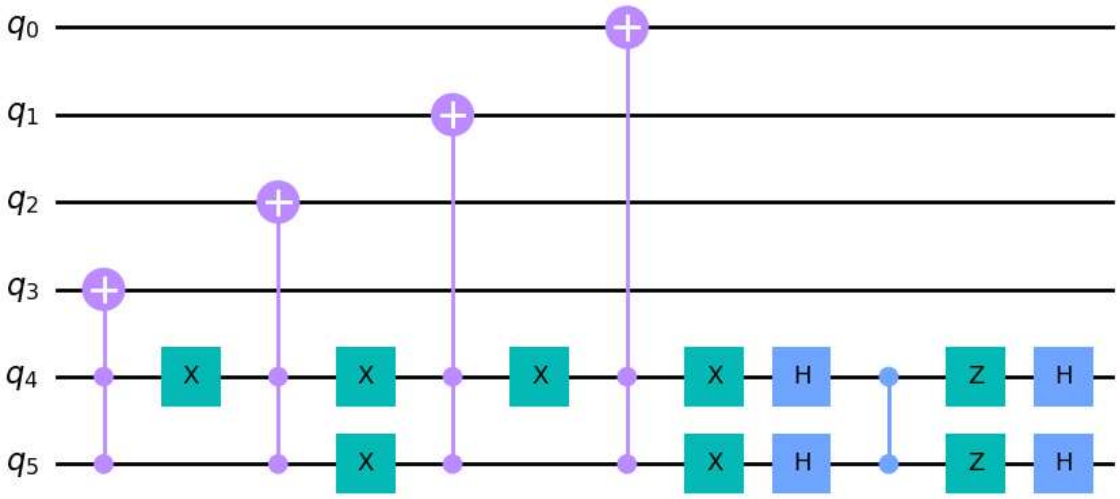
Out[17]:



4-dimensional hypercube quantum walk search

In [18]:
```
one_step_circuit.inverse().draw()
```

Out[18]:



Later, the phase estimation will be reversed using the inversed one step gate. Making controlled gates from both the one step gate and the two step gate

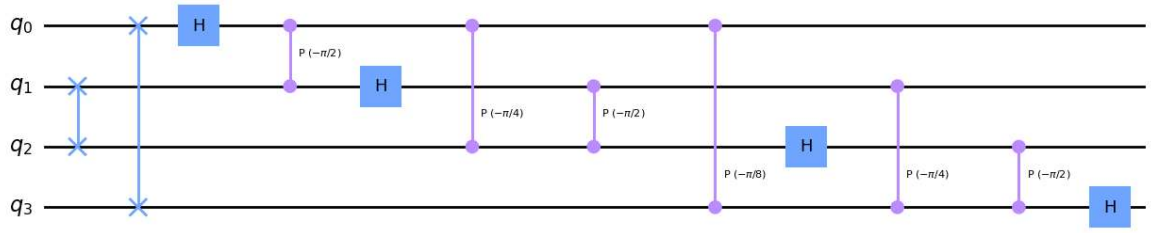
In [19]:
```
# Make controlled gates
inv_cont_one_step = one_step_circuit.inverse().control()
inv_cont_one_step_gate = inv_cont_one_step.to_instruction()
cont_one_step = one_step_circuit.control()
cont_one_step_gate = cont_one_step.to_instruction()
```

In the phase estimate, both the controlled one step gate and the controlled inversed one step gate will be applied. The Quantum Fourier Transform will also be used in the phase estimation process. The Quantum Fourier Transform is implemented via the Qiskit function

QFT. The inverse Quantum Fourier Transform is used for phase estimation, but we also need to employ the regular QFT to reverse the phase estimate.

In [20]:
```
inv_qft_gate = QFT(4, inverse=True).to_instruction()
qft_gate = QFT(4, inverse=False).to_instruction()

QFT(4, inverse=True).decompose().draw("mpl")
```

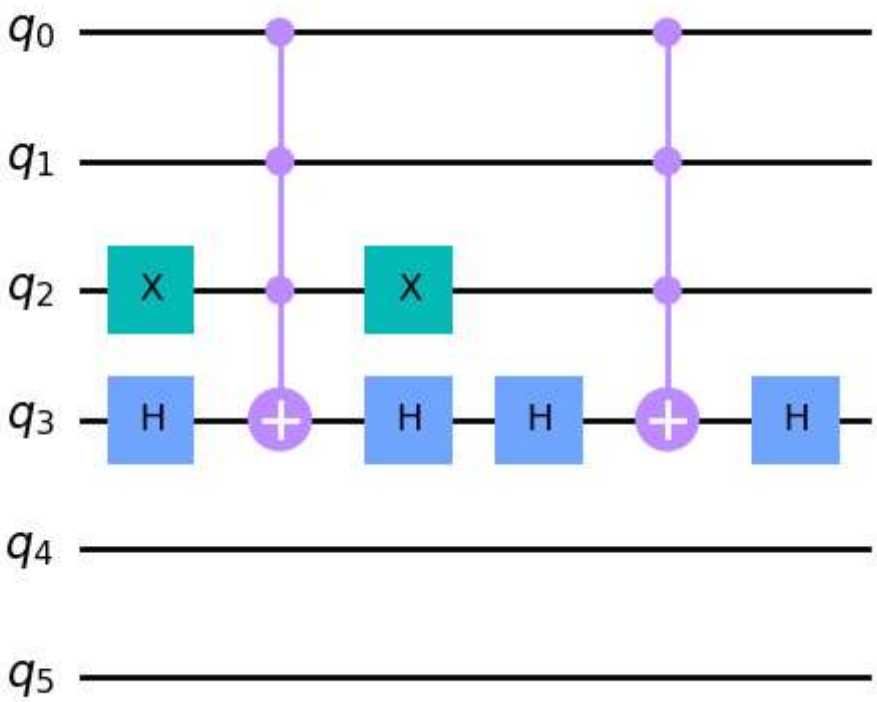
Out[20]:



A phase oracle that marks the states 1011 and 1111 is implemented before the phase estimation.

In [21]:
```
phase_circuit =  QuantumCircuit(6, name=' phase oracle ')
# Mark 1011
phase_circuit.x(2)
phase_circuit.h(3)
phase_circuit.mct([0,1,2], 3)
phase_circuit.h(3)
phase_circuit.x(2)
# Mark 1111
phase_circuit.h(3)
phase_circuit.mct([0,1,2],3)
phase_circuit.h(3)
phase_oracle_gate = phase_circuit.to_instruction()
# Phase oracle circuit
phase_oracle_circuit =  QuantumCircuit(11, name=' PHASE ORACLE CIRCUIT ')
phase_oracle_circuit.append(phase_oracle_gate, [4,5,6,7,8,9])
phase_circuit.draw()
```

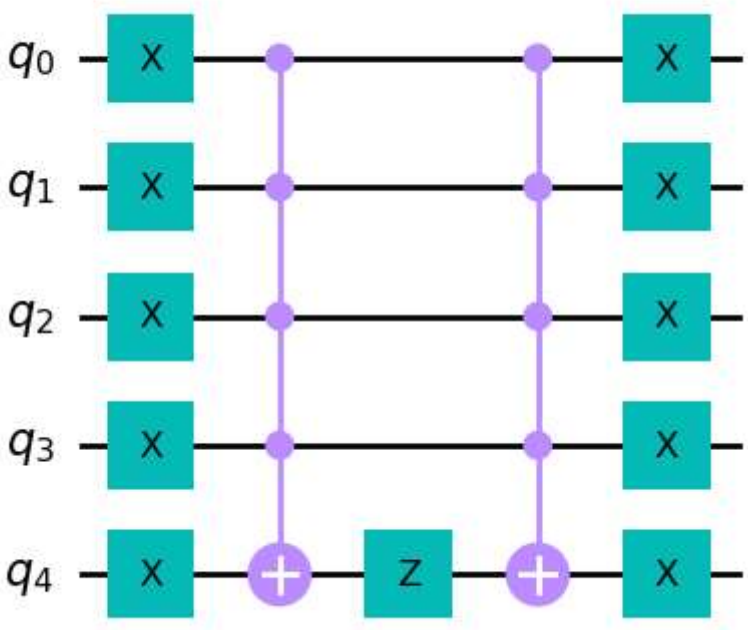
Out[21]:



Now that the other qubits are not zero, we will design a gate that rotates an auxiliary qubit. This gate will rotate the auxiliary qubit during phase estimation if 0 is true.

In [22]:
```
# Mark q_4 if the other qubits are non-zero
mark_auxiliary_circuit = QuantumCircuit(5, name=' mark auxiliary ')
mark_auxiliary_circuit.x([0,1,2,3,4])
mark_auxiliary_circuit.mct([0,1,2,3], 4)
mark_auxiliary_circuit.z(4)
mark_auxiliary_circuit.mct([0,1,2,3], 4)
mark_auxiliary_circuit.x([0,1,2,3,4])

mark_auxiliary_gate = mark_auxiliary_circuit.to_instruction()
mark_auxiliary_circuit.draw()
```

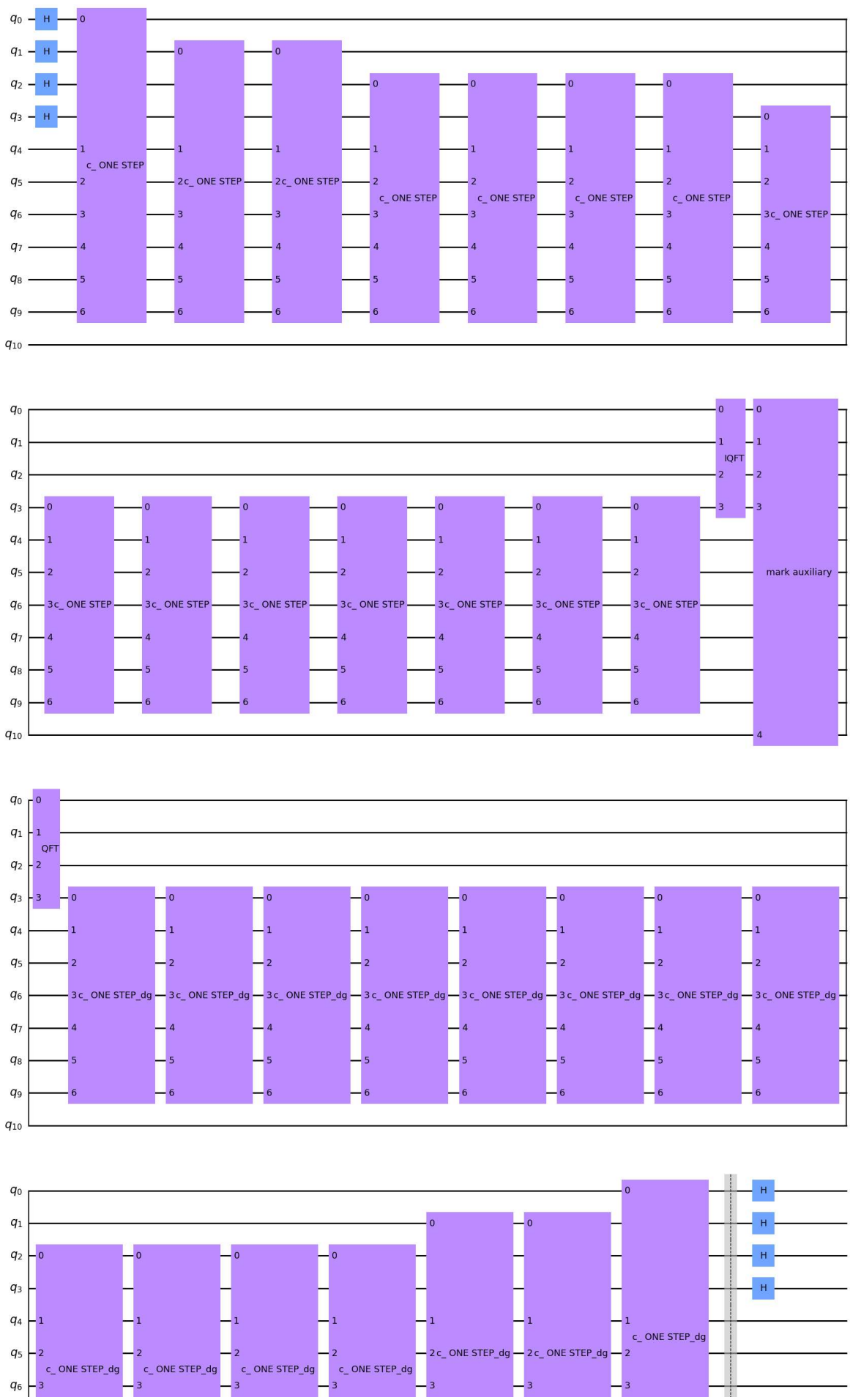
Out[22]:



Phase estimation for the first step of the quantum walk is followed by the rotation of an auxiliary qubit if 0 in this step. We employ the mark_auxiliary_gate we just made for this. Then, we do the phase estimation in reverse.

In [23]:
```
# Phase estimation
phase_estimation_circuit = QuantumCircuit(11, name=' phase estimation ')
phase_estimation_circuit.h([0,1,2,3])
for i in range(0,4):
    stop = 2**i
    for j in range(0,stop):
        phase_estimation_circuit.append(cont_one_step, [i,4,5,6,7,8,9])

# Inverse fourier transform
phase_estimation_circuit.append(inv_qft_gate, [0,1,2,3])

# Mark all angles theta that are not 0 with an auxiliary qubit
phase_estimation_circuit.append(mark_auxiliary_gate, [0,1,2,3,10])

# Reverse phase estimation
phase_estimation_circuit.append(qft_gate, [0,1,2,3])

for i in range(3,-1,-1):
    stop = 2**i
    for j in range(0,stop):
        phase_estimation_circuit.append(inv_cont_one_step, [i,4,5,6,7,8,9])
phase_estimation_circuit.barrier(range(0,10))
phase_estimation_circuit.h([0,1,2,3])

# Make phase estimation gate
phase_estimation_gate = phase_estimation_circuit.to_instruction()
phase_estimation_circuit.draw()
```

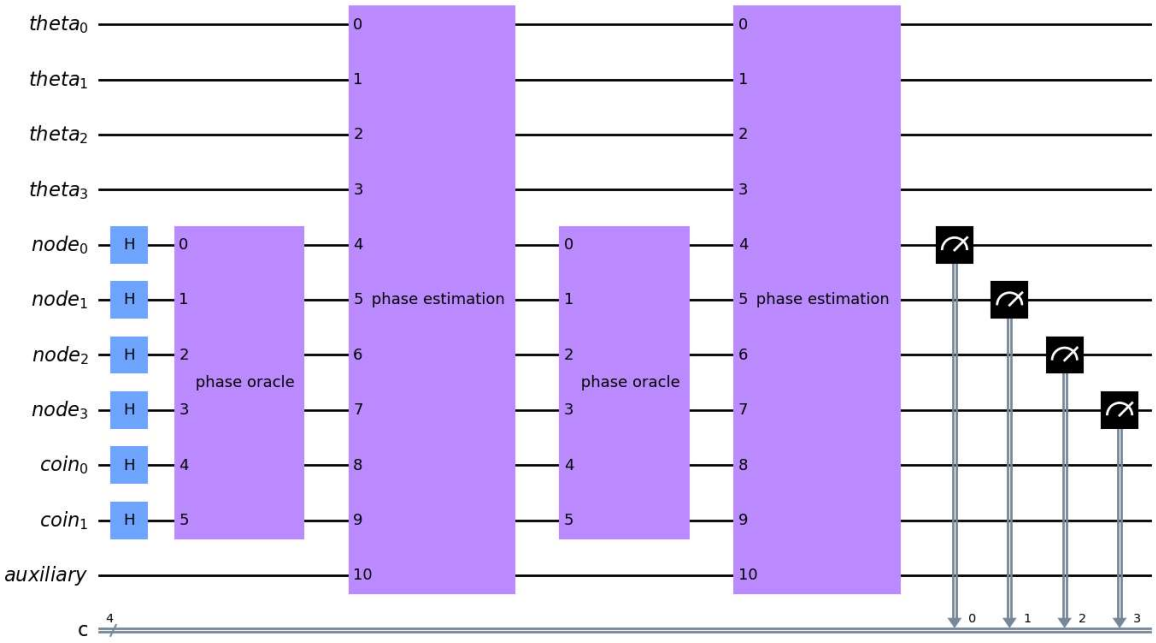
Out[23]:



Now we implement the whole quantum walk search algorithm using the gates we made previously. We start by applying Hadamard gates to node and coin qubits, which is step 1 in the algorithm. Thereafter, we iteratively apply the phase oracle gate and the phase estimation gate. We need O(1/√ε) iterations. Lastly, we measure the node qubits.

In [24]:
```
# Implementation of the full quantum walk search algorithm
theta_q = QuantumRegister(4, 'theta')
node_q = QuantumRegister(4, 'node')
coin_q = QuantumRegister(2, 'coin')
auxiliary_q = QuantumRegister(1, 'auxiliary')
creg_c2 = ClassicalRegister(4, 'c')
circuit = QuantumCircuit(theta_q, node_q, coin_q, auxiliary_q, creg_c2)
# Apply Hadamard gates to the qubits that represent the nodes and the coin
circuit.h([4,5,6,7,8,9])
iterations = 2

for i in range(0,iterations):
    circuit.append(phase_oracle_gate, [4,5,6,7,8,9])
    circuit.append(phase_estimation_gate, [0,1,2,3,4,5,6,7,8,9,10])

circuit.measure(node_q[0], creg_c2[0])
circuit.measure(node_q[1], creg_c2[1])
circuit.measure(node_q[2], creg_c2[2])
circuit.measure(node_q[3], creg_c2[3])
circuit.draw()
```

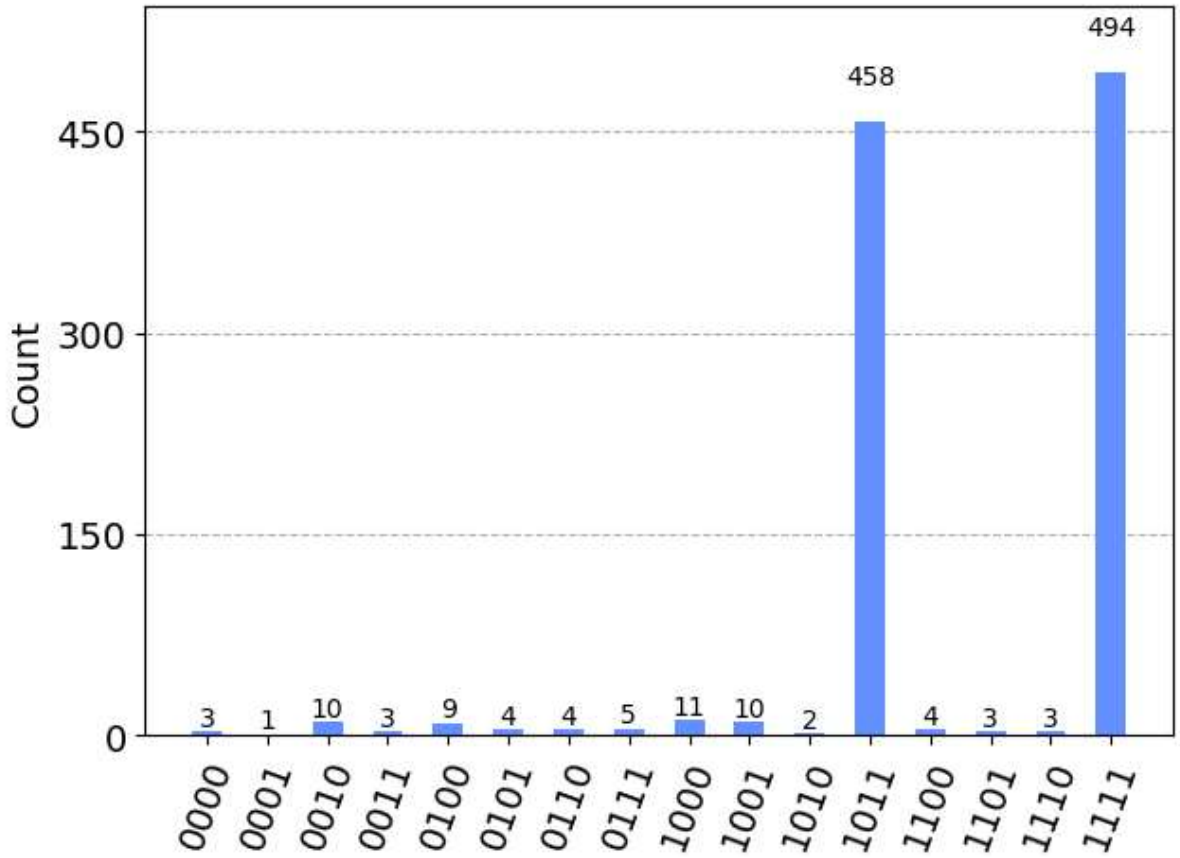
Out[24]:



Finally we run the implementation on the qasm simulator. We see that the circuit collapse to the marked states a clear majority of the times.

In [25]:
```
backend = Aer.get_backend('qasm_simulator')
job = execute( circuit, backend, shots=1024 )
hist = job.result().get_counts()
plot_histogram( hist )
```

Out[25]:



In [ ]:

In [ ]: