Assignment 1

Problem Statement:
Implementations of 16 Qubit Random Number Generator.

Objective:
1. Understand creation of Qubit circuit
2. Create 16 Qubit Random Number Generator

Outcome:
Displays 16 Qubit Random Number

Theory:
To create a Random Number Generator in qiskit for IBMs quantum computers using 16 qubits.

Requirements:

Python 3.x or above (available here: https://www.python.org/)

Pip: A package management system for Python (included with Python 3.x)

IBM Q Account: This is so you can run your programs on IBM quantum devices. You can sign up for one here: https://quantum-computing.ibm.com

Installation :

Install Python 3.x (Make sure Python is added to Path and Pip is checked)

Open Command Prompt and type in: pip install qiskit

Steps to perform:

STEP 1 : INITIALISE THE QUANTUM AND CLASSICAL REGISTERS

The first step is to initialise a 16 qubit register . This is done by the following code:

```
q = QuantumRegister(16,'q')
```

Next we initialise the 16 bit classical register with the following code:

```
c = ClassicalRegister(16,'c')
```

STEP 2 : CREATE THE CIRCUIT

Next we create a quantum circuit using the following code:

```
circuit = QuantumCircuit(q,c)
```

STEP 3 : APPLY A HADAMARD GATE TO ALL QUBITS

Then we need to apply a Hadamard gate. This gate is used to put a qubit in to a superposition of 1 and 0 such that when we measure the qubit it will be 1 or a 0 with equal probability.

This is done with the following code:

```
circuit.h(q)
```

STEP 4 : MEASURE THE QUBITS

After this we measure the qubits. This measurement will collapse the qubits superposition in to either a 1 or a 0.

This is done with the following code:

```
circuit.measure(q,c)
```

ALGORITHM :

1. Start
2. pip install qiskit
3. Initialise the quantum and classical registers
4. Create the circuit
5. Apply a hadamard gate to all qubits
6. Measure the qubits
7. Stop

How to run the program :

Write the code in to a python file.
Enter your API token in the IBMQ.enable_account('Insert API token here') part
Save and run.

Conclusion:

By this way, we can generate 16 Qubit Random Number.

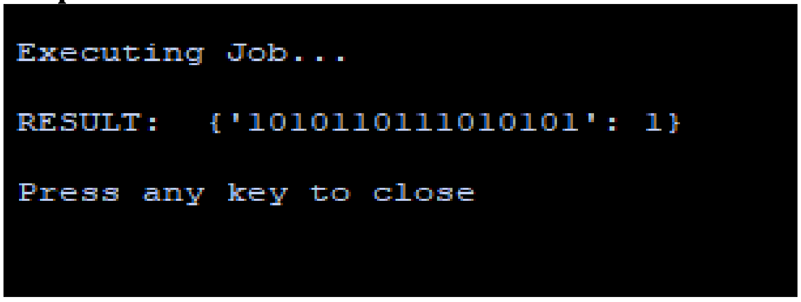
Oral Questions:

1. What is Qubit?
2. What are different steps performed to generate random number?
3. What are different types of registers?
4. What is quantum circuit?
5. How you measure the qubits?
6. What is Hadamard gate?

Code :

```
from qiskit import QuantumRegister, ClassicalRegister,
QuantumCircuit, execute,IBMQ

IBMQ.enable_account('ENTER API TOKEN HERE')
provider = IBMQ.get_provider(hub='ibm-q')

q = QuantumRegister(16,'q')
c = ClassicalRegister(16,'c')
circuit = QuantumCircuit(q,c)
circuit.h(q) # Applies hadamard gate to all qubits
circuit.measure(q,c) # Measures all qubits

backend = provider.get_backend('ibmq_qasm_simulator')
job = execute(circuit, backend, shots=1)

print('Executing Job...\n')
result = job.result()
counts = result.get_counts(circuit)

print('RESULT: ',counts,'\n')
print('Press any key to close')
input()
```

Output :



```
Executing Job...

RESULT:  {'1010110111010101': 1}

Press any key to close
```

Implementation :

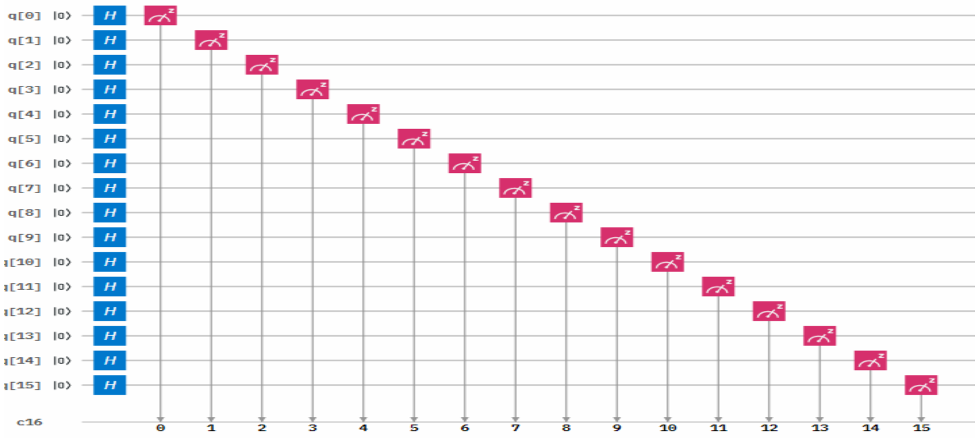


Figure 1: Circuit Diagram of the 16-qubit Random Number Generator

Assignment 2

Problem Statement:

Implement Quantum Teleportation algorithm in Python.

Objective:

1. Understand Quantum Teleportation algorithm.
2. Create quantum circuit.
3. Understand to perform Entanglement.

Outcome

Displays circuit for Quantum Teleportation

Theory:

What is Quantum Teleportation?

Quantum Computers are not really similar to Classical computers, the difference between the two is copying data because here Quantum bits is that they remain in the Quantum state till they are unobserved. As soon as we observe (or click) on them, they collapse to one of the known states. This is also called the no-cloning theorem. Hence, to copy data on a Quantum Computer, we need the process of Quantum Teleportation.

The theory behind the algorithm Let's assume there are two friends, Kartik and Sharanya. Sharanya wants to send some form of Quantum data, possibly a qubit to Kartik. Since she can't observe what the state of the qubit is due to the no-cloning theorem, she takes the help of the so-called 'portal', to transfer the data. So what the portal basically does is, that it creates entanglement between one qubit from Sharanya and one qubit of its own and sends the entangled pair towards Kartik. Then, Kartik would have to perform some actions to remove the entanglement and receive the output.

Algorithm:

Step 1: The portal creates an entangled pair of Qubits, which is a special pair known as Bell's pair. In order to create a Bell's pair using Quantum Circuits, we need to take one qubit and turn it into the (|+> or |->) state using a Hadamard gate and then using a CNOT gate on the other qubit, which will be controlled by the first qubit. One of the qubits is given to Sharanya (say Q1), the other to Kartik (say Q2).

Step 2: Let's say that the qubit Sharanya wants to send is |Ψ> = |∝> + |β>. She needs to applied a CNOT gate to Q1, controlled by |Ψ>. A CNOT gate is basically the 'if this, then that' condition of the Quantum world.

Step 3: Sharanya takes a measurement of the two qubits that she has, and stores them in two classical bits. She then sends this information to Kartik (A transfer can be made since classical bits are being sent). Since qubits can handle 2n classical bits, we can say that the outputs Sharanya will get with her calculation will always be a probabilistic answer containing 00, 01, 10, and 11 (all permutations of 0 and 1).

Step 4: Now, all that Kartik needs to do is perform certain transformations on the qubit he has, Q2, which is a part of an entangled pair. This part comes from Quantum Mechanics, so you can just know it as a fact, or the complexity of the article will increase manifolds. So, if Kartik gets a 00, he needs to apply for an I gate. For 01, a X gate needs to be applied, for 10, a Z gate needs to be applied and for 11, a ZX gate needs to be applied.

And there, we have it. Kartik now has a qubit in the same state as the state Sharanya initially had her qubit in.

Module needed Qiskit: Qiskit is an open-source framework for quantum computing. It provides tools for creating and manipulating quantum programs and running them on prototype quantum devices on IBM Q Experience or on simulators on a local computer. Let’s see how we can create a simple Quantum circuit and test it on a real Quantum computer or simulate it in our computer locally.

Installation:

```
pip install qiskit
```

Stepwise implementation :

Step 1: Creating the Quantum Circuit on which we will be doing operations.

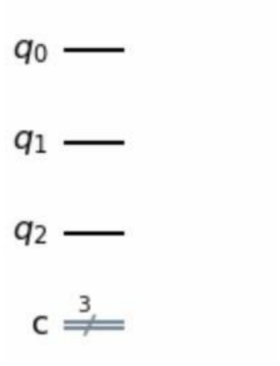
QuantumCircuit takes in 2 arguments, the number of qubits that we want to take and the number of classical bits that we want to take.

```
from qiskit import *
circuit = QuantumCircuit(3, 3)
%matplotlib inline      # enables the drawing of matplotlib figures in the IPython environment.
```

# Whenever during any point of the program we want to see how our circuit looks like, this is what we will be doing.

```
circuit.draw(output='mpl')
```

Output:



This is how our circuit looks right now. We have three quantum bits and 3 classical bits, which will be used to measure the values of these Qubits, whenever we want. Right now they don't have any value in them.

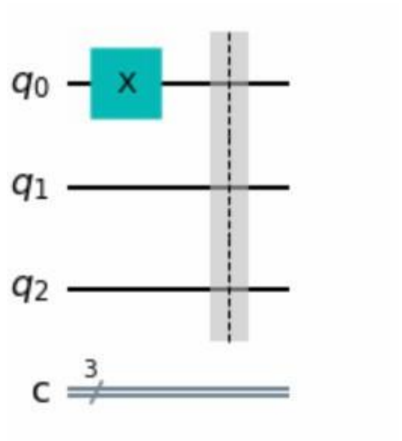
Step 2: Applying an X gate on the qubit which we have to teleport. We will also be adding a barrier, just to make the circuit more clear.

Now here, what we have done till now is that we have 3 qubits. What we'll be doing is that we will be using q1 to transport data from q0 to q3. For this, we will use an X gate to init q0 to 1 state

```
circuit.x(0)  # used to apply an X gate.
```

```
# This is done to make the circuit look more  organized and clear.
circuit.barrier()
circuit.draw(output='mpl')
```

Output:



This is how our circuit looks right now. On one of the Qubits, we have added an X gate, shown by the X. A barrier is added to make the circuit more organized as we keep on adding gates and other things. The classical bits are still untouched.

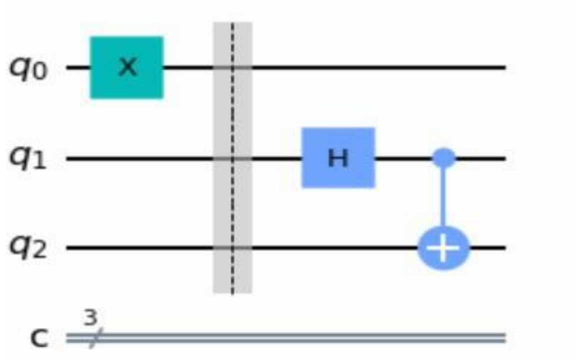
Step 3: Creating entanglement between Q1 and Q2 by applying a Hadamard gate on Q1, and a CX gate on Q1 and Q2 in such a way that the behavior of Q1 affects the behavior of Q2.

Here, what we'll do is that we will create entanglement so that the behavior of the first qubit affects the behavior of the second qubit.

```
# This is how we apply a Hadamard gate on Q1.
circuit.h(1)
```

```
# This is the CX gate, which takes two parameters, one being the control qubit and the  other being the target qubit.
circuit.cx(1, 2)
circuit.draw(output='mpl')
```

Output:



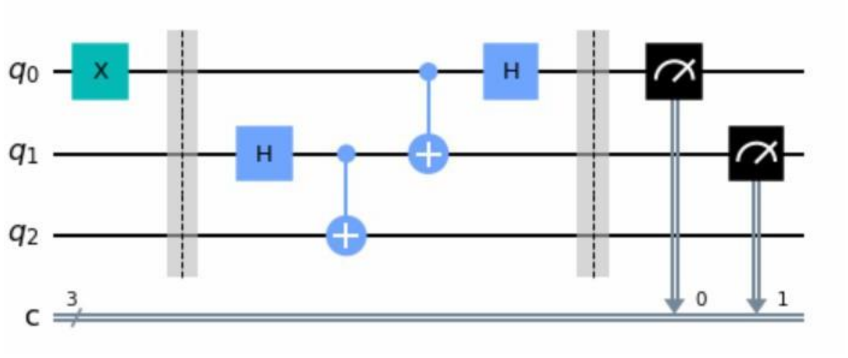
Here we can see that after this step, this is how our circuit looks. We have a Hadamard gate applied to Q1, shown by the 'H' Symbol, and a Controlled NOT(CX) gate on Q1 and Q2. The classical bits are still untouched.

Creating entanglement between Q0 and Q1 by applying a Hadamard gate on Q0, and a CX gate on Q0 and Q1 in such a way that the behavior of Q1 affects the behavior of Q0. So essentially, we have a system where the behavior of either of the Qubits will affect the behavior of all the Qubits.
Q1 can be considered as the portal we talked of, above. We will also now project the Qubits on the classical bits and measure the values of Q0 and Q1.

```
# The next step is to create a controlled  gate between qubit 0 and qubit 1.
# Also we will be applying a Hadamard gate to q0.

circuit.cx(0, 1)
circuit.h(0)

# Done for clarification of the circuit again.
circuit.barrier()
```

```
# the next step is to do the two measurements  on q0 and q1.
circuit.measure([0, 1], [0, 1])

# circuit.measure can take any number of arguments,  and has the following parameters:
# [qubit whose value is to be measured,  classical bit where the value is stored]
circuit.draw(output='mpl')
```

Output:



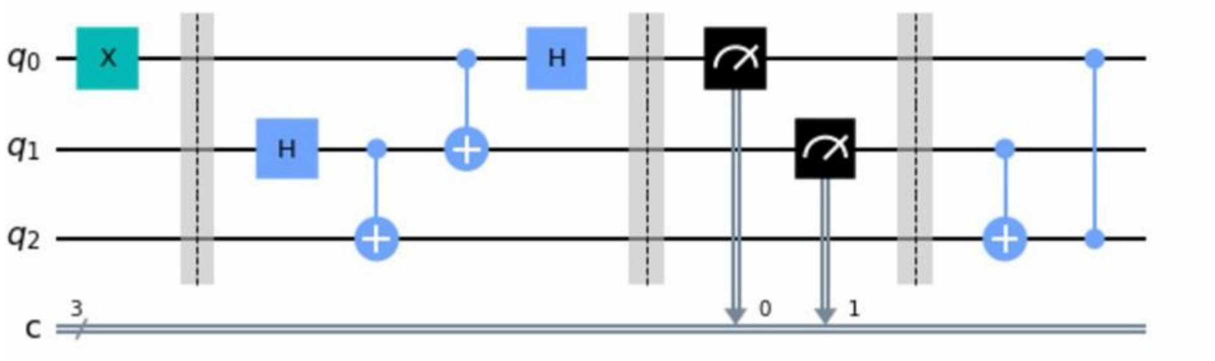
Here we can see how our circuit looks. With the help of barriers, you can easily distinguish what was done in this step. We have a Hadamard gate applied to Q0, shown by the ‘H’ Symbol, and a Controlled NOT(CX) gate on Q0 and Q1. The classical bits are now put to use, with the black symbols showing that we have taken the value of Q0 and Q1 and stored it in classical bit 1 and classical bit 2.

The actual explanation will require an understanding of Quantum mechanics, so one can just understand that for a 00 measurement of the classical bits, we need to apply an I gate. For 01, an X gate needs to be applied, for 10, a Z gate needs to be applied and for 11, a ZX gate needs to be applied. Since we don’t know what the value will be stored in classical bits, we are applying the more generalized Control X gates and Control Z gates.

The last step is to add two more gates, a controlled x gate and a controlled z gate (We have talked about this step in the text above, which tells what gates to apply for different measurements).

```
circuit.barrier()
circuit.cx(1, 2)
circuit.cz(0, 2)
circuit.draw(output='mpl')
```

Output:

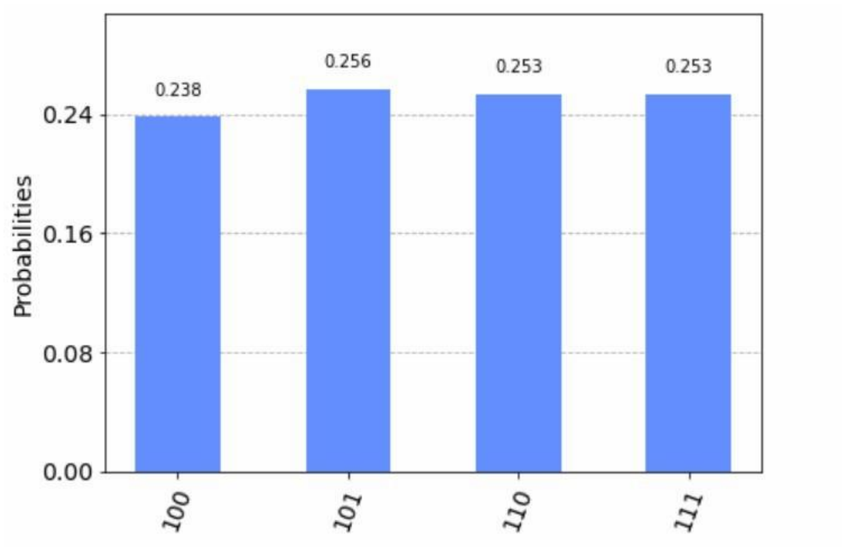


This is the final required circuit for Quantum Teleportation. After the barrier, we can see that a Controlled X gate has been applied on Q1 and Q2, such that the behavior of Q2 affects the behavior of Q1. Also, a Control Z gate has been applied as shown between Q0 and Q2.

Now that our circuit has been made, all we need to do is to pass that circuit into a simulator so that we can get the results from the circuit back. Once we get the results back, we are plotting a histogram from the values of the classical bits that we have received. You can think of whatever we have done now as an Abstract Data Type created by us, and the code below is the main function where we will put it to use. Each step has been explained in complete detail for easy understanding of the reader.

```
# The first step is to call a simulator
# which we will use to perform simulations.
from qiskit.tools.visualization import plot_histogram
sim = Aer.get_backend('qasm_simulator')

# here, like before, we have given the
# classical bit 2 the value of the Quantum bit 2.
circuit.measure(2, 2)

# Now, we run the execute function,
# which takes our quantum circuit,
# the backend which we are using and
# the number of shots we want
# (shots are to increase accuracy and
# mitigate errors in Quantum Computing).
# All of this is stored in a variable called result

result = execute(circuit, backend=sim, shots=1000).result()
counts = result.get_counts()

# This counts variable shows that for each possible combination,
# how many times the circuit gave a similar output
# (for example, 111 came x times, 101 came y times etc.)

# importing plot_histogram which will help us visualize the results.
plot_histogram(counts)
```

Output:



Algorithm:

1. Start
2. Creating the Quantum Circuit.
3. Applying an X gate on the qubit which we have to teleport.
4. Creating entanglement between Q1 and Q2 to create a controlled gate between qubit 0 and qubit 1.
5. Apply a Hadamard gate to q0.
6. Do the two measurements on q0 and q1.
7. Add two more gates, a controlled x gate and a controlled z gate
8. Stop.

Conclusion:

By this way, we have Implemented quantum teleportation algorithm in python.

Oral Questions:

1. What is entanglement?
2. What are different gates used in teleportation?
3. What is teleportation?
4. How Quantum circuit is designed?

Assignment 3

Problem Statement:

Implementing a 5 qubit Quantum Fourier Transform

Objective:

1. Understand Quantum Fourier Transform
2. Create 5 qubit Quantum Fourier Transform

Outcome

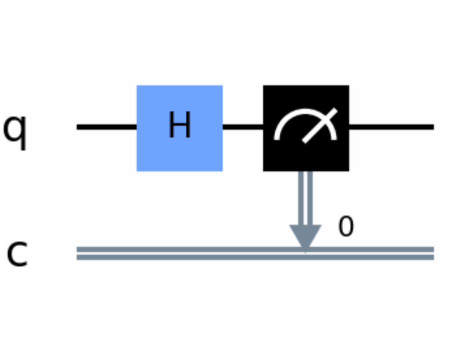
Displays circuit for 5 qubit Quantum Fourier Transform.

Theory:

What is the Quantum Fourier Transform?
The Quantum Fourier Transform (QFT) is a circuit that transforms the state of the qubit from the computational basis to the Fourier basis. *Note that the Fourier basis is just another term for the Hadamard basis.* As such the easiest way to implement a QFT is with Hadamard gates and Controlled U1 gates.

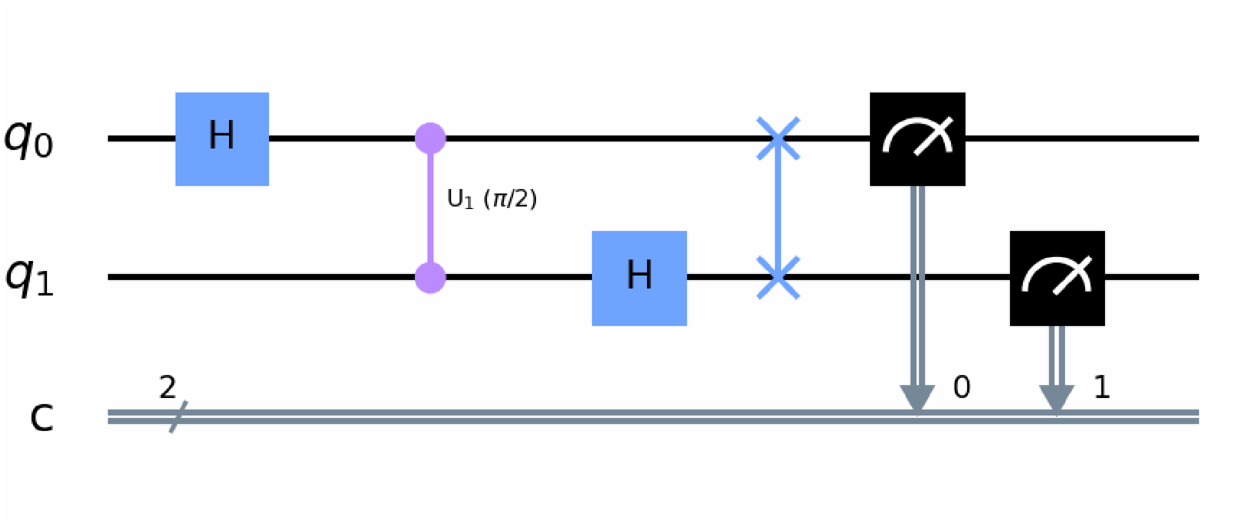
Note: A Controlled U1 gate is just a gate that implements a single rotation around the Z-axis (phase) of the target qubit if the control qubit is 1.

Circuit diagram of a 1 qubit QFT



The simplest QFT is a 1 qubit QFT which just implements a Hadamard gate.

However if we implement a 2 qubit QFT then you can see how the controlled U1 are used:



2 qubit QFT

First we implement a Hadamard gate which puts q0 in to superposition. Next we apply a controlled U1 gate with a rotation of pi/2 to q1. After this a Hadamard gate is applied to q1. Next we apply a swap gate to q0 and q1.

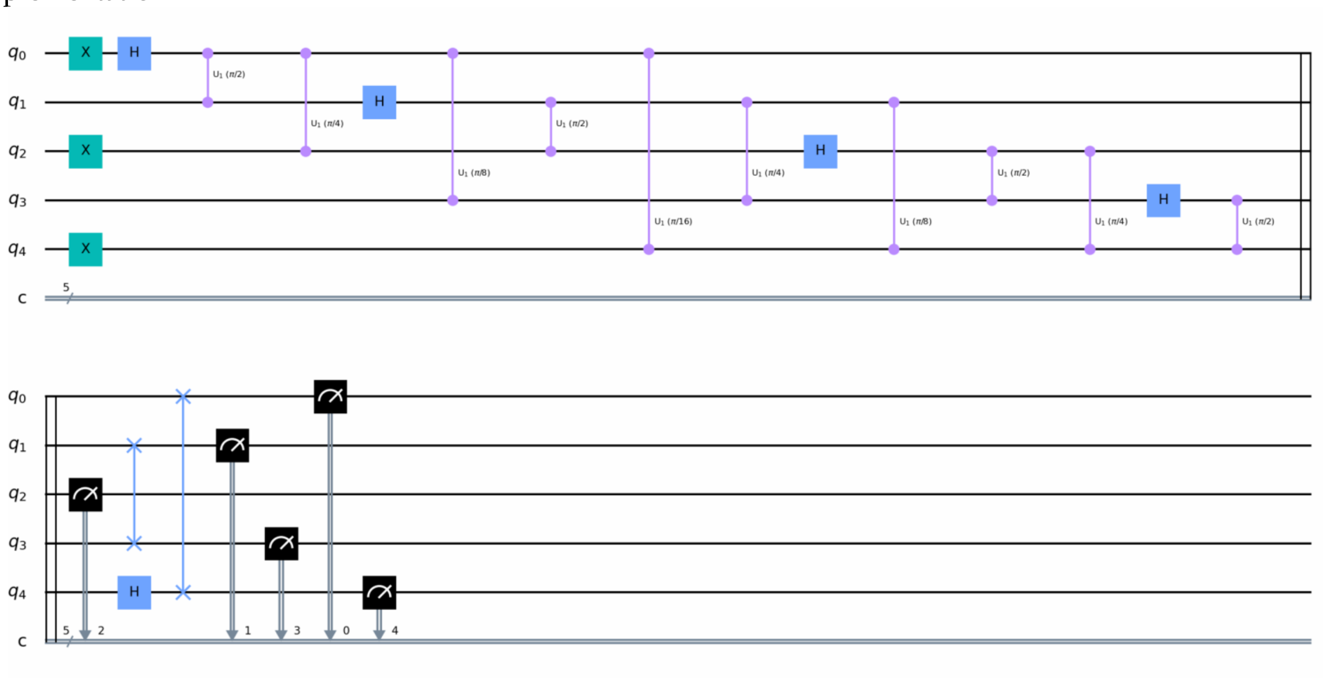
Note that these swap gates are not needed if the QFT is implemented at the end of your circuit.

After this both qubits will be in superposition but whatever computational value (1 or 0) will be encoded in to the Hadamard basis of the qubit.

To encode values on N qubits we have to double the rotation value of each qubit. For example the diagram below shows a 5 qubit QFT.

Notice how for q0 it applies a rotation of pi/2 for q1 then pi/4 for q2 then pi/8 for q3 and so on. This pattern repeats for each qubit. When all rotations have been applied to a qubit it is put in to superposition using a Hadamard gate. Then it can be used as a control qubit to apply rotations to target qubits below it.

Implementation



Circuit diagram of a 5 qubit QFT

Implementing a 5 qubit Quantum Fourier Transform in qiskit
In qiskit we could implement the 5 qubit QFT by implementing all the gates in the diagram above.

In qiskit you can use the QFT() function as follows:

QFT(num_qubits=None, approximation_degree=0, do_swaps=True, inverse=False, insert_barriers=False, name='qft')
Where:

num_qubits: The number of qubits we want to add to the QFT (in our case it is 5)

approximation_degree: This allows us to reduce circuit depth by ignoring phase rotations under a certain value

do_swaps: If set to true then we use swap gates in the QFT

inverse: If set to true we implement the inverse QFT

insert barrier: If set to true then we insert barriers

<u>For example in our 5 qubit QFT we implement the following:</u>

QFT(num_qubits=5, approximation_degree=0, do_swaps=True, inverse=False, insert_barriers=True, name='qft')

If we encode 1010 on to a QFT and then measure it we will get random values since the qubits have been put in to superposition and the values we encoded in to the computational basis are now encoded in the Hadamard basis of each qubit via the controlled U1 gates.

Inverse Quantum Fourier Transform
To get our values back we can use the inverse QFT. This reverses all the rotations done in the QFT above.

For example is there was a rotation of Pi in the QFT then the inverse QFT will do a rotation of -Pi.

In qiskit we can get the values back by implementing an inverse QFT by setting inverse to true.

For example:

QFT(num_qubits=5, approximation_degree=0, do_swaps=True, inverse=True, insert_barriers=True, name='qft')

How to run the program

Copy and paste the code below in to a python file

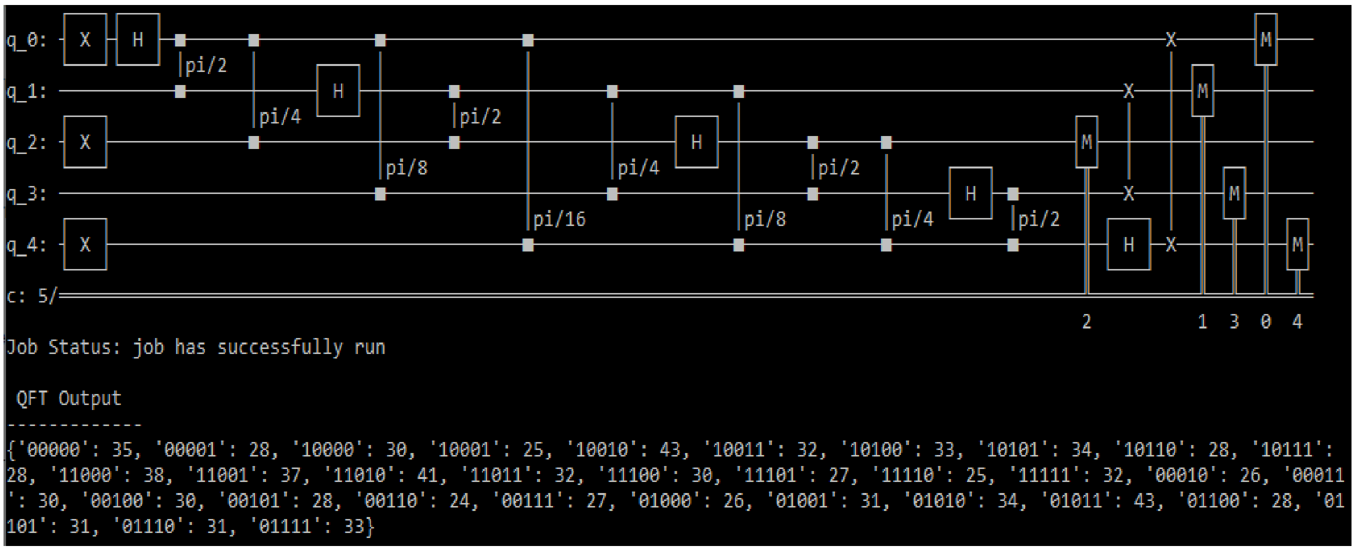
Enter your API token in the IBMQ.enable_account('Insert API token here') part

Save and run

Code :

```
from qiskit import QuantumRegister, ClassicalRegister
from qiskit import QuantumCircuit, execute,IBMQ
from qiskit.tools.monitor import job_monitor
from qiskit.circuit.library import QFT
import numpy as np

pi = np.pi

IBMQ.enable_account(‘ENTER API KEY HERE’)
provider = IBMQ.get_provider(hub='ibm-q')

backend = provider.get_backend('ibmq_qasm_simulator')

q = QuantumRegister(5,'q')
c = ClassicalRegister(5,'c')

circuit = QuantumCircuit(q,c)

circuit.x(q[4])
circuit.x(q[2])
circuit.x(q[0])
circuit += QFT(num_qubits=5, approximation_degree=0, do_swaps=True, inverse=False,
insert_barriers=False, name='qft')
circuit.measure(q,c)
circuit.draw(output='mpl', filename='qft1.png')
print(circuit)

job = execute(circuit, backend, shots=1000)

job_monitor(job)

counts = job.result().get_counts()

print("\n QFT Output")
print(" ------------ ")
print(counts)
input()
```

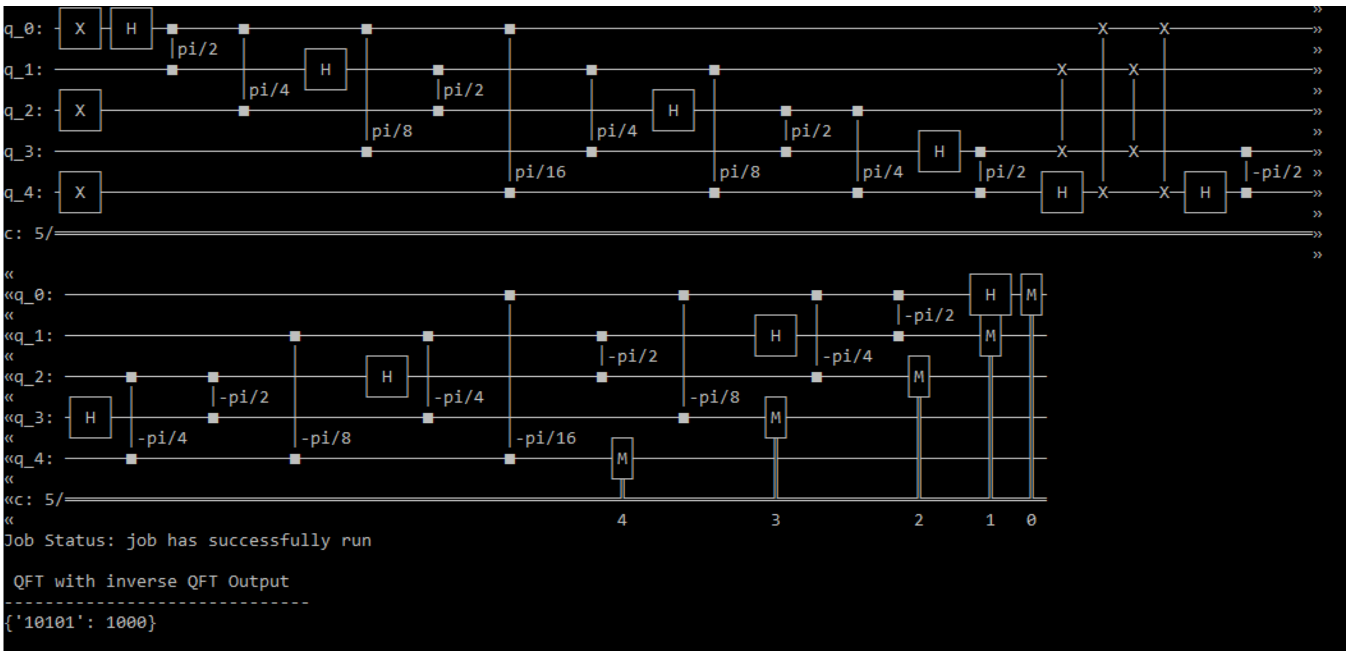
Output
Here is the output when running the QFT:


```
Job Status: job has successfully run

 QFT Output
-------------
{'00000': 35, '00001': 28, '10000': 30, '10001': 25, '10010': 43, '10011': 32, '10100': 33, '10101': 34, '10110': 28, '10111':
28, '11000': 38, '11001': 37, '11010': 41, '11011': 32, '11100': 30, '11101': 27, '11110': 25, '11111': 32, '00010': 26, '00011
': 30, '00100': 30, '00101': 28, '00110': 24, '00111': 27, '01000': 26, '01001': 31, '01010': 34, '01011': 43, '01100': 28, '01
101': 31, '01110': 31, '01111': 33}
```

Output when running the 5 qubit QFT. Notice how we get multiple values back since the qubits are in superposition and the value is encoded in the Hadamard basis.

Here is the output when running the QFT with the inverse QFT.

Oral Questions:


```
Job Status: job has successfully run

 QFT with inverse QFT Output
-----------------------------
{'10101': 1000}
```

Assignment 4

Problem Statement:

Tackle Noise with Error Correction

Objective:

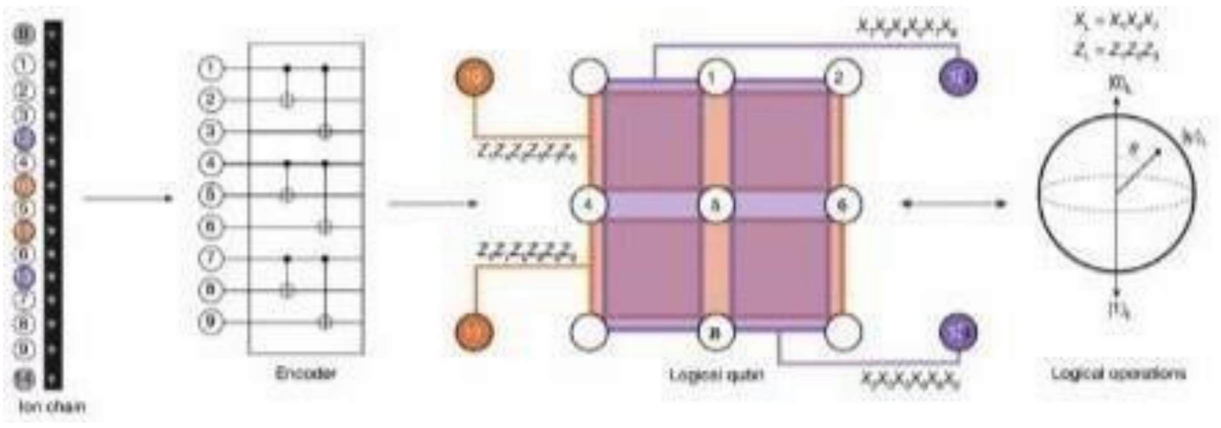
1. Understand Tackle Noise with Error Correction

Outcome

Software Requirement: Ubuntu OS, Quskit (Cloud based)

Theory:

Quantum error correction is theorised as essential to achieve fault tolerant quantum computing that can reduce the effects of noise on stored quantum information, faulty quantum gates, faulty quantum preparation, and faulty measurements. This would allow algorithms of greater circuit depth.
Noise is a major challenge in quantum computing, as it can cause errors in quantum computations. Quantum error correction (QEC) is a technique that can be used to tackle noise and improve the reliability of quantum computers.
QEC works by encoding a single logical quantum bit (qubit) into multiple physical qubits. This redundancy allows QEC to detect and correct errors that occur on the physical qubits.
There are a number of different QEC codes, each with its own strengths and weaknesses. Some QEC codes are more efficient, while others are more robust to noise.
QEC is still under development, but it has already been demonstrated in small-scale experiments. As QEC codes improve and become more efficient, they will be essential for building large-scale quantum computers.
Here is an example of how QEC can be used to tackle noise in a quantum communication system:
The sender encodes a single logical qubit into multiple physical qubits using a QEC code.
The sender transmits the physical qubits to the receiver.
The receiver uses the QEC code to detect and correct any errors that occurred during transmission.
The receiver decodes the physical qubits to recover the logical qubit.
If a small number of errors occur during transmission, the QEC code will be able to correct them and the receiver will be able to recover the logical qubit accurately. However, if too many errors occur, the QEC code will not be able to correct them and the receiver will not be able to recover the logical qubit accurately.
QEC can also be used to tackle noise in quantum computers. For example, QEC can be used to protect the qubits in a quantum computer from noise caused by faulty quantum gates or environmental interactions.
QEC is a powerful tool for tackling noise in quantum computing and communication. As QEC codes improve and become more efficient, they will play an essential role in building large-scale quantum computers and enabling reliable quantum communication.



How to run the program

Copy and paste the code below in to a python file

Enter your API token in the IBMQ.enable_account('Insert API token here') part

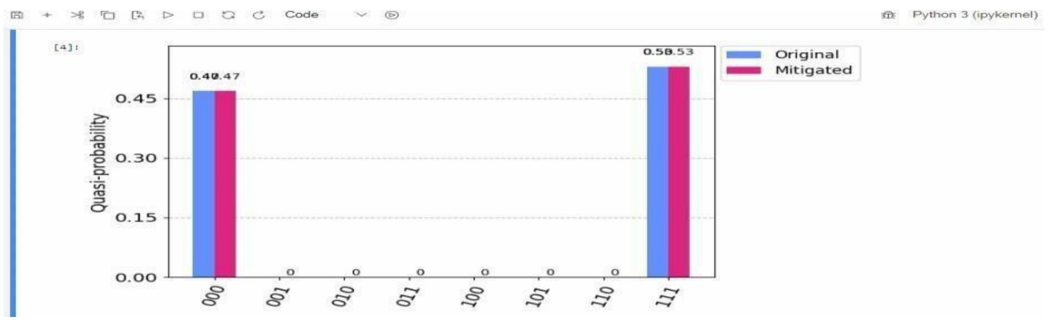
Save and run

Code:

```
!pip install qiskit-ignis


from qiskit import QuantumCircuit, assemble, Aer, transpile
from qiskit.visualization import plot_histogram
from qiskit.ignis.mitigation.import CompleteMeasFitter, complete_meas_cal, tensored_meas_cal

# Define the quantum circuit
qc = QuantumCircuit(3, 3)

# Apply gates and operations to the circuit
qc.h(0)
qc.cx(0, 1)
qc.cx(0, 2)
qc.measure([0, 1, 2], [0, 1, 2])

# Transpile the circuit
backend = Aer.get_backend('qasm_)
transpiled_qc = transpile(qc, backend)

# Simulate the noisy circuit
qobj = assemble(transpiled_qc, shots=1000)
job = backend.run(qobj)
result = job.result()
counts = result.get_counts()

# Perform error mitigation
cal_circuits, state_labels = complete_meas_cal(qubit_list=[, 1, 2])
cal_job = backend.run(assemble(cal_
cal_results = cal_job.result()
meas_fitter = CompleteMeasFitter(cal_
mitigated_counts = meas_fitter.filter.apply(

# Print the original counts
print("Original counts:")
print(counts)

# Print the mitigated counts
print("Mitigated counts:")
print(mitigated_counts)

# Plot the histograms of the original and mitigated counts
```

```
plot_histogram([counts, mitigated_counts], legend=['Original', 'Mitigated'])
```

Output:



Oral Questions:

Q1. How can we adapt error correction codes to different types of noise? How can we design error correction codes that are scalable to large numbers of qubits?
Q2. How can we implement error correction in a way that minimizes overhead?
Q3. How can we combine error correction with other techniques, such as quantum feedback control, to improve the performance of quantum devices?

Experiment No – 5

Aim: Implement Tarrataca’s quantum production system with the 3-puzzle problem

Outcome: At end of this experiment, student will be able to Write a Python Program using Tarrataca’s quantum production system with the 3-puzzle problem.

Software Requirement: Quiskit

Theory:

Tarrataca's quantum production system is a quantum algorithm for solving combinatorial optimization problems. It works by constructing a quantum circuit that represents the problem space, and then applying a sequence of quantum operations to the circuit to search for the optimal solution.

The 3-puzzle problem is a combinatorial optimization problem where the goal is to arrange three tiles in order, given an initial state. Each tile can be moved either left or right, and the cost of a move is equal to the distance that the tile is moved.

To implement Tarrataca's quantum production system for the 3-puzzle problem, we can use the following steps:

Encode the problem state as a quantum state. We can use a qubit to represent each tile, and the state of the qubit will represent the position of the tile. For example, if a tile is in the leftmost position, we can represent it with the qubit state $|0\rangle$, and if it is in the rightmost position, we can represent it with the qubit state $|1\rangle$.

Construct a quantum circuit that represents the problem space. The quantum circuit will have a qubit for each tile, and the gates will represent the possible moves that can be made. For example, we can use a CNOT gate to represent a move that exchanges the positions of two tiles.

Apply a sequence of quantum operations to the circuit to search for the optimal solution. We can use a variety of quantum algorithms to search for the optimal solution, such as Grover's algorithm or amplitude amplification.

Measure the qubits to obtain the optimal solution. Once we have found the optimal solution, we can measure the qubits to obtain the positions of the tiles.

Here is an example of a quantum circuit that can be used to solve the 3-puzzle problem:

q0: ---

q1: ---

q2: ---

CNOT q0 q1

CNOT q1 q2

H q0

H q1

H q2

Grover's algorithm

M q0

M q1

M q2

This circuit starts with the three qubits in the state |000⟩ . The first two CNOT gates exchange the positions of the first two qubits and the second two qubits, respectively. The three Hadamard gates put the qubits into a superposition of states.

Grover's algorithm is then used to search for the optimal solution. Grover's algorithm is a quantum algorithm that can amplify the probability of finding a solution to a search problem.

Finally, the three qubits are measured to obtain the optimal solution.

Tarrataca's quantum production system is a powerful algorithm for solving combinatorial optimization problems. It has the potential to solve problems that are intractable for classical computers.

Program:

```
from qiskit import Aer, transpile, assemble, QuantumCircuit from
qiskit.aqua import QuantumInstance
```

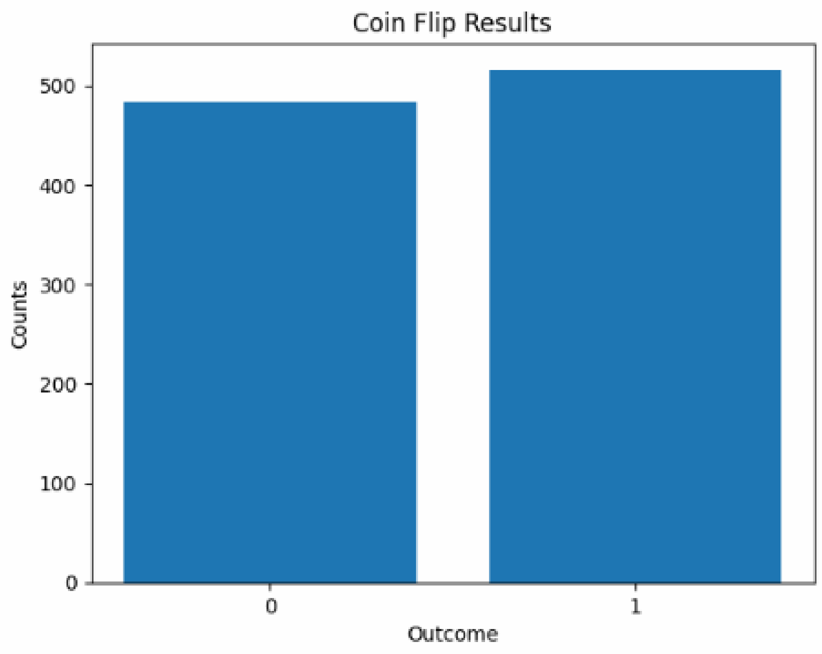
```
from qiskit.aqua.algorithms import QAOA

from qiskit.aqua.components.optimizers import COBYLA


# Define the objective function def
objective_function(x):

   cost = 0

   state = [[1, 2, 3], [4, 5, 6], [7, 8, None]]


   for i in range(3):

      for j in range(3):

         if state[i][j] is not None and state[i][j] != x[i][j]: cost += 1


   return cost


# Define the QAOA solver def
solve_3puzzle_qaoa():

   backend = Aer.get_backend('qasm_simulator') optimizer
   = COBYLA(maxiter=100)

   qaoa = QAOA(optimizer=optimizer, p=1,
quantum_instance=QuantumInstance(backend=backend))


   #    Generate    the    initial    state    circuit
   initial_state  =  QuantumCircuit(9)  for  i  in
   range(3):

      for j in range(3): initial_state.h(i *
         3 + j)

   initial_state.barrier()



  # Generate the mixer circuit mixer = QuantumCircuit(9)

    mixer.cz(0, 1)
    mixer.cz(0, 3)

    mixer.cz(1, 2)

    mixer.cz(1, 4)

    mixer.cz(2, 5)

    mixer.cz(3, 4)

    mixer.cz(3, 6)

    mixer.cz(4, 5)

    mixer.cz(4, 7)

    mixer.cz(5, 8)

    mixer.cz(6, 7)

    mixer.cz(7, 8)


    # Solve the problem using QAOA
    qaoa.initial_state = initial_state
    qaoa.mixer = mixer

    qaoa.objective_function = objective_function


    result = qaoa.compute_minimum_eigenvalue()
    solution = result.x


    return solution


# Solve the 3-puzzle problem using QAOA
solution = solve_3puzzle_qaoa()


# Print the solution
print("Solution found!") for i
in range(0, 9, 3):

    print(solution[i:i + 3])
```

```
from qiskit import QuantumCircuit, Aer, execute from
qiskit.visualization import plot_histogram


# Define the initial state

initial_state = [2, 5, 3, 1, 8, 6, 4, 7, None]


# Define the goal state

goal_state = [1, 2, 3, 4, 5, 6, 7, 8, None]


# Define the quantum circuit qc =
QuantumCircuit(18, 18)


# Initialize the circuit with the initial state for i,
tile in enumerate(initial_state):

  if tile is not None:

    qc.x(i)


# Perform swaps to reach the goal state for i,
tile in enumerate(goal_state):

  if tile is not None:

    initial_index = initial_state.index(tile)

     target_index = goal_state.index(tile)
    qc.swap(i, initial_index + 9) qc.swap(i,
    target_index + 9)


# Measure the final state
qc.measure(range(9), range(9))


# Simulate the circuit

backend = Aer.get_backend('qasm_simulator') job =
execute(qc, backend, shots=1024)
```

```
result = job.result()

counts = result.get_counts(qc)


# Find the most probable state (solution)
max_count = max(counts.values())

solution = [state for state, count in counts.items() if count == max_count][0]


# Print the solution
print("Solution found!") for i
in range(0, 9, 3):

   print(solution[i:i + 3])

from qiskit import Aer, execute, QuantumCircuit #

Define the quantum circuit


qc = QuantumCircuit(1, 1)

qc.h(0) # Apply Hadamard gate for superposition

qc.measure(0, 0) # Measure qubit and store result in classical bit


# Use the Aer simulator

simulator = Aer.get_backend('qasm_simulator')


# Set the number of shots (measurements)
num_shots = 1000


# Execute the quantum circuit on the simulator with multiple shots job =
execute(qc, simulator, shots=num_shots)


# Get the result of the measurements result
= job.result()

counts = result.get_counts()
```

```
# Print the coin flip results
print("Coin Flip Results:")

for outcome, count in counts.items():

   print(f"{outcome}: {count} ({count/num_shots*100:.2f}%)") from

qiskit import Aer, execute, QuantumCircuit


# Define the quantum circuit qc =
QuantumCircuit(1, 1)

qc.h(0) # Apply Hadamard gate for superposition

 qc.measure(0, 0) # Measure qubit and store result in classical bit


# Use the Aer simulator

simulator = Aer.get_backend('qasm_simulator')


# Set the number of shots (measurements)
num_shots = 1000


# Execute the quantum circuit on the simulator with multiple shots job =
execute(qc, simulator, shots=num_shots)


# Get the result of the measurements result
= job.result()

counts = result.get_counts()


# Print the coin flip results
print("Coin Flip Results:")

for outcome, count in counts.items():

   print(f"{outcome}: {count} ({count/num_shots*100:.2f}%)")


import matplotlib.pyplot as plt
```

```
from qiskit import Aer, execute, QuantumCircuit

# Define the quantum circuit qc =
QuantumCircuit(1, 1)

qc.h(0) # Apply Hadamard gate for superposition

qc.measure(0, 0) # Measure qubit and store result in classical bit

# Use the Aer simulator

simulator = Aer.get_backend('qasm_simulator')

# Set the number of shots (measurements)
num_shots = 1000

# Execute the quantum circuit on the simulatr with multiple shots job =
execute(qc, simulator, shots=num_shots)

# Get the result of the measurements result
= job.result()

counts = result.get_counts()

# Plot the coin flip results outcomes =
list(counts.keys()) counts_list =
list(counts.values())

plt.bar(outcomes, counts_list)
plt.xlabel('Outcome')
plt.ylabel('Counts') plt.title('Coin
Flip Results') plt.show()
```



Questions:

1. How does Tarrataca's quantum production system encode the problem state as a quantum state?

2. How does Tarrataca's quantum production system construct a quantum circuit that represents the problem space?

3. What are some of the quantum algorithms that can be used to search for the optimal solution to the 3-puzzle problem using Tarrataca's quantum production system?

4. How can we measure the qubits at the end of the quantum circuit to obtain the optimal solution to the 3-puzzle problem?

5. What are some of the advantages and disadvantages of using Tarrataca's quantum production system to solve the 3-puzzle problem?