

## ✓ Exam - Phase Estimation and Shor's Algorithm

This Jupyter Notebook contains a "?" at several places. Please substitute these question marks by the correct code.

Make sure you execute each cell.

## ✓ Your coordinates

First Name: Parsa

Last Name: Vares

## ✓ Installations (if needed)

```
%pip install qiskit[visualization]
```

```

Collecting qiskit[visualization]
  Downloading qiskit-1.3.1-cp39-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (12 kB)
Collecting rustworkx>=0.15.0 (from qiskit[visualization])
  Downloading rustworkx-0.15.1-cp38-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (9.9 kB)
Requirement already satisfied: numpy<3,>=1.17 in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (1.26.4)
Requirement already satisfied: scipy>=1.5 in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (1.13.1)
Requirement already satisfied: sympy>=1.3 in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (1.13.1)
Collecting dill>=0.3 (from qiskit[visualization])
  Downloading dill-0.3.9-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (2.8.2)
Collecting stevedore>=3.0.0 (from qiskit[visualization])
  Downloading stevedore-5.4.0-py3-none-any.whl.metadata (2.3 kB)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (4.12.2)
Collecting symengine<0.14,>=0.11 (from qiskit[visualization])
  Downloading symengine-0.13.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (1.2 kB)
Requirement already satisfied: matplotlib>=3.3 in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (3.8.0)
Requirement already satisfied: pydot in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (3.0.3)
Requirement already satisfied: Pillow>=4.2.1 in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (11.0.0)
Collecting pylatexenc>=1.4 (from qiskit[visualization])
  Downloading pylatexenc-2.10.tar.gz (162 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 162.6/162.6 kB 3.4 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: seaborn>=0.9.0 in /usr/local/lib/python3.10/dist-packages (from qiskit[visualization]) (0.13.2)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.3->qiskit[visualization]) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.3->qiskit[visualization]) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.3->qiskit[visualization]) (4.55.3)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.3->qiskit[visualization]) (1.4.7)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.3->qiskit[visualization]) (24.2)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib>=3.3->qiskit[visualization]) (3.2.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.0->qiskit[visualization]) (1.17.0)
Requirement already satisfied: pandas>=1.2 in /usr/local/lib/python3.10/dist-packages (from seaborn>=0.9.0->qiskit[visualization]) (2.2.2)
Collecting pbr>=2.0.0 (from stevedore>=3.0.0->qiskit[visualization])
  Downloading pbr-6.1.0-py2.py3-none-any.whl.metadata (3.4 kB)

```

```
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy>=1.3->qiskit[visualization]) (1.3.0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.2->seaborn>=0.9.0->qiskit[visualization]) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.2->seaborn>=0.9.0->qiskit[visualization]) (2024.2)
Downloading dill-0.3.9-py3-none-any.whl (119 kB)
 119.4/119.4 kB 5.7 MB/s eta 0:00:00
Downloading rustworkx-0.15.1-cp38-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.0 MB)
 2.0/2.0 MB 27.4 MB/s eta 0:00:00
Downloading stevedore-5.4.0-py3-none-any.whl (49 kB)
 49.5/49.5 kB 2.3 MB/s eta 0:00:00
Downloading symengine-0.13.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (49.7 MB)
 49.7/49.7 MB 17.8 MB/s eta 0:00:00
Downloading qiskit-1.3.1-cp39-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (6.7 MB)
 6.7/6.7 MB 40.2 MB/s eta 0:00:00
Downloading pbr-6.1.0-py2.py3-none-any.whl (108 kB)
 108.5/108.5 kB 5.8 MB/s eta 0:00:00
Building wheels for collected packages: pylatexenc
Building wheel for pylatexenc (setup.py) ... done
Created wheel for pylatexenc: filename=pylatexenc-2.10-py3-none-any.whl size=136816 sha256=3dc703416e631a562a3617aaaf68c3142d85f7f684ab48e14f6331207f70bea2
Stored in directory: /root/.cache/pip/wheels/d3/31/8b/e09b0386afd80cf556c00408c9aeea5c35c4d484a9c762fd5
Successfully built pylatexenc
Installing collected packages: pylatexenc, symengine, rustworkx, pbr, dill, stevedore, qiskit
Successfully installed dill-0.3.9 pbr-6.1.0 pylatexenc-2.10 qiskit-1.3.1 rustworkx-0.15.1 stevedore-5.4.0 symengine-0.13.0
```

```
%pip install qiskit-aer
```

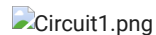
```
Collecting qiskit-aer
  Downloading qiskit_aer-0.15.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (8.0 kB)
Requirement already satisfied: qiskit>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer) (1.3.1)
Requirement already satisfied: numpy>=1.16.3 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer) (1.26.4)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer) (1.13.1)
Requirement already satisfied: psutil>=5 in /usr/local/lib/python3.10/dist-packages (from qiskit-aer) (5.9.5)
Requirement already satisfied: rustworkx>=0.15.0 in /usr/local/lib/python3.10/dist-packages (from qiskit>=1.1.0->qiskit-aer) (0.15.1)
Requirement already satisfied: sympy>=1.3 in /usr/local/lib/python3.10/dist-packages (from qiskit>=1.1.0->qiskit-aer) (1.13.1)
Requirement already satisfied: dill>=0.3 in /usr/local/lib/python3.10/dist-packages (from qiskit>=1.1.0->qiskit-aer) (0.3.9)
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/python3.10/dist-packages (from qiskit>=1.1.0->qiskit-aer) (2.8.2)
Requirement already satisfied: stevedore>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from qiskit>=1.1.0->qiskit-aer) (5.4.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from qiskit>=1.1.0->qiskit-aer) (4.12.2)
Requirement already satisfied: symengine<0.14,>=0.11 in /usr/local/lib/python3.10/dist-packages (from qiskit>=1.1.0->qiskit-aer) (0.13.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.0->qiskit>=1.1.0->qiskit-aer) (1.17.0)
Requirement already satisfied: pbr>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from stevedore>=3.0.0->qiskit>=1.1.0->qiskit-aer) (6.1.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy>=1.3->qiskit>=1.1.0->qiskit-aer) (1.3.0)
Downloading qiskit_aer-0.15.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.3 MB)
 12.3/12.3 MB 29.2 MB/s eta 0:00:00
Installing collected packages: qiskit-aer
Successfully installed qiskit-aer-0.15.1
```

## ✓ Single Qubit Phase Estimation

The phase is represented by an angle Theta. We want to find an estimate for Theta. Does the Phase Estimation Procedure allow us to do so?

Remember that  $QFT_1 = (1)$  in other words, a Matrix of the size  $1 \times 1$ .

[Please look at the picture of the Circuit to complete the code!](#)



```

from math import pi, cos, sin
from qiskit import QuantumCircuit

theta = 0.7
# Theta can be changed to any value between 0 and 1
# We want to find an estimate for theta.
# Does the Phase Estimation allows us to do so?

# 0.7 = 7/10
# This means: phase = 2*pi*theta = 7*pi/5

qc = QuantumCircuit(2, 1)

# Prepare eigenvector  $|\psi\rangle$ , which is the  $|1\rangle$  state

qc.x(1)
qc.barrier()

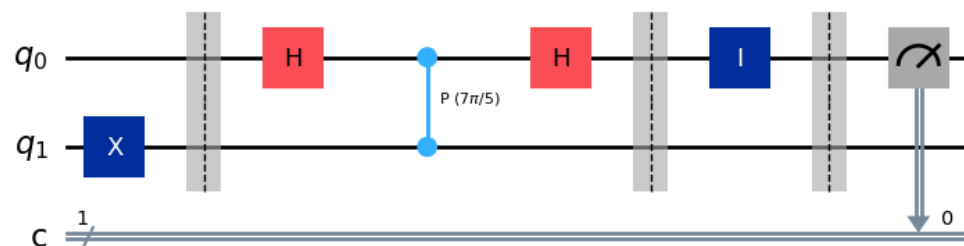
# Implement the estimation procedure
# The Controlled U-Gate is the Controlled Phase Gate ("cp") with phase 2*pi*Theta = 7*pi/5.
qc.h(0)
qc.cp(2 * pi * theta, 0, 1)
qc.h(0)
qc.barrier()

# The QFT-1 is simply the identity operator on Qubit q0
qc.id(0)
qc.barrier()

# Perform the final measurement
qc.measure(0,0)

# Draw the circuit
display(qc.draw('mpl'))

```

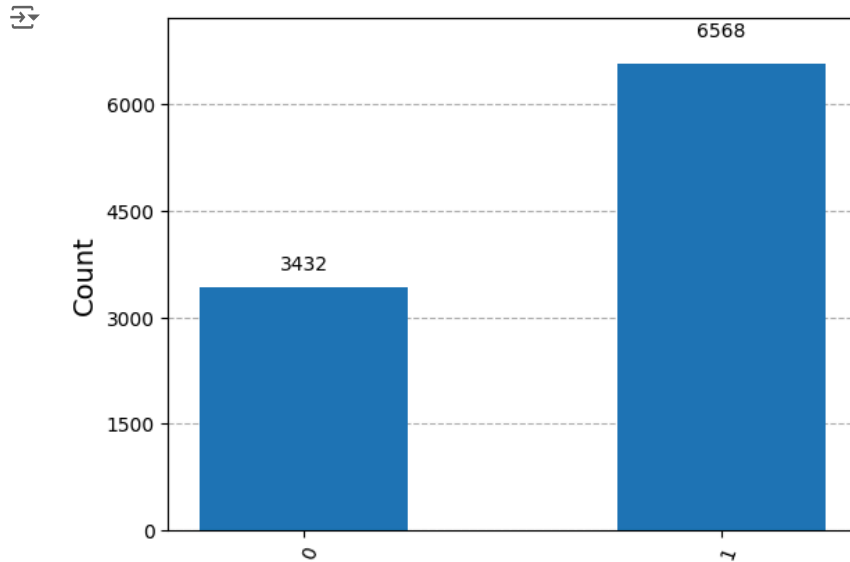


In this Circuit, the Eigenvector  $|\psi\rangle$  is  $|1\rangle$ .

```

from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
result = AerSimulator().run(qc, shots=10000).result()
statistics = result.get_counts()
plot_histogram(statistics)

```



```
print(statistics)
```

```
{'0': 3432, '1': 6568}
```

```
print("Probability to measure 0: ", statistics['0']/10000)
```

```
Probability to measure 0: 0.3432
```

```
print("Probability to measure 1: ", statistics['1']/10000)
```

```
Probability to measure 1: 0.6568
```

```
#We can now compare the results to the predicted values to see that they're correct.
```

```

# We make use of what we calculated mathematically in the course:
# p0 = [cos(pi*theta)]**2
# p1 = [sin(pi*theta)]**2

```

```

display({ # Calculate predicted results
    0: cos(pi * theta) ** 2,
    1: sin(pi * theta) ** 2
})


```

```
{0: 0.34549150281252616, 1: 0.6545084971874737}
```

## ✓ Quantum Fourier Transform for 2 Qubits

Remember that QFT2 = H, or that the 2-dimensional Quantum Fourier Transform is equivalent to the Hadamard Gate. We defined the QFT recursively as below, starting from the Hadamard Gate as QFT1.

Please look at the picture of the Circuit to complete the code!

Circuit2.png

```
# Define the 2-qubit Quantum Fourier Transform (QFT)
qfttwo = QuantumCircuit(2)

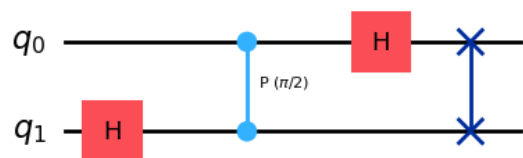
# Apply a Hadamard gate to the second qubit (q1)
qfttwo.h(1)

# Apply a controlled phase gate with angle  $\pi/2$ 
qfttwo.cp(pi / 2, 1, 0)

# Apply a Hadamard gate to the first qubit (q0)
qfttwo.h(0)

# Swap the qubits to complete the QFT
qfttwo.swap(0, 1)

# Draw the circuit
qfttwo.draw('mpl')
```



You will need this small thinking exercise in the next step.

## ✓ Two-Qubit Phase Estimation

Our eigenvector  $|\psi\rangle$  is  $|1\rangle$ .

We conditionally apply the eigenvalue once for qubit0.

We conditionally apply the eigenvalue twice for qubit1.

```

from math import pi
from qiskit import QuantumCircuit

theta = 0.7
qc = QuantumCircuit(3, 2)

# Prepare the eigenvector
qc.x(2)
qc.barrier()

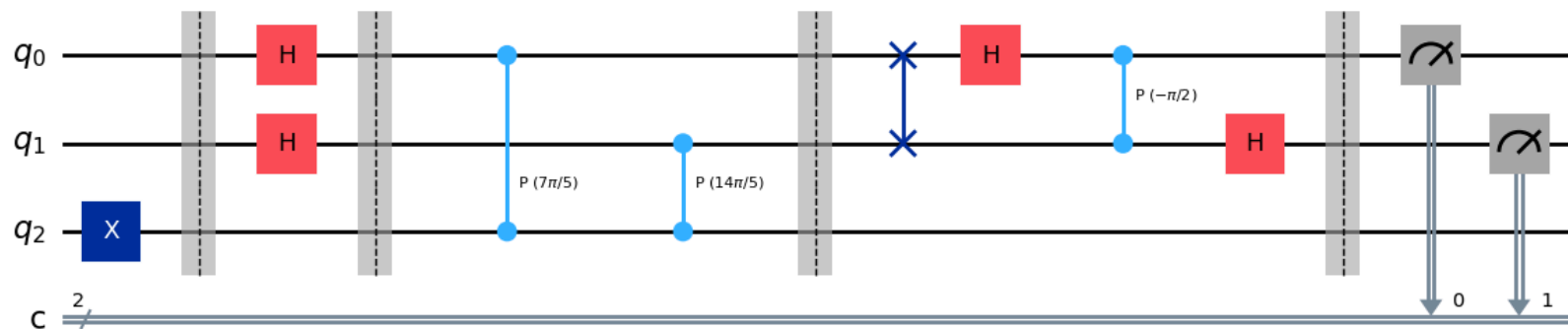
# The initial Hadamard gates
qc.h(0)
qc.h(1)
qc.barrier()

# The controlled unitary U gates
# Apply the U-Gate once if q0 is control Qubit.
# Apply the U-Gate twice if q1 is control Qubit.
qc.cp(2 * pi * theta, 0, 2)
qc.cp(2 * pi * (2 * theta), 1, 2)
qc.barrier()

# An implementation of the inverse of the two-qubit QFT2, in other words QFT-2
# Look to the previous exercise: apply the gates in the inverse sequence and the angle with a minus-sign.
qc.swap(0, 1) # First, apply the swap gate
qc.h(0)       # Reverse the Hadamard gate on q0
qc.cp(-pi / 2, 1, 0) # Controlled phase with negative angle
qc.h(1)       # Reverse the Hadamard gate on q1
qc.barrier()

# And finally the measurements
qc.measure([0, 1], [0, 1])
display(qc.draw('mpl'))

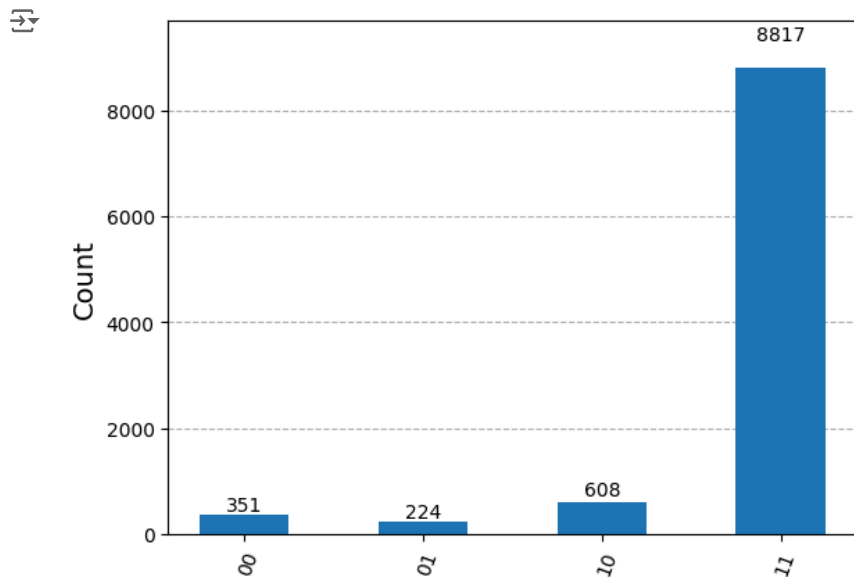
```



Execute the Circuit

```
result = AerSimulator().run(qc, shots=10000).result()
```

```
statistics = result.get_counts()
plot_histogram(statistics)
```

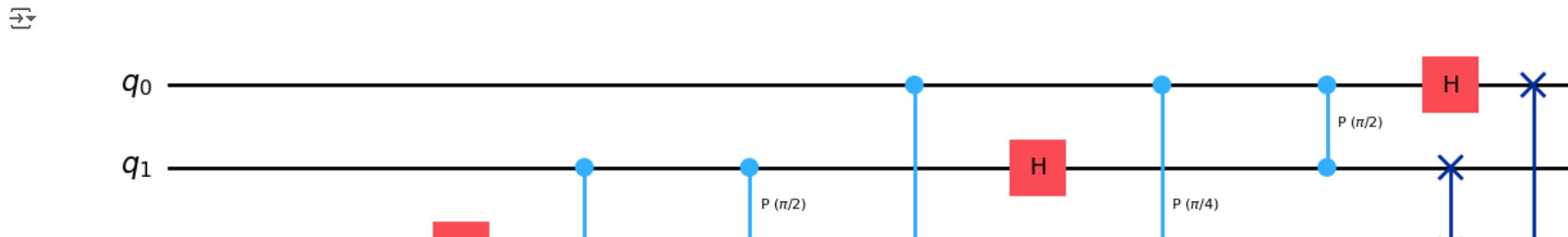


```
print("Probability to get decimal 0 = binary 00, or Theta = 0/4: ", statistics['00']/10000)
print("Probability to get decimal 1 = binary 01, or Theta = 1/4: ", statistics['01']/10000)
print("Probability to get decimal 2 = binary 10, or Theta = 2/4: ", statistics['10']/10000)
print("Probability to get decimal 3 = binary 11, or Theta = 3/4: ", statistics['11']/10000)
```

```
Probability to get decimal 0 = binary 00, or Theta = 0/4: 0.0351
Probability to get decimal 1 = binary 01, or Theta = 1/4: 0.0224
Probability to get decimal 2 = binary 10, or Theta = 2/4: 0.0608
Probability to get decimal 3 = binary 11, or Theta = 3/4: 0.8817
```

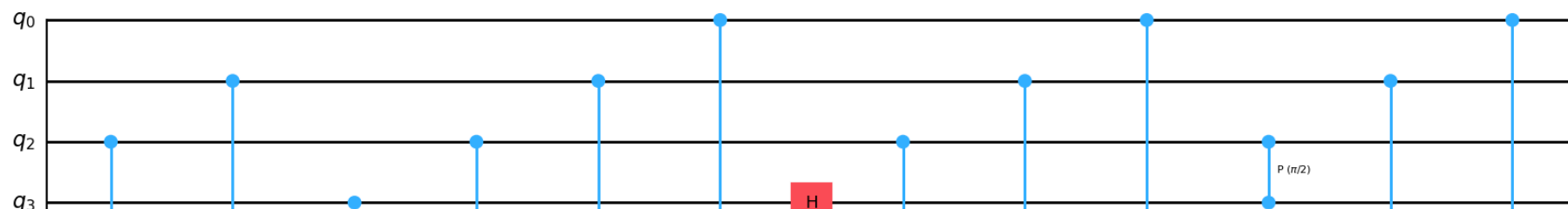
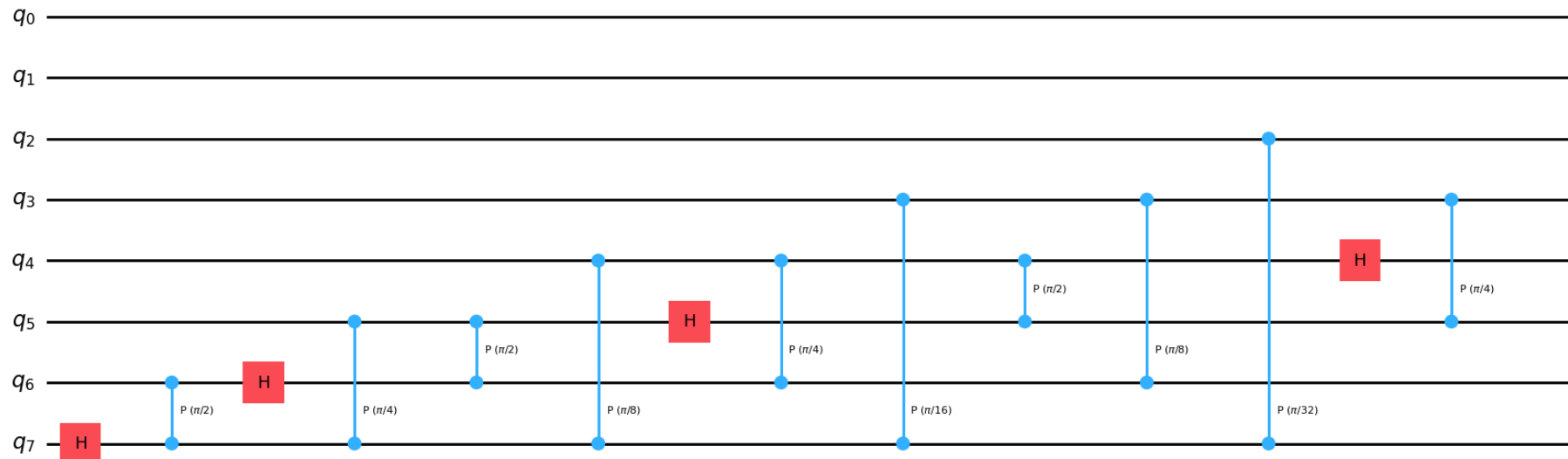
## QFTs in Qiskit

```
from qiskit.circuit.library import QFT
display(QFT(4).decompose().draw('mpl'))
```



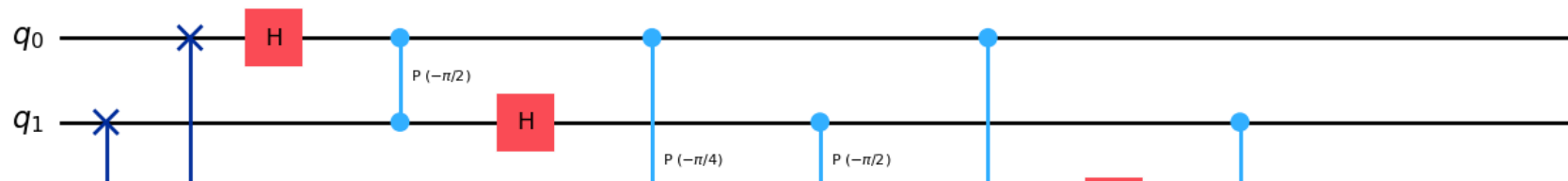
Now, we are going to draw the built-in QFT for 8 Qubits.

```
# Display the built-in QFT for 8 Qubits.
display(QFT(8).decompose().draw('mpl'))
```



```
#Now, we are going to draw the built-in Inverse QFT for 4 Qubits.
```

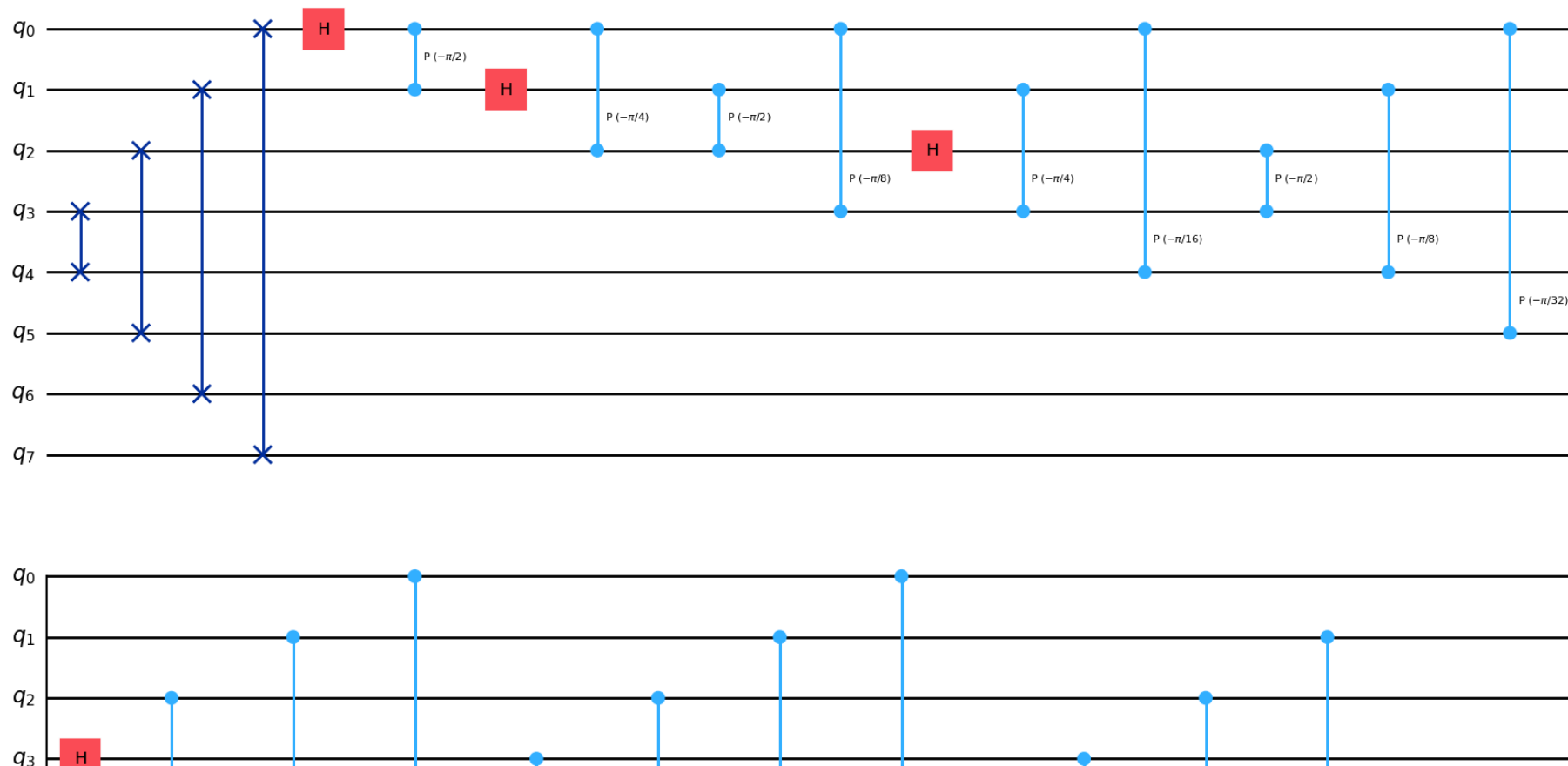
```
# Draw the built-in QFT4-1:
display(QFT(4, inverse=True).decompose().draw('mpl'))
```



```
For the fun, we are going to draw the built-in Inverse QFT for 8 Qubits.
```



```
# Draw the built-in QFT8-1:
display(QFT(8, inverse=True).decompose().draw('mpl'))
```



## Phase Estimation

Try adjusting  $\theta$  and the number of control qubits  $m$  to see how the results change.

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.circuit.library import QFT

theta = 0.7
m = 3 # Number of control qubits

control_register = QuantumRegister(m, name="Control")
target_register = QuantumRegister(1, name="|psi>")
output_register = ClassicalRegister(m, name="Result")
qc = QuantumCircuit(control_register, target_register, output_register)
```

```

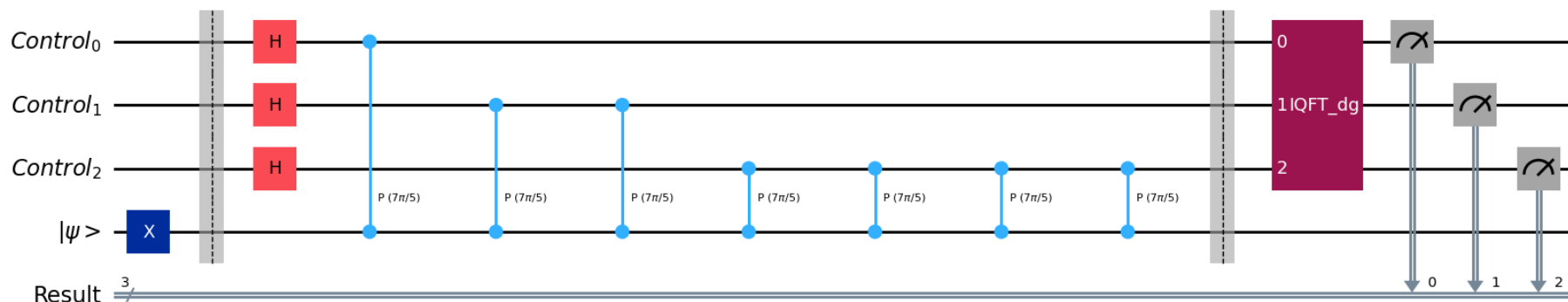
# Prepare the eigenvector  $|\psi\rangle$ 
qc.x(target_register)
qc.barrier()

# Perform phase estimation
for index, qubit in enumerate(control_register):
    qc.h(qubit)
    for _ in range(2**index):
        qc.cp(2 * pi * theta, qubit, target_register[0])
qc.barrier()

# Do inverse quantum Fourier transform
qc.compose(
    QFT(m, inverse=True), # Apply inverse QFT on the control register
    inplace=True
)

# Measure everything
qc.measure(range(m), range(m))
display(qc.draw('mpl'))

```



```

import warnings
warnings.filterwarnings("ignore")

```

```

from qiskit.primitives import Sampler

```

```

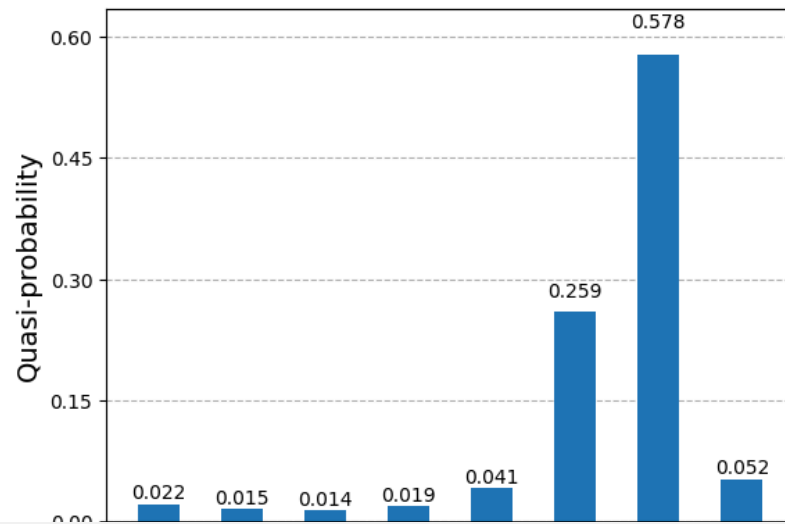
result = Sampler().run([qc]).result()

```

```

display(plot_histogram(result.quasi_dists))

```



```
most_probable = max(result.quasi_dists[0], key=result.quasi_dists[0].get)
```

```
print(f"Most probable output: {most_probable}")
```

```
print(f"Estimated theta: {most_probable/2**m}")
```



```
Most probable output: 6
Estimated theta: 0.75
```

## ✓ The Controlled Multiplication

First we'll hard code a controlled-multiplication operation for a given element  $a \in \mathbb{Z}_{15}$ .

In order to Hard-Code the controlled-multiplication, think about these aspects

- We represent bits with a decimal value between 0 and 15, for which we need 0000, ..., 1111.

*For multiplication by 2:*

- $0000 \times 2 = 0000$
- $0001 \times 2 = 0010$
- $0010 \times 2 = 0100$
- $0011 \times 2 = 0110$
- $0100 \times 2 = 1000$
- $0101 \times 2 = 1010$
- $0110 \times 2 = 1100$
- $0111 \times 2 = 1110$

- $1000 \times 2 = 0001$
- $1001 \times 2 = 0011$
- $1010 \times 2 = 0101$
- $1011 \times 2 = 0111$
- $1100 \times 2 = 1001$
- $1101 \times 2 = 1011$
- $1110 \times 2 = 1101$

*You can achieve this as follows with Bits 3210:*

- Swap Bit 2 and 3
- Swap Bit 1 and 2
- Swap Bit 0 and 1

The sequence of the Swaps is important and all of them must happen.

The same reasoning applies for the other multiplications by 4, 7, 8, 11, 13.

For some of them, we need even more Swaps.

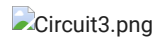
Please accept that the circuit below does the multiplication modulo 15, by applying smart Swaps.

Als, the multiplication modulo 15 is turned into a conditional Gate.

```
def c_amod15(a):
    """
    Controlled multiplication by a mod 15.
    This is hard-coded for simplicity.
    """
    if a not in [2, 4, 7, 8, 11, 13]:
        raise ValueError("'a' must not have common factors with 15")
    U = QuantumCircuit(4)
    if a in [2, 13]:
        U.swap(2, 3)
        U.swap(1, 2)
        U.swap(0, 1)
    if a in [7, 8]:
        U.swap(0, 1)
        U.swap(1, 2)
        U.swap(2, 3)
    if a in [4, 11]:
        U.swap(1, 3)
        U.swap(0, 2)
    if a in [7, 11, 13]:
        for q in range(4):
            U.x(q)
    U = U.to_gate()
    U.name = f"{a} mod 15"
    c_U = U.control()
    return c_U
```

## ✓ Algorithm of Shor - Illustration of the Circuit without Execution

Please look at the picture of the Circuit to complete the code!



```
N_COUNT = 8 # number of counting qubits
a = 7 # Relative prime with 15.
```

```
# Create QuantumCircuit with N_COUNT counting qubits
# plus 4 qubits for U to act on
qc = QuantumCircuit(N_COUNT + 4, N_COUNT)

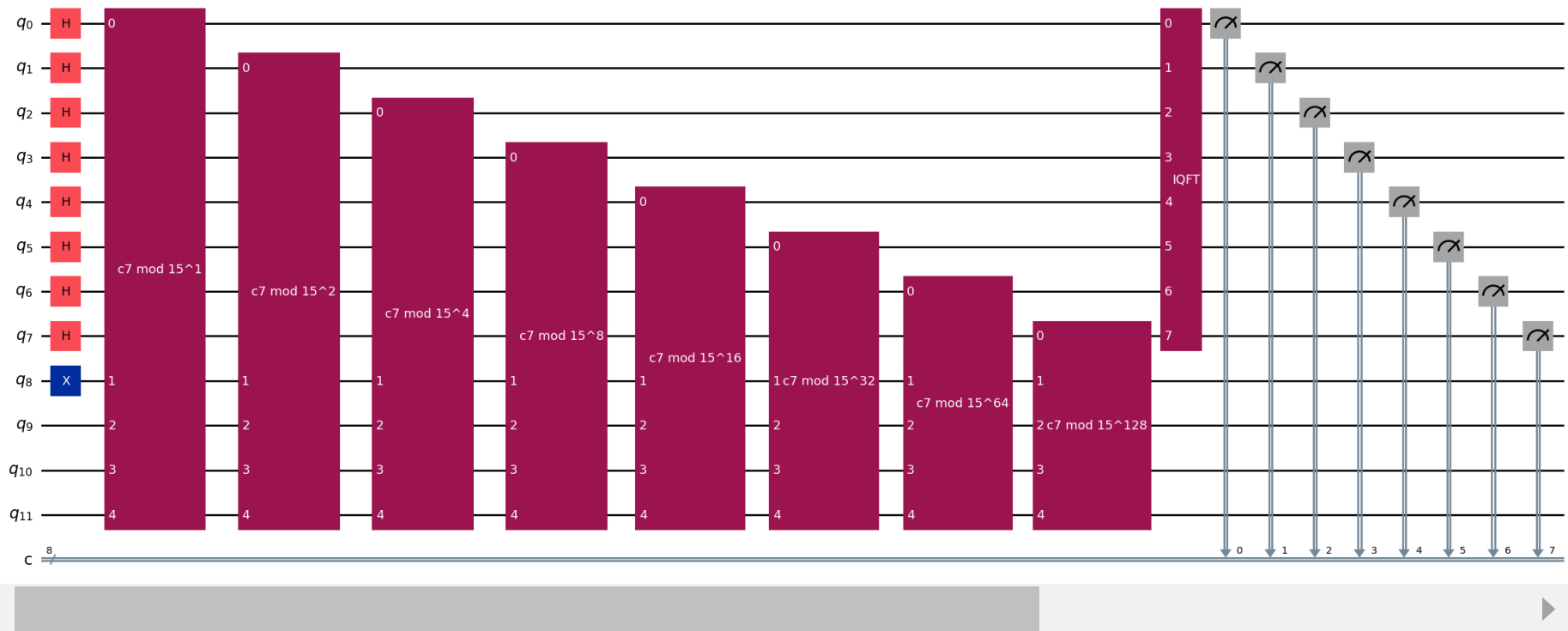
# Initialize counting qubits
# in state |+>
for q in range(N_COUNT):
    qc.h(q)

# And auxiliary register in state |1>
qc.x(N_COUNT)

# Do controlled-U operations
for q in range(N_COUNT):
    qc.append(c_amod15(a).power(2**q), # Apply controlled multiplication
              [q] + [i+N_COUNT for i in range(4)])

# Do inverse-QFT
qc.append(QFT(N_COUNT, inverse=True), range(N_COUNT))

# Measure circuit
qc.measure(range(N_COUNT), range(N_COUNT))
qc.draw('mpl', fold=-1) # -1 means 'do not fold'
```



## ✓ Algorithm of Shor - Finding the Prime Factors

Here's the phase estimation procedure from earlier implemented as a function.

```
def phase_estimation(
    controlled_operation: QuantumCircuit,
    psi_prep: QuantumCircuit,
    precision: int
):
    """
    Carry out phase estimation on a simulator.
    Args:
        controlled_operation: The operation to perform phase estimation on,
                               controlled by one qubit.
        psi_prep: Circuit to prepare  $|\psi\rangle$ 
        precision: Number of counting qubits to use
    Returns:
        float: Best guess for phase of  $U|\psi\rangle$ 
    """
    control_register = QuantumRegister(precision)
    output_register = ClassicalRegister(precision)
    target_register = QuantumRegister(psi_prep.num_qubits)
```

```

target_register = QuantumRegister(psi_prep.num_qubits)
qc = QuantumCircuit(control_register, target_register, output_register)

# Prepare  $|\psi\rangle$ 
qc.compose(psi_prep,
            qubits=target_register,
            inplace=True)

# Do phase estimation
for index, qubit in enumerate(control_register):
    qc.h(qubit)
    for _ in range(2**index):
        qc.compose(
            controlled_operation,
            qubits=[qubit] + list(target_register),
            inplace=True,
        )

qc.compose(
    QFT(precision, inverse=True),
    qubits=control_register,
    inplace=True
)

qc.measure(control_register, output_register)

measurement = Sampler().run(qc, shots=1).result().quasi_dists[0].popitem()[0]
return measurement / 2**precision

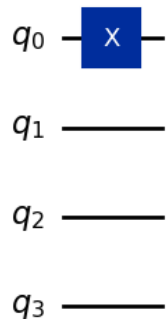
```

We can't easily prepare eigenvectors of the multiplication by  $a \in \mathbb{Z}_{15}$  operation, so we use  $|1\rangle$  as suggested.

```

psi_prep = QuantumCircuit(4)
psi_prep.x(0)
display(psi_prep.draw('mpl'))

```



And finally we can run the circuit to try to find a nontrivial factor of 15.

```

# Remind what the controlled_operation is about!
# Reminf what the psi_prep is about!
from fractions import Fraction
from math import gcd

a = 8
N = 15
# Define controlled modular multiplication
def controlled_modular_multiplication(a, n):
    return c_amod15(a)

FACTOR_FOUND = False
ATTEMPT = 0
while not FACTOR_FOUND:
    ATTEMPT += 1
    print(f"\nAttempt {ATTEMPT}")

    # Perform phase estimation
    phase = phase_estimation(
        controlled_modular_multiplication(a, N), # Controlled operation
        psi_prep, # Initial state preparation
        precision=8 # Number of control qubits
    )
    frac = Fraction(phase).limit_denominator(N)
    r = frac.denominator
    if phase != 0:
        # Guess for a factor is gcd(x^{r/2} - 1 , 15)
        guess = gcd(a ** (r // 2) - 1, N)
        if guess not in [1, N] and (N % guess) == 0:
            # Guess is a factor!
            print(f"Non-trivial factor found: {guess}")
            FACTOR_FOUND = True

```



```

Attempt 1
Non-trivial factor found: 3

```

#### ✓ Find the prime factors of N

```

# Output the factors
print("N =", N)
print("Prime Factor 1 =", guess)
print("Prime Factor 2 =", int(N / guess))

```