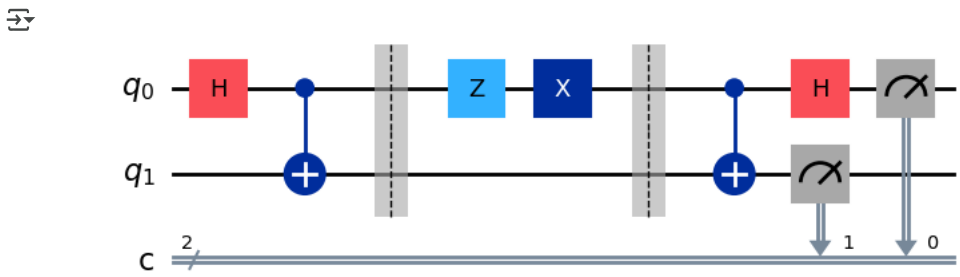
Superdense Coding

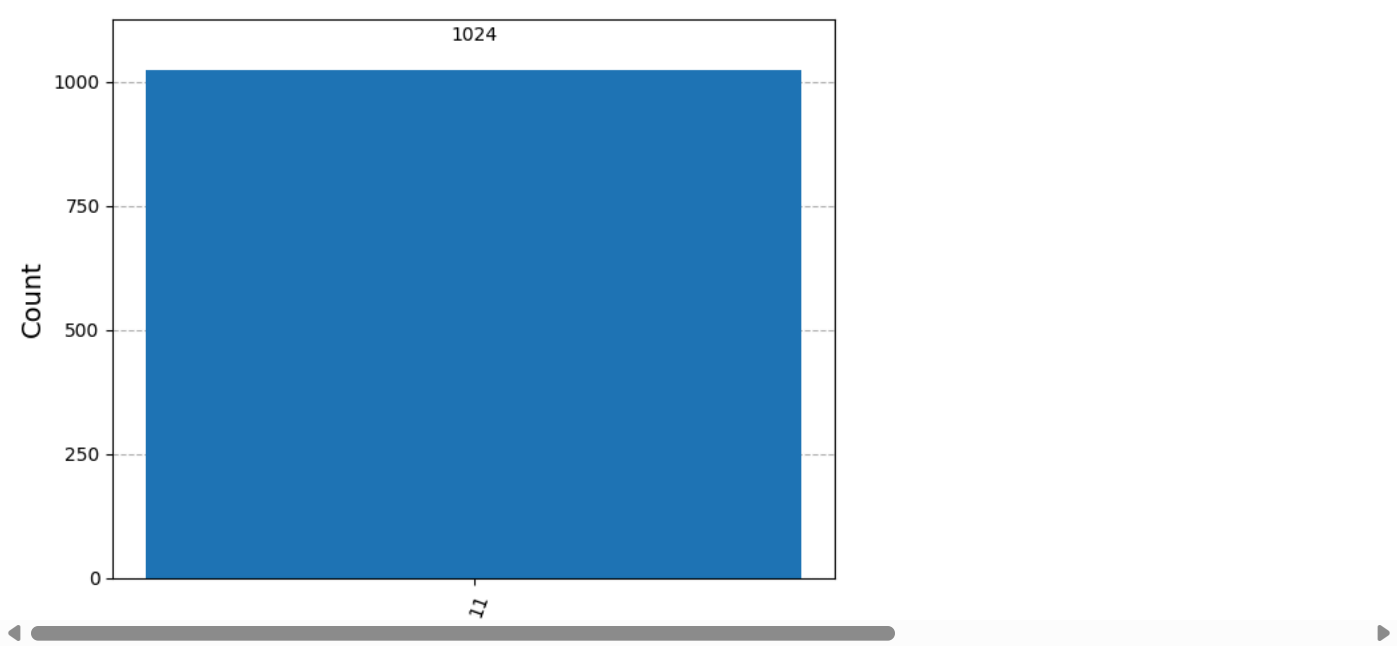
```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# Function to create the Superdense Coding circuit
def superdense_coding(message='10'):
    qc = QuantumCircuit(2, 2)

    # Step 1: Create Bell pair
    qc.h(0)
    qc.cx(0, 1)
    qc.barrier()

    # Step 2: Alice encodes her 2 classical bits
    if message == '00':
        pass
    elif message == '01':
        qc.x(0)
    elif message == '10':
        qc.z(0)
    elif message == '11':
        qc.z(0)
        qc.x(0)
    else:
        raise ValueError("Message must be one of: '00', '01', '10', or '11'")

    qc.barrier()

    # Step 3: Bob decodes
    qc.cx(0, 1)
    qc.h(0)

    # Step 4: Measure
    qc.measure(0, 0)
    qc.measure(1, 1)

    return qc

# Encode a message
message = '11'
qc = superdense_coding(message)

# Simulate using AerSimulator
simulator = AerSimulator()
compiled = transpile(qc, simulator)
result = simulator.run(compiled, shots=1024).result()
counts = result.get_counts()

# Display circuit
qc.draw('mpl')
```




```
# Output
print(f"Encoded message: {message}")
```

```
Encoded message: 11
```

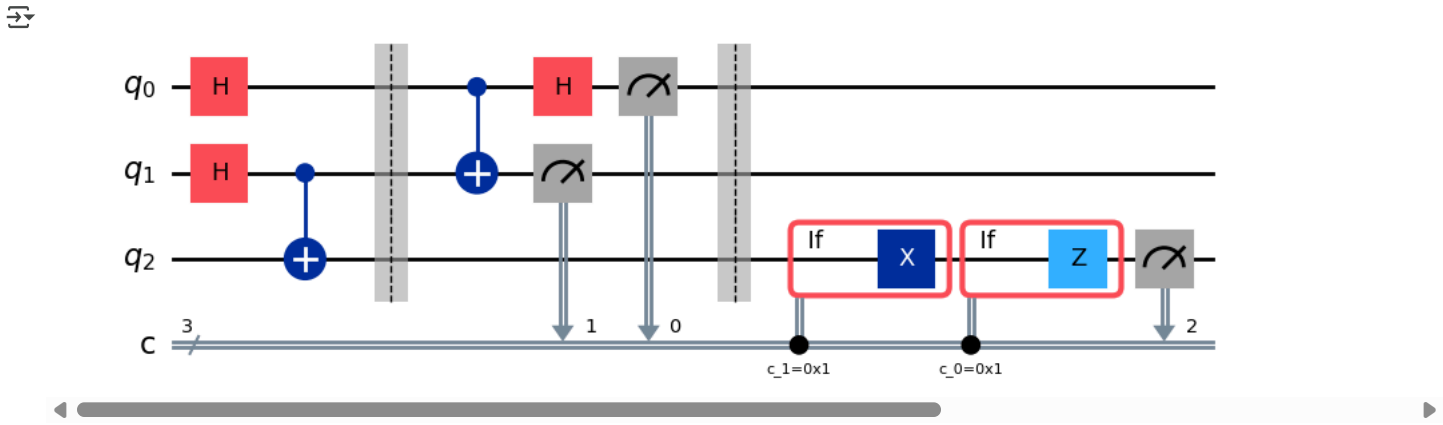
```
plot_histogram(counts)
```



Teleportation

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

def teleportation_circuit():
    # Define quantum and classical registers
    q = QuantumRegister(3, name='q')
    c = ClassicalRegister(3, name='c')
    qc = QuantumCircuit(q, c)

    # Step 1: Prepare qubit 0 in |+⟩ = (|0⟩ + |1⟩)/√2
    qc.h(q[0])

    # Step 2: Create Bell pair between qubits 1 and 2
    qc.h(q[1])
    qc.cx(q[1], q[2])
    qc.barrier()

    #Step 3: Alice performs operations on her qubits
    qc.cx(q[0], q[1])
    qc.h(q[0])

    # Step 4: Bell measurement on qubits 0 and 1
    qc.measure(q[0], c[0])
    qc.measure(q[1], c[1])
    qc.barrier()

    # Step 5: Conditional operations on Bob’s qubit
    with qc.if_test((c[1], 1)):
        qc.x(q[2])
    with qc.if_test((c[0], 1)):
        qc.z(q[2])

    # Step 6: Final measurement
    qc.measure(q[2], c[2])

    return qc

# Build and simulate
qc = teleportation_circuit()
```

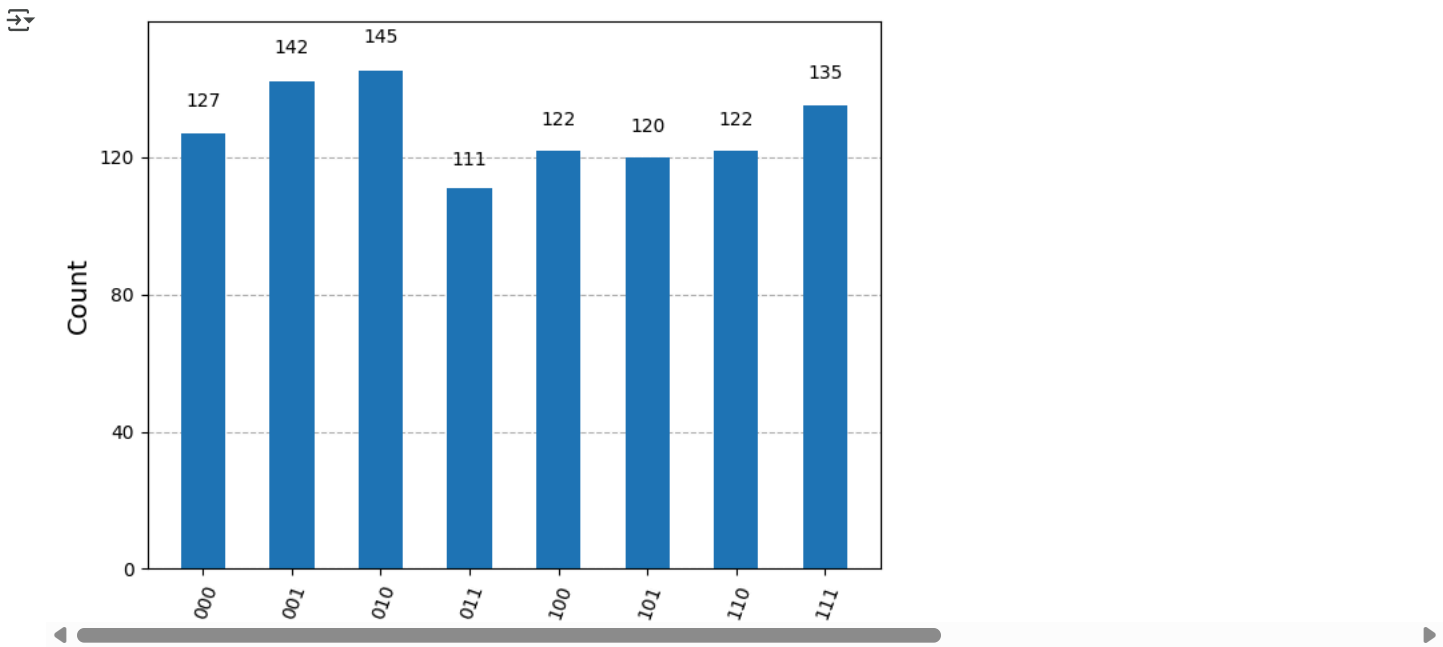
```
simulator = AerSimulator()
compiled = transpile(qc, simulator)
result = simulator.run(compiled, shots=1024).result()
counts = result.get_counts()

# Draw circuit
qc.draw('mpl')
```




```
# Display results
print("Measurement results:", counts)
```

```
Measurement results: {'100': 122, '011': 111, '111': 135, '110': 122, '001': 142, '000': 127, '010': 145, '101': 120}
```

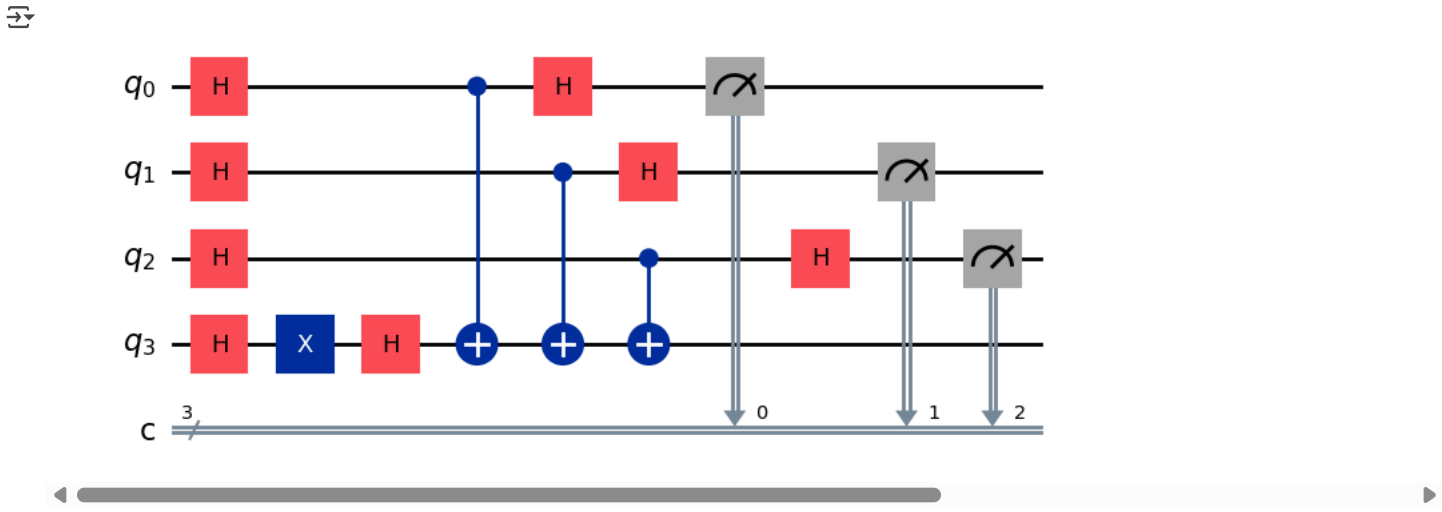
```
plot_histogram(counts)
```



Deutsch-Jozsa Algorithm

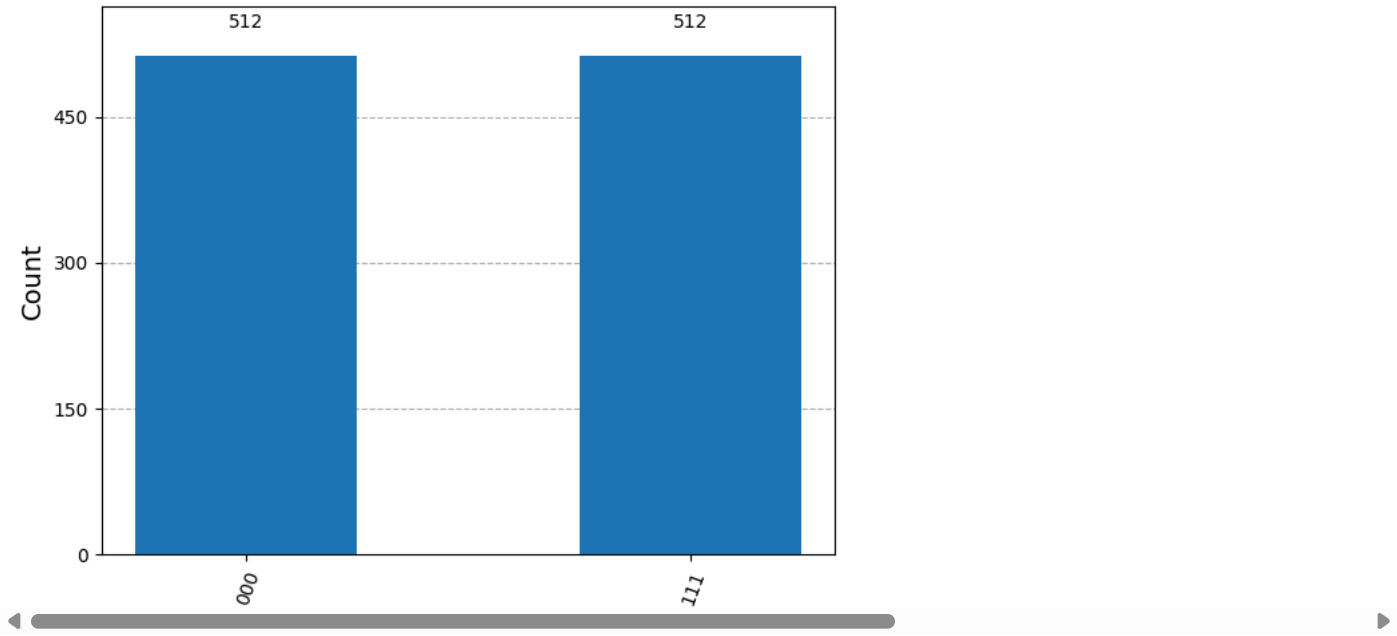
```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

#balanced oracle
def balanced_oracle(qc, n):
    for qubit in range(n):
        qc.cx(qubit, n)

#constant oracle
```

```
def constant_oracle_zero(qc, n):
    pass  # Do nothing

def constant_oracle_one(qc, n):
    qc.x(n)

#deutsch-jozsa algorithm
def deutsch_jozsa(n):
    # Create a quantum circuit with n+1 qubits (n qubits for input, 1 for output)
    qc = QuantumCircuit(n + 1, n)

    # Step 1: Apply Hadamard gate to all qubits (including the output qubit)
    qc.h(range(n))  # Apply H to the first n qubits
    qc.h(n)         # Apply H to the last qubit (output qubit)

    # Step 2: Apply X and H to the last qubit
    qc.x(n)         # Apply X gate to the last qubit (sets it to |1>)
    qc.h(n)         # Apply H gate to put it in superposition

    # Step 3: Apply the oracle
    balanced_oracle(qc, n)  # or constant_oracle_zero(qc, n)

    # Step 4: Apply Hadamard gate to the first n qubits again
    qc.h(range(n))

    # Step 5: Measure the first n qubits
    qc.measure(range(n), range(n))

    return qc

# Create and simulate the Deutsch-Jozsa circuit
n = 3  # Number of input qubits
qc = deutsch_jozsa(n)

# Use AerSimulator for simulation
simulator = AerSimulator()

# Transpile and run the circuit
compiled = transpile(qc, simulator)
result = simulator.run(compiled).result()
counts = result.get_counts()

# Draw and show the results
qc.draw('mpl')
```



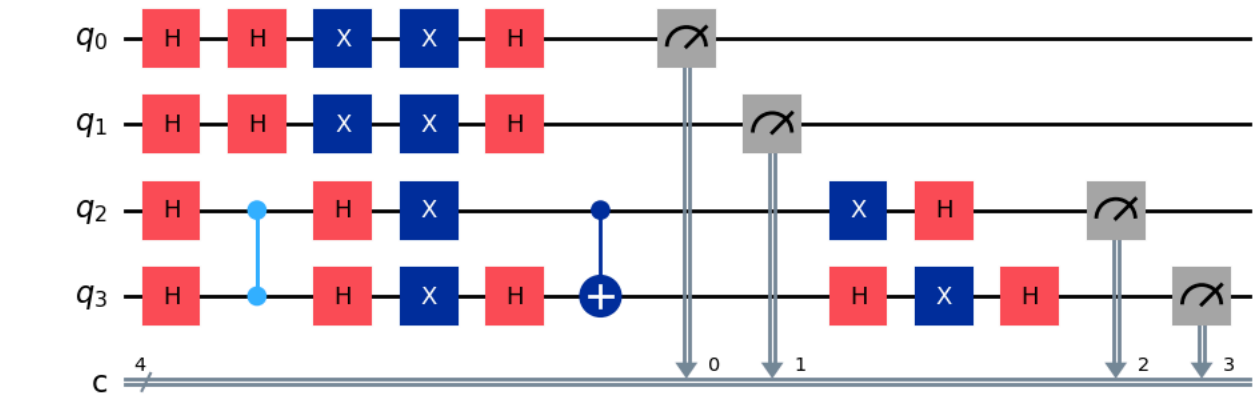

```
print("Measurement results:", counts)
plot_histogram(counts)
```

```
Measurement results: {'111': 512, '000': 512}
```



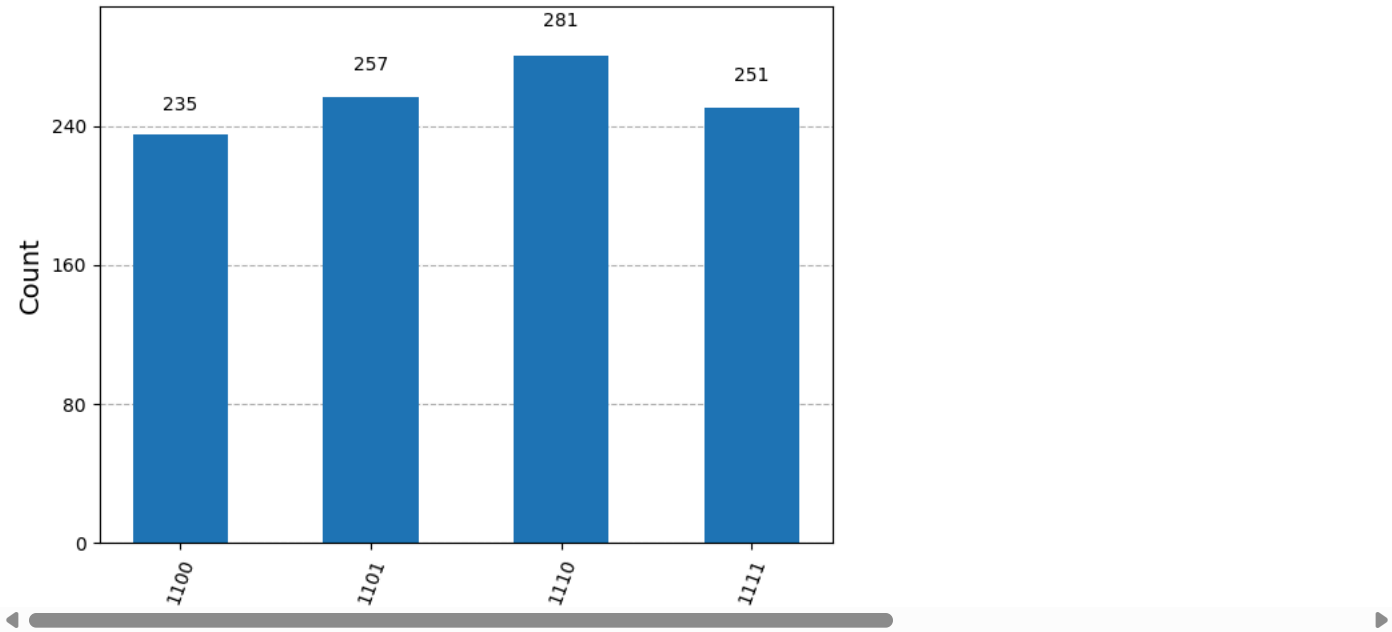
Grover's Algorithm

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

def grover_algorithm(n):
    # Create a quantum circuit with n qubits and n classical bits
    qc = QuantumCircuit(n, n)

    # Step 1: Apply Hadamard gates to all qubits (initialize to superposition)
    qc.h(range(n))

    # Step 2: Apply the oracle (in this case, we will use the oracle for |11> as the target)
    qc.cz(n-2, n-1)  # Oracle for marking the state |11>

    # Step 3: Apply the Grover diffusion operator (inversion about the average)
    qc.h(range(n))
    qc.x(range(n))
    qc.h(n-1)
    qc.cx(n-2, n-1)
    qc.h(n-1)
    qc.x(range(n))
    qc.h(range(n))

    # Step 4: Measure the qubits
    qc.measure(range(n), range(n))

    return qc

# Create and simulate the Grover's algorithm circuit
n = 4  # Number of qubits
qc = grover_algorithm(n)

# Use AerSimulator for simulation
simulator = AerSimulator()

# Transpile and run the circuit
compiled = transpile(qc, simulator)
result = simulator.run(compiled).result()
counts = result.get_counts()

# Draw and show the results
qc.draw('mpl')
```




```
# Display measurement results
print("Measurement results:", counts)
plot_histogram(counts)
```

```
Measurement results: {'1111': 251, '1101': 257, '1100': 235, '1110': 281}
```

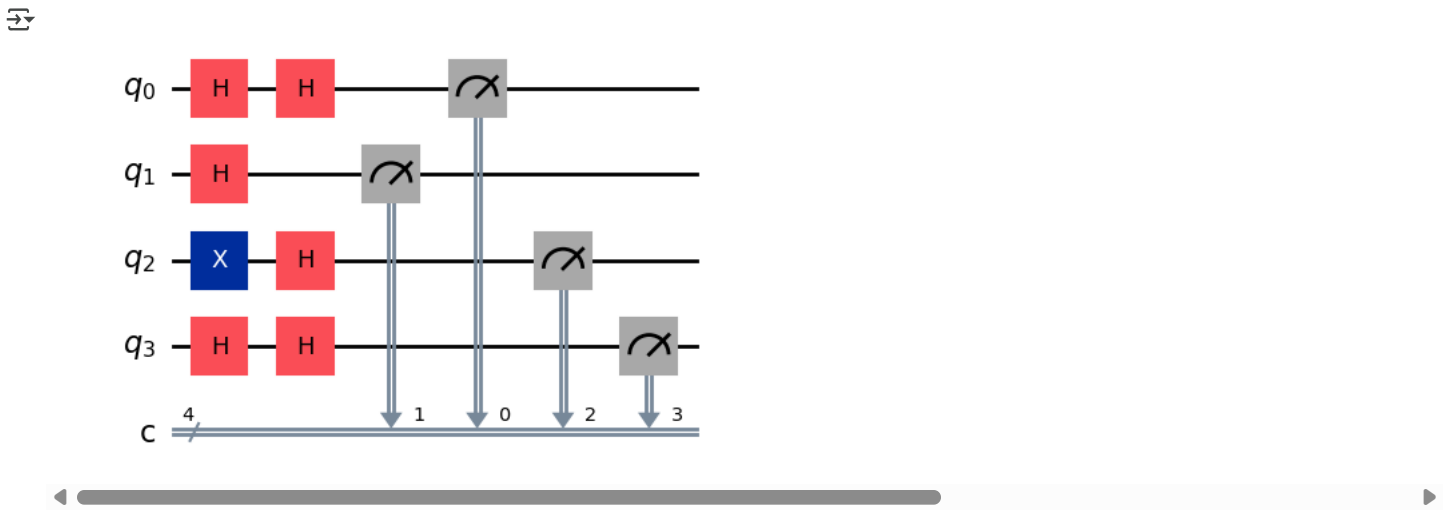


BB84

(Benett & Brassard 1984)

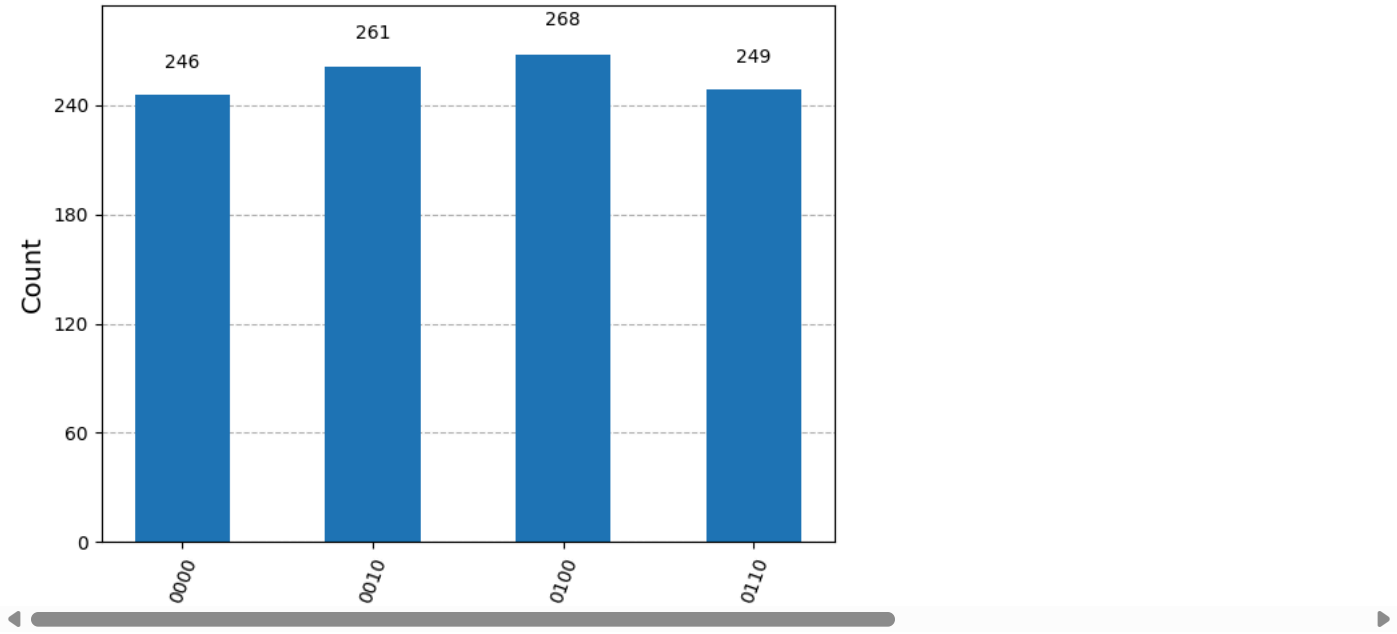
```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

def bb84_algorithm(n):
    # Alice's Preparation: Create a quantum circuit with n qubits and n classical bits
    qc = QuantumCircuit(n, n)

    # Step 1: Alice prepares qubits in random states (in this example, we'll set a few random bits)
    import random
    alice_bits = [random.randint(0, 1) for _ in range(n)]
    alice_bases = [random.randint(0, 1) for _ in range(n)]  # 0 = computational basis, 1 = diagonal basis

    # Alice prepares the qubits in either computational or diagonal basis based on the chosen basis
    for i in range(n):
        if alice_bases[i] == 0:  # Computational basis
```

```
            if alice_bits[i] == 1:
                qc.x(i)  # Apply X gate to flip the qubit
        else:  # Diagonal basis (Hadamard basis)
            if alice_bits[i] == 1:
                qc.x(i)
            qc.h(i)

    # Step 2: Alice sends the qubits to Bob (no need to simulate sending)
    # Step 3: Bob measures the qubits in random bases
    bob_bases = [random.randint(0, 1) for _ in range(n)]  # Bob chooses random bases

    for i in range(n):
        if bob_bases[i] == 0:  # Computational basis
            pass  # No change, Bob uses the qubit as is
        else:  # Diagonal basis
            qc.h(i)  # Apply Hadamard to switch to diagonal basis

    # Step 4: Bob measures the qubits and stores the result
    qc.measure(range(n), range(n))

    return qc, alice_bits, alice_bases, bob_bases

# Create the BB84 circuit with 4 qubits (for simplicity)
n = 4
qc, alice_bits, alice_bases, bob_bases = bb84_algorithm(n)

# Use AerSimulator for simulation
simulator = AerSimulator()

# Transpile and run the circuit
compiled = transpile(qc, simulator)
result = simulator.run(compiled).result()
counts = result.get_counts()

# Draw and show the results
qc.draw('mpl')
```




```
# Display measurement results and key
print("Measurement results:", counts)
plot_histogram(counts)
```

```
Measurement results: {'0110': 249, '0100': 268, '0000': 246, '0010': 261}
```




```
print("Alice's bits:", alice_bits)
print("Alice's bases:", alice_bases)
print("Bob's bases:", bob_bases)

# Extract the final shared key (where Alice's and Bob's bases match)
shared_key = []
for i in range(n):
    if alice_bases[i] == bob_bases[i]:
        shared_key.append(str(counts.get(bin(i)[2:].zfill(n), 0)))
print("Shared key:", ''.join(shared_key))
```

```
Alice's bits: [0, 0, 1, 0]
Alice's bases: [1, 0, 0, 1]
Bob's bases: [1, 1, 1, 1]
Shared key: 2460
```

Shor's Algorithm

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from numpy import pi
from fractions import Fraction
import math
from qiskit.circuit.library import UnitaryGate
from qiskit.quantum_info import Operator

def controlled_mod_mul(a, N, power, n_count, n_target):
    """Returns a controlled modular multiplication gate"""
    mod_a = pow(a, power, N)
    dim = 2 ** n_target
    matrix = [[0] * dim for _ in range(dim)]
    for i in range(dim):
        if i < N:
            new_i = (mod_a * i) % N
        else:
            new_i = i
        matrix[new_i][i] = 1
    U = Operator(matrix)
    gate = UnitaryGate(U.data, label=f"{a}^{power} mod {N}")
    cgate = gate.control(1)
    return cgate

def qft_dagger(n):
    qc = QuantumCircuit(n)
    for qubit in range(n // 2):
        qc.swap(qubit, n - qubit - 1)
    for j in range(n):
```

```
            for m in range(j):
                qc.cp(-pi / float(2 ** (j - m)), m, j)
            qc.h(j)
        qc.name = "QFT†"
        return qc

    def phase_estimation_mod_exp(a, N):
        n_count = 4
        n_target = 4
        qc = QuantumCircuit(n_count + n_target, n_count)

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

        # Set target register to |1⟩
        qc.x(n_count + n_target - 1)

        # Controlled modular exponentiation
        for i in range(n_count):
            power = 2 ** i
            cgate = controlled_mod_mul(a, N, power, n_count, n_target)
            control = i
            target_qubits = list(range(n_count, n_count + n_target))
            qc.append(cgate, [control] + target_qubits)

        # Apply inverse QFT
        qc.append(qft_dagger(n_count), range(n_count))

        # Measure counting qubits
        qc.measure(range(n_count), range(n_count))
        return qc

    # Parameters
    a = 7
    N = 15
    qc = phase_estimation_mod_exp(a, N)

    # Simulate
    backend = AerSimulator()
    compiled = transpile(qc, backend)
    result = backend.run(compiled, shots=1024).result()
    counts = result.get_counts()

    # Analyze
    print("Measurement counts:", counts)
    measured = max(counts, key=counts.get)
    decimal = int(measured, 2) / (2 ** len(measured))
    print(f"Measured value: {measured} => {decimal}")
    frac = Fraction(decimal).limit_denominator(N)
    r = frac.denominator
    print(f"Estimated period r = {r}")

    # Try factoring
    if r % 2 == 0:
        x = pow(a, r // 2, N)
        factor1 = math.gcd(x - 1, N)
        factor2 = math.gcd(x + 1, N)
        if factor1 * factor2 == N and factor1 != 1 and factor2 != 1:
            print(f"Factors of {N} are: {factor1} and {factor2}")
        else:
            print("GCDs did not yield non-trivial factors.")
    else:
        print("Period r is odd, factoring failed.")
```

```
Measurement counts: {'0100': 248, '1000': 229, '1100': 282, '0000': 265}
Measured value: 1100 => 0.75
Estimated period r = 4
Factors of 15 are: 3 and 5
```