# 通信實驗 Lab 3

B07901103 電機三 陳孟宏

## <Problem 1> Deutsch–Jozsa algorithm

### (a) Here I randomly generate output / input

1. Constant function : output = 0 ( y has no any gate )
2. Constant function : output = 1 ( y will have a “X-gate” )

```
# set the length of the n-bit input string.
n = 4
```

```
# Constant Oracle
const_oracle = QuantumCircuit(n+1)
output = np.random.randint(2)
if (output == 1):
    const_oracle.x(n)

print("Output: ", output)
const_oracle.draw()
```

```
Output:  0

q_0:
q_1:
q_2:
q_3:
q_4:
```

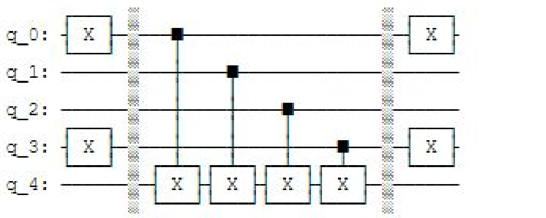
```
# Constant Oracle
const_oracle = QuantumCircuit(n+1)
output = np.random.randint(2)
if (output == 1):
    const_oracle.x(n)

print("Output: ", output)
const_oracle.draw()
```

```
Output:  1

q_0: ─────
q_1: ─────
q_2: ─────
q_3: ─────
q_4: ┤ X ├
```

3. Balanced function : output_0 : output_1 = 1 : 1

```
# Balanced Oracle
balanced_oracle = QuantumCircuit(n+1)

# Random generate 4-bit
b_str = ""
for i in range(4):
    temp = np.random.randint(2)
    b_str += str(temp)

print("Input: ", b_str)

# Place X-gates
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)

# Use barrier as divider
balanced_oracle.barrier()

# Controlled-NOT gates
for qubit in range(n):
    balanced_oracle.cx(qubit, n)

balanced_oracle.barrier()

# Place X-gates
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)

# Show oracle
balanced_oracle.draw()
```

```
Input:  1001
```



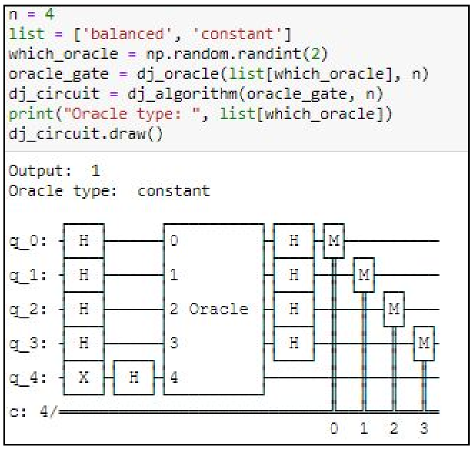
**(b)**

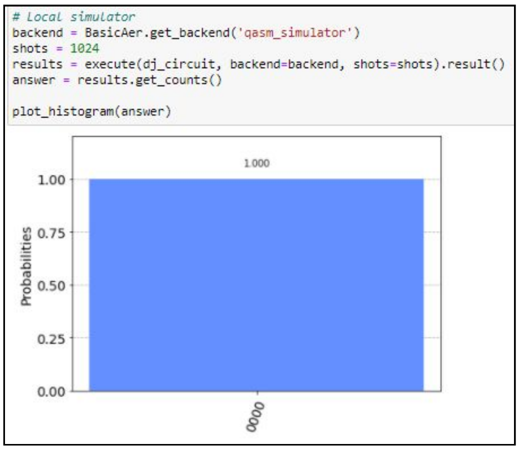
1. Define general oracle (constant / balanced) and apply the DJ-algorithm

```python
# General Case (case = 'balanced' / 'constant')
def dj_oracle(case, n):
    oracle_qc = QuantumCircuit(n+1)

    # First, let's deal with the case in which oracle is balanced
    if case == "balanced":
        # Random generate 4-bit
        b_str = ""
        for i in range(4):
            temp = np.random.randint(2)
            b_str += str(temp)

        print("Input: ", b_str)

        # Place X-gates
        for qubit in range(len(b_str)):
            if b_str[qubit] == '1':
                oracle_qc.x(qubit)

        # Controlled-NOT gates
        for qubit in range(n):
            oracle_qc.cx(qubit, n)

        # Place X-gates
        for qubit in range(len(b_str)):
            if b_str[qubit] == '1':
                oracle_qc.x(qubit)

    # Case in which oracle is constant
    if case == "constant":
        output = np.random.randint(2)
        if (output == 1):
            const_oracle.x(n)
        print("Output: ", output)

    # To show when we display the circuit
    oracle_gate = oracle_qc.to_gate()
    oracle_gate.name = "Oracle"
    return oracle_gate
```

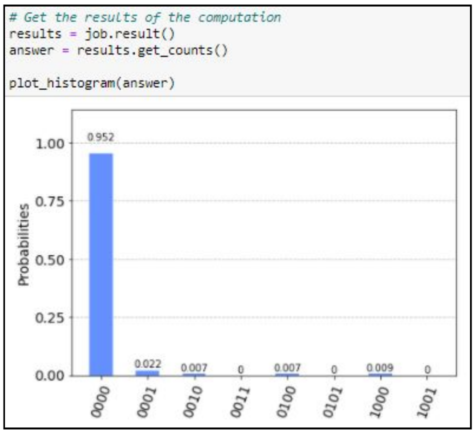
```python
# Performs the Deutsch-Joza algorithm
def dj_algorithm(oracle, n):
    dj_circuit = QuantumCircuit(n+1, n)
    # Set up the output qubit:
    dj_circuit.x(n)
    dj_circuit.h(n)

    for qubit in range(n):
        dj_circuit.h(qubit)

    dj_circuit.append(oracle, range(n+1))

    for qubit in range(n):
        dj_circuit.h(qubit)

    for i in range(n):
        dj_circuit.measure(i, i)

    return dj_circuit
```

2. When the oracle is a constant function, output = 0000



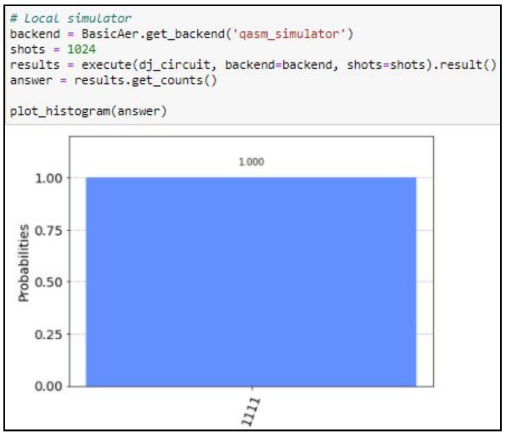



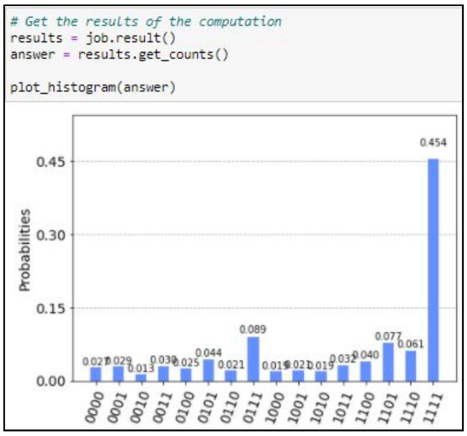

( local simulator )




( real IBMQ )

3. When the oracle is a balanced function, output = 1111



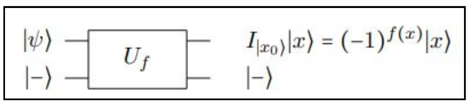




( local simulator ) ( real IBMQ )

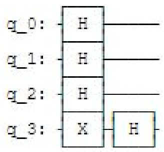
# <Problem 2> Grover's Search

## (a)

1. First, initialize the superposition state before entering the oracle U_f, and observe its state vector.

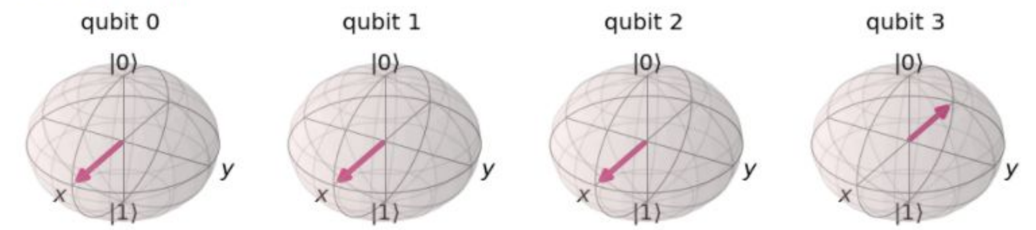


```python
# Inversion about mean
qc = QuantumCircuit(4)
qc.x(3)
# Apply transformation |s> -> |00..0> (H-gates)
for qubit in range(4):
    qc.h(qubit)

# Draw
qc.draw()
```

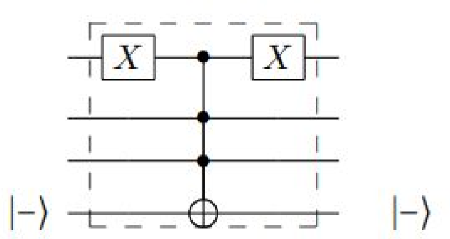



```python
# optional, verify qubit state by using the following commands to get the vector form
simulator = Aer.get_backend('statevector_simulator')
job = execute(qc, simulator)
result = job.result()
statevector1 = result.get_statevector()
print("Before oracle: ", statevector1)
plot_bloch_multivector(statevector1)
```

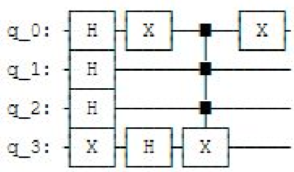
```
Before oracle:  [ 0.25-3.061617e-17j  0.25-3.061617e-17j  0.25-3.061617e-17j
  0.25-3.061617e-17j  0.25-3.061617e-17j  0.25-3.061617e-17j
  0.25-3.061617e-17j  0.25-3.061617e-17j -0.25+3.061617e-17j
 -0.25+3.061617e-17j -0.25+3.061617e-17j -0.25+3.061617e-17j
 -0.25+3.061617e-17j -0.25+3.061617e-17j -0.25+3.061617e-17j
 -0.25+3.061617e-17j]
```




2. Then, apply the given oracle and observe its state vector.

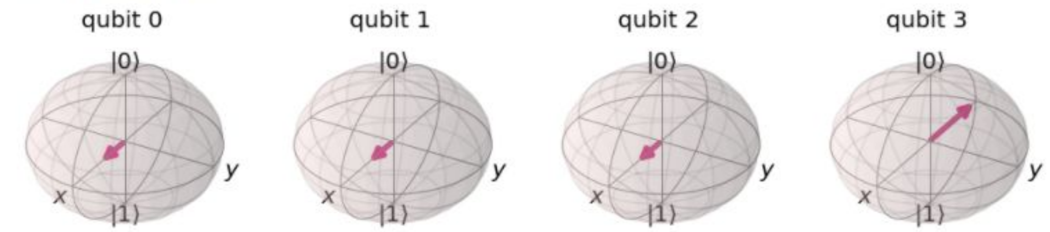



```python
# Inversion about mean
qc = QuantumCircuit(4)
qc.x(3)
# Apply transformation |s> -> |00..0> (H-gates)
for qubit in range(4):
    qc.h(qubit)
qc.x(0)
qc.mct(list(range(3)), 3)  # multi-controlled-toffoli
# Apply transformation |11..1> -> |00..0>
qc.x(0)

# Draw
qc.draw()
```



```
# optional, verify qubit state by using the following commands to get the vector form
simulator = Aer.get_backend('statevector_simulator')
job = execute(qc, simulator)
result = job.result()
statevector2 = result.get_statevector()
print("After oracle: ", statevector2)
plot_bloch_multivector(statevector2)
```

```
After oracle:  [ 0.25-6.12323400e-17j  0.25-6.12323400e-17j  0.25-6.12323400e-17j
  0.25-6.12323400e-17j  0.25-6.12323400e-17j  0.25-6.12323400e-17j
 -0.25+1.22464680e-16j  0.25-6.12323400e-17j -0.25+9.18485099e-17j
 -0.25+9.18485099e-17j -0.25+9.18485099e-17j -0.25+9.18485099e-17j
 -0.25+9.18485099e-17j -0.25+9.18485099e-17j  0.25-9.18485099e-17j
 -0.25+9.18485099e-17j]
```



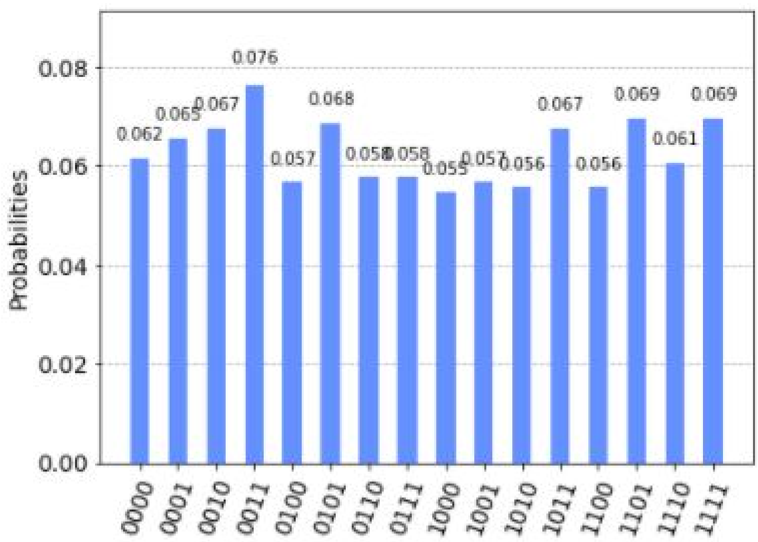

3. At the last, subtract the above two state vector ( before-oracle & after-oracle ).

```
print("After oracle - Before oracle: ", statevector1-statevector2)
```

```
After oracle - Before oracle:  [ 0. +3.0616170e-17j  0. +3.0616170e-17j  0. +3.0616170e-17j
  0. +3.0616170e-17j  0. +3.0616170e-17j  0. +3.0616170e-17j
  0.5-1.5308085e-16j  0. +3.0616170e-17j  0. -6.1232340e-17j
  0. -6.1232340e-17j  0. -6.1232340e-17j  0. -6.1232340e-17j
  0. -6.1232340e-17j  0. -6.1232340e-17j -0.5+1.2246468e-16j
  0. -6.1232340e-17j]
```

( The index number 7 and the index number 15, note that their subtraction results are +0.5, -0.5 respectively, which is a flip. )

```
qc.measure_all()
backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```



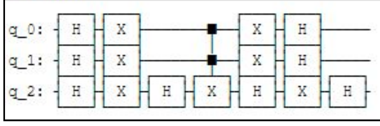

( Ignore the leftmost bit, We can see that the index number 7 is “011”, while the index number 15 is “011”, too. )

**(b)**

1. Design a “inversion about mean” via textbook.

$$I_{|\psi_0\rangle} := H^{\otimes n} \left(2(|0\rangle\langle 0|)^{\otimes n} - I\right) H^{\otimes n}$$

```
# Inversion about mean
nqubits = 3
qc = QuantumCircuit(nqubits)
# Apply transformation |s> -> |00..0> (H-gates)
for qubit in range(nqubits):
    qc.h(qubit)
# Apply transformation |00..0> -> |11..1> (X-gates)
for qubit in range(nqubits):
    qc.x(qubit)
# Do multi-controlled-Z gate
qc.h(nqubits-1)
qc.mct(list(range(nqubits-1)), nqubits-1)  # multi-controlled-toffoli
qc.h(nqubits-1)
# Apply transformation |11..1> -> |00..0>
for qubit in range(nqubits):
    qc.x(qubit)
# Apply transformation |00..0> -> |s>
for qubit in range(nqubits):
    qc.h(qubit)

# Draw
qc.draw()
```

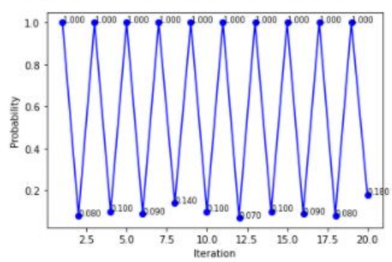



**(c)**

1. For every iteration, I measure the “shots=100000” with 100 times. If one time the maximum measurement is “011”, I mark it as “correct search” to calculate the successful probability.

```
prob = []
U_num = 1
for i in range(20):
    q2 = QuantumRegister(4)
    c2 = ClassicalRegister(4)
    qc2 = QuantumCircuit(q2, c2)
    qc2 = initialize_s(qc2, [0, 1, 2])
    qc2.append(oracle_ex3, [0, 1, 2])
    for i in range(U_num):
        qc2.append(diffuser(4), [0, 1, 2, 3])
    correct = 0
    for j in range(100):
        qc2.measure([0, 1, 2], [0, 1, 2])
        backend = Aer.get_backend('qasm_simulator')
        results = execute(qc2, backend=backend, shots=1024).result()
        answer = results.get_counts()
        # Count
        maxi = 0
        bit = ""
        for i in answer:
            if (answer[i] > maxi):
                maxi = answer[i]
                bit = i
        if (bit == "0110"):
            correct += 1
    prob.append(correct/100)
    U_num += 1
print("20 iteration's probability: ", prob)

20 iteration's probability:  [1.0, 0.08, 1.0, 0.1, 1.0, 0.09, 1.0, 0.14, 1.0, 0.1, 1.0, 0.07, 1.0, 0.1, 1.0, 0.09, 1.0, 0.08,
1.0, 0.18]
```

```
# Plot
plt.xlabel("Iteration")
plt.ylabel("Probability")

ite = []
for i in range(20): ite.append(i+1)

for x, y in zip(ite, prob):
    plt.text(x, y,'%.3f' % y,fontdict={'fontsize':8})

plt.plot(ite, prob, 'b-o')

[<matplotlib.lines.Line2D at 0x1cd843da190>]
```



2. Before executing the probability calculation, I need to define some functions first.

```python
# Oracle
qco = QuantumCircuit(3)
qco.x(0)
qco.h(2)
qco.mct(list(range(2)), 2)
qco.h(2)
qco.x(0)
oracle_ex3 = qco.to_gate()
oracle_ex3.name = "Oracle"
```

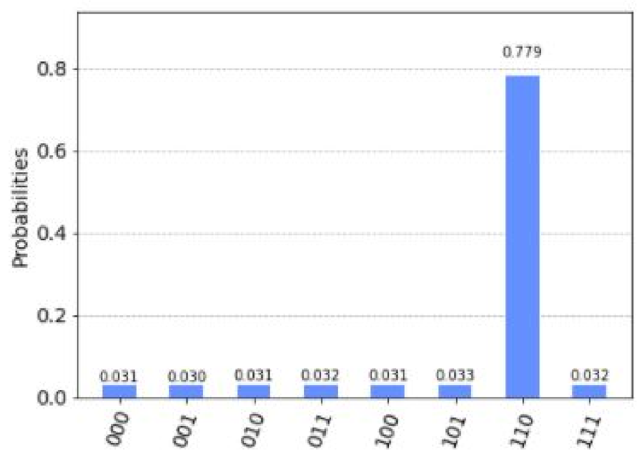
( Oracle )

```python
def initialize_s(qc, qubits):
    for q in qubits:
        qc.h(q)
    return qc

def diffuser(nqubits):
    qc = QuantumCircuit(nqubits)
    # Apply transformation |s> -> |00..0> (H-gates)
    for qubit in range(nqubits):
        qc.h(qubit)
    # Apply transformation |00..0> -> |11..1> (X-gates)
    for qubit in range(nqubits):
        qc.x(qubit)
    # Do multi-controlled-Z gate
    qc.h(nqubits-1)
    qc.mct(list(range(nqubits-1)), nqubits-1)  # multi-controlled-toffoli
    qc.h(nqubits-1)
    # Apply transformation |11..1> -> |00..0>
    for qubit in range(nqubits):
        qc.x(qubit)
    # Apply transformation |00..0> -> |s>
    for qubit in range(nqubits):
        qc.h(qubit)
    # We will return the diffuser as a gate
    U_s = qc.to_gate()
    U_s.name = "U_inv"
    return U_s
```

( Initialize superposition states / Inversion about mean / Amplifier )

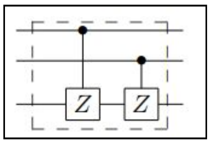
3. Also, I use “qasm simulator” to verify the oracle is correct for “011”.

```python
# qasm simulator
n = 3
qc1 = QuantumCircuit(n)
qc1 = initialize_s(qc1, [0,1,2])
qc1.append(oracle_ex3, [0,1,2])
qc1.append(diffuser(n), [0,1,2])
qc1.measure_all()
backend = Aer.get_backend('qasm_simulator')
results = execute(qc1, backend=backend, shots=100000).result()
answer = results.get_counts()
plot_histogram(answer)
```



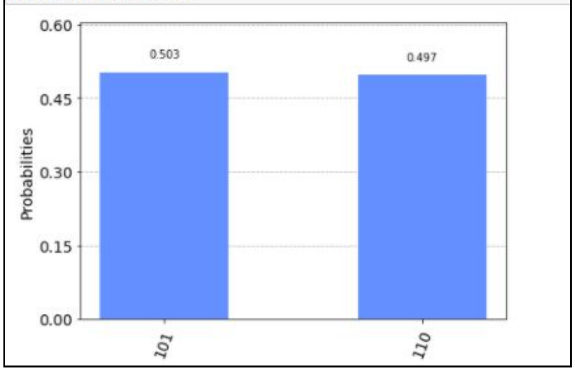

**(d)**

1. Use the given oracle to get our new targets: “011”, “101”.




2. Based on 2-(a) and 2-(b)(c)(d), if I apply my specific oracle to the Grover’s search device, the target will be “inversed” and “amplified”. Hence, now assume that I apply the oracle of (“011”, “101”) into my search device, I should get two peaks: “011” and “101”.

```
qc = QuantumCircuit(3)
qc = initialize_s(qc, [0,1,2])
qc.cz(0, 2)
qc.cz(1, 2)
qc.append(diffuser(3), [0,1,2])
qc.measure_all()

backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```




**(e)**

1. Same as 2-(c), for every iteration, I measure the “shots=100000” with 100 times. If one time the maximum measurement is “011”, I mark it as “correct search” to calculate the successful probability.
2. In addition, I calculate the query times until there is one iteration making the successful probability to be “1”. Otherwise, I will put one more “Grover’s search device” to the quantum circuit.

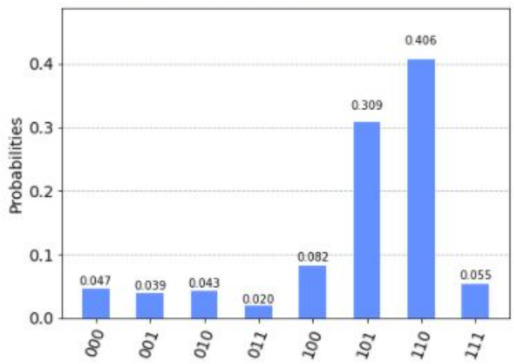
```
prob, count, U_num = 0, 0, 0
U_num += 1
q2 = QuantumRegister(3)
c2 = ClassicalRegister(3)
qc = QuantumCircuit(q2, c2)
qc = initialize_s(qc, [0, 1, 2])
qc.cz(0, 2)
qc.cz(1, 2)
while (prob != 1):
    for i in range(U_num):
        qc.append(diffuser(3), [0, 1, 2])
    correct = 0
    for j in range(100):
        qc.measure([0, 1, 2], [0, 1, 2])
        backend = Aer.get_backend('qasm_simulator')
        results = execute(qc, backend=backend, shots=1024).result()
        answer = results.get_counts()
        # Count
        maxi = 0
        bit = ""
        for i in answer:
            if (answer[i] > maxi):
                maxi = answer[i]
                bit = i
        if (bit == "110" or bit == "101"):
            correct += 1
    prob = int(correct/100)
    count += 1
print("Query: ", count)

Query:  1
```

( Query times: 1 )

3. Also, I run the circuit at the real IBMQ, we can notice that the peaks are "101" and "011", although there are still some states (trivial error).

```
# Monitoring our job
from qiskit.tools.monitor import job_monitor

q2 = QuantumRegister(3)
c2 = ClassicalRegister(3)
qc = QuantumCircuit(q2, c2)
qc = initialize_s(qc, [0, 1, 2])
qc.cz(0, 2)
qc.cz(1, 2)
qc.append(diffuser(3), [0, 1, 2])
qc.measure([0, 1, 2], [0, 1, 2])

# Run our circuit
job = execute(qc, backend=backend, shots=shots)
job_monitor(job)

# Plotting our result
result = job.result()
count = result.get_counts()
plot_histogram(count)
```

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

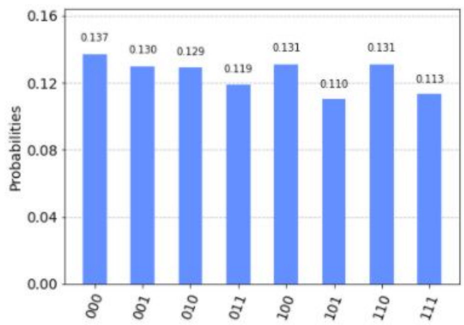



**(f)**

1. Use the same operation as 2-(c), now target are "011", "110", "101", "111", because we flip the signs of 4 out of 8 states, the mean of 000~111 will be zero, so we cannot amplify some specific bit strings, and they will have the same probability.

```
q2 = QuantumRegister(3)
c2 = ClassicalRegister(3)
qc = QuantumCircuit(q2, c2)
qc = initialize_s(qc, [0, 1, 2])
qc.cz(0, 2)
qc.cz(1, 2)
qc.cz(1, 0)
qc.append(diffuser(3), [0, 1, 2])
qc.measure([0, 1, 2], [0, 1, 2])
backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend=backend, shots=1024).result()
answer = results.get_counts()
print(answer)

plot_histogram(answer)
```

```
{'000': 140, '001': 133, '010': 132, '011': 122, '100': 134, '101': 113, '110': 134, '111': 116}
```

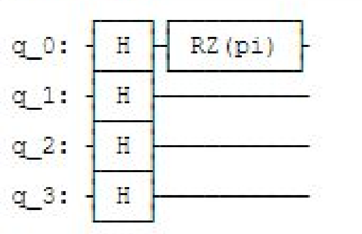


# <Problem 3> Quantum Fourier Transform

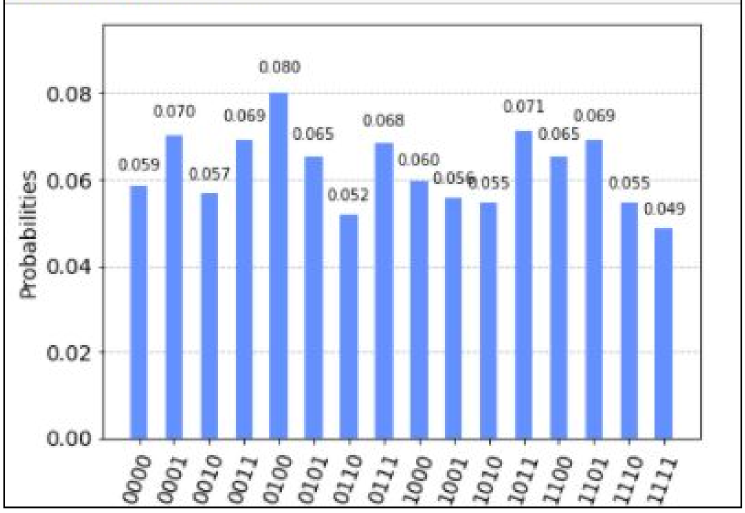
## (a) Directly measure $|\phi 1\rangle$, $|\phi 2\rangle$, and $|\phi 3\rangle$

1. $|\phi 1\rangle$: measure = equi-probability

```python
# phi_1
qc_a = QuantumCircuit(4)
for i in range(4):
    qc_a.h(i)
qc_a.rz(pi, 0)
qc_a.draw()
```




```python
qc_a.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
results = execute(qc_a, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```

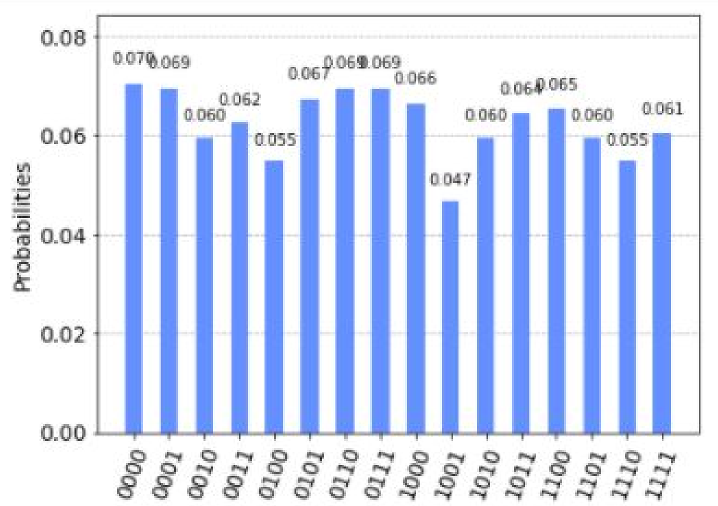



2. $|\phi 2\rangle$: measure = equi-probability

```python
#phi_2
qc_b = QuantumCircuit(4)
for i in range(4):
    qc_b.h(i)
qc_b.rz(pi/2, 0)
qc_b.ry(pi, 1)
qc_b.draw()
```

```
     ┌───┐┌──────────┐
q_0: ┤ H ├┤ RZ(pi/2) ├
     ├───┤└┬────────┬┘
q_1: ┤ H ├─┤ RY(pi) ├─
     ├───┤ └────────┘
q_2: ┤ H ├────────────
     ├───┤
q_3: ┤ H ├────────────
     └───┘
```

```python
qc_b.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
results = execute(qc_b, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```

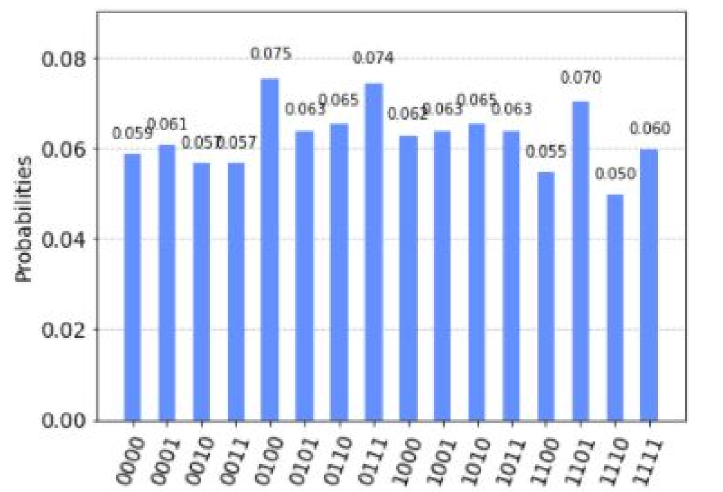



3. $|\phi 3\rangle$: measure = equi-probability

```python
# phi_3
qc_c = QuantumCircuit(4)
for i in range(4):
    qc_c.h(i)
qc_c.rz(pi/4, 0)
qc_c.rz(pi/2, 1)
qc_c.rz(pi, 2)
qc_c.draw()
```

```
     ┌───┐┌─────────┐
q_0: ┤ H ├┤ RZ(pi/4) ├
     ├───┤├─────────┤
q_1: ┤ H ├┤ RZ(pi/2) ├
     ├───┤└┬───────┬┘
q_2: ┤ H ├─┤ RZ(pi) ├─
     ├───┤ └───────┘
q_3: ┤ H ├────────────
     └───┘
```

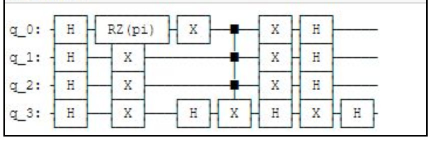
```python
qc_c.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
results = execute(qc_c, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```



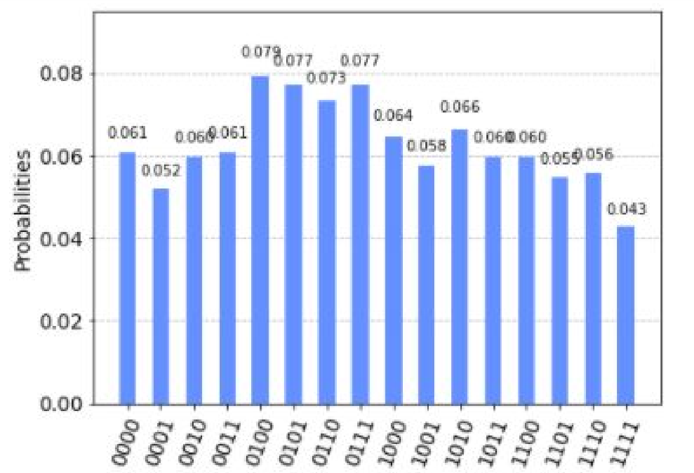

## (b) Apply amplitude amplifier to $|\phi 1\rangle$, $|\phi 2\rangle$, and $|\phi 3\rangle$

1. $|\phi 1\rangle$: measure = equi-probability

```python
# phi_1
qc_a = QuantumCircuit(4)
for i in range(4):
    qc_a.h(i)
qc_a.rz(pi, 0)

for qubit in range(4):
    qc_a.x(qubit)
# Do multi-controlled-Z gate
qc_a.h(3)
qc_a.mct(list(range(3)), 3)  # multi-controlled-toffoli
qc_a.h(3)
# Apply transformation |11..1> -> |00..0>
for qubit in range(4):
    qc_a.x(qubit)
# Apply transformation |00..0> -> |s>
for qubit in range(4):
    qc_a.h(qubit)

qc_a.draw()
```

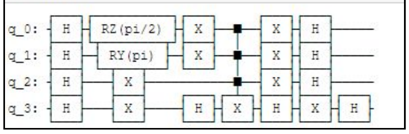



```python
qc_a.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
results = execute(qc_a, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```

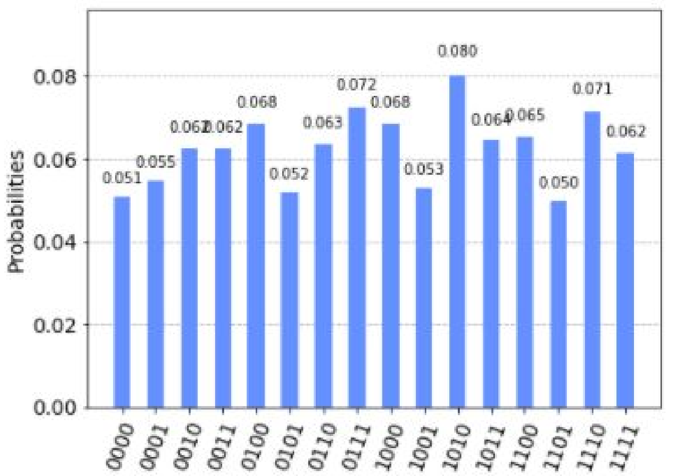



2. $|\phi 2\rangle$: measure = equi-probability

```python
# phi_2
qc_b = QuantumCircuit(4)
for i in range(4):
    qc_b.h(i)
qc_b.rz(pi/2, 0)
qc_b.ry(pi, 1)

for qubit in range(4):
    qc_b.x(qubit)
# Do multi-controlled-Z gate
qc_b.h(3)
qc_b.mct(list(range(3)), 3)  # multi-controlled-toffoli
qc_b.h(3)
# Apply transformation |11..1> -> |00..0>
for qubit in range(4):
    qc_b.x(qubit)
# Apply transformation |00..0> -> |s>
for qubit in range(4):
    qc_b.h(qubit)

qc_b.draw()
```

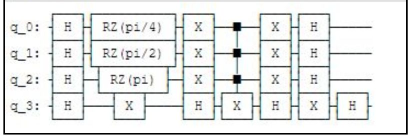



```python
qc_b.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
results = execute(qc_b, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```

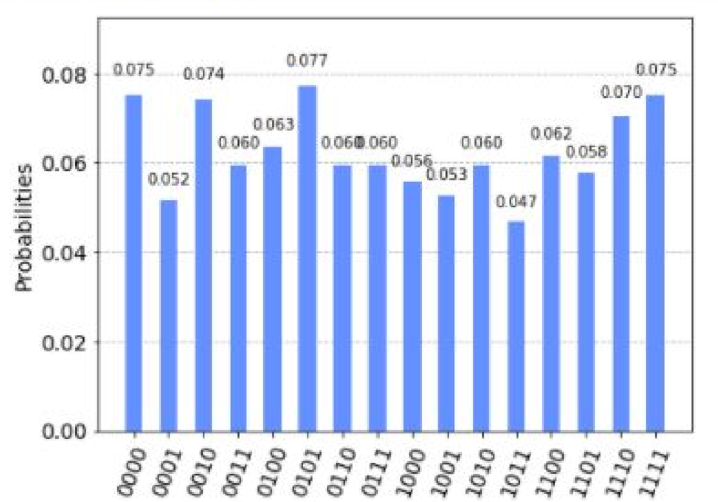



3. $|\phi 3\rangle$: measure = equi-probability

```python
# phi_3
qc_c = QuantumCircuit(4)
for i in range(4):
    qc_c.h(i)
qc_c.rz(pi/4, 0)
qc_c.rz(pi/2, 1)
qc_c.rz(pi, 2)

for qubit in range(4):
    qc_c.x(qubit)
# Do multi-controlled-Z gate
qc_c.h(3)
qc_c.mct(list(range(3)), 3)  # multi-controlled-toffoli
qc_c.h(3)
# Apply transformation |11..1> -> |00..0>
for qubit in range(4):
    qc_c.x(qubit)
# Apply transformation |00..0> -> |s>
for qubit in range(4):
    qc_c.h(qubit)

qc_c.draw()
```




```python
qc_c.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
results = execute(qc_c, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```




## (c) Apply qft_dagger to $|\phi 1\rangle$, $|\phi 2\rangle$, and $|\phi 3\rangle$

```python
def qft(n):
    """n-qubit QFT the first n qubits in circuit"""
    qc = QuantumCircuit(n)
    # Don't forget the Swaps!
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cu1(np.pi/float(2**(j-m)), m, j)
        qc.h(j)
    qc.name = "QFT"
    return qc

def qft_dagger(n):
    """n-qubit QFTdagger the first n qubits in circuit"""
    qc = QuantumCircuit(n)
    # Don't forget the Swaps!
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cu1(-np.pi/float(2**(j-m)), m, j)
        qc.h(j)
    qc.name = "QFT"
    return qc
```

(Sample code of QFT)

1. $|\phi 1\rangle$

```python
# phi_1
qc_a = QuantumCircuit(4, 4)
for i in range(4):
    qc_a.h(i)
qc_a.rz(pi, 0)

q_1 = qc_a + qft_dagger(4)
q_1.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
answer_1 = execute(q_1, backend=backend).result().get_counts()
print(answer_1)
```

```
{'1000': 1024}
```

2. $|\phi 2\rangle$

```python
# phi_2
qc_b = QuantumCircuit(4, 4)
for i in range(4):
    qc_b.h(i)
qc_b.rz(pi/2, 0)
qc_b.ry(pi, 1)

q_2 = qc_b + qft_dagger(4)
q_2.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
answer_2 = execute(q_2, backend=backend).result().get_counts()
print(answer_2)
```

```
{'0100': 1024}
```

3. $|\phi 3\rangle$

```python
# phi_3
qc_c = QuantumCircuit(4, 4)
for i in range(4):
    qc_c.h(i)
qc_c.rz(pi/4, 0)
qc_c.rz(pi/2, 1)
qc_c.rz(pi, 2)

q_3 = qc_c + qft_dagger(4)
q_3.measure([0, 1, 2, 3], [0, 1, 2, 3])
backend = Aer.get_backend('qasm_simulator')
answer_3 = execute(q_3, backend=backend).result().get_counts()
print(answer_3)
```

```
{'0010': 1024}
```

# <Problem 4> Period-Finding Algorithm

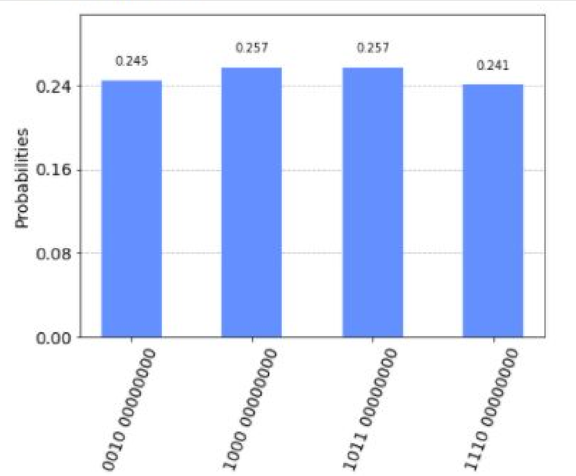
## (a) Step 1: Construct a uniform superposition (sample code)

```python
def c_amod15(a, power):
    """Controlled multiplication by a mod 15"""
    if a not in [2,7,8,11,13]:
        raise ValueError("'a' must be 2,7,8,11 or 13")
    U = QuantumCircuit(4)
    for iteration in range(power):
        if a in [2,13]:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
        if a in [7,8]:
            U.swap(2,3)
            U.swap(1,2)
            U.swap(0,1)
        if a == 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 = "%i^%i mod 15" % (a, power)
    c_U = U.control()
    return c_U
```

```python
# Create QuantumCircuit with n_count counting qubits plus 4 qubits for U
# to act on n_count = 8 # number of counting qubits
n_count = 8
# a variable that can be adjusted later
a = 7
# The first 8-qubit register for storing x
qr1 = QuantumRegister(n_count, name="q1")
# The second 4-qubit register for storing f(x)
qr2 = QuantumRegister(4, name="q2")
cr1 = ClassicalRegister(n_count, name="c1")
cr2 = ClassicalRegister(4, name="c2")

qc = QuantumCircuit(qr1, qr2, cr1, cr2)

# Initialize counting qubits in uniform superposition
for q in range(n_count):
    qc.h(q)

# And ancilla register in state |1>
qc.x(3+n_count)

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

**(b) Step 2: Measure the second register**

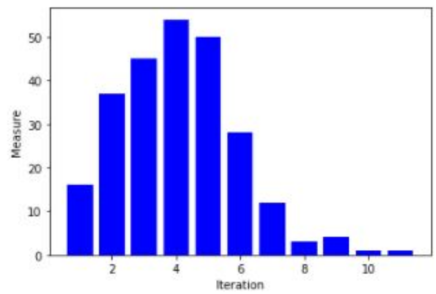
```python
qc.measure(qr2, cr2)
backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend=backend, shots=1024).result()
answer = results.get_counts()
plot_histogram(answer)
```




**(c) Step 3: Verify its "uniform" randomization**

```
qc.measure(qr1, cr1)
backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend=backend, shots=1024).result()
answer = results.get_counts()

# Uniform
maxi, mini = -1, 2**8
for i in answer:
    if (answer[i] > maxi): maxi = answer[i]
    elif (answer[i] < mini): mini = answer[i]
print("Max: ", maxi)
print("min: ", mini)

delta = (maxi - mini)/10
print("Delta: ", delta)
print("Total data numbers: ", len(answer))

lis = [0]*11
for i in answer:
    lis[int((answer[i]-mini)/delta)] += 1

# Plot
plt.xlabel("Iteration")
plt.ylabel("Measure")

ite = []
for i in range(11): ite.append(mini+i*delta)

plt.bar(ite, lis, color='blue')
```

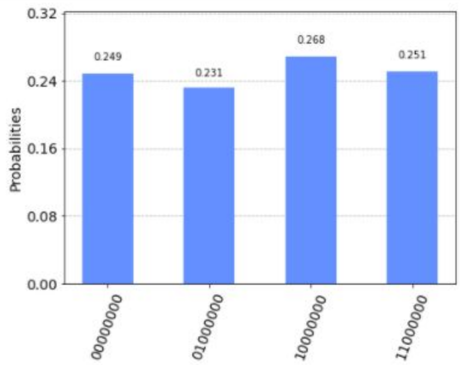
```
Max:  11
min:  1
Delta:  1.0
Total data numbers:  251

<BarContainer object of 11 artists>
```




(Every state has its measured number, I extract the maximum and the minimum, subtract them and the result is divided by 10 to get the 10 groups data, each group has the difference of (max-min)/10. We can see that the distribution is an "uniform" distribution.)

## (d) Step 4: Apply QFT† N

```
# Do inverse-QFT
qc.append(qft_dagger(n_count), range(n_count))

# Measure circuit
qc.measure(range(n_count), range(n_count))

backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend, shots=2048).result()
counts = results.get_counts()
plot_histogram(counts)
```




```
def qft_dagger(n):
    """n-qubit QFTdagger the first n qubits in circ"""
    qc = QuantumCircuit(n)
    # Don't forget the Swaps!
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cu1(-np.pi/float(2**(j-m)), m, j)
        qc.h(j)
    qc.name = "QFT†"
    return qc
```

## (e) Step 5: Get the probability of right period in this case

1. a = 7

```python
import pandas as pd
from fractions import Fraction

rows, measured_phases = [], []
for output in counts:
    decimal = int(output, 2) # Convert (base 2) string to decimal
    phase = decimal/(2**n_count) # Find corresponding eigenvalue
    measured_phases.append(phase)
for phase in measured_phases:
    frac = Fraction(phase).limit_denominator(15)
    rows.append([phase, "%i/%i" % (frac.numerator, frac.denominator), frac.denominator])

# Print as a table
headers=["Phase", "Fraction", "Guess for r"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

```
   Phase Fraction  Guess for r
0   0.00      0/1            1
1   0.25      1/4            4
2   0.50      1/2            2
3   0.75      3/4            4
```

2. a = 2

```
   Phase Fraction  Guess for r
0   0.00      0/1            1
1   0.25      1/4            4
2   0.50      1/2            2
3   0.75      3/4            4
```

3. a = 8

```
   Phase Fraction  Guess for r
0   0.00      0/1            1
1   0.25      1/4            4
2   0.50      1/2            2
3   0.75      3/4            4
```

4. a = 11

```
   Phase Fraction  Guess for r
0   0.00      0/1            1
1   0.25      1/4            4
2   0.50      1/2            2
3   0.75      3/4            4
```

5. a = 13

```
   Phase Fraction  Guess for r
0   0.00      0/1            1
1   0.25      1/4            4
2   0.50      1/2            2
3   0.75      3/4            4
```