Test_for_Magic_box_implementation

June 28, 2020

In [4]:
```
#import library
from qiskit.circuit.library import QFT
from qiskit.extensions import UnitaryGate
import numpy as np
```

In [13]:
```
# In our final Jupyter notebook we won't have to
# break things up functionally like this, but it
# should help for now.

# Verification user input to construct problem to be solved
#
def verify(a,b,N):
    if a<0 or b<0 or N<0:
        print("Invalid input")
        return 0
    else:
        print("Valid input")

#test verify 8=2^3%14
verify(2,8,14)
```

```
Valid input
```

In [15]:
```
# Calculates the multiplicative inverse of x mod N
# (the number y such that xy = 1 (mod N)) using
# the extended Euclidean algorithm.
def invert(x, N):
    q = [0, 0]
    r = [N, x]
    t = [0, 1]

    while r[-1] != 0:
        q.append(r[-2]//r[-1])
        r.append(r[-2] - (q[-1]*r[-1]))
        t.append(t[-2] - (q[-1]*t[-1]))

    if r[-2] != 1:
```

```
                raise Exception

            return t[-2] % N

        #test invert 2*3=1(mod5)
        ans=invert(2,5)
        #should be 3
        ans
```

Out[15]: 3

In [16]:
```
# Returns a unitary matrix which has the effect of multiplying each
# input |x> by a in mod N, resulting in the state |ax>.
def create_unitary(a, N):
    dim = 2**int(np.ceil(np.log(N)/np.log(2)) + 1)
    U = np.zeros((dim, dim))
    # Generate a permutation of the multiplicative group of Z_N.
    for i in range(int(dim/2)):
        U[i,i] = 1
    for i in range(N):
        U[int(dim/2) + i, ((a*i) % N)+int(dim/2)] = 1
    # The remaining states are irrelevant.
    for i in range(N, int(dim/2)):
        U[int(dim/2) + i, int(dim/2) + i] = 1
    print("Multiply by", a)
    print(U)
    return U

#test create_unitary
ans=create_unitary(2, 5)
ans
```

```
Multiply by 2
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
```

```
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

```
Out[16]: array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

```
In [17]: # b is some power of a, and the oracle outputs m,
         # where b = a^m (mod N) with >50% probability.
         # (this is where our main algorithm goes)
         def oracle(a, b, N):

             # Calculate the order of a
             r = 1
             product = a
             while product != 1:
                 product = (product * a) % N
                 r += 1

             # Find number of bits(n) needed to store a value from 0 to N-1
             # and initialize 2 quantum registers of size n
             n = int(np.ceil(np.log(N)/np.log(2)))
             qr1, qr2 = QuantumRegister(n), QuantumRegister(n)
             cr1, cr2 = [ClassicalRegister(1) for i in range(n)], [ClassicalRegister(1) for i i
             qc = QuantumCircuit(qr1, qr2)
             for register in cr1:
                 qc.add_register(register)
             for register in cr2:
                 qc.add_register(register)

             #Change second register to state |00...01>
             qc.x(qr2[n-1])

             #Add H gate to first register
             for i in range(n):
```

```
    qc.h(qr1[i])

# We need log_2(n) different matrices U_(a^(2^x))
for i in range(n):
    U = create_unitary(a**(2**(n-i)) % N, N)
    qubits = [qr1[i]] + [qr2[j] for j in range(n)]
    qc.iso(U, qubits, [])


qc.append(QFT(n), [qr1[i] for i in range(n)])

for i in range(n):
    qc.measure(qr1[i], cr1[i])

# Now cr1 is in state y. We define k to be the closest integer to y*r / 2**n.
# Reuse the first quantum register, because we don't need it anymore.
for i in range(n):
    qc.x(qr1[i]).c_if(cr1[i], 1)

qc.h(qr1[0])

# I don't think there's any way to get the result of the measurement mid-circuit
# in qiskit. So this is a stop-gap method for now.
for y in range(2**n):
    k = int(np.round(y*r/(2**n))) % r
    kInv = invert(k, r)
    print(k, kInv, r)
    print(kInv)



print(qc.draw(output="text"))

# # Phase 2 Starts here
# # Calculate k^-1 and find its binary representation
# k_inv_bin = bin(invert(k, r))

# # Step 1: Initialize a 1 qubit register to |0>
# qr3 = QuantumRegister(1)
# cr3 = ClassicalRegister(1)
# qc.add_register(qr3)
# qc.add_register(cr3)

# # Step 2: Add H gate to new register
# qc.h(qr3[0])

# # Step 3: applying controlled U operation
# for pos, bit in enumerate(k_inv_bin):
```

```
    #     if(bit == '1'):
    #         #apply U operation here

    # # Step 4: Applying a controlled phase shift of -i to
    # # to second register
    # qc.rz(-pi/2 , qr3[0])

    # # Step 5 & 6: Apply H-get to 2nd register and measure
    # qc.h(qr3[0])
    # qc.measure(qr3[0], cr3[0])

    return 0
#test this function
oracle(3, 1, 13)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-17-fbdf8947e50b> in <module>()
 88     return 0
 89 #test this function
---> 90 oracle(3, 1, 13)


<ipython-input-17-fbdf8947e50b> in oracle(a, b, N)
 14     # and initialize 2 quantum registers of size n
 15     n = int(np.ceil(np.log(N)/np.log(2)))
---> 16     qr1, qr2 = QuantumRegister(n), QuantumRegister(n)
 17     cr1, cr2 = [ClassicalRegister(1) for i in range(n)], [ClassicalRegister(1) for
 18     qc = QuantumCircuit(qr1, qr2)


NameError: name 'QuantumRegister' is not defined
```

In [6]:
```
# oracle(3, 1, 13)

# Solves the discrete logarithm problem for
# b = a^m (mod N) using repeated calls to the
# oracle defined above.
def logarithm(a, b, N):
    return 0
```