Week 7- Quantum communications

(c) Ariel Guerreiro 2023

In [ ]:
```
import numpy as np
import math as mt
import matplotlib.pyplot as plt
import random
import re             # regular expressions module

from pylab import plot
from qiskit import *
from qiskit.visualization import *

from qiskit import QuantumCircuit, Aer, transpile, assemble
from qiskit.visualization import plot_histogram, plot_bloch_multivector
from numpy.random import randint
from qiskit.quantum_info import state_fidelity, entropy, mutual_information
from qiskit.extensions import UnitaryGate
from matplotlib import cm
from matplotlib.ticker import LinearLocator
print("Imports Successful")
```

```
Imports Successful
```

Teleportation

Alice wants to send quantum information to Bob. Specifically, suppose she wants to transfer the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ to Bob, which is assumed to be spatially separated. This is called state teleportation and its protocol can be decomposed in the following steps:

Step 1: Alice and Bob create an entangled state of two quantum registers. This is typically an element of the Bell basis, which for sake of simplicity we assume to be

$$\ket{\Phi^+} := \frac{1}{\sqrt{2}}(\ket{00} + \ket{11})$$

.

One quantum register remains with Alice (say, the register q_1) and the other with Bob (say, the register q_2).

Step 2: Alice applies a CNOT gate on q_1 controlled by $|\psi\rangle$ (the state she is trying to send Bob).

Step 3: Next, Alice applies a Hadamard gate to $|\psi\rangle$;

Step 4: Alice applies a measurement to both her registers (q_1 and $|\psi\rangle$).

Step 5: Then, Alice use a classical channel to communicate with Bob the results of the measurements (for example, via the phone). Note that this transfer of information is classical.

Step 6: Based on what Alice obtains form her measurements and tells Bob over the classical channel, he applies specific gates to his quantum register, q_2. The gates to be applied are as follows :

00 → Do nothing

01 → Apply X gate

10 → Apply Z gate

11 → Apply ZX gate

where these gates correspond to the Pauli matrices

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \; Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \; Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

As a result of this protocol the quantum state in Bob's register will be the state $|\psi\rangle$

The pair they create is a special pair called a Bell pair. In quantum circuit language, the way to create a Bell pair between two qubits is to first transfer one of them to the Bell basis (|+⟩ and |−⟩) by using a Hadamard gate, and then to apply a CNOT gate onto the other qubit controlled by the one in the Bell basis.

In [ ]:
```
# Simple function that applies a series of unitary gates from a given string

def Unitary(transformation_list, register, quantum_circuit, dagger):
    # transformation_list is a string of transformations to be applied to a register
    # register is the register where to apply the transformation
    # quantum_circuit is the circuit that contains the registers
    # dagger is 0 or 1 depending on whether we wand a transform or its H.c.
    functionmap = {
        #set of the possible transformation to be applied to the state

                    'x':quantum_circuit.x, #Pauli X
                    'y':quantum_circuit.y, #Pauli Y
                    'z':quantum_circuit.z, #Pauli Z
                    'h':quantum_circuit.h, #Hadamard
                    't':quantum_circuit.t, #T gate
    }
       #How to apply the Hermitc conjugate of the unitary transformation
    if dagger:
        # Notice that only the T gate is not self-adjoint,
        # hence is the only one that needs to be altered in the case of dagger ==1
        functionmap['t'] = quantum_circuit.tdg

    if dagger:
        # Applies the sequence of transforms in the case of dagger ==1
        [functionmap[unitary](register) for unitary in transformation_list]
    else:
        # Applies the sequence of transforms in the case of dagger ==o (notice the reverse order)
        [functionmap[unitary](register) for unitary in transformation_list[::-1]]
```

Code Explained:

This code defines a function called "Unitary" that applies a series of unitary gates to a quantum register in a quantum circuit. The function takes in four arguments: the list of transformations to be applied, the register where the transformations are to be applied, the quantum circuit that contains the register, and a boolean variable called "dagger" that indicates whether the function should apply the transformation or its Hermitian conjugate.

The function starts by creating a dictionary called "functionmap" that maps the transformation characters to their corresponding functions in the quantum circuit library. The dictionary contains functions for the Pauli X, Y, and Z gates, the Hadamard gate, and the T gate. If "dagger" is set to 1, the dictionary maps the T gate to its Hermitian conjugate, which is denoted as "tdg".

If "dagger" is set to 1, the function applies the transformations in the order they are given in the list to the register. If "dagger" is set to 0, the function applies the transformations in the reverse order. The function uses list comprehension to iterate over the transformation list and apply each transformation to the register using the corresponding function from the "functionmap" dictionary.

The registers used in the protocol are as follows:

$q[0]$: register containing the state to be teleported.

$q[1]$: Alice's second register, which will be entangled with Bob's

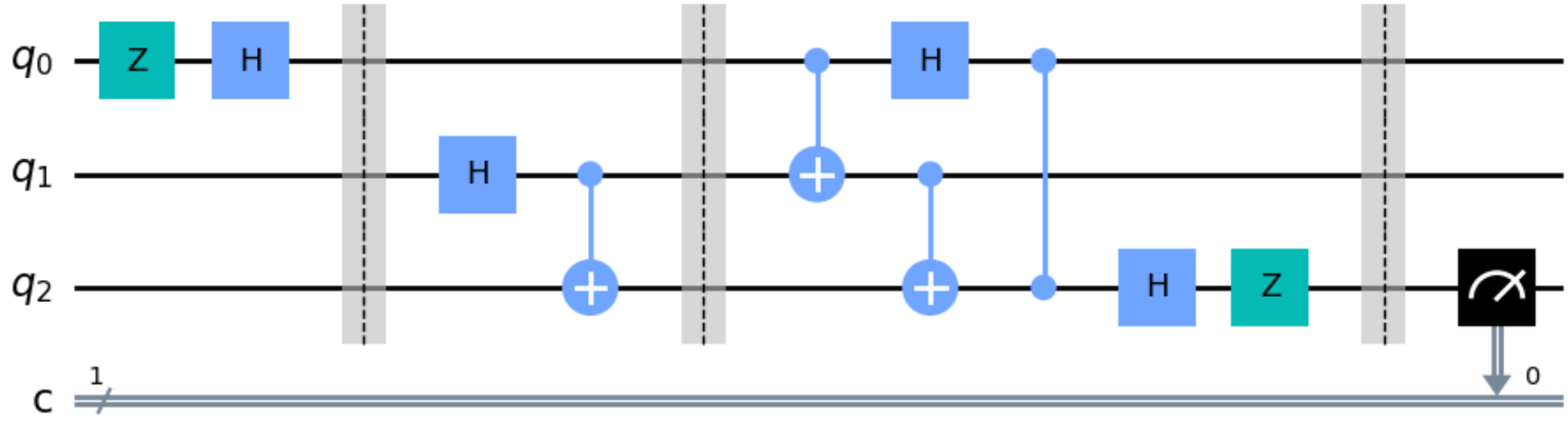
$q[2]$: Bob's register, which will be the destination for the teleportation

Beside these quantum register, the protocol also requires a classical register to contain the results from the measures

In [ ]:
```
# Create the quantum circuit with 3 qubits and 1 classical bit
circuit5 = QuantumCircuit(3, 1)

#Generate a secret state for Alice using a series of unitary gates on a qubit  initialized in the state |0) .
secret = 'hz'
Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)
circuit5.barrier()

#Redy to start teleportation protocol....Beam me up Scotty! (STR)

# Step 1: Alice and Bob create an entangled state of two quantum registers.
# Hadamard followed by CX generates the intended Bell state
circuit5.h(1)
circuit5.cx(1, 2)
circuit5.barrier()

#Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
circuit5.cx(0, 1)

#Step 3: Alice applies a Hadamard gate to q0
circuit5.h(0)

#Step 4: Alice applies a measurement to both her registers
#step 5: Alice use a classical channel to communicate with Bob the results of the measurements
#Step 6: Bob applies specific gates to his quantum register q2.

#All of these steps are done simultaneously,using these quantum operators
#Why are there no actual measurments but transformations applied to Bob
#controled by the states in Alice registers?
circuit5.cx(1, 2)
circuit5.cz(0, 2)


#Let us verify if the teleportation s as worked by inverting the unitary
#transformation that generateb the secret state in Alice's register  but
#now applied to Bob's state. If everything went of it should reverse the
#state back to |0>
Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

circuit5.barrier()
circuit5.measure(2, 0)
```

```
circuit5.measure(2, 0)

circuit5.draw(output='mpl')
```

Out[ ]:



Code Explained:

This code demonstrates a quantum state teleportation protocol using Qiskit. The code starts by creating a quantum circuit called "circuit5" with 3 qubits and 1 classical bit.

In the next step, a secret quantum state is generated for Alice using a series of unitary gates on a qubit initialized in the state |0⟩. The "Unitary" function is called with the secret state string and the first qubit in the circuit as arguments to apply the sequence of unitary gates. A barrier is added to the circuit to separate the initialization step from the teleportation protocol.

The teleportation protocol is implemented in the next steps. Alice and Bob create an entangled state of two quantum registers using a Hadamard gate followed by a CX gate on qubits 1 and 2. Alice applies a CNOT gate on q1 controlled by q0 and a Hadamard gate to q0. Alice then measures both of her qubits and communicates the measurement results to Bob using a classical channel. Bob applies specific gates to his quantum register q2 based on the measurement results he receives from Alice. These gates are chosen to "teleport" the quantum state of q0 to q2.

To verify if the teleportation was successful, the "Unitary" function is called again with the secret state string and the third qubit in the circuit as arguments, but with "dagger" set to 1 to apply the inverse transformations. The circuit then measures the third qubit and stores the result in the classical bit.

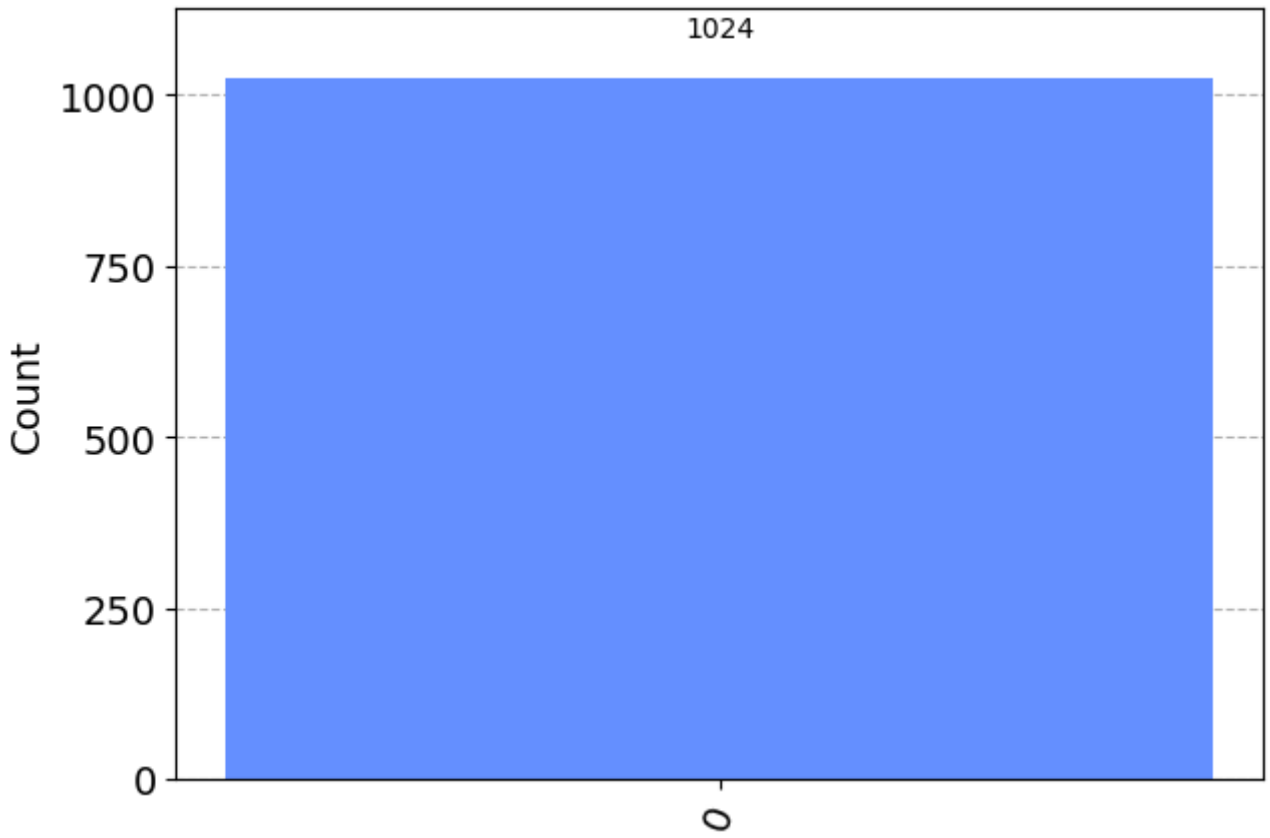
In the next snippet, the circuit is run on a simulator backend and the measurement results are plotted using a histogram.

If the teleportation was successful, all of the measurement results should be in the state |0⟩, indicating that the secret state was successfully teleported from Alice's qubit to Bob's qubit.

In [ ]:
```
backend = Aer.get_backend('qasm_simulator')
job = execute(circuit5, backend, shots=1024)
result = job.result()

measurement_result = result.get_counts(circuit5)
print(measurement_result)
plot_histogram(measurement_result)
#perfect teleportation should provide all the counts in the state |0>
```

```
{'0': 1024}
```

Out[ ]:



Exercises

1. Verify whether this protocol works for a wide range of secret states.

In [ ]:
```
def protocol(secret):
    circuit5 = QuantumCircuit(3, 1)
    Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)
    circuit5.barrier()

    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

    circuit5.barrier()
    circuit5.measure(2, 0)

    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, shots=1024)
```

```
    job = execute(circuit5, backend, shots=1024)
    result = job.result()

    measurement_result = result.get_counts(circuit5)
    return measurement_result
    #perfect teleportation should provide all the counts in the state |0>
```

In [ ]:
```
from itertools import combinations_with_replacement
# Create the quantum circuit with 3 qubits and 1 classical bit


#Generate a secret state for Alice using a series of unitary gates on a qubit  initialized in the state |0) .
secret_list=[]
for i in range(2,10):
    secret=combinations_with_replacement(['x', 'y', 'h','z',], i)
    for j in secret:
        n=j[0]
        for k in range(1,len(j)):
            n+=j[k]
        secret_list.append(n)
```

In [ ]:
```
n=0
for i in secret_list:
    result=protocol(i)
    if result["0"]==1024:
        n+=1
if n==len(secret_list):
    print('Sucess')
```

```
Sucess
```

2. Test the protocol in the IBM computer a measure the fidelity for several states.

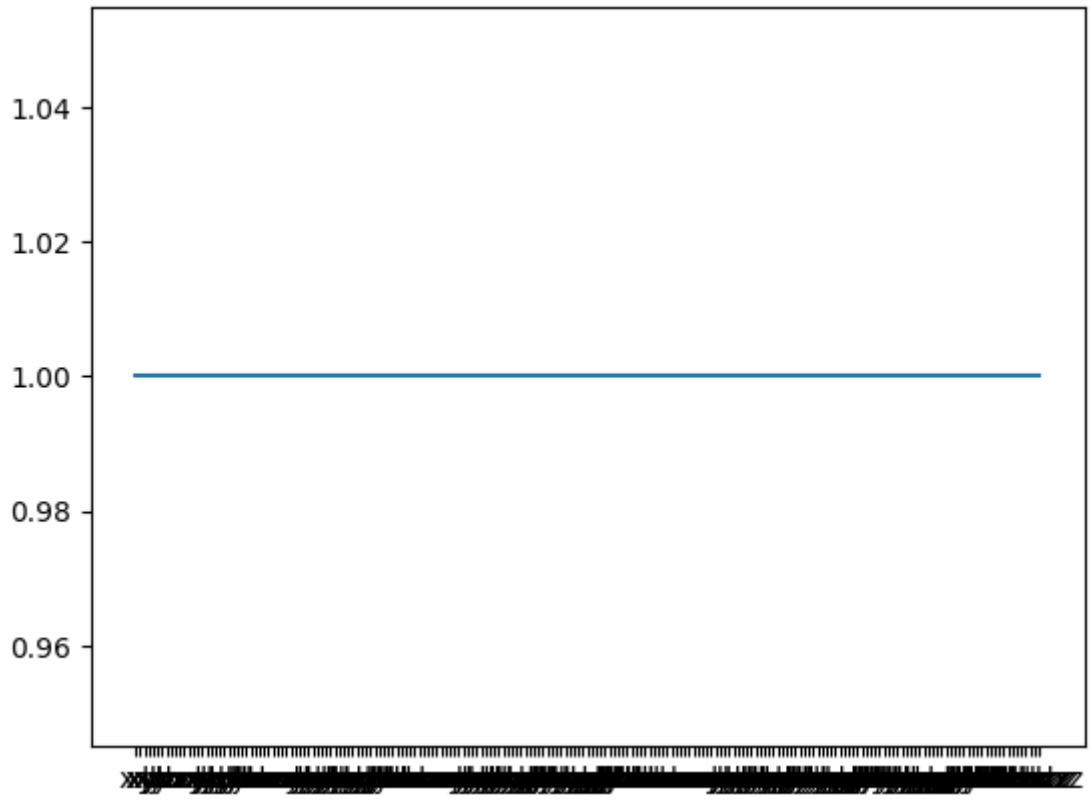
Question 1: Does the fidelity changes if have a more complex unitary transformation to generate the secret state? Why?

In [ ]:
```
def protocol_fidelity(secret):
    circuit5 = QuantumCircuit(3, 1)
    Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)

    circuit5.barrier()


    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)
    circuit5.barrier()


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

    circuit5.barrier()
    circuit5.measure(2, 0)
```

```
    circuit5.measure(2, 0)
    shots=1024
    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, shots=1024)
    result = job.result()
    measurement_result = result.get_counts(circuit5)
    fidelity=measurement_result["0"]/shots

    return fidelity
    #perfect teleportation should provide all the counts in the state |0>
```

In [ ]:
```
#Generate a secret state for Alice using a series of unitary gates on a qubit  initialized in the state |0) .
secret_list=[]
for i in range(2,7):
    secret=combinations_with_replacement(['x', 'y', 'h','z',], i)
    for j in secret:
        n=j[0]
        for k in range(1,len(j)):
            n+=j[k]
        secret_list.append(n)


result=[]
for i in secret_list:
    result.append(protocol_fidelity(i))
plt.plot(secret_list,result)
plt.show
```

Out[ ]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



We can see that the fidelity no matter how much complicated the unitary transformation gets, it never increases above 0.25. It is to be expected because the state will always remain, no matter how many gates are presented, in the type

$$\alpha|0\rangle + \beta|1\rangle$$

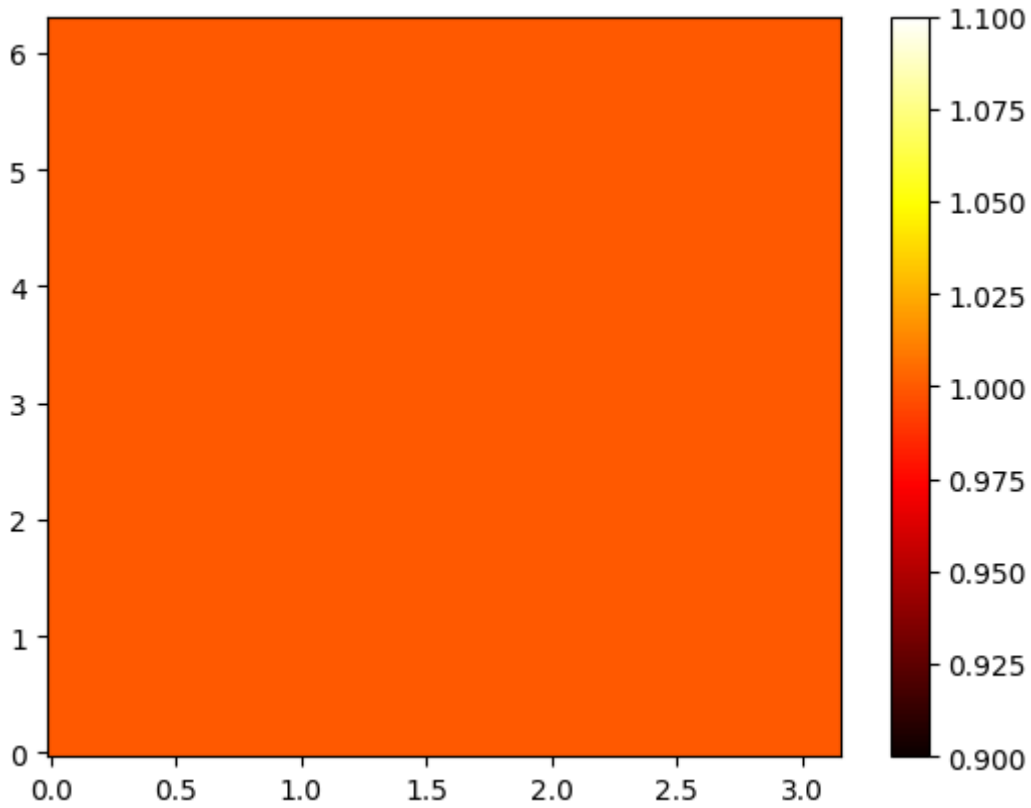
Question 2: Does the fidelity depends on the position of of the secret state in the Bloch sphere? If so, why?

In [ ]:
```
def protocol_fidelity_getbloch(theta,phi,lamda):
    circuit5 = QuantumCircuit(3, 1)
    circuit5.u(theta, phi, lamda, 0)

    circuit5.barrier()


    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)
    circuit5.barrier()


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    u_adj=np.array([[np.cos(theta/2),np.exp(-1j*phi)*np.sin(theta/2)],[-np.sin(theta/2),np.exp(-1j*phi)*np.cos(thet
    circuit5.unitary(u_adj,2)
    circuit5.barrier()
    circuit5.measure(2, 0)
```

```
    circuit5.measure(2, 0)
    shots=1024
    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, shots=1024)
    result = job.result()
    measurement_result = result.get_counts(circuit5)
    fidelity=measurement_result["0"]/shots

    return fidelity
    #perfect teleportation should provide all the counts in the state |0>
```

In [ ]:
```
theta=np.linspace(0,np.pi,100)
phi=np.linspace(0,2*np.pi,100)
lamda=0
result=np.zeros((100,100))
x=[]
y=[]
z=[]

for i in range(len(theta)):
    for j in range(len(phi)):
        result[i,j]=protocol_fidelity_getbloch(theta[i],phi[j],lamda)
        x.append(np.sin(theta)*np.cos(phi))
        y.append(np.sin(theta)*np.sin(phi))
        z.append(np.cos(theta))
```

In [ ]:
```
plt.pcolormesh(theta,phi,result)
plt.colorbar()
plt.show()
```



When thinking of the secret state it's going to be something like this

$$|\psi\rangle = \cos(\theta/2)|0\rangle + e^{i\phi}\sin(\theta/2)|1\rangle$$

$$\rho = \begin{pmatrix} \cos^2(\theta/2) & e^{i\phi}\sin(\theta/2)\cos(\theta/2) \\ e^{-i\phi}\sin(\theta/2)\cos(\theta/2) & \sin^2(\theta/2) \end{pmatrix}$$

Advanced exercises

3. Alter this protocol to teleport a 2-qubit state.

In [ ]:
```
def protocol_mult_qubit(secret,N):
    circuit5 = QuantumCircuit(3*N, N)
    for i in range(N):
        Unitary(secret[i], circuit5.qubits[i], circuit5, dagger = 0)
        circuit5.barrier()

    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
        circuit5.h(i+N)
        circuit5.cx(i+N, i+2*N)
        circuit5.barrier()

        circuit5.cx(i, i+N)
        circuit5.h(i)   #Step 3: Alice applies a Hadamard gate to q0
        circuit5.cx(i+N, i+2*N)
        circuit5.cz(i, i+2*N)
        Unitary(secret[i], circuit5.qubits[i+2*N], circuit5, dagger=1)

        circuit5.draw(output='mpl')
        circuit5.barrier()
        circuit5.measure(i+2*N, i)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>


    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, shots=1024)
```

```
    job = execute(circuit5, backend, shots=1024)
    result = job.result()

    measurement_result = result.get_counts(circuit5)
    return measurement_result
    #perfect teleportation should provide all the counts in the state |0>
```

In [ ]:
```
from itertools import permutations
import random
# Create the quantum circuit with 3 qubits and 1 classical bit

N=2
#Generate a secret state for Alice using a series of unitary gates on a qubit  initialized in the state |0) .
secret_list=[]
for l in range(N):
    secret_list_1=[]
    for i in range(2,10):
        secret=permutations(['x', 'y', 'h','z',], i)
        for j in secret:
            n=j[0]
            for k in range(1,len(j)):
                n+=j[k]
            secret_list_1.append(n)
    secret_list.append(random.sample(secret_list_1, len(secret_list_1)))
```

In [ ]:
```
j=0
for i in np.transpose(secret_list):
    result=protocol_mult_qubit(i,N)
    if result["00"]==1024:
        j+=1
if j==len(np.transpose(secret_list)):
    print('Sucess')
```

```
Sucess
```

4. Alter this protocol to teleport a 1-qubit mixed state.

In [ ]:
```
def protocol_mixed(secret):
    circuit5 = QuantumCircuit(3, 1)
    Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)
    circuit5.cx(1,0)
    circuit5.reset(1)
    circuit5.barrier()

    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

    circuit5.barrier()
    circuit5.measure(2, 0)
```

```
    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, shots=1024)
    result = job.result()

    measurement_result = result.get_counts(circuit5)
    return measurement_result
    #perfect teleportation should provide all the counts in the state |0>
```

In [ ]:
```
from itertools import permutations
# Create the quantum circuit with 3 qubits and 1 classical bit


#Generate a secret state for Alice using a series of unitary gates on a qubit  initialized in the state |0) .
secret_list=[]
for i in range(1,10):
    secret=permutations(['x', 'y', 'h','z',], i)
    for j in secret:
        n=j[0]
        for k in range(1,len(j)):
            n+=j[k]
        secret_list.append(n)
```

In [ ]:
```
n=0
for i in secret_list:
    result=protocol_mixed(i)
    if result["0"]==1024:
        n+=1
if n==len(secret_list):
    print('Sucess')
```

```
Sucess
```

Question 3: Does the initial entropy influences the fidelity of the teleportation?

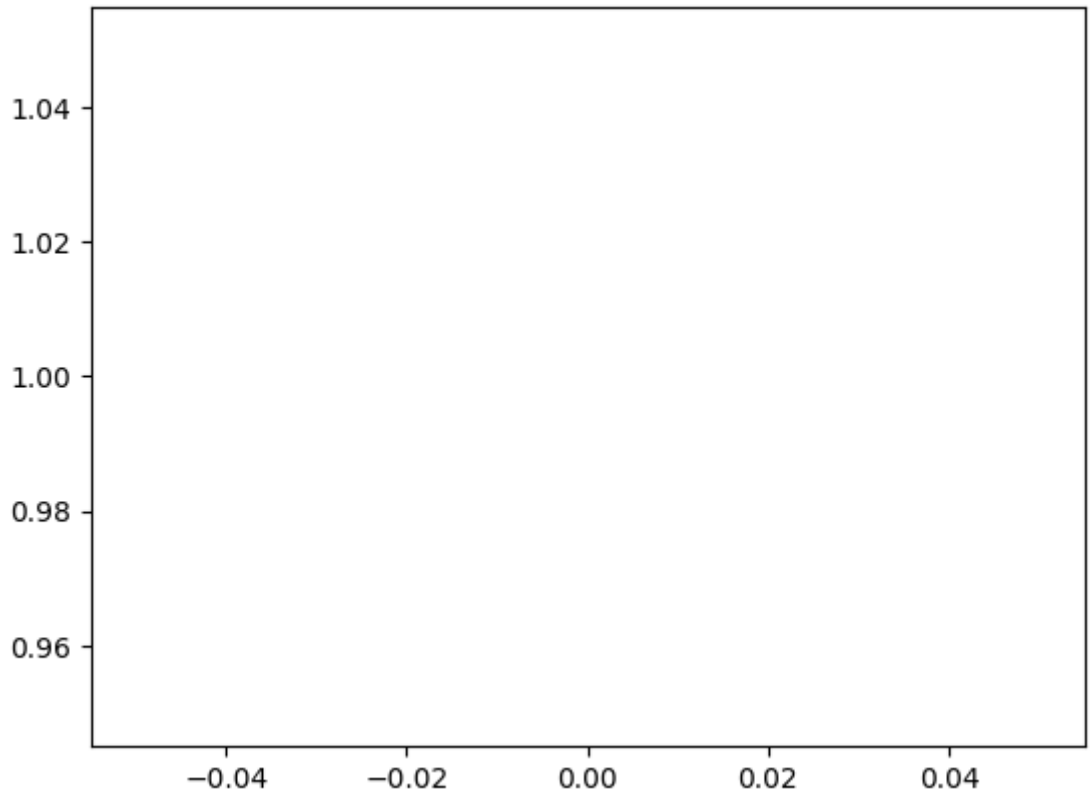
In [ ]:
```
def protocol_fidelity_entropy(secret):
    circuit5 = QuantumCircuit(3, 1)
    Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)

    circuit5.barrier()
    backend = Aer.get_backend('statevector_simulator')
    state_i = execute(circuit5, backend).result().get_statevector(circuit5)
    entropy_i=entropy(state_i)
    circuit5.barrier()

    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)
    circuit5.barrier()


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)
```

```
    unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

    circuit5.barrier()
    circuit5.measure(2, 0)
    shots=1024
    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, shots=1024)
    result = job.result()
    counts=result.get_counts(circuit5)
    fidelity=counts["0"]/shots

    return fidelity,entropy_i
    #perfect teleportation should provide all the counts in the state |0>
```

In [ ]:
```
#Generate a secret state for Alice using a series of unitary gates on a qubit  initialized in the state |0) .
secret_list=[]
for i in range(1,7):
    secret=combinations_with_replacement(['x', 'y', 'h','z',], i)
    for j in secret:
        n=j[0]
        for k in range(1,len(j)):
            n+=j[k]
        secret_list.append(n)


result=[]
entropy_list=[]
for i in secret_list:
    entropy_list.append(protocol_fidelity_entropy(i)[1])
    result.append(protocol_fidelity_entropy(i)[0])
```

In [ ]:
```
plt.plot(entropy_list,result,"-k")
plt.show
```

Out[ ]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



3. Alter the previous code to describe quantum state teleportation in a noisy channel.

In [ ]:
```
from qiskit.providers.aer import AerSimulator

# Qiskit Aer noise module imports
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors import QuantumError, ReadoutError
from qiskit.providers.aer.noise.errors import pauli_error
from qiskit.providers.aer.noise.errors import depolarizing_error
from qiskit.providers.aer.noise.errors import thermal_relaxation_error
```

In [ ]:
```
# Example error probabilities
p_reset = 0.02
p_gate1 = 0.05

# QuantumError objects
error_reset = pauli_error([('X', p_reset), ('I', 1 - p_reset)])
error_gate1 = pauli_error([('X',p_gate1), ('I', 1 - p_gate1)])
error_gate2 = error_gate1.tensor(error_gate1)

# Add errors to noise model
noise_bit_flip = NoiseModel()
noise_bit_flip.add_all_qubit_quantum_error(error_reset, "reset")
noise_bit_flip.add_all_qubit_quantum_error(error_gate1, ["h"])
```

In [ ]:
```
def protocol_noise(secret):
    circuit5 = QuantumCircuit(3, 1)
    Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)
    circuit5.barrier()

    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

    circuit5.barrier()
    circuit5.measure(2, 0)

    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend basis_gates=noise_bit_flip.basis_gates noise_model=noise_bit_flip, shots=1024)
```

```
    job = execute(circuit5, backend,basis_gates=noise_bit_flip.basis_gates,noise_model=noise_bit_flip, shots=1024)
    result = job.result()

    measurement_result = result.get_counts(circuit5)
    return measurement_result
    #perfect teleportation should provide all the counts in the state |0>
```

Question 3: How does the noise alter the fidelity between the input and output state?

BIT FLIP NOISE

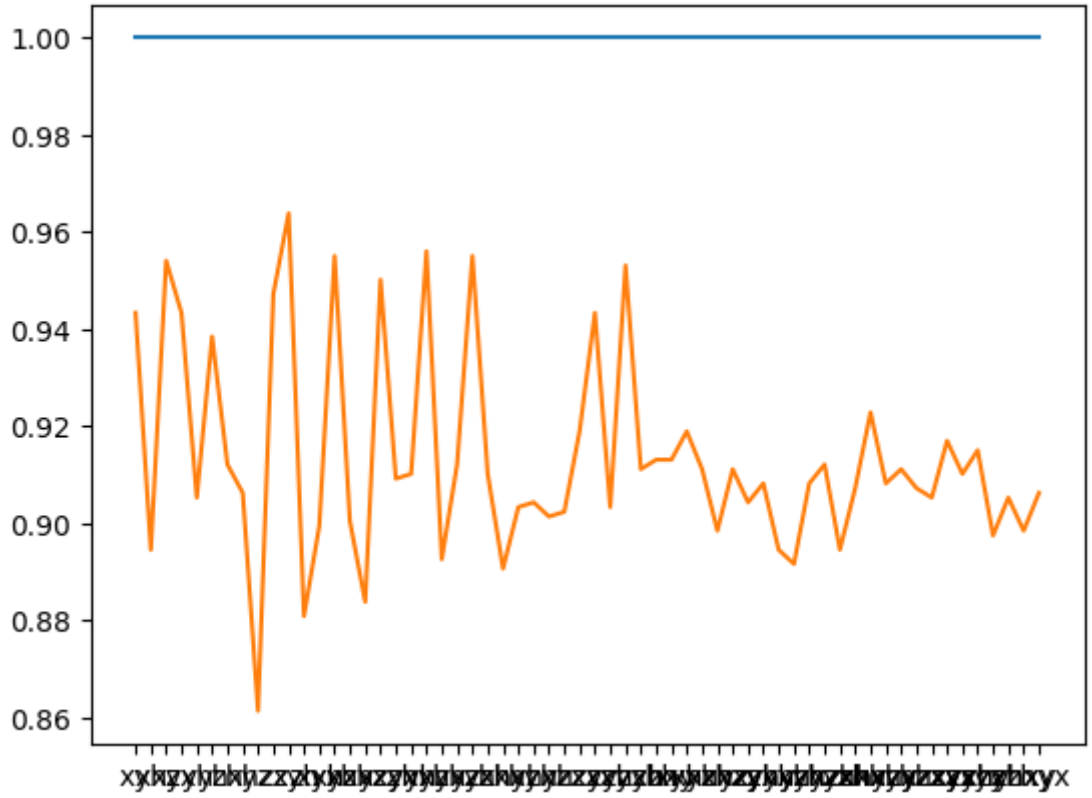
In [ ]:
```
def protocol_fidelity_noise(secret):
    circuit5 = QuantumCircuit(3, 1)
    Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)

    circuit5.barrier()


    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)
    circuit5.barrier()


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

    circuit5.barrier()
    circuit5.measure(2, 0)
```

```
    circuit5.measure(2, 0)
    shots=1024
    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, basis_gates=noise_bit_flip.basis_gates,noise_model=noise_bit_flip,shots=1024)
    result = job.result()
    measurement_result = result.get_counts(circuit5)
    fidelity=measurement_result["0"]/shots

    return fidelity
    #perfect teleportation should provide all the counts in the state |0>
```

In [ ]:
```
from itertools import permutations
#Generate a secret state for Alice using a series of unitary gates on a qubit  initialized in the state |0) .
secret_list=[]
for i in range(2,7):
    secret=permutations(['x', 'y', 'h','z',], i)
    for j in secret:
        n=j[0]
        for k in range(1,len(j)):
            n+=j[k]
        secret_list.append(n)
```

In [ ]:
```
result_noise=[]
result=[]
for i in secret_list:
    result_noise.append(protocol_fidelity_noise(i))
    result.append(protocol_fidelity(i))


plt.plot(secret_list,result)
plt.plot(secret_list,result_noise)
plt.show
```

Out[ ]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



THERMAL NOISE

In [ ]:
```
n_qubits=3
```

In [ ]:
```
# T1 and T2 values for qubits 0-3
T1s = np.random.normal(50e3, 10e3, n_qubits) # Sampled from normal distribution mean 50 microsec
T2s = np.random.normal(50e3, 10e3, n_qubits)  # Sampled from normal distribution mean 50 microsec

# Truncate random T2s <= T1s
T2s = np.array([min(T2s[j], 2 * T1s[j]) for j in range(n_qubits)])

# Instruction times (in nanoseconds)
time_h = 100 # (two X90 pulses)
time_cx = 300
time_reset = 1000  # 1 microsecond
time_measure = 1000 # 1 microsecond

# QuantumError objects
errors_reset = [thermal_relaxation_error(t1, t2, time_reset)
                for t1, t2 in zip(T1s, T2s)]
errors_measure = [thermal_relaxation_error(t1, t2, time_measure)
                  for t1, t2 in zip(T1s, T2s)]
errors_h  = [thermal_relaxation_error(t1, t2, time_h)
             for t1, t2 in zip(T1s, T2s)]
errors_cx = [[thermal_relaxation_error(t1a, t2a, time_cx).expand(
             thermal_relaxation_error(t1b, t2b, time_cx))
              for t1a, t2a in zip(T1s, T2s)]
               for t1b, t2b in zip(T1s, T2s)]

# Add errors to noise model
noise_thermal = NoiseModel()
for j in range(n_qubits):
    noise_thermal.add_quantum_error(errors_reset[j], "reset", [j])
    noise_thermal.add_quantum_error(errors_measure[j], "measure", [j])
    noise_thermal.add_quantum_error(errors_h[j], "h", [j])
    for k in range(n_qubits):
        noise_thermal.add_quantum_error(errors_cx[j][k], "cx", [j, k])

print(noise_thermal)
```

```
NoiseModel:
  Basis gates: ['cx', 'h', 'id', 'rz', 'sx']
  Instructions with noise: ['cx', 'measure', 'h', 'reset']
  Qubits with noise: [0, 1, 2]
  Specific qubit errors: [('reset', (0,)), ('reset', (1,)), ('reset', (2,)), ('measure', (0,)), ('measure', (1,)),
('measure', (2,)), ('h', (0,)), ('h', (1,)), ('h', (2,)), ('cx', (0, 0)), ('cx', (0, 1)), ('cx', (0, 2)), ('cx',
(1, 0)), ('cx', (1, 1)), ('cx', (1, 2)), ('cx', (2, 0)), ('cx', (2, 1)), ('cx', (2, 2))]
```

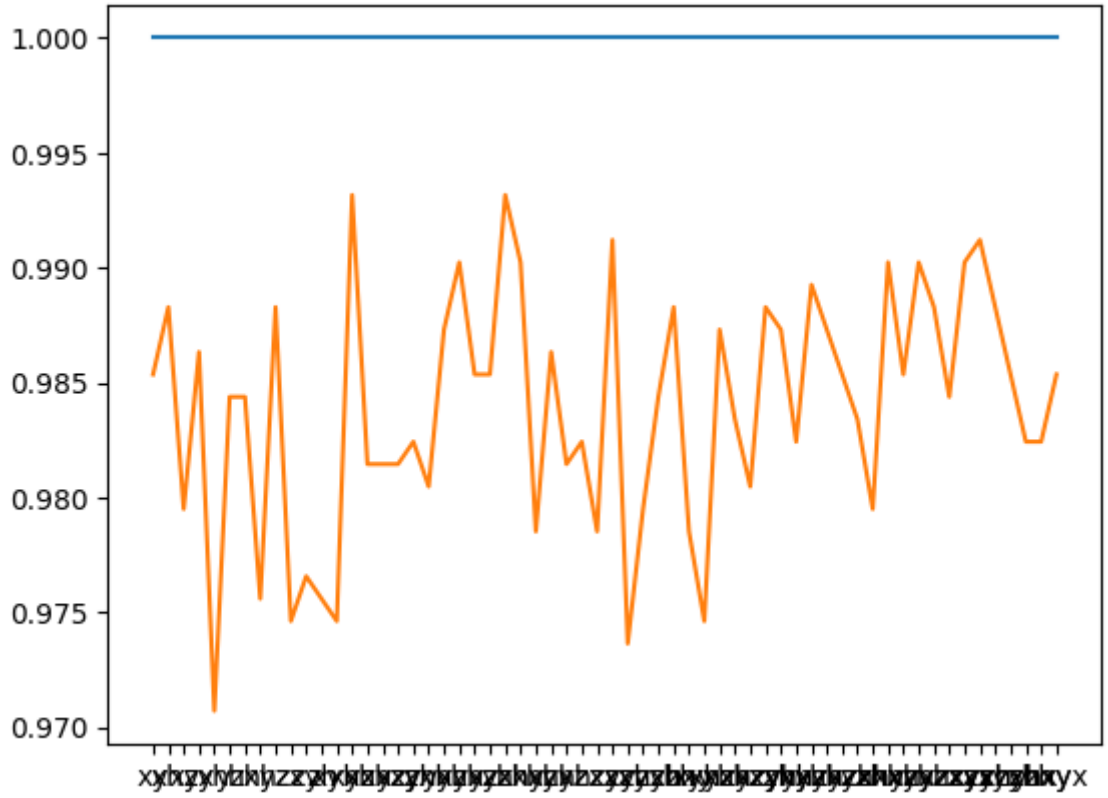
In [ ]:
```
def protocol_fidelity_noise(secret):
    circuit5 = QuantumCircuit(3, 1)
    Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)

    circuit5.barrier()


    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)
    circuit5.barrier()


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

    circuit5.barrier()
    circuit5.measure(2, 0)
```

```
    circuit5.measure(2, 0)
    shots=1024
    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, basis_gates=noise_thermal.basis_gates,
              noise_model=noise_thermal,shots=1024)
    result = job.result()
    measurement_result = result.get_counts(circuit5)
    fidelity=measurement_result["0"]/shots

    return fidelity
    #perfect teleportation should provide all the counts in the state |0>
```

In [ ]:
```
from itertools import permutations
#Generate a secret state for Alice using a series of unitary gates on a qubit  initialized in the state |0⟩ .
secret_list=[]
for i in range(2,7):
    secret=permutations(['x', 'y', 'h','z',], i)
    for j in secret:
        n=j[0]
        for k in range(1,len(j)):
            n+=j[k]
        secret_list.append(n)
```

In [ ]:
```
result_noise=[]
result=[]
for i in secret_list:
    result_noise.append(protocol_fidelity_noise(i))
    result.append(protocol_fidelity(i))


plt.plot(secret_list,result)
plt.plot(secret_list,result_noise)
plt.show
```

Out[ ]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



4. Compute the channel capacity in function of the noise level.

Thermal Noise

In [ ]:
```
# T1 and T2 values for qubits 0-3
def thermal_noise():
    T1s = np.random.normal(50e3, 10e3, n_qubits) # Sampled from normal distribution mean 50 microsec
    T2s = np.random.normal(50e3, 10e3, n_qubits)  # Sampled from normal distribution mean 50 microsec

# Truncate random T2s <= T1s
    T2s = np.array([min(T2s[j], 2 * T1s[j]) for j in range(n_qubits)])

# Instruction times (in nanoseconds)
    time_h = 100 # (two X90 pulses)
    time_cx = 300
    time_reset = 1000  # 1 microsecond
    time_measure = 1000 # 1 microsecond

# QuantumError objects
    errors_reset = [thermal_relaxation_error(t1, t2, time_reset)
                    for t1, t2 in zip(T1s, T2s)]
    errors_measure = [thermal_relaxation_error(t1, t2, time_measure)
                      for t1, t2 in zip(T1s, T2s)]
    errors_h  = [thermal_relaxation_error(t1, t2, time_h)
                 for t1, t2 in zip(T1s, T2s)]
    errors_cx = [[thermal_relaxation_error(t1a, t2a, time_cx).expand(
             thermal_relaxation_error(t1b, t2b, time_cx))
              for t1a, t2a in zip(T1s, T2s)]
               for t1b, t2b in zip(T1s, T2s)]

# Add errors to noise model
    noise_thermal = NoiseModel()
    for j in range(n_qubits):
        noise_thermal.add_quantum_error(errors_reset[j], "reset", [j])
        noise_thermal.add_quantum_error(errors_measure[j], "measure", [j])
        noise_thermal.add_quantum_error(errors_h[j], "h", [j])
        for k in range(n_qubits):
            noise_thermal.add_quantum_error(errors_cx[j][k], "cx", [j, k])
    return T1s
```

In [ ]:
```
def protocol_noise_mutual_entropy(secret):
    circuit5 = QuantumCircuit(3, 1)
    Unitary(secret, circuit5.qubits[0], circuit5, dagger = 0)


    #Ready to start teleportation protocol....Beam me up Scotty! (STR)

    # Step 1: Alice and Bob create an entangled state of two quantum registers.
    # Hadamard followed by CX generates the intended Bell state
    circuit5.h(1)
    circuit5.cx(1, 2)
    circuit5.barrier()

    #Step 2: Alice applies a CNOT gate on  q1  controlled by  q0
    circuit5.cx(0, 1)

    #Step 3: Alice applies a Hadamard gate to q0
    circuit5.h(0)

    #Step 4: Alice applies a measurement to both her registers
    #step 5: Alice use a classical channel to communicate with Bob the results of the measurements
    #Step 6: Bob applies specific gates to his quantum register q2.

    #All of these steps are done simultaneously,using these quantum operators
    #Why are there no actual measurments but transformations applied to Bob
    #controled by the states in Alice registers?
    circuit5.cx(1, 2)
    circuit5.cz(0, 2)
    circuit5.barrier()


    #Let us verify if the teleportation s as worked by inverting the unitary
    #transformation that generateb the secret state in Alice's register  but
    #now applied to Bob's state. If everything went of it should reverse the
    #state back to |0>
    Unitary(secret, circuit5.qubits[2], circuit5, dagger=1)

    circuit5.barrier()
    circuit5.measure(2, 0)
    shots=1024
    backend = Aer.get_backend('qasm_simulator')
```

```
    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit5, backend, basis_gates=noise_thermal.basis_gates,
                  noise_model=noise_thermal,shots=1024)
    result = job.result()

    return
    #perfect teleportation should provide all the counts in the state |0>
```

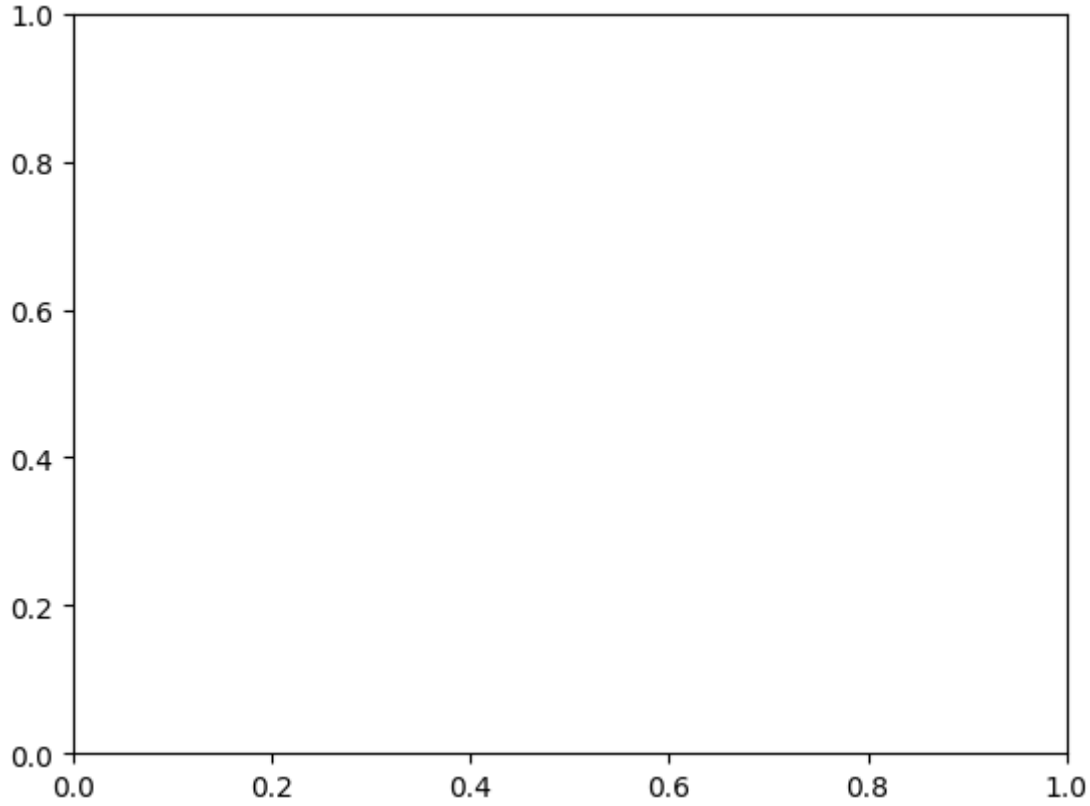
In [ ]:
```
noise_lvl_thermal=[]
cap=[]
for i in range(100):
    noise_lvl_thermal.append(thermal_noise())
    me=[]
    for secret in secret_list:
        me.append(protocol_noise_mutual_entropy(secret))
    cap.append(max(cap))
```

```
---------------------------------------------------------------------------
QiskitError                               Traceback (most recent call last)
Cell In[239], line 7
      5 me=[]
      6 for secret in secret_list:
----> 7     me.append(protocol_noise_mutual_entropy(secret))
      8 cap.append(max(cap))

Cell In[238], line 55, in protocol_noise_mutual_entropy(secret)
     51 job = execute(circuit5, backend, basis_gates=noise_thermal.basis_gates,
     52           noise_model=noise_thermal,shots=1024)
     53 result = job.result()
---> 55 return mutual_information([state_i,state_f])

File ~/.local/lib/python3.10/site-packages/qiskit/quantum_info/states/measures.py:154, in mutual_information(stat
e, base)
    131 def mutual_information(state, base=2):
    132     r"""Calculate the mutual information of a bipartite state.
    133
    134     The mutual information :math:`I` is given by:
   (...)
    152         QiskitError: if input is not a bipartite QuantumState.
    153     """
--> 154     state = _format_state(state, validate=True)
    155     if len(state.dims()) != 2:
    156         raise QiskitError("Input must be a bipartite quantum state.")

File ~/.local/lib/python3.10/site-packages/qiskit/quantum_info/states/utils.py:136, in _format_state(state, valida
te)
    134             state = DensityMatrix(state)
    135 if not isinstance(state, (Statevector, DensityMatrix)):
--> 136     raise QiskitError("Input is not a quantum state")
    137 if validate and not state.is_valid():
    138     raise QiskitError("Input quantum state is not a valid")

QiskitError: 'Input is not a quantum state'
```

In [ ]:
```
plt.plot(noise_lvl_thermal,cap)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[237], line 1
----> 1 plt.plot(noise_lvl_thermal,cap)

File ~/.local/lib/python3.10/site-packages/matplotlib/pyplot.py:2812, in plot(scalex, scaley, data, *args, **kwarg
s)
   2810 @_copy_docstring_and_deprecators(Axes.plot)
   2811 def plot(*args, scalex=True, scaley=True, data=None, **kwargs):
-> 2812     return gca().plot(
   2813         *args, scalex=scalex, scaley=scaley,
   2814         **({"data": data} if data is not None else {}), **kwargs)

File ~/.local/lib/python3.10/site-packages/matplotlib/axes/_axes.py:1688, in Axes.plot(self, scalex, scaley, data,
*args, **kwargs)
   1445 """
   1446 Plot y versus x as lines and/or markers.
   1447
   (...)
   1685 (``'green'``) or hex strings (``'#008000'``).
   1686 """
   1687 kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
-> 1688 lines = [*self._get_lines(*args, data=data, **kwargs)]
   1689 for line in lines:
   1690     self.add_line(line)

File ~/.local/lib/python3.10/site-packages/matplotlib/axes/_base.py:311, in _process_plot_var_args.__call__(self,
data, *args, **kwargs)
    309     this += args[0],
    310     args = args[1:]
--> 311 yield from self._plot_args(
    312     this, kwargs, ambiguous_fmt_datakey=ambiguous_fmt_datakey)

File ~/.local/lib/python3.10/site-packages/matplotlib/axes/_base.py:504, in _process_plot_var_args._plot_args(sel
f, tup, kwargs, return_kwargs, ambiguous_fmt_datakey)
    501     self.axes.yaxis.update_units(y)
    503 if x.shape[0] != y.shape[0]:
--> 504     raise ValueError(f"x and y must have same first dimension, but "
    505                      f"have shapes {x.shape} and {y.shape}")
    506 if x.ndim > 2 or y.ndim > 2:
```

```
    507     raise ValueError(f"x and y can be no greater than 2D, but have "
    508                      f"shapes {x.shape} and {y.shape}")

ValueError: x and y must have same first dimension, but have shapes (1, 3) and (0,)
```



Quantum Key Distribution

(adapted from: https://qiskit.org/textbook/ch-algorithms/quantum-key-distribution.html)

When Alice and Bob need to send secret messages, they must encrypt the message to ensure its confidentiality. One approach to achieve that is to use symmetric-key cryptography, which requires Alice and Bob to share a secret key.

However, if Alice and Bob want to share their secret key using a classical communication channel controlled by Eve, they have to trust that Eve will not copy the key. In contrast, if Alice and Bob use a quantum communication channel, which utilizes individual photons to represent a qubit, they can detect if Eve tries to read Bob's message before it reaches Alice. In this channel, the polarization of photons can be in one of two states.

The protocol exploits the fundamental principle of quantum mechanics that measuring a qubit can change its state. If Alice sends Bob a qubit and Eve attempts to measure it before Bob, there is a possibility that Eve's measurement will change the qubit's state, and Bob will receive a different state than Alice sent. In this case, Bob now has a 50% chance of measuring 1, which signals to him and Alice that there is something wrong with their channel.

To guarantee that an eavesdropper has a negligible chance of intercepting the key, the quantum key distribution protocol repeats this process multiple times.

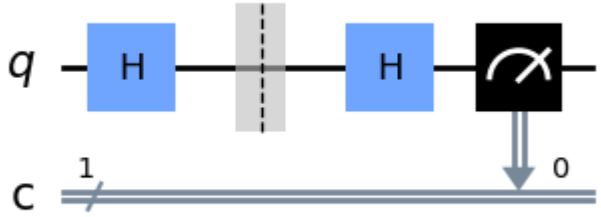
Protocol Overview

The protocol makes use of the fact that measuring a qubit can change its state. If Alice sends Bob a qubit, and an eavesdropper (Eve) tries to measure it before Bob does, there is a chance that Eve's measurement will change the state of the qubit and Bob will not receive the qubit state Alice sent.

If Alice prepares a qubit in the state $|+\rangle$ (`0` in the X-basis), and Bob measures it in the X-basis, Bob is sure to measure `0` :

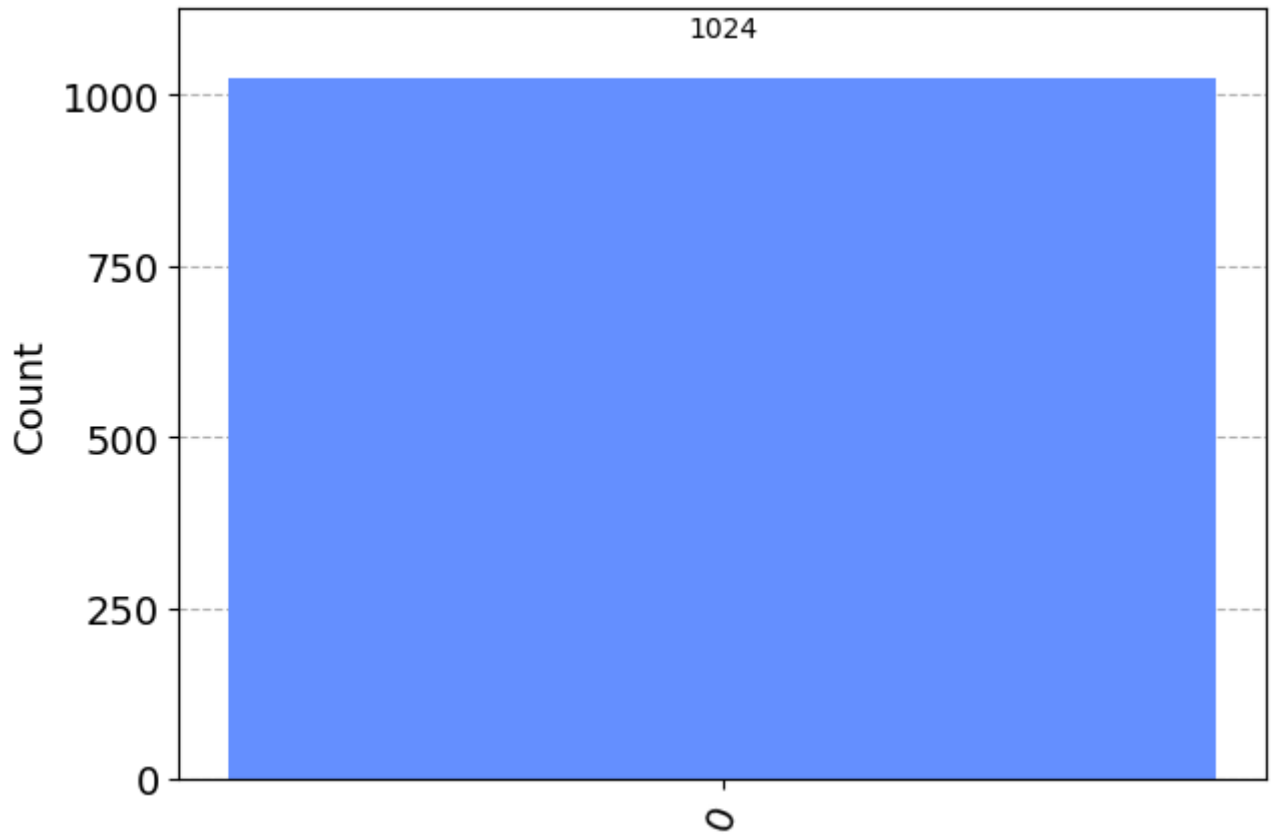
In [ ]:
```
qc = QuantumCircuit(1,1)
# Alice prepares qubit in state |+>
qc.h(0)
qc.barrier()
# Alice now sends the qubit to Bob
# who measures it in the X-basis
qc.h(0)
qc.measure(0,0)
qc.draw(output='mpl')
```

Out[ ]:



In [ ]:
```
# Simulate circuit
aer_sim = Aer.get_backend('aer_simulator')
job = aer_sim.run(qc, shots=1024)
plot_histogram(job.result().get_counts())
```

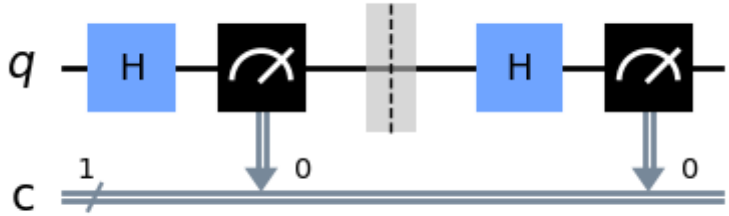
Out[ ]:



But if Eve tries to measure this qubit in the Z-basis before it reaches Bob, she will change the qubit's state from $|+\rangle$ to either $|0\rangle$ or $|1\rangle$, and Bob is no longer certain to measure `0`:

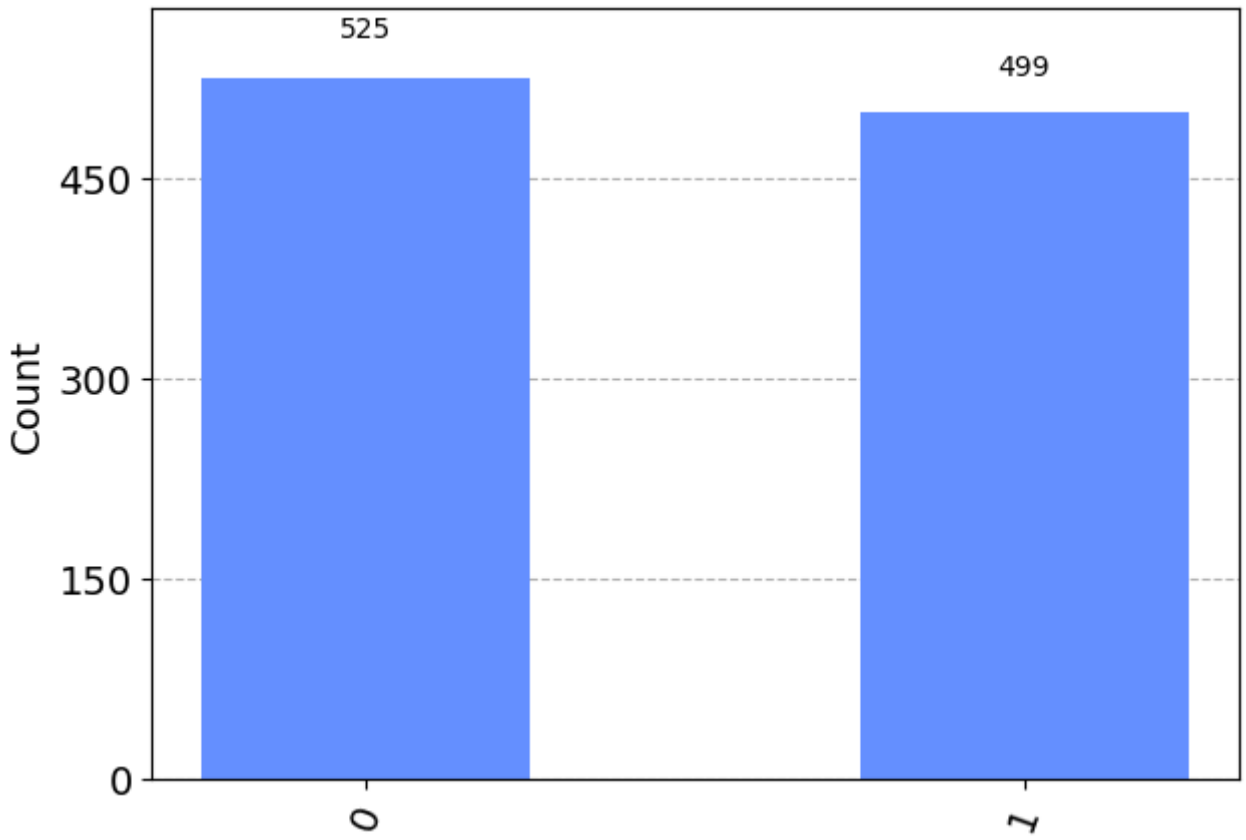
In [ ]:
```
qc = QuantumCircuit(1,1)
# Alice prepares qubit in state |+>
qc.h(0)
# Alice now sends the qubit to Bob
# but Eve intercepts and tries to read it
qc.measure(0, 0)
qc.barrier()
# Eve then passes this on to Bob
# who measures it in the X-basis
qc.h(0)
qc.measure(0,0)

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

Out[ ]:



In [ ]:
```
# Simulate circuit
aer_sim = Aer.get_backend('aer_simulator')
job = aer_sim.run(qc)
plot_histogram(job.result().get_counts())
```

Out[ ]:



We can see here that Bob now has a 50% chance of measuring 1, and if he does, he and Alice will know there is something wrong with their channel.

The quantum key distribution protocol involves repeating this process enough times that an eavesdropper has a negligible chance of getting away with this interception. It is roughly as follows:

- Step 1

Alice chooses a string of random bits, e.g.:

`1000101011010100`

And a random choice of basis for each bit:

`ZZXZXXXZXZXXXXXX`

Alice keeps these two pieces of information private to herself.

- Step 2

Alice then encodes each bit onto a string of qubits using the basis she chose; this means each qubit is in one of the states $|0\rangle$, $|1\rangle$, $|+\rangle$ or $|-\rangle$, chosen at random. In this case, the string of qubits would look like this:

$$|1\rangle|0\rangle|+\rangle|0\rangle|-\rangle|+\rangle|-\rangle|0\rangle|-\rangle|1\rangle|+\rangle|-\rangle|+\rangle|-\rangle|+\rangle|+\rangle$$

This is the message she sends to Bob.

- Step 3

Bob then measures each qubit at random, for example, he might use the bases:

`XZZZXZXZXZXZZZXZ`

And Bob keeps the measurement results private.

- Step 4

Bob and Alice then publicly share which basis they used for each qubit. If Bob measured a qubit in the same basis Alice prepared it in, they

Bob and Alice then publicly share which basis they used for each qubit. If Bob measured a qubit in the same basis Alice prepared it in, they use this to form part of their shared secret key, otherwise they discard the information for that bit.

- Step 5

Finally, Bob and Alice share a random sample of their keys, and if the samples match, they can be sure (to a small margin of error) that their transmission is successful.

Running the QKD protocol: Without Interception

Let's first see how the protocol works when no one is listening in, then we can see how Alice and Bob are able to detect an eavesdropper.

- Step 1

Alice generates her random set of bits:

In [ ]:
```
np.random.seed(seed=0)
n = 100
## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)
print(alice_bits)
```

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

To generate pseudo-random keys, we will use the `randint` function from numpy. To make sure you can reproduce the results on this page, we will set the seed to 0. In this example, Alice will send a message 100 qubits long.

At the moment, the set of bits '`alice_bits`' is only known to Alice. We will keep track of what information is only known to Alice, what information is only known to Bob, and what has been sent over Eve's channel in a table like this:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		

- Step 2

Alice chooses to encode each bit on qubit in the X or Z-basis at random, and stores the choice for each qubit in `alice_bases` . In this case, a `0` means "prepare in the Z-basis", and a `1` means "prepare in the X-basis":

In [ ]:
```
np.random.seed(seed=0)
n = 100
## Step 1
#Alice generates bits
alice_bits = randint(2, size=n)

## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
print(alice_bases)
```

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

Alice also keeps this knowledge private:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		

The function `encode_message` below, creates a list of `QuantumCircuit` s, each representing a single qubit in Alice's message:

In [ ]:
```
def encode_message(bits, bases):
    message = []
    for i in range(n):
        qc = QuantumCircuit(1,1)
        if bases[i] == 0: # Prepare qubit in Z-basis
            if bits[i] == 0:
                pass
            else:
                qc.x(0)
        else: # Prepare qubit in X-basis
            if bits[i] == 0:
                qc.h(0)
            else:
                qc.x(0)
                qc.h(0)
        qc.barrier()
        message.append(qc)
    return message


    np.random.seed(seed=0)



n = 100

## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)

## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)
```

We can see that the first bit in `alices_bits` is `0`, and the basis she encodes this in is the X-basis (represented by `1`):

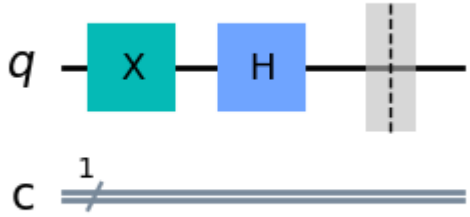
In [ ]:
```
print('bit = %i' % alice_bits[0])
print('basis = %i' % alice_bases[0])
```

```
bit = 1
basis = 1
```

And if we view the first circuit in `message` (representing the first qubit in Alice's message), we can verify that Alice has prepared a qubit in the state $|+\rangle$:

In [ ]:
```
message[0].draw(output='mpl')
```

Out[ ]:

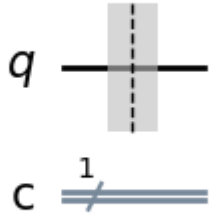


As another example, we can see that the fourth bit in `alice_bits` is `1`, and it is encoded in the Z-basis, Alice prepares the corresponding qubit in the state $|1\rangle$:

In [ ]:
```
print('bit = %i' % alice_bits[4])
print('basis = %i' % alice_bases[4])
message[4].draw(output='mpl')
```

```
bit = 0
basis = 0
```

Out[ ]:



This message of qubits is then sent to Bob over Eve's quantum channel:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message

- Step 3

Bob then measures each qubit in the X or Z-basis at random and stores this information:

In [ ]:
```
np.random.seed(seed=0)
n = 100

## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)

## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)

## Step 3
# Decide which basis to measure in:
bob_bases = randint(2, size=n)
print(bob_bases)
```

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

`bob_bases` stores Bob's choice for which basis he measures each qubit in.

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases

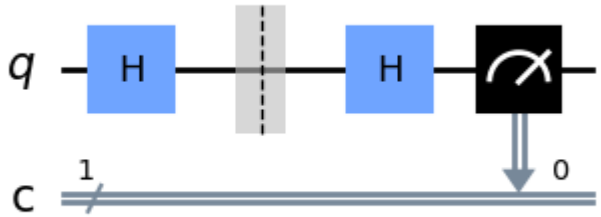
Below, the function `measure_message` applies the corresponding measurement and simulates the result of measuring each qubit. We store the measurement results in `bob_results` .

In [ ]:
```
def measure_message(message, bases):
    backend = Aer.get_backend('aer_simulator')
    measurements = []
    for q in range(n):
        if bases[q] == 0: # measuring in Z-basis
            message[q].measure(0,0)
        if bases[q] == 1: # measuring in X-basis
            message[q].h(0)
            message[q].measure(0,0)
        result = backend.run(message[q], shots=1, memory=True).result()
        measured_bit = int(result.get_memory()[0])
        measurements.append(measured_bit)
    return measurements


np.random.seed(seed=0)
n = 100

## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)

## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)

## Step 3
# Decide which basis to measure in:
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)
```

We can see that the circuit in message[0] (representing the 0th qubit) has had an X -measurement added to it by Bob. Since Bob has by chance chosen to measure in the same basis Alice encoded the qubit in, Bob is guaranteed to get the result `0` . For the 6th qubit (shown below), Bob's random choice of measurement is not the same as Alice's, and Bob's result has only a 50% chance of matching Alices'.

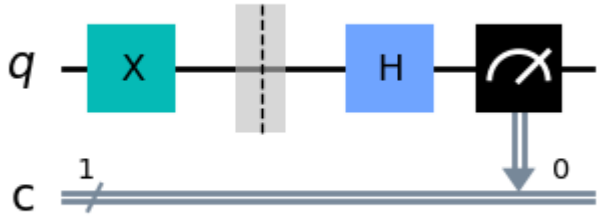
In [ ]:
```
message[0].draw(output='mpl')
```

Out[ ]:



In [ ]:
```
message[6].draw(output='mpl')
```

Out[ ]:



In [ ]:
```
print(bob_results)
```

```
[0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1,
1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1,
1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1]
```

Bob keeps his results private.

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results

- Step 4

After this, Alice reveals (through Eve's channel) which qubits were encoded in which basis:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases

And Bob reveals which basis he measured each qubit in:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases

bob_bases bob_bases

If Bob happened to measure a bit in the same basis Alice prepared it in, this means the entry in `bob_results` will match the corresponding entry in `alice_bits`, and they can use that bit as part of their key. If they measured in different bases, Bob's result is random, and they both throw that entry away. Here is a function `remove_garbage` that does this for us:

In [ ]:
```
def remove_garbage(a_bases, b_bases, bits):
    good_bits = []
    for q in range(n):
        if a_bases[q] == b_bases[q]:
            # If both used the same basis, add
            # this to the list of 'good' bits
            good_bits.append(bits[q])
    return good_bits


np.random.seed(seed=0)
n = 100

## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)

## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)

## Step 3
# Decide which basis to measure in:
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)

## Step 4
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
print(alice_key)
```

```
[0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0,
0, 1, 1, 1, 1, 0, 0, 1, 0, 0]
```

Alice and Bob both discard the useless bits, and use the remaining bits to form their secret keys.

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases
bob_bases	bob_bases	
alice_key		

In [ ]:
```
np.random.seed(seed=0)
n = 100

## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)

## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)

## Step 3
# Decide which basis to measure in:
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)

## Step 4
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
bob_key = remove_garbage(alice_bases, bob_bases, bob_results)
print(bob_key)
```

```
[0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0,
0, 1, 1, 1, 1, 0, 0, 1, 0, 0]
```

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases
bob_bases	bob_bases	
alice_key		bob_key

- Step 5

Finally, Bob and Alice compare a random selection of the bits in their keys to make sure the protocol has worked correctly:

In [ ]:
```
def sample_bits(bits, selection):
    sample = []
    for i in selection:
        # use np.mod to make sure the
        # bit we sample is always in
        # the list range
        i = np.mod(i, len(bits))
        # pop(i) removes the element of the
        # list at index 'i'
        sample.append(bits.pop(i))
    return sample


np.random.seed(seed=0)
n = 100

## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)

## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)

## Step 3
# Decide which basis to measure in:
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)

## Step 4
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
bob_key = remove_garbage(alice_bases, bob_bases, bob_results)

## Step 5
sample_size = 15
bit_selection = randint(n, size=sample_size)

bob_sample = sample_bits(bob_key, bit_selection)
print("  bob_sample = " + str(bob_sample))
```

```
print("   bob_sample =   " + str(bob_sample))
alice_sample = sample_bits(alice_key, bit_selection)
print("alice_sample = "+ str(alice_sample))
```

```
  bob_sample = [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
alice_sample = [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

Alice and Bob both broadcast these publicly, and remove them from their keys as they are no longer secret.

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases
bob_bases	bob_bases	
alice_key		bob_key
bob_sample	bob_sample	bob_sample
alice_sample	alice_sample	alice_sample

If the protocol has worked correctly without interference, their samples should match.

If their samples match, it means (with high probability) `alice_key == bob_key` . They now share a secret key they can use to encrypt their messages!

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases
bob_bases	bob_bases	
alice_key		bob_key
bob_sample	bob_sample	bob_sample

alice_sample alice_sample alice_sample

shared_key shared_key

In [ ]:
```
bob_sample == alice_sample
```

Out[ ]:
```
True
```

In [ ]:
```
print(bob_key)
print(alice_key)
print("key length = %i" % len(alice_key))
```

```
[1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0]
[1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0]
key length = 33
```

Running the QKD protocol: With Interception

Let's now see how Alice and Bob can tell if Eve has been trying to listen in on their quantum message. We repeat the same steps as without interference, but before Bob receives his qubits, Eve will try and extract some information from them. Let's set a different seed so we get a specific set of reproducible 'random' results.

- Step 1

Alice generates her set of random bits:

In [ ]:
```
np.random.seed(seed=3)
## Step 1
alice_bits = randint(2, size=n)
print(alice_bits)
```

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

- Step 2

Alice encodes these in the Z and X-bases at random, and sends these to Bob through Eve's quantum channel:

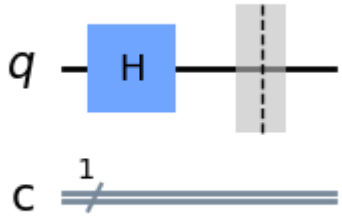
In [ ]:
```
np.random.seed(seed=3)
## Step 1
alice_bits = randint(2, size=n)
## Step 2
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)
print(alice_bases)
```

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

In this case, the first qubit in Alice's message is in the state $|+\rangle$:

In [ ]:
```
message[0].draw(output='mpl')
```

Out[ ]:



Oh no! Bob's key and Alice's key do not match. We know this is because Eve tried to read the message between steps 2 and 3, and changed the qubits' states. For all Alice and Bob know, this could be due to noise in the channel, but either way they must throw away all their results and try again- Eve's interception attempt has failed.

Risk Analysis

For this type of interception, in which Eve measures all the qubits, there is a small chance that Bob and Alice's samples could match, and Alice sends her vulnerable message through Eve's channel. Let's calculate that chance and see how risky quantum key distribution is.

- For Alice and Bob to use a qubit's result, they must both have chosen the same basis. If Eve chooses this basis too, she will successfully intercept this bit without introducing any error. There is a 50% chance of this happening.
- If Eve chooses the *wrong* basis, i.e. a different basis to Alice and Bob, there is still a 50% chance Bob will measure the value Alice was trying to send. In this case, the interception also goes undetected.
- But if Eve chooses the *wrong* basis, i.e. a different basis to Alice and Bob, there is a 50% chance Bob will not measure the value Alice was trying to send, and this *will* introduce an error into their keys.

If Alice and Bob compare 1 bit from their keys, the probability the bits will match is 0.75, and if so they will not notice Eve's interception. If they measure 2 bits, there is a $0.75^2 = 0.5625$ chance of the interception not being noticed. We can see that the probability of Eve going undetected can be calculated from the number of bits (x) Alice and Bob chose to compare:

$$P(\text{undetected}) = 0.75^x$$

If we decide to compare 15 bits as we did above, there is a 1.3% chance Eve will be undetected. If this is too risky for us, we could compare 50 bits instead, and have a 0.00006% chance of being spied upon unknowingly.

You can retry the protocol again by running the cell below. Try changing `sample_size` to something low and see how easy it is for Eve to intercept Alice and Bob's keys.

In [ ]:
```
n = 100
# Step 1
alice_bits = randint(2, size=n)
alice_bases = randint(2, size=n)
# Step 2
message = encode_message(alice_bits, alice_bases)
# Interception!
eve_bases = randint(2, size=n)
intercepted_message = measure_message(message, eve_bases)
# Step 3
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)
# Step 4
bob_key = remove_garbage(alice_bases, bob_bases, bob_results)
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
# Step 5
sample_size = 15 # Change this to something lower and see if
                 # Eve can intercept the message without Alice
                 # and Bob finding out
bit_selection = randint(n, size=sample_size)
bob_sample = sample_bits(bob_key, bit_selection)
alice_sample = sample_bits(alice_key, bit_selection)

if bob_sample != alice_sample:
    print("Eve's interference was detected.")
else:
    print("Eve went undetected!")
```

```
Eve's interference was detected.
```

Exercises

5. Test the QKD protocol with a noise model.

In [ ]:
```
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors import pauli_error
# Example error probabilities
p_reset = 0.04
p_gate1 = 0.5

# QuantumError objects
error_reset = pauli_error([('X', p_reset), ('I', 1 - p_reset)])
error_gate1 = pauli_error([('X',p_gate1), ('I', 1 - p_gate1)])
error_gate2 = error_gate1.tensor(error_gate1)

# Add errors to noise model
noise_bit_flip = NoiseModel()
noise_bit_flip.add_all_qubit_quantum_error(error_reset, "reset")
noise_bit_flip.add_all_qubit_quantum_error(error_gate1, ["h"])
```

In [ ]:
```
def measure_message(message, bases):
    backend = Aer.get_backend('aer_simulator')
    measurements = []
    for q in range(n):
        if bases[q] == 0: # measuring in Z-basis
            message[q].measure(0,0)
        if bases[q] == 1: # measuring in X-basis
            message[q].h(0)
            message[q].measure(0,0)
        result = backend.run(message[q], shots=1, memory=True,basis_gates=noise_bit_flip.basis_gates,noise_model=no
        measured_bit = int(result.get_memory()[0])
        measurements.append(measured_bit)
    return measurements
```

In [ ]:
```
np.random.seed(seed=3)
## Step 1
alice_bits = randint(2, size=n)
print(alice_bits)
```

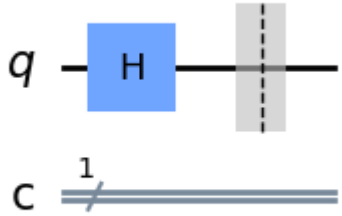
```
[0 0 1 1 0 0 0 1 1 1 0 1 1 1 0 1 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 1 0 1 1 0 1
 0 0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 0 0 1 1 1 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1
 1 0 1 1 1 1 1 1 0 1 0 0 1 1 0 1 0 0 0 0 0 1 1 0 1 1]
```

In [ ]:
```
np.random.seed(seed=3)
## Step 1
alice_bits = randint(2, size=n)
## Step 2
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)
print(alice_bases)
```

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

In [ ]:
```
message[0].draw(output='mpl')
```

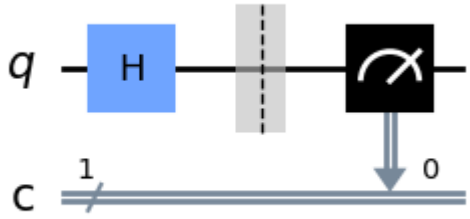
Out[ ]:



In [ ]:
```
### Interception!

np.random.seed(seed=3)
## Step 1
alice_bits = randint(2, size=n)
## Step 2
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)
## Interception!!
eve_bases = randint(2, size=n)
intercepted_message = measure_message(message, eve_bases)
print(intercepted_message)

message[0].draw(output='mpl')
```

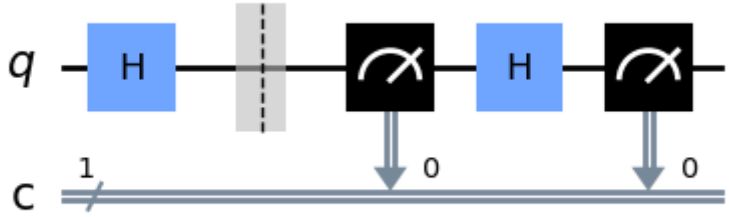
```
[0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1,
1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0,
1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0]
```

Out[ ]:



In [ ]:
```
np.random.seed(seed=3)
## Step 1
alice_bits = randint(2, size=n)
## Step 2
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)
## Interception!!
eve_bases = randint(2, size=n)
intercepted_message = measure_message(message, eve_bases)
## Step 3
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)
message[0].draw(output='mpl')
```

Out[ ]:



In [ ]:
```
np.random.seed(seed=3)
## Step 1
alice_bits = randint(2, size=n)
## Step 2
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)
## Interception!!
eve_bases = randint(2, size=n)
intercepted_message = measure_message(message, eve_bases)
## Step 3
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)
## Step 4
bob_key = remove_garbage(alice_bases, bob_bases, bob_results)
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
```

In [ ]:
```
np.random.seed(seed=3)
## Step 1
alice_bits = randint(2, size=n)
## Step 2
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)
## Interception!!
eve_bases = randint(2, size=n)
intercepted_message = measure_message(message, eve_bases)
## Step 3
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)
## Step 4
bob_key = remove_garbage(alice_bases, bob_bases, bob_results)
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
## Step 5
sample_size = 15
bit_selection = randint(n, size=sample_size)
bob_sample_noise = sample_bits(bob_key, bit_selection)
print("  bob_sample = " + str(bob_sample_noise))
alice_sample_noise = sample_bits(alice_key, bit_selection)
print("alice_sample = "+ str(alice_sample_noise))
bob_sample_noise == alice_sample_noise
```

```
  bob_sample = [1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1]
alice_sample = [1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

Out[ ]: False

In [ ]:
```
print("  bob_sample = " + str(bob_sample))
print("alice_sample = "+ str(alice_sample))
```

```
  bob_sample = [1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1]
alice_sample = [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1]
```

Question 4: Is it possible to distinguish noise from evesdropping? If so, how?

Today's afterthoughts:

What has changed, if anything, in your perspective about quantum measurements after this notebook?