Quantum Key Distribution for a Channel

- **OBJECTIVES** : Generating random keys and sending it over through simulation

```
In [1]:
import numpy as np
from qiskit import *
import random
```

- **KEY** : It is important for the sender and reciever to communicate through a KEY, something secret that is only known to them. KEY is used for encoding and decoding messages that are sent
- QKD allows us to establish a shared key which is RANDOM.
- Both have the same sequence of **0s and 1s**
- Alice and Bob **DON'T HAVE TO MEET EACH OTHER EVEN ONCE**
- This key is STILL TRANSFERRED SECRETLY.

GOAL

- To **ESTABLISH A CONNECTION WITH A SECRET and RANDOM KEY**
- This is a **symmetric key distribution**

- This is a 100% secure key distribution technique

- It employs quantum mechanics to get the knowledge about whether your key was tampered with or not
- Note : the information being transferred is just **CLASSICAL** but **KEY BEING TRANSFERRED IS TRANSFERRED THROUGH A QUANTUM CHANNEL**

STEPS

```
In [1]:
from qiskit import *
from random import getrandbits
```

Pre - Knowledge

- **You need to establish the length of the KEY**
- Here -> **KEY : 600 bits**
- Also we have a quantum and a classical channel

```
In [2]:
length_key = 600
q_channel = []
c_channel = []
```

Step1 : Alice selects bit-string and bases

- Generate 600 bits that are random
- Then generate another 600 bits to specify the **basis**

```
In [3]:
def select_encoding(length):
    # To store the random 0 / 1
    alice_bits = ""
    # to store the random vertical or diagonal bases
    alice_bases = ""

    for _ in range(length):
        # just use randbits and append to the end of the string
        alice_bits += str(getrandbits(1))
        alice_bases += str(getrandbits(1))

    return alice_bits, alice_bases
```

```
In [4]:
alice_bits, alice_bases = select_encoding(length_key)
print("Alice's bits :",alice_bits[:20],"\n")
print("Alice's basis string :",alice_bases[:20])
```

```
Alice's bits : 11010101001111011101

Alice's basis string : 11101101000001100001
```

Step2 : Bob selects bases

- BOB NEEDS TO SELECT **only bases**
- Why? **BOB ONLY SELECTS THE BASES IN WHICH HE NEEDS TO MEASURE**

```
In [5]:
def select_bob_measurement(length):
    bob_bases = ""
    # bob selects the bases in which he wants to measure
    for _ in range(length):
        bob_bases += str(getrandbits(1))

    return bob_bases
```

```
In [6]:
bob_bases = select_bob_measurement(length_key)
print("Bob's string:",bob_bases[:20])
```

```
Bob's string: 11000010101110100000
```

2.3. Step 3: Encode : All alice

Alice now uses her random list of numbers to generate a bunch of quantum states: In this excercise we are going to represent the creation of a qubit as an individual `QuantumCircuit` object.

The table below summarizes the qubit states Alice sends, based on the bit of Alice's `alice_bitstring` the corresponding bit of `selected_bases` :

Bit in Alice's `alice_bitstring`	Corresponding bit in `selected_bases`	Encoding basis	Qubit state sent
0	0	Z	0
0	1	X	+
1	0	Z	1
1	1	X	-

```
In [7]:
def generate_zqubit(q):
    Q = QuantumCircuit(1,1)
    if(int(q) == 1):
        Q.x(0)
    return Q

def generate_xqubit(q):
    Q = QuantumCircuit(1,1)
    if(int(q) == 0):
        Q.h(0)
    else:
        Q.x(0)
        Q.h(0)
    return Q
```

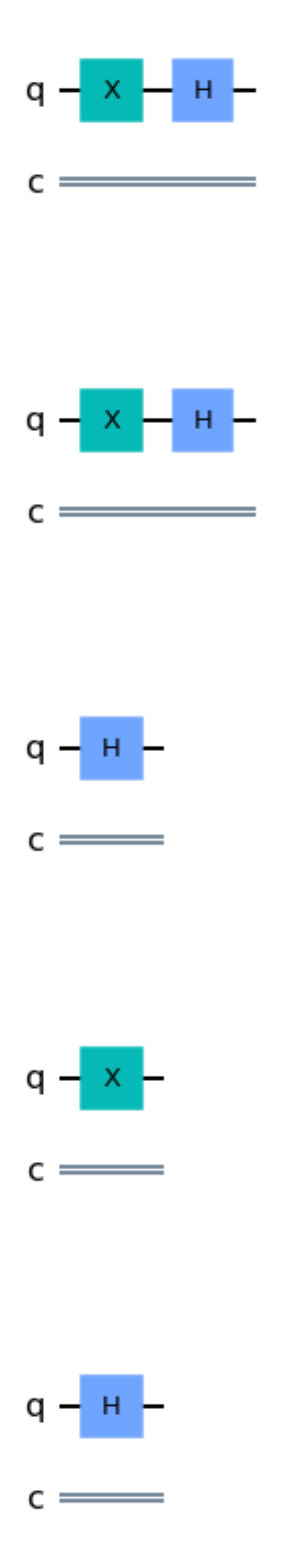
```
In [8]:
def encoded_qubits(alice_bits,alice_bases):
    '''Encoded qubits : returns a list of QuantumCircuits with
                        each QuantumCircuit representing a qubit'''
    encoded = []
    for i,k in zip(alice_bits,alice_bases):

#         print(" Alice bit :",i,"Basis :",k)

        if(i == '0' and k == '0'):
            # encoding is + and qubit would be |0>
            q = generate_zqubit(i)
        if(i == '1' and k == '0'):
            # encoding is + and qubit would be |1>
            q = generate_zqubit(i)
        if(i == '0' and k == '1'):
            # encoding is x and qubit would be |+>
            q = generate_xqubit(i)
        if(i == '1' and k == '1'):
            q = generate_xqubit(i)
#         display(q.draw('mpl'))
        # now append the qubit in the channel
        encoded.append(q)

    return encoded
```

```
In [9]:
encoded_alice_bits = encoded_qubits(alice_bits,alice_bases)
```

Step4 : All alice

- Send the qubits that you have to the **q_channel**
- These qubits are just assigned to the quantum channel

```
In [10]:
q_channel = encoded_alice_bits
for k in q_channel[:5]:
    display(k.draw('mpl'))
```



EVE ENTERS!

E1. Eve selects bases

```
In [11]:
def select_eve_bases(length):
    eve_bases = ""
    for _ in range(length):
        b = getrandbits(1)
        eve_bases+=(str(b))

    return eve_bases
```

```
In [12]:
eve_bases = select_eve_bases(length_key)
print("First 20 random bases that Eve selected :",eve_bases[:20])
```

```
First 20 random bases that Eve selected : 01010111001110100010
```

E2. Eve intercepts the Channel

```
In [13]:
def measure_eve(eve_bases,encoded_qubits,backend):
    # Eve would perform a measurement on the recieved state in the quantum channel

    eves_bitstring = ""

    for i in range(len(eve_bases)):
        b = eve_bases[i]
        q = encoded_qubits[i]

        # see which basis did Eve select
        if b == '0':
            # means Eve chose Z basis
            q.measure(0,0)
        else:
            # means Eve chose the X basis
            q.h(0)
            q.measure(0,0)

        counts = execute(q,backend=backend,shots=1).result().get_counts()
        # get the counts of values measured
        measure_bit = max(counts, key = counts.get)
        eves_bitstring+= str(measure_bit)

    return eves_bitstring
```

```
In [14]:
backend = Aer.get_backend('qasm_simulator')
eve_bits = measure_eve(eve_bases,q_channel,backend)
print("Eve measured :",eve_bits[:30],"...")
```

```
Eve measured : 011111010001010111111110001010 ...
```

E3. Now Eve encodes

```
In [15]:
def encode_eve(eve_bitstring, eve_bases):
    # encode the bit string that you measured
    encoded_bits = []
    for i in range(len(eve_bases)):
        bit = eve_bitstring[i]
        base = eve_bases[i]
        # check the base
        if(base == '0'): # z basis
            q = generate_zqubit(bit)
        else: # x basis
            q = generate_xqubit(bit)
        # append qubit to channel
        encoded_bits.append(q)

    return encoded_bits
```

```
In [16]:
encoded_eve_bits = encode_eve(eve_bits,eve_bases)
q_channel = encoded_eve_bits
```

```
In [17]:
print("Eve intercepted, Let us see what Bob does now")
```

```
Eve intercepted, Let us see what Bob does now
```

Step5 : Bob Measures

- Bob now has to measure the qubits in the **RANDOM BASES THAT HE CHOSE IN PART 2**
- Note : At the end of the measurement at Bob's end, **he does not have the key, just HIS MEASUREMENTS**

SHOTS = 1 is important for simulation

```
In [18]:
def measure_key(bob_bases, encoded_qubits,backend):
    # Perform the measurement on the qubits you recieved from Alice

    # This is because Bob is going to need to measure the qubit he
    # has recieved

    # Bob's bases need not be the same as the bases of Alice

    bob_bitstring = ""
    for i,k in zip(bob_bases,encoded_qubits):
        if i == '0':
            # bob chose the Z basis
            k.measure(0,0)
        else:
            # bob chose the x basis
            k.h(0) # why? because this is going to bring our |+> as a |0> # and it is going
 to bring |-> into |1>
            k.measure(0,0)

        counts = execute(k,backend=backend,shots= 1).result().get_counts()

        # max with a key is just a max function which gets the key with the maximum value
        measure_bit = max(counts, key = counts.get)
        bob_bitstring += str(measure_bit)

    return bob_bitstring
```

```
In [19]:
sim_backend = Aer.get_backend('qasm_simulator')
bob_string = measure_key(bob_bases,q_channel,sim_backend)
```

```
In [20]:
bob_string
```

```
'1111110010010101110111110001010000111001100111101110001000100000000110101100111101101010110111000111100010011100110001001110011010
1101010111110001110010011110010011001010110010110000101010001000110100000000100011101000001010011111110010110000110010111001110011001
1100010111111011100111101100011101111010111110100110001100011011100110111000010010011011111001001101011000111011011100000000110100
0011001110100100100110110010110101100011110100000001011110110011101111011000111010000000100100110101011101001111011000000101001011
01110100101111010111110000100010011100001000000011001100101111110111101111101111011101000100110'
```

- The measured string is **not** the same as Alice's key

```
In [21]:
alice_bits == bob_string
```

```
False
```

Step 6 : All Alice

- Announce the BASES to bob via a classical channel.
  - Note: **This all happened AFTER THE STATE HAS BEEN SENT SO EVE, EVEN IF SHE HEARD THE BASIS, CAN'T SET UP HER BASES AND THEN MEASURE , BECAUSE THE STATE HAS ALREADY BEEN PASSED THROUGH**

```
In [22]:
c_channel = alice_bases
```

Step 7 : Find the symmetric key

- Now , given you have the bases of ALICE with you, you may identify **which bases you had same as ALICE**

```
In [23]:
def compare_bases(bob_bases,alice_bases):
    '''compare_bases function just returns the
       list of indices that Bob and Alice agree on
       AFTER ALICE ANNOUNCES her bases
       '''
    indices = []

    for i in range(len(bob_bases)):
        if(bob_bases[i] == alice_bases[i]):
            indices.append(i)

    return indices
```

```
In [24]:
indices = compare_bases(bob_bases,c_channel)
c_channel = indices
```

Now, Bob will send the information to Alice that **THEY AGREED ON SO AND SO INDICES**

```
In [25]:
def construct_key_from_index(bit_string,indices):
    '''Returns the final key for both alice and bob
       given that they both know where they agreed upon'''
    final_key =""
    for k in indices:
        final_key+= str(bit_string[k])

    return final_key
```

```
In [26]:
alice_key = construct_key_from_index(alice_bits,indices)
```

```
In [27]:
bob_key = construct_key_from_index(bob_string,indices)
```

```
In [28]:
print("Alice's Key :",alice_key)
print("\nBob's Key :  ",bob_key)
```

```
Alice's Key : 1110011101100101011011101100000100110011010010111011111111100101001110010111111010100110100000111100100101010111100111111100011101011101111110111100110110011001010010110010111100111010000101111000010101100111000001001111000010100000011101000000011011110010010111000101110111001011000010010101011010110010

Bob's Key :   1110011101100101011011101000001010111010010110101011111111011001101000101101111000000000010000010010010011110011000011111010001111101101011101111110010111101000000101111000111100111010000000100011001110110011100111100111110001101101011101000101010110011001011010110010110111100001000000000101111111111000010
```

Why are these keys same?

- This is because these keys are formed according to certain indices.
- Which indices? Those indices **WHERE THEIR BASES MATCHED ACTUALLY**
- So, if their bases matched, then WE USED THOSE MATCHED BASES TO ACTUALLY CONSTRUCT OUR KEY

```
In [29]:
print("Length of original key :",length_key)
print("Length of final key :",len(alice_key))
```

```
Length of original key : 600
Length of final key : 300
```

```
In [30]:
alice_key == bob_key
```

```
False
```

Even after the reveal of the bases, we can see that the key was not same!

They need to discard this channel and go on another one where they can be sure to have no eavesdropping

```
In [47]:
def loader():
    for _ in range(10):
        print(".",end='')
        time.sleep(0.1)
        print(".",end='')
        time.sleep(0.1)
        print(".",end='\r')
    print()
```

Putting it all together

- This code block is a simulation of all the above functions. **NOTE** - all above functions need to be defined before running this code block

```
In [49]:
import time
# compiling it all together
KEY_LENGTH = 500

#1. alice selects basis and her key
alice_bits, alice_bases = select_encoding(length_key)
print("Alice generating bases")
loader()

#2. bob selects basis
bob_bases = select_bob_measurement(length_key)
print("Bob generating bases")
loader()
#3. alice encodes
encoded_alice_bits = encoded_qubits(alice_bits,alice_bases)
print("Alice is encoding")
loader()
#4. alice sends
q_channel = encoded_alice_bits
backend = Aer.get_backend('qasm_simulator')
#5. eve intercepts
eaves_drop = getrandbits(1)
if eaves_drop == 1:
    # eve interferes
    print("Eve is here!")
    # selects bases
    print("Eve is selecting bases")
    loader()
    eve_bases = select_eve_bases(length_key)
    # intercept the key
    print("Eve measures!")
    loader()
    eve_bits = measure_eve(eve_bases,q_channel,backend)
    # re-encode and send off to Bob
    print("Eve is encoding")
    loader()
    encoded_eve_bits = encode_eve(eve_bits,eve_bases)
    # send to Bob
    print("Eve has sent!")
    q_channel = encoded_eve_bits

#6. Now Bob Measures
print("Bob is measuring")
loader()
bob_bit_string = measure_key(bob_bases,q_channel,backend)

#7. now Alice reveals the bases with which she sent
print("Alice sending her bases")
loader()
c_channel = alice_bases

print("Bob comparing bases")
loader()
#8. Bob now sends alice which bases they agreed upon
indices = compare_bases(bob_bases,c_channel)

print("Bob sending common bases")
loader()
#9. send the bases on which they agree upon
c_channel = indices

#10. re-construct the key
print("Keys being constructed")
loader()
alice_key = construct_key_from_index(alice_bits,c_channel)
bob_key = construct_key_from_index(bob_bit_string,c_channel)

if eaves_drop == 1:
    print("Eve intercepted the Channel.")
    print("Are resulting keys equal ? ")
    loader()
    print("\nAlice Key :",alice_key)
    print("\nBob Key :",bob_key)
    print(alice_key == bob_key)
    print("Length of keys :",len(alice_key))

else:
    print("Eve did not intercept the Channel.")
    print("Are resulting keys equal ? ")
    loader()
    print("\nAlice Key :",alice_key)
    print("\nBob Key :",bob_key)
    print(alice_key == bob_key)
    print("Length of keys :",len(alice_key))
```

```
Alice generating bases
...............
Bob generating bases
..............
Alice is encoding
..........
Eve is here!
Eve is selecting bases
................
Eve measures!
............
Eve is encoding
.....................
Eve has sent!
Bob is measuring
...............
Alice sending her bases
...............
Bob comparing bases
................
Bob sending common bases
.................
Keys being constructed
.....................
Eve intercepted the Channel.
Are resulting keys equal ?
..................

Alice Key : 0001111010010111110100110010101111010111100001100011100000010111111011110110011111100011110001001010001110000010000110100101001011110011011111001111110111111011111111100010000111110110010100111010101011100010001010010011001010010101010101011011001011101010001001011000101111101101010111010110001110100101

Bob Key : 1000010010010101110101110101001011011101101101001011101001010111111001010000000110110000111000101101000111010101101011010111111010111100000101100110111001111100100110110101011011101111001110011100011011001001010011001001100101000011000000001101000111010101100100011110010001100111101001100011100011010011 0
False
Length of keys : 304
```