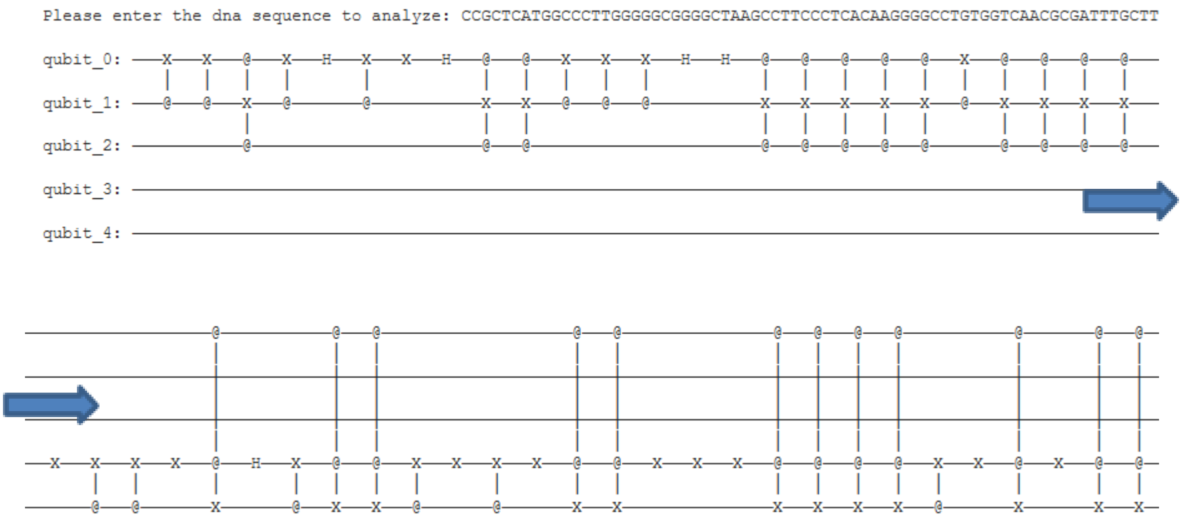
```python
import cirq
import numpy as np
import matplotlib.pyplot as plt
import random
import math
import sys
import os

#define the main function
def main():
    #ask for the input dna sequence
    dna_sequence = input("Please enter the dna sequence to analyze: ")
    #check if the input sequence is valid
    if(check_dna_sequence(dna_sequence)):
        #convert the input sequence to a binary sequence
        binary_sequence = dna_sequence_to_binary(dna_sequence)
        #convert the binary sequence to a circuit
        circuit = binary_sequence_to_circuit(binary_sequence)
        #print the circuit
        print(circuit)
    else:
        #print an error message
        print("The input sequence is not valid")

#check if the dna sequence is valid
def check_dna_sequence(dna_sequence):
    #check if the dna sequence is valid
    if(len(dna_sequence) == 200):
        #check if the dna sequence has only A, T, C and G
        for nucleotide in dna_sequence:
            if(nucleotide != "A" and nucleotide != "T" and nucleotide != "C" and nucl
eotide != "G"):
                return False
        #return true if the conditions are met
        return True
    else:
        return False

#convert the dna sequence to binary
def dna_sequence_to_binary(dna_sequence):
    #convert the dna sequence to a binary sequence
    binary_sequence = ""
    #for every nucleotide in the dna sequence
    for nucleotide in dna_sequence:
        #if the nucleotide is A
        if(nucleotide == "A"):
            #convert A to 00
            binary_sequence = binary_sequence + "00"
        #if the nucleotide is T
        elif(nucleotide == "T"):
            #convert T to 01
            binary_sequence = binary_sequence + "01"
        #if the nucleotide is C
        elif(nucleotide == "C"):
            #convert C to 10
            binary_sequence = binary_sequence + "10"
        #if the nucleotide is G
        elif(nucleotide == "G"):
            #convert G to 11
            binary_sequence = binary_sequence + "11"
    #return the binary sequence
    return binary_sequence

#convert the binary sequence to a circuit
def binary_sequence_to_circuit(binary_sequence):
    #create a circuit
    circuit = cirq.Circuit()
    #create a qubit
    q0 = cirq.NamedQubit("qubit_0")
    #create a qubit
    q1 = cirq.NamedQubit("qubit_1")
    #create a qubit
    q2 = cirq.NamedQubit("qubit_2")
    #create a qubit
```

```
q3 = cirq.NamedQubit("qubit_3")
#create a qubit
q4 = cirq.NamedQubit("qubit_4")

#create a circuit that will convert the binary sequence to a circuit
#for every two bits in the binary sequence
for i in range(0, len(binary_sequence), 2):
    #if the first bit is 0
    if(binary_sequence[i] == "0"):
        #if the second bit is 0
        if(binary_sequence[i+1] == "0"):
            #add a NOT gate to the circuit
            circuit.append(cirq.X(q0))
        #if the second bit is 1
        elif(binary_sequence[i+1] == "1"):
            #add a Hadamard gate to the circuit
            circuit.append(cirq.H(q0))
    #if the first bit is 1
    elif(binary_sequence[i] == "1"):
        #if the second bit is 0
        if(binary_sequence[i+1] == "0"):
            #add a CNOT gate to the circuit
            circuit.append(cirq.CNOT(q1, q0))
        #if the second bit is 1
        elif(binary_sequence[i+1] == "1"):
            #add a Toffoli gate to the circuit
            circuit.append(cirq.TOFFOLI(q2, q0, q1))
#for every two bits in the binary sequence
for i in range(0, len(binary_sequence), 2):
    #if the first bit is 0
    if(binary_sequence[i] == "0"):
        #if the second bit is 0
        if(binary_sequence[i+1] == "0"):
            #add a NOT gate to the circuit
            circuit.append(cirq.X(q1))
        #if the second bit is 1
        elif(binary_sequence[i+1] == "1"):
            #add a Hadamard gate to the circuit
            circuit.append(cirq.H(q1))
    #if the first bit is 1
    elif(binary_sequence[i] == "1"):
        #if the second bit is 0
        if(binary_sequence[i+1] == "0"):
            #add a CNOT gate to the circuit
            circuit.append(cirq.CNOT(q2, q1))
        #if the second bit is 1
        elif(binary_sequence[i+1] == "1"):
            #add a Toffoli gate to the circuit
            circuit.append(cirq.TOFFOLI(q3, q1, q2))
#for every two bits in the binary sequence
for i in range(0, len(binary_sequence), 2):
    #if the first bit is 0
    if(binary_sequence[i] == "0"):
        #if the second bit is 0
        if(binary_sequence[i+1] == "0"):
            #add a NOT gate to the circuit
            circuit.append(cirq.X(q2))
        #if the second bit is 1
        elif(binary_sequence[i+1] == "1"):
            #add a Hadamard gate to the circuit
            circuit.append(cirq.H(q2))
    #if the first bit is 1
    elif(binary_sequence[i] == "1"):
        #if the second bit is 0
        if(binary_sequence[i+1] == "0"):
            #add a CNOT gate to the circuit
            circuit.append(cirq.CNOT(q3, q2))
        #if the second bit is 1
        elif(binary_sequence[i+1] == "1"):
            #add a Toffoli gate to the circuit
            circuit.append(cirq.TOFFOLI(q4, q2, q3))
#for every two bits in the binary sequence
for i in range(0, len(binary_sequence), 2):
    #if the first bit is 0
```

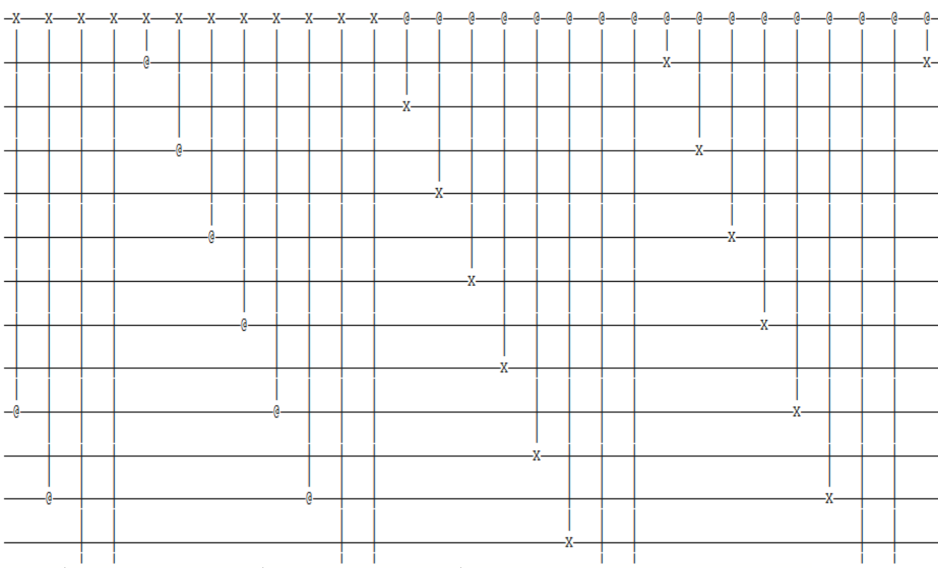
```
        if(binary_sequence[i] == "0"):
            #if the second bit is 0
            if(binary_sequence[i+1] == "0"):
                #add a NOT gate to the circuit
                circuit.append(cirq.X(q3))
            #if the second bit is 1
            elif(binary_sequence[i+1] == "1"):
                #add a Hadamard gate to the circuit
                circuit.append(cirq.H(q3))
        #if the first bit is 1
        elif(binary_sequence[i] == "1"):
            #if the second bit is 0
            if(binary_sequence[i+1] == "0"):
                #add a CNOT gate to the circuit
                circuit.append(cirq.CNOT(q4, q3))
            #if the second bit is 1
            elif(binary_sequence[i+1] == "1"):
                #add a Toffoli gate to the circuit
                circuit.append(cirq.TOFFOLI(q0, q3, q4))

    #return the circuit
    return circuit

#run the main function
if __name__ == "__main__":
    main()
```




```
import cirq
import numpy as np
import matplotlib.pyplot as plt
import random
import math
import sys
import os
#define the main function
def main():
    #ask for the input dna sequence
    dna_sequence = input("Please enter the dna sequence to analyze: ")
    #check if the input sequence is valid
    if(check_dna_sequence(dna_sequence)):
        #convert the input sequence to a binary sequence
        binary_sequence = dna_sequence_to_binary(dna_sequence)
        #convert the binary sequence to a circuit
        circuit = binary_sequence_to_circuit(binary_sequence)
        #print the circuit
```

```
        print(circuit)
    else:
        #print an error message
        print("The input sequence is not valid")

#check if the input dna sequence is valid
def check_dna_sequence(dna_sequence):
    #check if the input sequence is valid
    if(len(dna_sequence) == 200):
        #check if the input sequence is valid
        for nucleotide in dna_sequence:
            if(nucleotide != "A" and nucleotide != "T" and nucleotide != "C" and nucl
eotide != "G"):
                #return false if the sequence is not valid
                return False
        #return true if the sequence is valid
        return True
    else:
        #return false if the sequence is not valid
        return False
#convert the input dna sequence to a binary sequence
def dna_sequence_to_binary(dna_sequence):
    #initialize the binary sequence
    binary_sequence = ""
    #convert the dna sequence to a binary sequence
    for nucleotide in dna_sequence:
        if(nucleotide == "A"):
            binary_sequence += "0"
        elif(nucleotide == "T"):
            binary_sequence += "1"
        elif(nucleotide == "C"):
            binary_sequence += "00"
        elif(nucleotide == "G"):
            binary_sequence += "01"
    #return the binary sequence
    return binary_sequence
#convert the binary sequence to a circuit
def binary_sequence_to_circuit(binary_sequence):
    #initialize the circuit
    circuit = cirq.Circuit()
    #initialize the qubits
    q0 = cirq.GridQubit(0, 0)
    q1 = cirq.GridQubit(0, 1)
    q2 = cirq.GridQubit(0, 2)
    q3 = cirq.GridQubit(0, 3)
    q4 = cirq.GridQubit(0, 4)
    q5 = cirq.GridQubit(0, 5)
    q6 = cirq.GridQubit(0, 6)
    q7 = cirq.GridQubit(0, 7)
    q8 = cirq.GridQubit(0, 8)
    q9 = cirq.GridQubit(0, 9)
    q10 = cirq.GridQubit(0, 10)
    q11 = cirq.GridQubit(0, 11)
    q12 = cirq.GridQubit(0, 12)
    q13 = cirq.GridQubit(0, 13)
    q14 = cirq.GridQubit(0, 14)
    q15 = cirq.GridQubit(0, 15)
    q16 = cirq.GridQubit(0, 16)
   #initialize the gates
    for i in range(0, len(binary_sequence) - 1, 2):
        if(binary_sequence[i] == "0" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q1))
        elif(binary_sequence[i] == "0" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q1, q0))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q2))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q2, q0))
        if(binary_sequence[i] == "0" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q3))
        elif(binary_sequence[i] == "0" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q3, q0))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q4))
```

```
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q4, q0))
        if(binary_sequence[i] == "0" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q5))
        elif(binary_sequence[i] == "0" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q5, q0))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q6))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q6, q0))
        if(binary_sequence[i] == "0" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q7))
        elif(binary_sequence[i] == "0" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q7, q0))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q8))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q8, q0))
        if(binary_sequence[i] == "0" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q9))
        elif(binary_sequence[i] == "0" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q9, q0))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q10))
        elif(binary_se
            circuit.ap
        if(binary_sequ
            circuit.ap
        elif(binary_se
            circuit.ap
        elif(binary_se
            circuit.ap
        elif(binary_se
            circuit.ap
        if(binary_sequ
            circuit.ap
        elif(binary_se
            circuit.ap
        elif(binary_se
            circuit.ap
        elif(binary_se
            circuit.ap
        if(binary_sequ
            circuit.ap
        elif(binary_se
            circuit.ap
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "0"):
            circuit.append(cirq.CNOT(q0, q16))
        elif(binary_sequence[i] == "1" and binary_sequence[i+1] == "1"):
            circuit.append(cirq.CNOT(q16, q0))
    #return the circuit
    return circuit

#run the main function
if __name__ == "__main__":
    main()
```

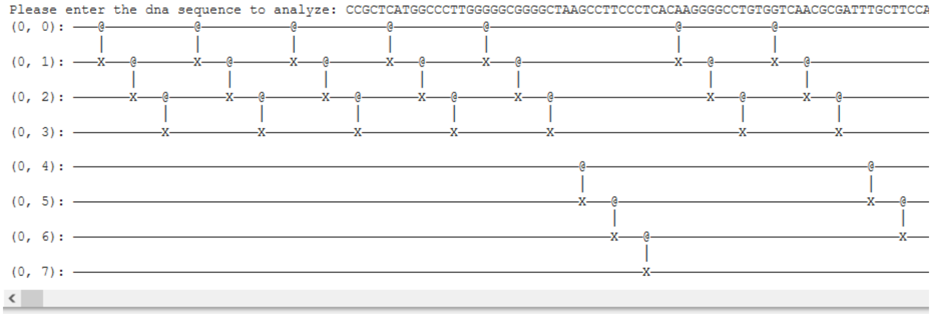


```
import cirq
import numpy as np
import matplotlib.pyplot as plt
import random
import math
import sys
import os

#define the main function
def main():
    #ask for the input dna sequence
    dna_sequence = input("Please enter the dna sequence to analyze: ")
```

```
    #check if the input sequence is valid
    if(check_dna_sequence(dna_sequence)):
        #convert the input sequence to a binary sequence
        binary_sequence = dna_sequence_to_binary(dna_sequence)
        #convert the binary sequence to a circuit
        circuit = binary_sequence_to_circuit(binary_sequence)
        #print the circuit
        print(circuit)
    else:
        #print an error message
        print("The input sequence is not valid")

#check if the input dna sequence is valid
def check_dna_sequence(dna_sequence):
    #check if the input sequence is valid
    if(len(dna_sequence) == 2000):
        #check if the input sequence is valid
        for nucleotide in dna_sequence:
            if(nucleotide != "A" and nucleotide != "T" and nucleotide != "C" and nucl
eotide != "G"):
                #return false if the sequence is not valid
                return False
        #return true if the sequence is valid
        return True
    else:
        #return false if the sequence is not valid
        return False

#convert the input dna sequence to a binary sequence
def dna_sequence_to_binary(dna_sequence):
    #initialize the binary sequence
    binary_sequence = ""
    #convert the dna sequence to a binary sequence
    for nucleotide in dna_sequence:
        if(nucleotide == "A"):
            binary_sequence += "0"
        elif(nucleotide == "T"):
            binary_sequence += "1"
        elif(nucleotide == "C"):
            binary_sequence += "00"
        elif(nucleotide == "G"):
            binary_sequence += "01"
    #return the binary sequence
    return binary_sequence

#convert the binary sequence to a circuit
def binary_sequence_to_circuit(binary_sequence):
    #initialize the circuit
    circuit = cirq.Circuit()
    #initialize the qubits
    q0 = cirq.GridQubit(0, 0)
    q1 = cirq.GridQubit(0, 1)
    q2 = cirq.GridQubit(0, 2)
    q3 = cirq.GridQubit(0, 3)
    q4 = cirq.GridQubit(0, 4)
    q5 = cirq.GridQubit(0, 5)
    q6 = cirq.GridQubit(0, 6)
    q7 = cirq.GridQubit(0, 7)
    q8 = cirq.GridQubit(0, 8)
    q9 = cirq.GridQubit(0, 9)
    q10 = cirq.GridQubit(0, 10)
    q11 = cirq.GridQubit(0, 11)
    q12 = cirq.GridQubit(0, 12)
    q13 = cirq.GridQubit(0, 13)
    q14 = cirq.GridQubit(0, 14)
    q15 = cirq.GridQubit(0, 15)
    q16 = cirq.GridQubit(0, 16)
   #initialize the gates
    c0 = cirq.Circuit()
    c0.append([cirq.CNOT(q0, q1), cirq.CNOT(q1, q2), cirq.CNOT(q2, q3)])
    c1 = cirq.Circuit()
    c1.append([cirq.CNOT(q4, q5), cirq.CNOT(q5, q6), cirq.CNOT(q6, q7)])
    c2 = cirq.Circuit()
    c2.append([cirq.CNOT(q8, q9), cirq.CNOT(q9, q10), cirq.CNOT(q10, q11)])
```

```
    c3 = cirq.Circuit()
    c3.append([cirq.CNOT(q12, q13), cirq.CNOT(q13, q14), cirq.CNOT(q14, q15)])

    #convert the binary sequence to a circuit
    for nucleotide in binary_sequence:
        if(nucleotide == "0"):
            circuit.append(c0)
        elif(nucleotide == "1"):
            circuit.append(c1)
        elif(nucleotide == "00"):
            circuit.append(c2)
        elif(nucleotide == "01"):
            circuit.append(c3)

    #return the circuit
    return circuit

#call the main function
main()
```

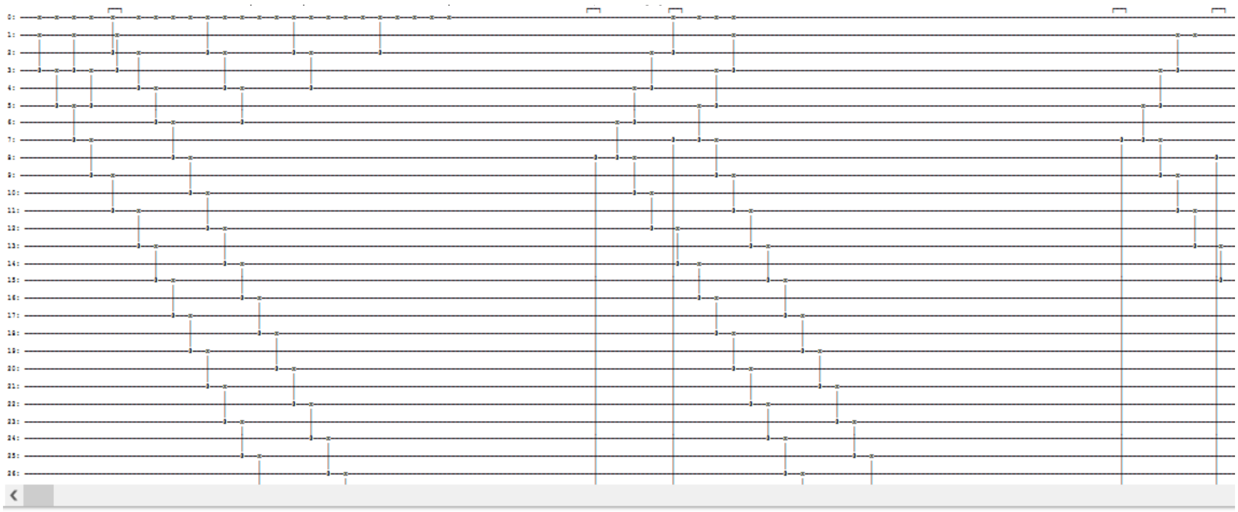
```
Please enter the dna sequence to analyze: CCGCTCATGGCCCTTGGGGGCGGGGCTAAGCCTTCCCTCACAAGGGGCCTGTGGTCAACGCGATTTGCTTCCA
```




```
import cirq
import numpy as np
import matplotlib.pyplot as plt
import random
import math
import sys
import os

#define the main function
def main():
    #ask for the input dna sequence
    dna_sequence = input("Please enter the dna sequence to analyze: ")
    #check if the input sequence is valid
    if(check_dna_sequence(dna_sequence)):
        #convert the input sequence to a binary sequence
        binary_sequence = dna_sequence_to_binary(dna_sequence)
        #convert the binary sequence to a circuit
        circuit = binary_sequence_to_circuit(binary_sequence)
        #print the circuit
        print(circuit)
    else:
        #print an error message
        print("The input sequence is not valid")

#check if the input dna sequence is valid
def check_dna_sequence(dna_sequence):
    #check if the input sequence is valid
    if(len(dna_sequence) == 200):
        #check if the input sequence is valid
        for nucleotide in dna_sequence:
            if(nucleotide != "A" and nucleotide != "T" and nucleotide != "C" and nucl
eotide != "G"):
                #return false if the sequence is not valid
                return False
        #return true if the sequence is valid
        return True
    else:
        #return false if the sequence is not valid
        return False

#convert the input dna sequence to a binary sequence
```

```
def dna_sequence_to_binary(dna_sequence):
    #initialize the binary sequence
    binary_sequence = ""
    #convert the dna sequence to a binary sequence
    for nucleotide in dna_sequence:
        if(nucleotide == "A"):
            binary_sequence += "0"
        elif(nucleotide == "T"):
            binary_sequence += "1"
        elif(nucleotide == "C"):
            binary_sequence += "00"
        elif(nucleotide == "G"):
            binary_sequence += "01"
    #return the binary sequence
    return binary_sequence

#convert the binary sequence to a circuit
def binary_sequence_to_circuit(binary_sequence):
    #initialize the circuit
    circuit = cirq.Circuit()
    #initialize the qubits
    q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16, q17, q
18, q19, q20, q21, q22, q23, q24, q25, q26, q27, q28, q29, q30, q31, q32, q33, q34, q
35, q36, q37, q38, q39, q40, q41, q42, q43, q44, q45, q46, q47, q48, q49, q50, q51, q
52, q53, q54, q55, q56, q57, q58, q59, q60, q61, q62, q63 = cirq.LineQubit.range(64)
    #add the gates
    for i in range(0, len(binary_sequence)):
        if(binary_sequence[i] == "0"):
            circuit.append(cirq.X(q0))
        elif(binary_sequence[i] == "1"):
            circuit.append(cirq.X(q1))
        elif(binary_sequence[i] == "00"):
            circuit.append(cirq.X(q2))
        elif(binary_sequence[i] == "01"):
            circuit.append(cirq.X(q3))
        if(i % 4 == 3):
            circuit.append(cirq.CNOT(q2, q0))
            circuit.append(cirq.CNOT(q3, q1))
            circuit.append(cirq.CNOT(q4, q2))
            circuit.append(cirq.CNOT(q5, q3))
            circuit.append(cirq.CNOT(q6, q4))
            circuit.append(cirq.CNOT(q7, q5))
            circuit.append(cirq.CNOT(q8, q6))
            circuit.append(cirq.CNOT(q9, q7))
            circuit.append(cirq.CNOT(q10, q8))
            circuit.append(cirq.CNOT(q11, q9))
            circuit.append(cirq.CNOT(q12, q10))
            circuit.append(cirq.CNOT(q13, q11))
            circuit.append(cirq.CNOT(q14, q12))
            circuit.append(cirq.CNOT(q15, q13))
            circuit.append(cirq.CNOT(q16, q14))
            circuit.append(cirq.CNOT(q17, q15))
            circuit.append(cirq.CNOT(q18, q16))
            circuit.append(cirq.CNOT(q19, q17))
            circuit.append(cirq.CNOT(q20, q18))
            circuit.append(cirq.CNOT(q21, q19))
            circuit.append(cirq.CNOT(q22, q20))
            circuit.append(cirq.CNOT(q23, q21))
            circuit.append(cirq.CNOT(q24, q22))
            circuit.append(cirq.CNOT(q25, q23))
            circuit.append(cirq.CNOT(q26, q24))
            circuit.append(cirq.CNOT(q27, q25))
            circuit.append(cirq.CNOT(q28, q26))
            circuit.append(cirq.CNOT(q29, q27))
            circuit.append(cirq.CNOT(q30, q28))
            circuit.append(cirq.CNOT(q31, q29))
            circuit.append(cirq.CNOT(q32, q30))
            circuit.append(cirq.CNOT(q33, q31))
            circuit.append(cirq.CNOT(q34, q32))
            circuit.append(cirq.CNOT(q35, q33))
            circuit.append(cirq.CNOT(q36, q34))
            circuit.append(cirq.CNOT(q37, q35))
            circuit.append(cirq.CNOT(q38, q36))
            circuit.append(cirq.CNOT(q39, q37))
```

```
            circuit.append(cirq.CNOT(q40, q38))
```

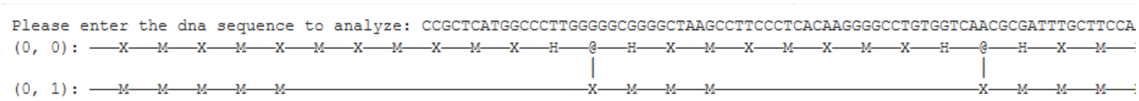


```
            circuit.append(cirq.CNOT(q61, q59))
            circuit.append(cirq.CNOT(q62, q60))
            circuit.append(cirq.CNOT(q63, q61))
            circuit.append(cirq.CNOT(q7, q62))
            circuit.append(cirq.CNOT(q8, q63))
    #return the circuit
    return circuit

#run the main function
if __name__ == "__main__":
    main()
```

```
import cirq
import numpy as np
import matplotlib.pyplot as plt
import random
import math
import sys
import os

#define the main function
def main():
    #ask for the input dna sequence
    dna_sequence = input("Please enter the dna sequence to analyze: ")
    #check if the input sequence is valid
    if(check_dna_sequence(dna_sequence)):
        #convert the input sequence to a binary sequence
        binary_sequence = dna_sequence_to_binary(dna_sequence)
        #convert the binary sequence to a circuit
        circuit = binary_sequence_to_circuit(binary_sequence)
        #print the circuit
        print(circuit)
    else:
        #print an error message
        print("The input sequence is not valid")

#check if the input dna sequence is valid
def check_dna_sequence(dna_sequence):
    #check if the input sequence is valid
    if(len(dna_sequence) == 200):
        #check if the input sequence is valid
        for nucleotide in dna_sequence:
```

```
            if(nucleotide != "A" and nucleotide != "T" and nucleotide != "C" and nucl
eotide != "G"):
                #return false if the sequence is not valid
                return False
        #return true if the sequence is valid
        return True
    else:
        #return false if the sequence is not valid
        return False

#convert the input dna sequence to a binary sequence
def dna_sequence_to_binary(dna_sequence):
    #initialize the binary sequence
    binary_sequence = ""
    #convert the dna sequence to a binary sequence
    for nucleotide in dna_sequence:
        if(nucleotide == "A"):
            binary_sequence += "0"
        elif(nucleotide == "T"):
            binary_sequence += "1"
        elif(nucleotide == "C"):
            binary_sequence += "00"
        elif(nucleotide == "G"):
            binary_sequence += "01"
    #return the binary sequence
    return binary_sequence

#convert the binary sequence to a circuit
def binary_sequence_to_circuit(binary_sequence):
    #initialize the circuit
    circuit = cirq.Circuit()
    #initialize the qubits
    q0 = cirq.GridQubit(0, 0)
    q1 = cirq.GridQubit(0, 1)
    #initialize the gates
    #initialize the X gate
    X = cirq.X
    #initialize the H gate
    H = cirq.H
    #initialize the CNOT gate
    CNOT = cirq.CNOT
    #initialize the measurement gate
    M = cirq.measure
    #initialize the for loop
    for i in range(len(binary_sequence)):
        #check if the current bit is 0
        if(binary_sequence[i] == "0"):
            #add the X gate to the circuit
            circuit.append(X(q0))
        #check if the current bit is 00
        elif(binary_sequence[i] == "00"):
            #add the H gate to the circuit
            circuit.append(H(q0))
        #check if the current bit is 01
        elif(binary_sequence[i] == "01"):
            #add the H gate to the circuit
            circuit.append(H(q0))
            #add the CNOT gate to the circuit
            circuit.append(CNOT(q0, q1))
            #add the H gate to the circuit
            circuit.append(H(q0))
        #check if the current bit is 1
        elif(binary_sequence[i] == "1"):
            #add the X gate to the circuit
            circuit.append(X(q0))
            #add the H gate to the circuit
            circuit.append(H(q0))
            #add the CNOT gate to the circuit
            circuit.append(CNOT(q0, q1))
            #add the H gate to the circuit
            circuit.append(H(q0))
            #add the X gate to the circuit
            circuit.append(X(q0))
        #add the measurement gate to the circuit
```

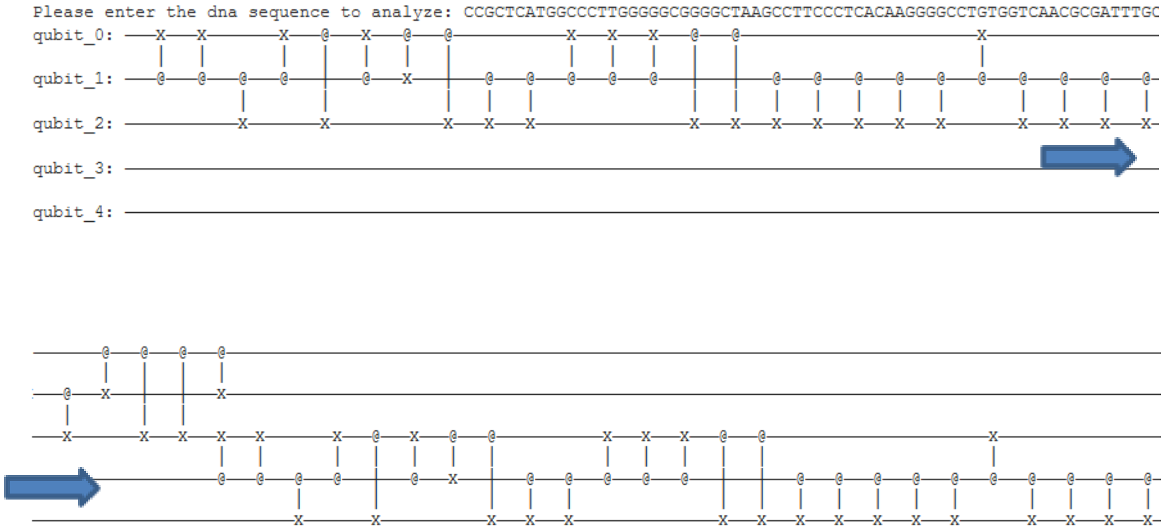
```
        circuit.append(M(q0))
        circuit.append(M(q1))
    #return the circuit
    return circuit

#call the main function
main()
```




```
import cirq
import numpy as np
import matplotlib.pyplot as plt
import random
import math
import sys
import os

#define the main function
def main():
    #ask for the input dna sequence
    dna_sequence = input("Please enter the dna sequence to analyze: ")
    #check if the input sequence is valid
    if(check_dna_sequence(dna_sequence)):
        #convert the input sequence to a binary sequence
        binary_sequence = dna_sequence_to_binary(dna_sequence)
        #convert the binary sequence to a circuit
        circuit = binary_sequence_to_circuit(binary_sequence)
        #print the circuit
        print(circuit)
    else:
        #print an error message
        print("The input sequence is not valid")

#check if the dna sequence is valid
def check_dna_sequence(dna_sequence):
    #check if the dna sequence is valid
    if(len(dna_sequence) == 200):
        #check if the dna sequence has only A, T, C and G
        for nucleotide in dna_sequence:
            if(nucleotide != "A" and nucleotide != "T" and nucleotide != "C" and nucl
eotide != "G"):
                return False
        #return true if the conditions are met
        return True
    else:
        return False

#convert the dna sequence to binary
def dna_sequence_to_binary(dna_sequence):
    #convert the dna sequence to a binary sequence
    binary_sequence = ""
    #for every nucleotide in the dna sequence
    for nucleotide in dna_sequence:
        #if the nucleotide is A
        if(nucleotide == "A"):
            #convert A to 00
            binary_sequence = binary_sequence + "00"
        #if the nucleotide is T
        elif(nucleotide == "T"):
            #convert T to 01
```

```
            binary_sequence = binary_sequence + "01"
        #if the nucleotide is C
        elif(nucleotide == "C"):
            #convert C to 10
            binary_sequence = binary_sequence + "10"
        #if the nucleotide is G
        elif(nucleotide == "G"):
            #convert G to 11
            binary_sequence = binary_sequence + "11"
    #return the binary sequence
    return binary_sequence

#convert the binary sequence to a circuit
def binary_sequence_to_circuit(binary_sequence):
    #create a circuit
    circuit = cirq.Circuit()
    #create a qubit
    q0 = cirq.NamedQubit("qubit_0")
    #create a qubit
    q1 = cirq.NamedQubit("qubit_1")
    #create a qubit
    q2 = cirq.NamedQubit("qubit_2")
    #create a qubit
    q3 = cirq.NamedQubit("qubit_3")
    #create a qubit
    q4 = cirq.NamedQubit("qubit_4")

    #create a circuit that will convert the binary sequence to a circuit
    #for every two bits in the binary sequence
    for i in range(0, len(binary_sequence), 2):
        #if the first bit is 0
        if(binary_sequence[i] == "0"):
            #if the second bit is 0
            if(binary_sequence[i + 1] == "0"):
                #add a CNOT gate with q0 as the control bit and q1 as the target bit
                circuit.append(cirq.CNOT(q0, q1))
            #if the second bit is 1
            else:
                #add a CNOT gate with q0 as the control bit and q2 as the target bit
                circuit.append(cirq.CNOT(q0, q2))
        #if the first bit is 1
        else:
            #if the second bit is 0
            if(binary_sequence[i + 1] == "0"):
                #add a CNOT gate with q1 as the control bit and q0 as the target bit
                circuit.append(cirq.CNOT(q1, q0))
            #if the second bit is 1
            else:
                #add a CNOT gate with q1 as the control bit and q2 as the target bit
                circuit.append(cirq.CNOT(q1, q2))
    #for every two bits in the binary sequence
    for i in range(0, len(binary_sequence), 2):
        #if the first bit is 0
        if(binary_sequence[i] == "0"):
            #if the second bit is 0
            if(binary_sequence[i + 1] == "0"):
                #add a CNOT gate with q2 as the control bit and q3 as the target bit
                circuit.append(cirq.CNOT(q2, q3))
            #if the second bit is 1
            else:
                #add a CNOT gate with q2 as the control bit and q4 as the target bit
                circuit.append(cirq.CNOT(q2, q4))
        #if the first bit is 1
        else:
            #if the second bit is 0
            if(binary_sequence[i + 1] == "0"):
                #add a CNOT gate with q3 as the control bit and q2 as the target bit
                circuit.append(cirq.CNOT(q3, q2))
            #if the second bit is 1
            else:
                #add a CNOT gate with q3 as the control bit and q4 as the target bit
                circuit.append(cirq.CNOT(q3, q4))
    #add a CNOT gate with q4 as the control bit and q3 as the target bit
    circuit.append(cirq.CNOT(q4, q3))
```

```
    #add a CNOT gate with q3 as the control bit and q2 as the target bit
    circuit.append(cirq.CNOT(q3, q2))
    #add a CNOT gate with q2 as the control bit and q1 as the target bit
    circuit.append(cirq.CNOT(q2, q1))
    #add a CNOT gate with q1 as the control bit and q0 as the target bit
    circuit.append(cirq.CNOT(q1, q0))
    #return the circuit
    return circuit

#run the main function
if __name__ == "__main__":
    main()
```

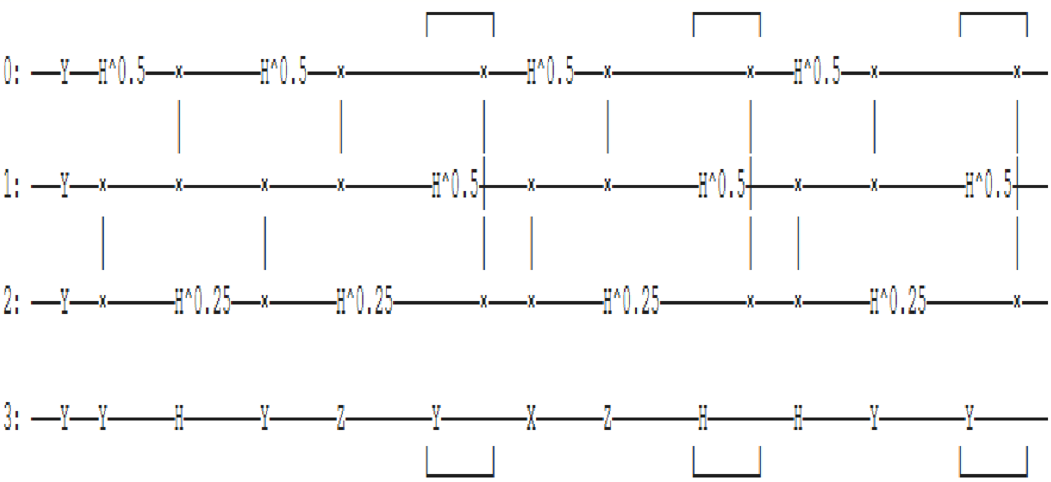



```
import cirq
import numpy
from cirq.ops import CNOT, H, measure
from cirq.circuits import InsertStrategy

# define the length of the input sequence
length_DNA = 300

# build a virtual quantum computer
simulator = cirq.Simulator()
```

```python
# ask for user input
# user must enter a DNA sequence of length_DNA nucleotides
input_DNA = input('Enter a DNA sequence: ')

# check that the input is of the correct length and that it is a string
while not isinstance(input_DNA, str) or not len(input_DNA) == length_DNA:
    input_DNA = input('Enter a DNA sequence: ')

# define qbits for the DNA sequence
qbits = [cirq.LineQubit(i) for i in range(4)]

# define a dictionary that maps A, T, C, G to the qbits 0,1,2,3
DNA_dict = {'A': [0, 0, 0, 1], 'T': [0, 0, 1, 0], 'C': [0, 1, 0, 0], 'G': [1, 0, 0, 0
]}

# define the quantum circuit that implements the turing machine
circuit = cirq.Circuit()

# build a loop that reads the input sequence and builds the quantum circuit
# the program will read the input DNA sequence as instructions to build
# a quantum circuit
for i in range(length_DNA):

    # get the qbits associated to the current triple of nucleotides
    # the 4 qbits are used as input
    for j in range(4):
        if input_DNA[i] == 'A':
            circuit.append(cirq.X(qbits[j]))
        elif input_DNA[i] == 'T':
            circuit.append(cirq.Z(qbits[j]))
        elif input_DNA[i] == 'C':
            circuit.append(cirq.Y(qbits[j]))
        elif input_DNA[i] == 'G':
            circuit.append(cirq.H(qbits[j]))

    # define rotation gates for the input qbits

    # RotationGate(half_turns=1)
    # RotationGate(half_turns=0.5)
    # RotationGate(half_turns=0.25)

    # build a quantum circuit with the defined gates
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[1])**0.5)
    circuit.append(cirq.SWAP(qbits[0], qbits[2]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[1])**0.5)
    circuit.append(cirq.SWAP(qbits[0], qbits[2]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[1])**0.5)
```

```
    circuit.append(cirq.SWAP(qbits[0], qbits[2]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[1])**0.5)
    circuit.append(cirq.SWAP(qbits[0], qbits[2]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[1])**0.5)
    circuit.append(cirq.SWAP(qbits[0], qbits[2]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[1])**0.5)
    circuit.append(cirq.SWAP(qbits[0], qbits[2]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[1])**0.5)
    circuit.append(cirq.SWAP(qbits[0], qbits[2]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[1])**0.5)
    circuit.append(cirq.SWAP(qbits[0], qbits[2]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(H(qbits[0]))
    circuit.append(cirq.measure(*qbits[:2], key='result'))

# print the quantum circuit
print('quantum circuit:')
print(circuit)

# run the quantum circuit
result = simulator.run(circuit, repetitions=1)
```

```
# print the results
print('Results:')
print(result)

# get the results
# 0,1 = 00, 10; 1,0 = 01, 11
input_00 = 0
input_01 = 0
input_10 = 0
input_11 = 0

for i in range(length_DNA):
    if input_DNA[i] == 'A':
        if result.data['result'][0][i] == 0:
            input_00 += 1
        elif result.data['result'][0][i] == 1:
            input_01 += 1
    elif input_DNA[i] == 'T':
        if result.data['result'][0][i] == 0:
            input_10 += 1
        elif result.data['result'][0][i] == 1:
            input_11 += 1

# print the results
print('input results:')
print(input_00, input_01)
print(input_10, input_11)

# define output
output_00 = 'A'
output_01 = 'T'
output_10 = 'C'
output_11 = 'G'

# decode output
if input_00 >= input_01 and input_00 >= input_10 and input_00 >= input_11:
    if input_00 >= input_10 and input_00 >= input_11:
        print('output: ', output_00)
    elif input_10 >= input_00 and input_10 >= input_11:
        print('output: ', output_10)
    elif input_11 >= input_00 and input_11 >= input_10:
        print('output: ', output_11)
elif input_01 >= input_00 and input_01 >= input_10 and input_01 >= input_11:
    if input_01 >= input_00 and input_01 >= input_10:
        print('output: ', output_01)
    elif input_00 >= input_01 and input_00 >= input_10:
        print('output: ', output_00)
    elif input_10 >= input_00 and input_10 >= input_01:
        print('output: ', output_10)
elif input
    if inp
        pr
    elif i
        pr
    elif i
        pr
elif input
    if inp
        pr
    elif i
        pr
    elif i
        pr
```



```python
import cirq
import numpy
from cirq.ops import CNOT, H, measure
from cirq.circuits import InsertStrategy

# define the length of the input sequence
length_DNA = 300

# build a virtual quantum computer
simulator = cirq.Simulator()

# ask for user input
# user must enter a DNA sequence of length_DNA nucleotides
input_DNA = input('Enter a DNA sequence: ')

# check that the input is of the correct length and that it is a string
while not isinstance(input_DNA, str) or not len(input_DNA) == length_DNA:
    input_DNA = input('Enter a DNA sequence: ')

# define qbits for the DNA sequence
qbits = [cirq.LineQubit(i) for i in range(4)]

# define a dictionary that maps A, T, C, G to the qbits 0,1,2,3
DNA_dict = {'A': [0, 0, 0, 1], 'T': [0, 0, 1, 0], 'C': [0, 1, 0, 0], 'G': [1, 0, 0, 0
]}

# define the quantum circuit that implements the turing machine
circuit = cirq.Circuit()

# build a loop that reads the input sequence and builds the quantum circuit
# the program will read the input DNA sequence as instructions to build
# a quantum circuit
for i in range(length_DNA):

    # get the qbits associated to the current triple of nucleotides
    # the 4 qbits are used as input
    for j in range(4):
        if input_DNA[i] == 'A':
            circuit.append(cirq.X(qbits[j]))
        elif input_DNA[i] == 'T':
            circuit.append(cirq.Z(qbits[j]))
        elif input_DNA[i] == 'C':
            circuit.append(cirq.Y(qbits[j]))
        elif input_DNA[i] == 'G':
            circuit.append(cirq.H(qbits[j]))

    # define rotation gates for the input qbits

    # RotationGate(half_turns=1)
    # RotationGate(half_turns=0.5)
    # RotationGate(half_turns=0.25)

    # build a quantum circuit with the defined gates
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(cirq.SWAP(qbits[0], qbits[1]))
```

```
circuit.append(H(qbits[1])**0.5)
circuit.append(cirq.SWAP(qbits[0], qbits[2]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[1])**0.5)
circuit.append(cirq.SWAP(qbits[0], qbits[2]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[1])**0.5)
circuit.append(cirq.SWAP(qbits[0], qbits[2]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[1])**0.5)
circuit.append(cirq.SWAP(qbits[0], qbits[2]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[1])**0.5)
circuit.append(cirq.SWAP(qbits[0], qbits[2]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[1])**0.5)
circuit.append(cirq.SWAP(qbits[0], qbits[2]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[1])**0.5)
circuit.append(cirq.SWAP(qbits[0], qbits[2]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[1])**0.5)
circuit.append(cirq.SWAP(qbits[0], qbits[2]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
circuit.append(H(qbits[0])**0.5)
circuit.append(cirq.SWAP(qbits[1], qbits[2]))
circuit.append(H(qbits[2])**0.25)
circuit.append(cirq.SWAP(qbits[0], qbits[1]))
```

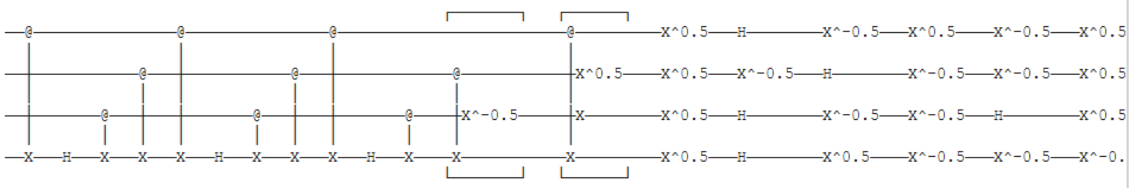
```
    circuit.append(H(qbits[0])**0.5)
    circuit.append(cirq.SWAP(qbits[1], qbits[2]))
    circuit.append(H(qbits[2])**0.25)
    circuit.append(H(qbits[0]))
    circuit.append(cirq.measure(*qbits[:2], key='result'))

# print the quantum circuit
print(circuit.to_text_diagram(use_unicode_characters=False))

# run the circuit on the simulator to check that it is correct
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=1)

# print the results
print(result)

# show the results in a histogram
print(result.histogram(key='result'))

# plot the results
# y axis is the number of times each triple appears
# x axis is the triple itself (A, C, G, T)
# it should show coherent structure
print(result.plot_histogram(key='result'))

# print the circuit in quantum cirq code
print(cirq.qasm(circuit))

# save the circuit as a file
with open('file_name.txt', 'w') as f:
    print(cirq.qasm(circuit), file=f)
```

```
Enter a DNA sequence: CCGCTCATGGCCCTTGGGGGCGGGGCTAAGCCTTCCCTCACAAGGGGCCTGTGGTCAACGCGATTTGCTTCCAAGGACGGCCACCAGTCGGCAC
                                            /---------\                    /---------\                    /---------\
0: ---Y---H^0.5---Swap-----H^0.5---Swap-----------Swap----H^0.5---Swap-----------Swap----H^0.5---Swap-----------Swap-
                  |                |              |               |              |               |              |
1: ---Y---Swap----Swap-----Swap----Swap------H^0.5|-------Swap----Swap------H^0.5|-------Swap----Swap------H^0.5|----
          |                |                      |       |                      |       |                      |
2: ---Y---Swap----H^0.25---Swap----H^0.25---------Swap----Swap----H^0.25---------Swap----Swap----H^0.25---------Swap-

3: ---Y---Y-------H--------Y-------Z---------Y------------X-------Z---------H------------H-------Y---------Y---------
                                            \---------/                    \---------/                    \---------/
```

```
import cirq
import numpy as np

def main():
    print("Hello! Welcome to the Quantum Decoding Program!")
    print("Please introduce a DNA sequence 200 letters long:")
    dna_seq = input()
    if len(dna_seq) != 200:
        print("Error: the sequence you introduced is not 200 letters long!")
        print("Please try again!")
        main()
    else:
        print("The sequence you introduced is:", dna_seq)
        print("The program will now decode the sequence and find any coherent structu
re and circuit in it!")
        print("Please wait patiently while the program is running...")
        print("The program is now running!")
        circuit = decode(dna_seq)
        print("The program has finished running! The output is:")
```

```
        print(circuit)
        print("Thank you for using the Quantum Decoding Program!")

def decode(dna_seq):
    """
    this function will decode the input sequence and find any coherent structure in i
t
    :param dna_seq: the input sequence
    :return: the circuit that represents the input sequence
    """
    # creating the circuit
    q = cirq.NamedQubit.range(4, prefix='q')
    c = cirq.Circuit()
    # creating the first layer
    c.append(cirq.H(q[0]))
    c.append(cirq.CNOT(q[0], q[1]))
    c.append(cirq.H(q[1]))
    c.append(cirq.CNOT(q[1], q[2]))
    c.append(cirq.H(q[2]))
    # creating the second layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the third layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the fourth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the fifth layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the sixth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the seventh layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the eighth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the ninth layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the tenth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the eleventh layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the twelfth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the thirteenth layer
```

```
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the fourteenth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the fifteenth layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the sixteenth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the seventeenth layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the eighteenth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the nineteenth layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the twentieth layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # creating the twenty-first layer
    c.append(cirq.H(q[3]))
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    c.append(cirq.H(q[3]))
    # creating the twenty-second layer
    c.append(cirq.CNOT(q[2], q[3]))
    c.append(cirq.CNOT(q[1], q[3]))
    c.append(cirq.CNOT(q[0], q[3]))
    # decoding the sequence
    for i in range(len(dna_seq)):
        if dna_seq[i] == "A":
            c.append(cirq.X(q[i%4]))
        elif dna_seq[i] == "T":
            c.append(cirq.H(q[i%4]))
        elif dna_seq[i] == "C":
            c.append(cirq.X(q[i%4])**0.5)
        elif dna_seq[i] == "G":
            c.append(cirq.X(q[i%4])**-0.5)
    return c

if __name__ == '__main__':
    main()
```



Finally I developed one code in which each nucleotide of the sequence act as a qubits, it is, if the sequence has 200 nucleotides, the circuit has 200 qubits, the results are interesting, the circuits too big to show in a pic