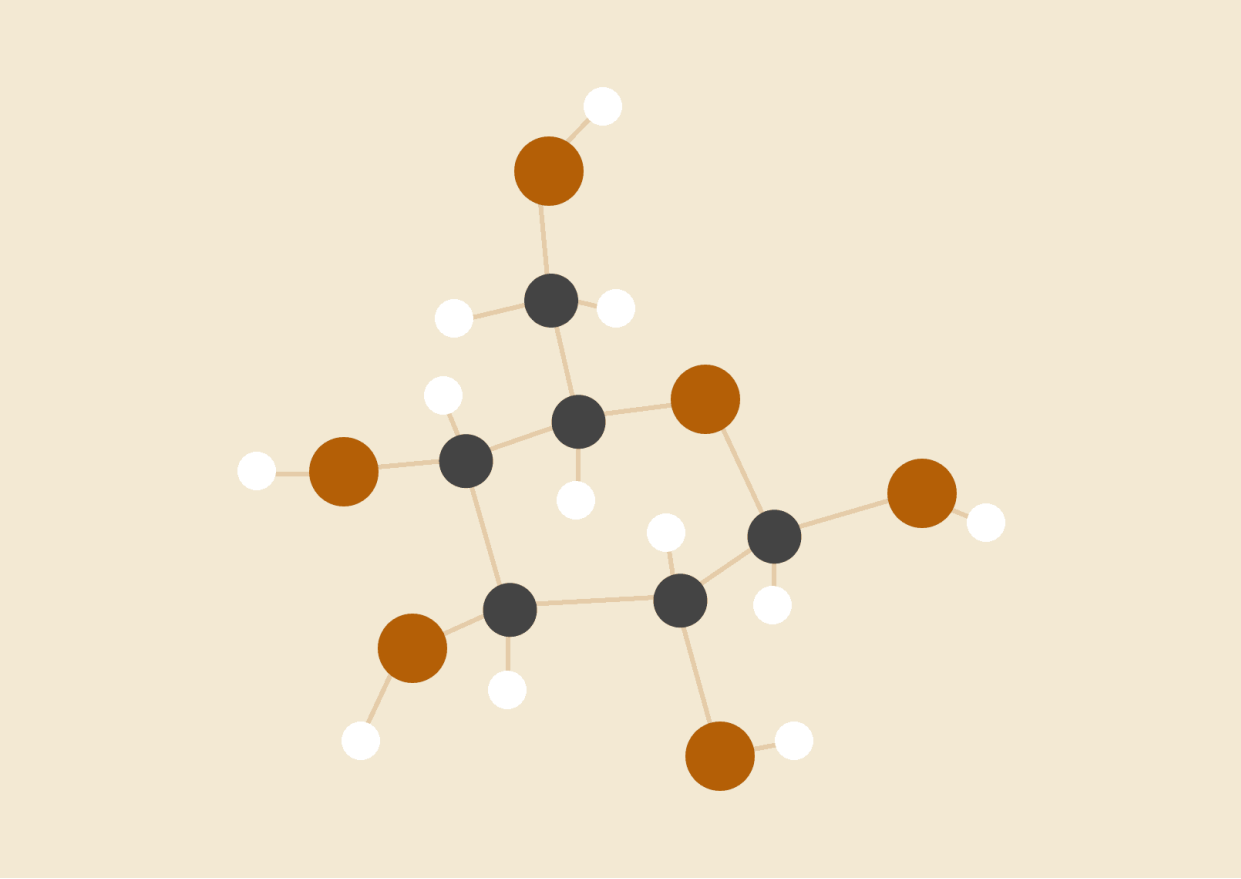
How It Works and Overhead Analysis

A code that is easy to code for the programmer is difficult to execute by the computer.



Sagar B Dollin

04.02.1998

sagardollin@gmail.com

How The Solution Works?

It's quite easy to understand. First of all let's see what are the different packages we are using.

In [1]:
```
#This is a solution to Task 3 where in we are supposed to implement circuits consisting of (H,I,X,Y,Z,Rx,Ry,Rz,CNot,Cz)
# using only these three gates Rx,Rz,Cz . This ipynb file illustartes how this compiling can be done

from qiskit import QuantumCircuit,Aer,execute                                  #the packages used here are mostly from qiskit c
from qiskit.visualization import plot_histogram, plot_bloch_multivector
from qiskit.extensions import Initialize
from qiskit_textbook.tools import random_state, array_to_latex
from math import pi
```

I installed these when I was studying Qiskit Textbook. So I'm assuming the reader is pretty much familiar with the Qiskit and Qiskit Textbook Packages.

Next, Let us implement a quantum circuit of say 4 qubits, as shown below

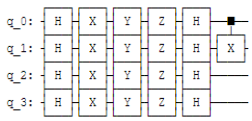
In [2]:
```
#Now let us first create a quantum circuit using as many gates as posiible(H,I,X,Y,Z,Ry,CNot)
qc = QuantumCircuit(4)

for i in range(4):
    qc.h(i)
    qc.x(i)
    qc.y(i)
    qc.z(i)
    qc.h(i)
qc.cnot(0,1)

#This is a just random circuit that I created to illustrate my solution

qc.draw()  #draw the circuit
```

Out[2]:



Let’s see the unitary of the above Quantum circuit.

```
In [3]: backend = Aer.get_backend('unitary_simulator')
unitary = execute(qc,backend).result().get_unitary()
# Display the unitary:
array_to_latex(unitary, pretext="\\text{Circuit = } ")

#Lets have a look at how the unitary matrix of this circuit looks like
```

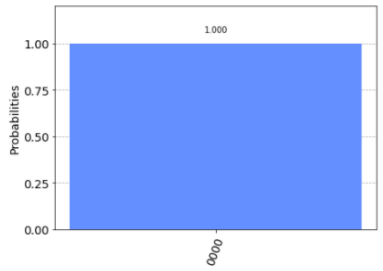
$$\text{Circuit} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}$$

It's beautiful, isn't it? That is because it looks like a 16x16 Identity Matrix. This is the point where I realised XYZ=I. Now since our Quantum circuit has a unitary of the Identity Matrix we can be sure that the state of the system doesn't change, i.e., we began with |0000> state and after measuring the circuit at the end we will get a result of |0000>. Lets visualise the result

```
In [6]: backend = Aer.get_backend('statevector_simulator')
final_counts = execute(qc,backend).result().get_counts()
plot_histogram(final_counts)

#finally the outcome which is run on a statevector simulator, so if we were to run the same on ibm computer the results may vary
```

Out[6]:



Now keeping in mind the result of our Quantum circuit let's design a logic that can convert this Circuit or perhaps any Quantum circuit(H,I,X,Y,Z,Rx,Ry,Rz,Cnot,Cz) into a Circuit comprising of only three basic gates i.e., Rx,Rz,Cz. The idea is to define functions to compile each gate into its equivalent form defined by using only Rx,Rz and Cz gates using some identities. These identities can change the global phase but they should yield the same result as yielded by the original Quantum Circuit.

Have look at the below code defining functions to convert each gate to its equivalent form ignoring the global phase difference

```
In [7]: #Now lets define all the gates used above in terms of only Rx,Rz,Cz Gates
        #We will be using some identities to do it

        def compile_h(qc,qubit):            #Identity used : H = Rz(pi/2)Rx(pi/2)Rz(pi/2)
            qc.rz(pi/2,qubit)
            qc.rx(pi/2,qubit)
            qc.rz(pi/2,qubit)

        def compile_x(qc,qubit):            #Identity used : Rx(pi)=-iX (global phase difference exists)
            qc.rx(pi,qubit)


        def compile_y(qc,qubit):             #Identity used : Y = SXSdg where S = Rz(pi/2) and Sdg = Rz(-pi/2)
            qc.rz(pi/2,qubit)
            compile_x(qc,qubit)
            qc.rz(-pi/2,qubit)

        def compile_z(qc,qubit):             #Identity used : Rz(pi) = -Z (global phase difference exists)
            qc.rz(pi,qubit)

        def compile_i(qc,qubit):             #Identity used : XX = I since X is its own inverse
            compile_x(qc, qubit)
            compile_x(qc, qubit)

        def compile_ry(qc,theta,qubit):      #Identity used : Ry = SdgRx(-thetha)S
            qc.rz(-pi/2,qubit)
            qc.rx(-theta,qubit)
            qc.rz(pi/2,qubit)

        def compile_cnot(qc,c,t):            #Indentiy used: Cnot = HCzH where we simplify H in terms of Rx and Rz
            compile_h(qc,t)
            qc.cz(c,t)
            compile_h(qc,t)
```

Now we **override** the default methods of the gates from Qiskit Quantum Circuit with the above user defined functions that we defined.

```
In [8]: #Now this is the most important step
        #Here we override our user defined functions to compile on the actual funtion gates from qiskit
        #So once we override whenever the user calls for an H gate using h() funtion then internally the compile_h() is called to break
        #Similarly for all other functions that we override with our user defined compile_<name> functions

        import sys

        module = sys.modules['qiskit']
        module.QuantumCircuit.h = compile_h            #Overriding h() with compile_h()
        module.QuantumCircuit.x = compile_x            #Overriding x() with compile_x()
        module.QuantumCircuit.y = compile_y            #Overriding y() with compile_y()
        module.QuantumCircuit.z = compile_z            #and so on for all the gates given here
        module.QuantumCircuit.ry = compile_ry
        module.QuantumCircuit.cnot = compile_cnot
        module.QuantumCircuit.i = compile_i
        sys.modules['qiskit'] = module

        #Note: This function overriding is local to the session so it won't cause any changes to the original functions of QuantumCircuit
        #What I mean is if u end this session and run the program again without doin the overriding part then the program runs as it sho
```

What we have done in the above code is we are overriding the h() function that is used to apply hadamard gate by our user defined function compile_h(). This compile_h() function gives the same result as h() but it is implemented using Rx and Rz gates. So once we run this code , from then onwards whenever we call h() method the compile_h() is called and h is implemented using these. The same is true for all the methods we are overriding here. Note that this overriding is local to the session therefore it will not cause any changes to the qiskit's original default methods.

Now, after overriding let's implement the same quantum circuit that we did in the beginning, as shown

```
In [12]: qc = QuantumCircuit(4)

         for i in range(4):
             qc.h(i)
             qc.x(i)
             qc.y(i)
             qc.z(i)
             qc.h(i)


         qc.draw()
```

Out[12]:

```
q_0: ┤ RZ(pi/2) ├┤ RX(pi/2) ├┤ RZ(pi/2) ├┤ RX(pi) ├┤ RZ(pi/2) ├┤ RX(pi) ├»
q_1: ┤ RZ(pi/2) ├┤ RX(pi/2) ├┤ RZ(pi/2) ├┤ RX(pi) ├┤ RZ(pi/2) ├┤ RX(pi) ├»
q_2: ┤ RZ(pi/2) ├┤ RX(pi/2) ├┤ RZ(pi/2) ├┤ RX(pi) ├┤ RZ(pi/2) ├┤ RX(pi) ├»
q_3: ┤ RZ(pi/2) ├┤ RX(pi/2) ├┤ RZ(pi/2) ├┤ RX(pi) ├┤ RZ(pi/2) ├┤ RX(pi) ├»
«
«q_0: ┤ RZ(-pi/2) ├┤ RZ(pi) ├┤ RZ(pi/2) ├┤ RZ(pi/2) ├┤ RZ(pi/2) ├
«
«q_1: ┤ RZ(-pi/2) ├┤ RZ(pi) ├┤ RZ(pi/2) ├┤ RZ(pi/2) ├┤ RZ(pi/2) ├
«
«q_2: ┤ RZ(-pi/2) ├┤ RZ(pi) ├┤ RZ(pi/2) ├┤ RZ(pi/2) ├┤ RZ(pi/2) ├
«
«q_3: ┤ RZ(-pi/2) ├┤ RZ(pi) ├┤ RZ(pi/2) ├┤ RZ(pi/2) ├┤ RZ(pi/2) ├
«
```

This gives us all the gates defined only in terms of what we wanted i.e., as per the requirement of the Task.

Now to check if the results are correct lets have a look at the Unitary matrix and the state vector probability

```
In [13]: #And Voila, Our circuit is all interms of Rx,Rz,Cz
         #But one task still remains that is to check if the behaviour of the circuit is same? i.e., to see if it yeilds the same result

         backend = Aer.get_backend('unitary_simulator')
         unitary = execute(qc,backend).result().get_unitary()
         # Display the Unitary:
         array_to_latex(unitary, pretext="\\text{Circuit = } ")

         #Lets have a look at its unitary (This may at some points appear different from original circuit but that's just because of globa
```

$$\text{Circuit} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The Unitary is the same, although in this example we don't have any global phase difference but if we take different examples we will come across examples where we will encounter global phase difference.

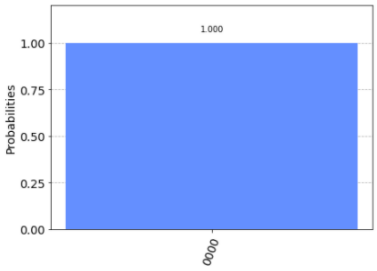
Let's have a look at the State Vector probability

```
In [16]: backend = Aer.get_backend('statevector_simulator')
         final_counts = execute(qc,backend).result().get_counts()

         plot_histogram(final_counts)

         #This is the moment we have been waiting for, when u run it on a statevector simulator the results are same. This is true for any
         #You can try for different quantum circuits yourself using this code and it should give the same results for all
         #as it preserves the Universality.
         #The results on ibm quantum computer can get a little different from the original because of noise but the circuit still remains
```

Out[16]:



And allow me to say “YIPEEEEEEEEE!!”

But our task here is not yet done let's dig into the overhead analysis

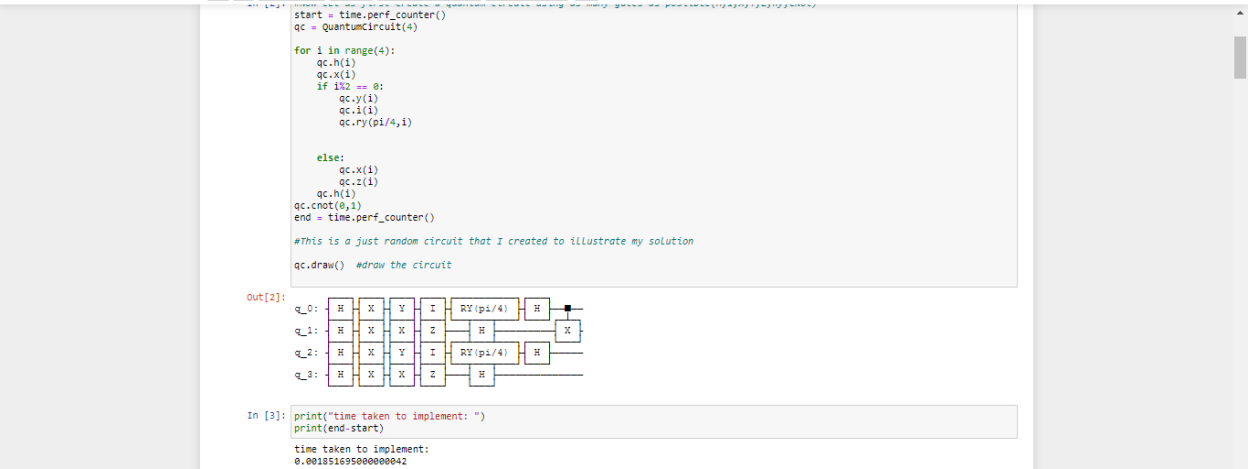
Overhead Analysis

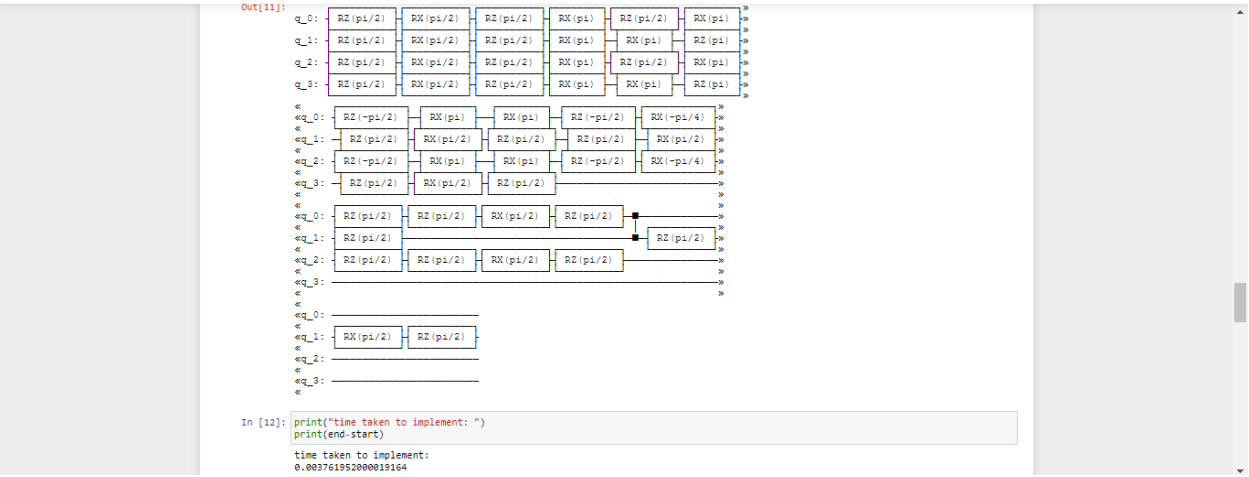
In my understanding I’m considering these things as the overhead

1. Time Complexity
2. Number of gates or the depth or the circuits

Time Complexity

In my solution every gate is converted to its equivalent in some constant time, so i can say that if there are n gates in the circuit my implementation takes O(n) to implement the gates , which is the same as without my logic so implementation doesn't add any overhead in asymptotic notations but let's see quantitatively what is the difference in time for both(this could be machine dependent)





As we can see it takes almost double the time required for implementing the circuit in terms of rx,rz,cz , also notice I have used a more complex circuit than the earlier example. Double time means O(2n) which is equivalent to O(n) which is still a linear solution to the problem.

Number of Gates(Depth of the Circuit)

The number of gates implemented in the circuit can be given by the formula:

Total number of gates = 2h+2y+i+2ry+6cnot+original_gates

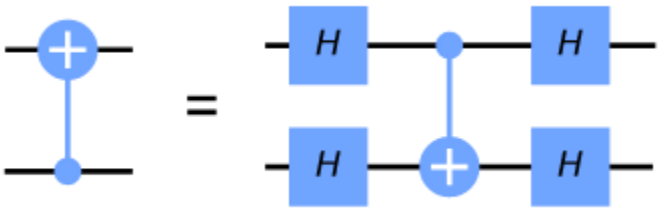
Where h is number of hadamard gate in original input circuit.

Similarly y,i,ry,and cnot are all numbers of corresponding gates in the original input gate and Original_gates is the total number of gates in the original input gates.

As we can see this increase in the number is not optimal. Can we do better? And the answer is yes of course we can, let me discuss some approaches that can be used to optimize our solution in terms of the

number of gates used.

First we need to keep track of **phase kick back**, for example see this image



source: Qiskit textbook

Whenever we encounter an input like the one in the left side we can just implement the gate on the right side, so instead of using 5 gates we use only one of them. This can be achieved using variables or lists to keep track of the structure of the circuit and replace it with appropriate optimised circuits.

Next approach is a more generalised form of what we saw above , we use identities to identify circuits that are equivalent to a smaller circuits for example

If the Input is HZH then by the identity we know that

HZH = X

And X can be implemented using Rx(pi) so this way we achieve our output circuit more compact than the input circuit.

All these can be achieved using more advanced algorithms by keeping track of the input gates using appropriate data structures.

Another interesting fact to note is we are implementing cnot gate using cz gate, but in reality on ibm quantum computer cnot is the only 2 qubit gate that can be implemented and all other 2-qubit gates are implemented using some combination of cnot and other gates. So if we run our solution on a ibm quantum computer, whenever we encounter a cnot in our input circuit we are implementing cz to achieve cnot, we use 7 gates to do this and again ibm quantum computer uses another 3 gates to apply cz that we produced to implement in terms of cnot.