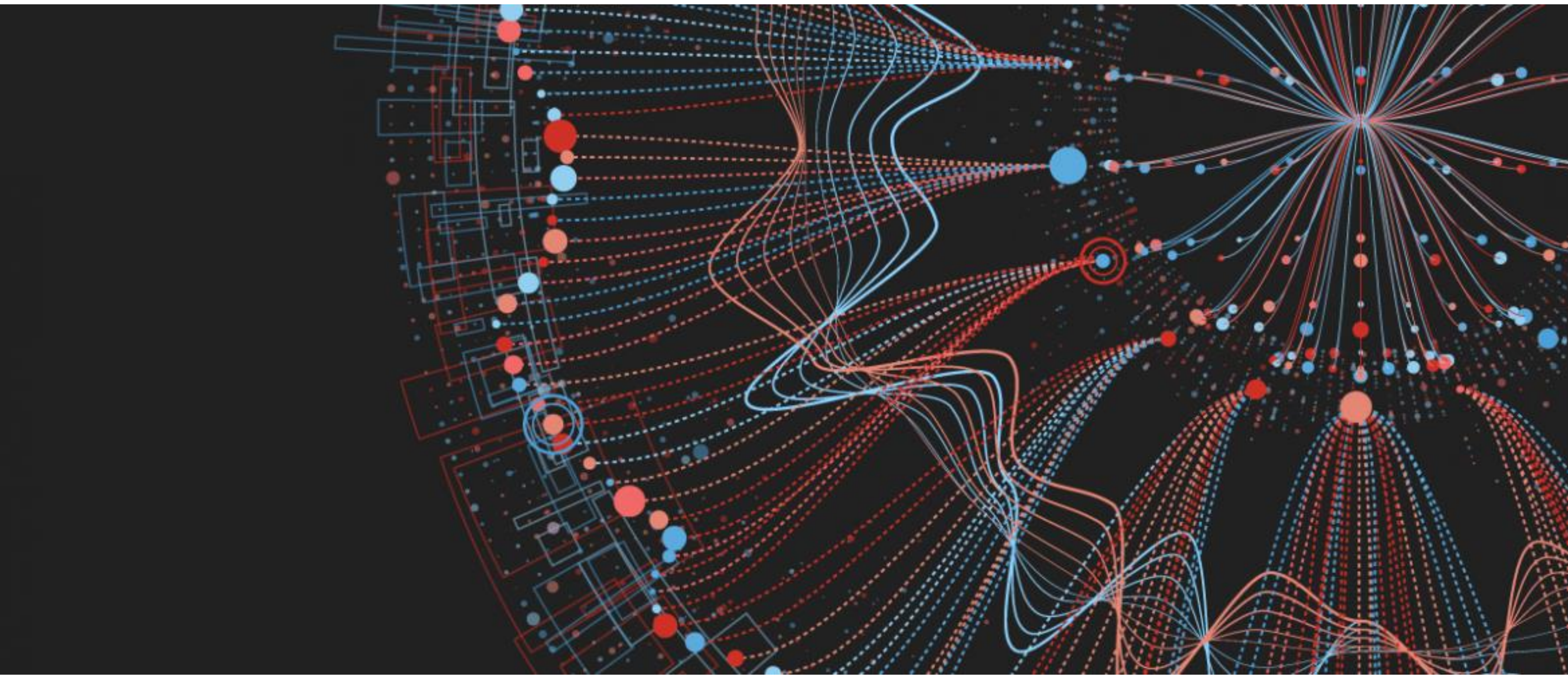
EECE 571K Project Report

April 13, 2021

An Introduction to Quantum Computing and Shor's Algorithm



Zitian (Daniel) Tong
M.Eng Student   Electrical and Computer Engineering (ECE)

THE UNIVERSITY OF BRITISH COLUMBIA
Electrical and Computer Engineering

1 Abstract

Current Internet security heavily relies on cryptographic methods such as the Secure Hash Algorithm(SHA, SHA-256, SHA-512 ...)[1]. Most cryptographic techniques are based on the simple fact that there is no efficient non-quantum algorithm that can efficiently factorize a large prime number[2][1]. In 1994, MIT professor Peter Shor came up with a polynomial-time quantum algorithm that can efficiently break this rule where factorizing a significant and meaningful prime number becomes promising[3]. However, the current best quantum computer can still not use this algorithm to factorize a cryptographically substantial number due to technology constraints[4]. This research project report will provide readers with a quick introduction to the integer factorization problem, quantum computing and quantum algorithms. Further, we will focus on the quantum part of Shor's algorithm, which can solve a crucial part of the integer factorization — period finding.

2 Introduction

Nowadays, small to intermediate-scale quantum computers already exist in university or industry laboratories(Noisy Intermediate Scale Quantum Devices, often been called NISQ)[5]. Such noisy devices with about 50 qubits are promising to demonstrate quantum supremacy in the following years[4]. However, even if successful, such devices still have a long way to benefit industries from solving real-time or mission-critical problems[6]. Researchers need to find a way to implement quantum algorithms to solve practical problems. Such algorithms should not require extensive error correction. They should not need an enormous number of qubits and quantum logic gates.

To successfully implement Shor's Algorithm without even considering error correction requires more than 5,000 qubits to factor cryptographically significant numbers. With error correction, it goes up to nearly a million[6]. Despite many qubits, it is needed; it also requires hundreds of millions of gate operations. This requirement is almost impossible for the status quo. However, with more and more quantum optimization algorithms come out, personally I think it is the right time now to consider post-quantum cryptography and study how quantum algorithms work. Therefore, the focus of this research is to explore this area(program a sample quantum code) and explain readers the fundamentals behind Shor's algorithm.

3 Quantum Computing and Quantum Algorithms

Probably the most famous idea of quantum computing is Schrodinger's cat experiment. Schrodinger's Cat is a thought experiment that describes putting a cat into a black box that contains a flask of poison and a radioactive source. If the internal radioactive monitor detects radioactivity (probability with 50%), the flasks will release poison, killing the cat. Since the outside cannot observe the situation inside the black box, we could say the cat is in the superposition of either dead or alive[7]. In terms of a bra-ket notation or Dirac notation:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \longrightarrow$$

3.1 Quantum Superposition

$|\psi\rangle$ here represents the quantum state, and $|1\rangle$ and $|0\rangle$ are two basis states. α and β are the square roots of the possibility of resulting state $|0\rangle$ and $|1\rangle$. Why square root? because we need to make in a vector space that $\alpha^2 + \beta^2 = 1$.

In quantum computing, the basic unit is qubit or quantum bit, similar to a classical bit; after the measurement, the qubit result's observation is 0 or 1. However, during our experiment, each qubit could be in a quantum superposition of either 0 or 1 with a probability associated with each. Imagine we have a system with n qubits, then we can represent this superposition of states as[3]:

$$\sum_{i=0}^{2^n-1} a_i \left|S_i\right\rangle$$

The sum of the a_i^2 represents the total probability 1, and each $|Si\rangle$ is a basis vector of outcome in a Hilbert Space. We need to notice that, unlike a classical computational system, after measurement, the entire superposition will be collapsed to a single state $|Si\rangle$ with a probability of a_i^2. When we are debugging a code in a classical computer, we usually add a print function to see the calculation's intermediate results. On the contrary, you can only measure a quantum algorithm at the very end. If you add measure operations in the middle of the algorithm, the whole system will collapse to a fixed result. In a nutshell, quantum computers are leveraging quantum superpositions properties to perform calculations in a parallel manner.

Let's discuss one example of quantum operation[3], where (the following matrix representation is dirac notation, dirac notation can be used to simiply the matrix representation):

$$input : \left|00\right\rangle, outputs : \left|00\right\rangle$$

$$input : \left|01\right\rangle, outputs : \left|01\right\rangle$$

$$input : \left|10\right\rangle, outputs : \frac{1}{\sqrt{2}}(\left|10\right\rangle + \left|11\right\rangle)$$

$$input: |11\rangle, outputs: \frac{1}{\sqrt{2}}(|10\rangle - |11\rangle)$$

The description above then matches the following matrix operation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

We can also describe the above matrix operation as a quantum gate logic where each qubit input corresponds to another. A quantum gate is feasible if and only if the corresponding matrix is unitary (dagger represents the Hermitian adjoint of a matrix)[3]:

$$U^\dagger U = UU^\dagger = I$$

3.2 Quantum Fourier Transform

When we perform a quantum operation, we are connecting input quantum qubits with quantum logic gates. By combining different types of gates, we can build certain practical unitary transformations. Now, let us check out the Quantum Fourier Transform(QFT). The Discrete Fourier Transform(classical computer) acts on a vector $(x_0, ..., x_{N-1})$ and maps it to the vector $(y_0, ..., y_{N-1})$ according to the formula[8]:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_N^{jk}$$

where $\omega_N^{jk} = e^{2\pi i \frac{jk}{N}}$. In quantum superposition, our input is $\sum_{i=0}^{N-1} x_i |i\rangle$ and maps it to

$\sum_{i=0}^{N-1} y_i \left| i \right\rangle$, note here we use N represents the total number of basis, and $\left| i \right\rangle$ to represent the basis state. Then according the to Discrete Fouries Transform, we can find out that only the amplitudes of the state will be transformed, or we can expressed it as[8]:

$$\left| x \right\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \omega_N^{xy} \left| y \right\rangle \tag{1}$$

Similar to how Discrete Fourier Transform works in classical computer, QFT is a tool to convert an input between two basses. We consider the input basis as the computational basis and the output basis as fourier basis[8].

$$\begin{gathered} |\textbf{State in Computational Basis}\rangle \xrightarrow{\textbf{QFT}} |\textbf{State in Fourier Basis}\rangle \\ \textbf{QFT}|\boldsymbol{x}\rangle = |\tilde{\boldsymbol{x}}\rangle \end{gathered} \tag{2}$$

Let's walk through one simple example of 1-qubit QFT, consider a single qubit $\left| \psi \right\rangle = \alpha \left| 0 \right\rangle + \beta \left| 1 \right\rangle$, α is the square roots of the probability of resulting basis $\left| 0 \right\rangle$ and β is the square roots of the probability of the resulting basis $\left| 1 \right\rangle$, refering to previous QFT input definition, α is X_0 and β is X_1 and N = 2. To calculate y_0 and y_1[8]:

$$y_0 = \frac{1}{\sqrt{2}}(\alpha e^{2\pi i \frac{0\times 0}{2}} + \beta e^{2\pi i \frac{1\times 0}{2}}) = \frac{1}{\sqrt{2}}(\alpha + \beta)$$

$$y_1 = \frac{1}{\sqrt{2}}(\alpha e^{2\pi i \frac{0\times 1}{2}} + \beta e^{2\pi i \frac{1\times 1}{2}}) = \frac{1}{\sqrt{2}}(\alpha - \beta)$$

Therefore, the entire procedure is:

$$U_{QFT} \left| \psi \right\rangle = \frac{1}{\sqrt{2}}(\alpha + \beta) \left| 0 \right\rangle + \frac{1}{\sqrt{2}}(\alpha - \beta) \left| 1 \right\rangle$$

It is worth to mention that the above operation is exactly same as applying a H-gate(Hadamard Operator):

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

NOTE: the below is extra reading to make the above equation to be more generalized. There are a lot of mathematical trcks that only been used by physcis, without a deep understanding of the below derivation, you can still understand the next step. Back to the QFT, if N is a fairly large number what is the general equation. Let's derive the general representation when N = 2^n, define $|x\rangle = |x_1.....x_n\rangle$ and x_1 is the most significant bit[8][9]:

$$QFT_N\,|x\rangle = \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1}\omega_N^{xy}\,|y\rangle$$

Due to $\omega_N^{xy} = e^{2\pi i\frac{xy}{N}}$ and $N = 2^n$[8][9]:

$$QFT_N\,|x\rangle = \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1}e^{2\pi i\frac{xy}{2^n}}\,|y\rangle$$

Also, in fractional binary notation $y = y_1....y_n$, $\frac{y}{2^n} = \sum_{k=1}^{n}\frac{y_k}{2^k}$[8][9]

$$QFT_N\,|x\rangle = \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1}e^{2\pi ix(\sum_{k=1}^{n}\frac{y_k}{2^k})}\,|y_1...y_n\rangle$$

$$QFT_N\,|x\rangle = \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1}\prod_{k=1}^{n}e^{2\pi ix\frac{y_k}{2^k}}\,|y_1...y_n\rangle$$

We need to apply a trick here to convert the above complex equation into tensor product format, which is expanding $\sum_{y=0}^{N-1} = \sum_{y_1=0}^{1}\sum_{y_2=0}^{1}\sum_{y_3=0}^{1}\cdots\sum_{y_n=0}^{1}$[8][9]:

$$QFT_N\,|x\rangle = \frac{1}{\sqrt{N}}\bigotimes_{k=1}^{n}(|0\rangle + e^{2\pi i\frac{x}{2^k}}\,|1\rangle)$$

$$= \frac{1}{\sqrt{N}}(|0\rangle + e^{\frac{2\pi i}{2}x}|1\rangle) \bigotimes (|0\rangle + e^{\frac{2\pi i}{2^2}x}|1\rangle) \bigotimes ... \bigotimes (|0\rangle + e^{\frac{2\pi i}{2^{n-1}}x}|1\rangle) \bigotimes (|0\rangle + e^{\frac{2\pi i}{2^n}x}|1\rangle)$$

4 Integer Factorization Problem

After introducing quantum computing basics, quantum superposition, and a more complicated Quantum Fourier Transformation(QFT) model, it is time for us to dive into the integer factorization problem, which is the problem that Shor's Algorithm was trying to solve. In number theory, integer factorization is to decompose an integer into a product of smaller numbers; if we restricted the smaller numbers to prime numbers, the process is also called prime factorization[10]. As been discussed in the lectures, when numbers are sufficiently large, there is no efficient non-quantum algorithm that can perform integer factorization problems. An efficient quantum algorithm Shor's algorithm appeared in 1994; before that, the best integer factoring algorithm asymptotically is the number field service, which to factor an integer n takes asymptotic running time $exp(clog(n)^{\frac{1}{3}}loglog(n)^{\frac{2}{3}})$ for some constant N[3]. One impressive thing is that Shor's algorithm is not directly solving the integer factorization issue; we are not directly inputting the target integer N into the quantum algorithm and produces output factor1 and factor2[3]. Shor's algorithm is targeting a reduced problem called finding the order of an element. Therefore, the complexity by using Shor's algorithm to factorizing a large integer number N has two steps:

1. solving reduced problem complexity is asymptotically $exp(clog(n)^{\frac{1}{3}}loglog(n)^{\frac{2}{3}})$ on a quantum computer(find the p value)
2. converting the reduced problem result into factors of N running on a classical computer - logn, also been called post-processing time(using p-value to calculate factors)

4.1 Reducing Integer Factorization

Let us first assume a large integer number we need to factorize, which is N. Besides; we will randomly select a guess number g (non zero), which is smaller than N and does not share a common factor with N[11].

$$\underbrace{g, N}_{share\ no\ common\ factors}$$

Of course, if our selected g share a common factor with N, we can easily find N's factors by using the Euclidean algorithm. You are fortunate; you already solved the integer factorization for a specific case. Generally speaking, the possibility of finding such a case is super slight. There is a number theorem that states for our case if g repeatedly time itself p times, the following relation must be satisfied[11]:

$$\underbrace{g, N}_{share\ no\ common\ factors} \quad \rightarrow \underbrace{g \times g \times g ... \times g}_{\text{p times}} = m \times N + 1 \tag{3}$$

The reduced problem that we need to use Shor's algorithm to find out is: there should exist an integer p that results in $g^p \equiv 1(modN)$.

$$\underbrace{g, N}_{share\ no\ common\ factors} \quad \rightarrow \quad \underbrace{g^p \equiv 1(modN)}_{\text{there should exist an integer p}} \quad \rightarrow g^p = mN + 1 \tag{4}$$

Moving the +1 from the right side of the equation to the left side:

$$g^p - 1 = mN \tag{5}$$

Further, we can turn equation 5 into another format[11]:

$$(g^{\frac{p}{2}} - 1)(g^{\frac{p}{2}} + 1) = mN \rightarrow \underbrace{(g^{\frac{p}{2}} - 1)}_{\textbf{factor 1}}\underbrace{(g^{\frac{p}{2}} + 1)}_{\textbf{factor 2}} \equiv 0(modN) \tag{6}$$

From equation 6, we can find out that once we find the p that satisfied the condition in equation 3. We can successfully factorize the N into two factors: $g^{\frac{p}{2}}+1$ and $g^{\frac{p}{2}}-1$. (NOTE: the factor1 and factor2 might not be the actual factors of N but they do share common factors with N)

Let's practice a simple example to make sure you understand the above problem conversion. When g=7 ,N=15 and p=4 (given for practice purposes).

$$7^{\frac{4}{2}}+1=50$$

$$7^{\frac{4}{2}}-1=48$$

However, both 50 and 48 are bigger than 15, how should we factorize the 15 into the appropriate answers. The answer is Euclidean algorithm since factor1 and factor2 both share common factors with N, we can apply Euclidean algorithm to find out the greatest common divisor:

$$gcd(50,15)=5 \rightarrow \textbf{real factor 1: 5}$$

$$gcd(48,15)=3 \rightarrow \textbf{real factor 2: 3}$$

$$\textbf{and factor 1} \times \textbf{factor 2} = \textbf{N}$$

Let's consider some edge cases in the above approach:

Case1:

$$\underbrace{(g^{\frac{p}{2}}-1)}_{\textbf{a N}}\underbrace{(g^{\frac{p}{2}}+1)}_{\textbf{b}} \equiv 0(modN) \qquad (7)$$

From equation 7, we can determine if factor 1 is equal to an integer multiple of N. If so, then we have to pick another p to try the entire quantum process again because, in that case, we are factoring the index before N instead of the N - the significantly large number.

Case2:

$$\textbf{If p is an odd number} \rightarrow g^{\frac{p}{2}} \rightarrow \textbf{not an integer} \tag{8}$$

Equation 8 tells us if the input p is an odd number. Then the two factors will not be able to be an integer number. Thus, when this situation happens, we need to pick another p to try the entire quantum process again.

Case 1 and Case 2 are inevitable since the p-value generated by the quantum process is entirely random. In practice, designers have to repeatedly run the quantum algorithms to find out the possible p-values(by using QFT and Quantum Phase Estimation) and try them individually on a classical computer. Luckily, the possibility of not showing case 1 and case 2 is the following:

$$1 - P(\textbf{case 1 + case 2}) = 37.5\%$$

4.2 Periodic Properties of P-value

Our magic p has an attractive property that can be shown in the following steps:

$$g^p = m \cdot N + 1 \tag{9}$$

Let's randomly choose a value of c that makes the remainder becomes 3:

$$g^c = m_1 \cdot N + 3 \tag{10}$$

Let equation 9 time equation 10:

$$g^c \cdot g^p = (m_1 \cdot N + 3)(m \cdot N + 1)$$

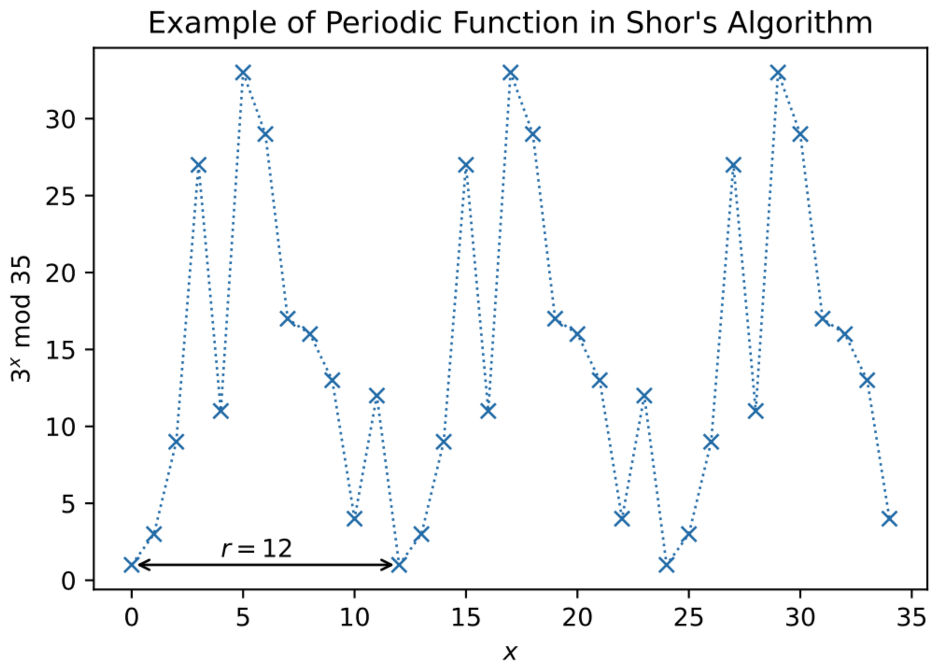
$$g^{c+p} = m_1 \cdot m \cdot N^2 + (m_1 + 3m)N + 3$$

$$g^{c+p} \equiv 3(modN) \tag{11}$$

Let equation 9 and equation 11 time together:

$$g^{c+2p} \equiv 3(modN) \tag{12}$$

Notice the remainder of equations 9, 10, 11 and 12; we can find an exciting property that the p has a repetitive property. We can draw a simple diagram to show this property. The x-axis represents the changing value of p, and the y-axis represents the remainder values. We can obtain the following diagram(assume N=35)[12]:



5 Shor's Algorithm and Implementation

From the previous section, we converted the Integer factorization problem to finding the modular exponentiation function period. In order word, we need to find out the value of p as shown in the following equation:

$$g^p = m \cdot N + 1 \tag{13}$$

Besides, we find out that p has a periodic characteristic. Therefore, by using frequency analysis, we can easily find out the value of P. However, in a classical computer, calculating Discrete Fourier Transformation is time-consuming. It is not even better than just randomly guess one. On the contrary, it is straightforward to calculate in a frequency domain by using quantum computer due to quantum superposition[8].

The below four steps shows how we should break a sizeable prime number if a large-scale and low noise quantum computer exists:

STEP 1: convert the problem into finding the order
STEP 2: find N and randomly select a number g, which should not share a common factor with N (if it does, then u apply euclidean algorithm to break the elements)
STEP 3: run QFT to generate p-value
STEP 4: testing all possible p-values generated by the quantum algorithm

Since quantum computers are unstable and errors are inevitable, we need to run quantum programs many times. Besides, everything in quantum is a probability. Usually, we need to analyze the quantum computer's output in a histogram manner to figure out the correct result.

6 Code Explanation

To successfully implementing a quantum circuit to test the Shor's algorithm. I used some open source libraries: NumPy, Matplotlib and qiskit. IBM Q provided users with a learning tutorial. I highly recommend readers to checking out IBM Quantum Experience first to study necessary knowledge about building Quantum Circuit and setting up a test environment. In this section, I will explain the sample code that I designed to successfully factorize 15 into 5 and 3!

Building Helper Function - cmod15: hard code a quantum circuit to convert questions to finding the period

```
'''
convert the integer factorization problem into a period finding problem
modular exponantiation function, below function is a hard-coded version
of factorizing 15 and random guess 7

input: a - random guess
        power -  value p
'''
def c_amod15(a, power):
    U = QuantumCircuit(4)
    for iteration in range(power):
        U.swap(2, 3)
        U.swap(1, 2)
        U.swap(0, 1)
        for q in range(4):
            U.x(q)
        U.to_gate()
        U.name = '%i^%i mode 15' % (a, power)
        c_U = U.control()
        return c_U
```

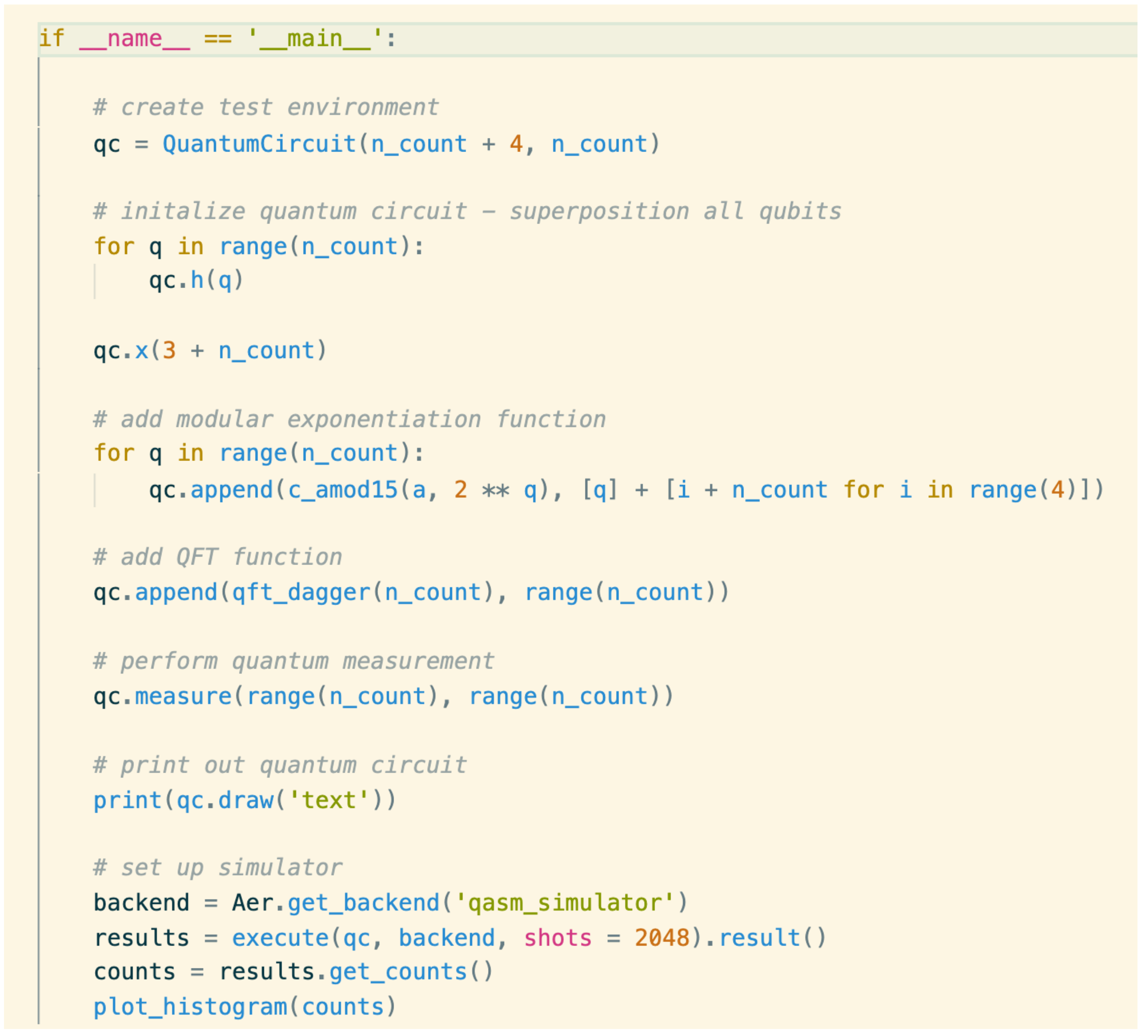
Building Helper Function - qft function: quantum fourier transform quantum circuit

```
'''
quantum circuit for QFT
input: n - number of deisgned qubit
'''
def qft_dagger(n):
    qc = QuantumCircuit(n)
    for qubit in range(n // 2):
        qc.swap(qubit, n - qubit - 1)
    for j in range(n):
        for m in range(j):
            qc.cp(-np.pi / float(2 ** (j - m)), m, j)
        qc.h(j)
    qc.name = 'QFT dagger'
    return qc
```

Setting the system requires 8 qubits and define g to be 7(randomly guess one):

```
# define 8-qubit
n_count = 8
# define our guess is 7
a = 7
```

Program Main Logic:



```python
if __name__ == '__main__':

    # create test environment
    qc = QuantumCircuit(n_count + 4, n_count)

    # initalize quantum circuit - superposition all qubits
    for q in range(n_count):
        qc.h(q)

    qc.x(3 + n_count)

    # add modular exponentiation function
    for q in range(n_count):
        qc.append(c_amod15(a, 2 ** q), [q] + [i + n_count for i in range(4)])

    # add QFT function
    qc.append(qft_dagger(n_count), range(n_count))

    # perform quantum measurement
    qc.measure(range(n_count), range(n_count))

    # print out quantum circuit
    print(qc.draw('text'))

    # set up simulator
    backend = Aer.get_backend('qasm_simulator')
    results = execute(qc, backend, shots = 2048).result()
    counts = results.get_counts()
    plot_histogram(counts)
```

The main program logic is similar to what I concluded in the last section. In the beginning, a quantum computer is not in superposition. Therefore, we need to apply the Hadamard operation to each of the qubits. Next, we add the first helper function designed to turn superpositioned quantum bits into a modular exponentiation function. The upcoming step is running QFT to generate a possible p-value. In the end, we measure the possible outcomes.

The sample program outputs 00000000, 01000000,10000000 and 11000000. These are the binary reading of in total eight quantum bits. To successfully get the p-value, we first turn binary representation into decimal representation 0, 64, 128 and 192. QFT will output possible values in reverse format(out of $2^{\textbf{total number of qbits}}$). Therefore, potential $\frac{1}{p}$ values are 0(0), $\frac{64}{256}(\frac{1}{4})$, $\frac{128}{256}(\frac{1}{2})$ and $\frac{192}{256}(\frac{2}{3})$. Plugging each of the estimation into the equation 6, we can now get the factors of 15 are 5 and 3:

case: when $\frac{1}{p}$ is $\frac{1}{4}$:

$$7^{\frac{4}{2}} + 1 = 50$$

$$7^{\frac{4}{2}} - 1 = 48$$

$$gcd(50, 15) = 5 \rightarrow \textbf{real factor 1: 5}$$

$$gcd(48, 15) = 3 \rightarrow \textbf{real factor 2: 3}$$

$$\textbf{and factor 1} \times \textbf{factor 2} = \textbf{N}$$

7 Future Work

The existence and advancement of quantum computers will bring a huge shift to our current tech industry, from Cryptography[1], Artificial Intelligence(AI) [13] to Computer Architecture[14]. However, to successfully implement Shor's algorithm, without even considering the error correction, requires more than 5,000 qubits to factor cryptographically significant numbers. With error correction, it goes up to nearly a million[6]. Despite a large number of qubits is required, it also requires hundreds of millions of gate operations[6]. This requirement is nearly impossible for the status quo. Researchers need to find a way to implement quantum algorithms to solve useful problems, such algorithms should not need extensive error correction and they should not need enormous numbers of qubits and quantum logic gates. One thing caught my attention recently is QAOA[15], an optimization method for quantum algorithms.

> *"If QAOA really works, one thing that we should consider is designing programming languages and compilers of everything just for QAOA and not for general quantum algorithms because if they were designed specifically for QAOA they would be more efficient for that problem than general compilers ...... right now quantum computers are so small, and may not be able to afford generally programming languages and generally architectures as well ......"[2]*
>
> ***Peter Shor, ISCA 2018***

Since its publication, QAOA attracts a lot of researchers across the industry and a lot of papers have given out many user cases and performance comparisons between QAOA and the state-of-art classical solver. However, the complexity of QAOA depends on how we design the shallow circuit, how the quantum computer was been designed and how we interact with our quantum logic. From Peter Shor's speech on ISCA 2018, instead of thinking of a general-purpose quantum computer for doing useless tasks, the industry needs to develop QAOA based quantum computer to optimize its performance in order to have an impact on their customer's problems.

The future work in this area including exploring new optimization algorithms that may outperform QAOA and more research on QAOA based quantum computers[2].

References

[1] T. M. Fernández-Caramès and P. Fraga-Lamas. Towards post-quantum blockchain: A review on blockchain cryptography resistant to quantum computing attacks. *IEEE Access*, 8:21091–21116, 2020.

[2] QAOA (Peter Shor, ISCA 2018). https://www.youtube.com/watch?v=HHIWUi3GmdMt=1531s. Accessed: 2020-12-06.

[3] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, Oct 1997.

[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando Brandao, David Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, and John Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 10 2019.

[5] T. Tanimoto, S. Matsuo, S. Kawakami, Y. Tabuchi, M. Hirokawa, and K. Inoue. How many trials do we need for reliable nisq computing? In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 288–290, 2020.

[6] G. G. Guerreschi and A. Y. Matsuura. Qaoa for max-cut requires hundreds of qubits for quantum speed-up. *Scientific Reports*, 9(1), May 2019.

[7] Schrödinger's cat, Apr 2021.

[8] The Qiskit Team. Quantum fourier transform, Apr 2021.

[9] Motohiko Ezawa. Electric circuits for universal quantum gates and quantum fourier transformation. *Physical Review Research*, 2(2), Jun 2020.

[10] Integer factorization, Mar 2021.

[11] Franklin de Lima Marquezino, Renato Portugal, and Carlile Lavor. Shor's algorithm for integer factorization. *SpringerBriefs in Computer Science*, page 57–77, 2019.

[12] The Qiskit Team. Shor's algorithm, Apr 2021.

[13] J. Choi, S. Oh, and J. Kim. The useful quantum computing techniques for artificial intelligence engineers. In *2020 International Conference on Information Networking (ICOIN)*, pages 1–3, 2020.

[14] L. Riesebos, X. Fu, A. A. Moueddenne, L. Lao, S. Varsamopoulos, I. Ashraf, J. van Someren, N. Khammassi, C. G. Almudever, and K. Bertels. Quantum accelerated computer architectures. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2019.

[15] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014.