Understanding what Enables Shor's Algorithm to Break the RSA Algorithm

Candidate Code: kdp683

Page Count: 18

1 Introduction

In the summer of 2023, I underwent an internship at a local quantum computing company named TerraNexum, a company focused on using the power of quantum computing to aid in fighting the climate crisis. Specifically, the company focuses on using quantum computing for solving optimization problems such as designing more efficient pathways for energy resources to flow along with planning out the most efficient way to transition to renewable sources of energy while still remaining feasible. Through this internship, I learned about the immense power that quantum computing holds and I even got to write code that used quantum resources through cloud services like Amazon's Braket. During my internship, I worked on an open-source project called OpenQGO that took in layers of geospatial information and connected individual nodes to each other based on the result of a quantum optimization algorithm called QAOA (Quantum Approximate Optimization Algorithm). This project gave me beginning insight into how we could use quantum computing and software together to redefine the way we solve our existing sustainability problems. I was able to see the power that quantum computing contained for solving really complex problems like how to best transition our current energy infrastructure to more sustainable fuels. However, during my internship I realized that my mathematical knowledge about how quantum algorithms work was very limited. Specifically, during a presentation my internship mentor gave in front of other quantum computing enthusiasts, I remember struggling to understand what even half of the symbols on the screen meant. Since this point, I have been working on trying my best to understand the intricacies under the hood of quantum computing and specifically the algorithms that come with it that could change the way we solve problems.

1.1 Shor's Algorithm and Encryption

When I was researching topics for my IA, I knew that I wanted to write a paper about about a quantum algorithm. Eventually my research allowed me to stumble onto Shor's Algorithm. An algorithm which if implemented could change the security status of the Internet as we know it. On the surface, Shor's Algorithm seems to serve a very basic, if not seemingly useless function. Given a semi-prime input number I, the algorithm returns the two prime numbers, a and b which when multiplied together produce I. However, in practice this algorithm has the ability to undermine our current encryption systems that allow us to maintain privacy and control of the data we send over the Internet.

To understand why, it is necessary to have a basic understanding of the RSA encryption system that is primarily used when sending sensitive information over the Internet. RSA works through generating both a private and public key to secure your information. Both of these keys consist of two numbers, one of which is created by multiplying two large prime numbers together. [1] The public key is then used to run an encryption algorithm to convert your recognizable data into data that is indecipherable. Conversely, if one wants to decipher the data back into its recognizable form, they need the private key which then can be used to apply the decryption algorithm. The strength of RSA comes from the fact that it is computationally easy to multiply two very large prime numbers together to create both keys, but it is very hard to factorize that same prime number back into its prime components. [1] RSA keys are generally around 2048 bits long making any classical computing factoring method rendered useless given the time it would take to factor a number that large. If one could factor this number though, they could run the decryption algorithm on a set of data and gain access to sensitive information.

While classical computing fails to accomplish this, Shor's Algorithm comes in. Utilizing the power of quantum computing principles, Shor's Algorithm massively speeds up the process of factoring extremely large semi-prime numbers (numbers made from multiplying two prime numbers) through the use of a classical Euclidean Algorithm, and a Quantum Fourier Transform. I decided to explore more on both of these topics and investigate how they are used to power Shor's Algorithm and how they could be used to break the security of the Internet. The aim of this investigation is to gain understanding of the mathematical concepts that provide basis for Shor's Algorithm.

2 The Very Basics of Quantum Computing

2.1 The Mathematics Behind Quantum Computers

Quantum computers operate through qubits, particles that can assume states of zero, one, or even both (known as superposition). [2] Qubits can really be anything [2] but in this case, we'll assume that we're talking about atoms being used as qubits. Qubits can go through similar operations or logic gates as classical bits on a computer can (think of AND, OR, XOR gates). However, these logic gates differ for quantum computers as they work by applying a rotation to the atom to change their state. There are two types of logic gates in the quantum world, single-qubit and two-qubit. Single-qubit gates can flip the state of a qubit between 0, 1, or superposition. Two-qubit gates are created to allow qubits to interact with each other, otherwise known as quantum entanglement. Quantum entanglement is a phenomenon that occurs when two particles are tethered together and one particle cannot be fully defined without taking into consideration the other. [3] This feature of qubits allows quantum computation to exponentially increase in terms of processing power when more qubits are added as they can link and perform operations together through interactions. Combined with the power of superposition which essentially allows qubits to view multiple possibilities of a calculation at the same time gives an amazing performance boost over classical computers and allows quantum computers to handle such powerful algorithms and tackle complex problems. [4]

While quantum entanglement and superposition are out of scope for this paper, I mentioned their existence because the play a huge part in why Shor's Algorithm is so powerful. The mathematical concepts we will explore later use these fundamental quantum computing capabilities to run much faster than their classical counterparts.

2.2 Expressing Qubits Mathematically

Qubits can be represented as complex 2-dimensional vectors where α and β represent the x and y component respectively of the vector. Both α and β can be real or complex numbers, but the absolute magnitude of the vector must always be one. The value of α represents the probability that a qubit will be measured as a state of zero while the value of β represents the probability that a qubit will be measured as a state of one.

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

$$|\alpha^2| + |\beta^2| = 1$$

[5]

Vector Notation for a Qubit

For example, the following vectors represent qubit vectors for values of 0 and 1 which are known as the computational basis states for a single qubit.

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

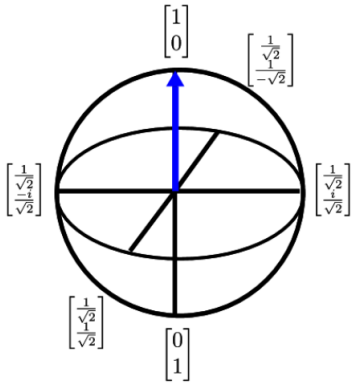
Vector Notation for Qubit State 0

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

[5]

Vector Notation for Qubit State 1

Plotting these 2-dimensional vectors on a 3D sphere representation known as the Bloch sphere allows us to visualize single-qubit states.



[5]

The x, y, and z axes all correlate to different aspects about the state of the qubit. [6] The z-axis corresponds to the probability that a qubit is measured as either a 0 or 1. The y-axis corresponds to the imaginary part of the state vector. The x-axis corresponds to the real part of the state vector.

You may be wondering how we plot the qubit vector on a Bloch Sphere. To do this, we need to rewrite our qubit into the form:

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

[7]

We can express any qubit in this form by using the $|0\rangle$ and $|1\rangle$ states. θ in this case represents the angle between the z and y axis while ϕ represents the angle between the x and y axis. With these two angle measurements and values for our state vectors, we can plot any qubit on the Bloch Sphere and visualize how logic gates act on qubits.

2.3 Quantum Logic Gates

Logic gates in quantum are defined by matrices. By applying these matrices to qubits, you rotate the qubit state vector leading to a change in state. [5] This essentially changes the probability that a qubit state is measured as a zero or a one, but this is not always the case as seen later. One of the most important logic gates in the quantum world is the Hadamard gate. The Hadamard gate puts a qubit into superposition where it can assume the value of both 0 and 1 as the probability that the measured state of a qubit as 0 or 1 is 0.5. [8] In the Bloch Sphere, this would lead to a rotation of $\frac{\pi}{2}$ radians on the y-axis and a rotation of π radians in the x-axis. The matrix definition of the Hadamard gate is as follows:

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

[5]

Another important gate to know for Shor's Algorithm is the S-gate which is a $\frac{\pi}{2}$ radian rotation about the z-axis. The matrix representation for this gate is as follows:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

[9]

Note that because we are only rotating about the z-axis, the probability that a qubit is measured as a 0 or 1 before and after going through an S-gate is actually the same. This is because movement up or down the z-axis changes the probability of being measured as a zero or one but since rotation about the z-axis keeps the z-position the same, we actually are not changing the probability at all! However, we are changing our values on the x and y-axis and this is known as a phase shift in quantum computing terms. [8]

3 The Components of Shor's Algorithm

3.1 Euclidean Algorithm

Before starting with the Quantum Fourier Transform process, Shor's Algorithm makes use of the classical Euclidean Algorithm. Given two integers, a and b, the Euclidean Algorithm seeks to make the following equation to solve for the greatest common divisor of both a and b:

$$a = q_0 b + r_0$$

[6]

where q_0 is the number of times that the number b fits into a with integer division, and r_0 is the remainder from that division. The algorithm then repeats this process but instead of using a and b, it uses b and then r_0, and then uses the subsequent remainders until the last remainder is zero leading to the general formula:

$$r_n = q_{n+2}r_{n+1} + 0$$

[6]

The answer to the problem then is r_{n+1}. [6] This algorithm is efficient for small values of a and b but it is less efficient for larger values hence leading to the need of quantum computing algorithms which will be explored later.

3.1.1 Example

Take for example, the numbers 100 and 28. 28 goes into 100 three times with a remainder of 16.

$$100 = 3(28) + 16$$

Now, using b and the remainder, we continue to form the next equation. The new q value would be 1 as 16 fits once into 28 with a remainder of 12.

$$28 = 1(16) + 12$$

After this equation, we use the subsequent remainders to form equations until the new remainder is zero.

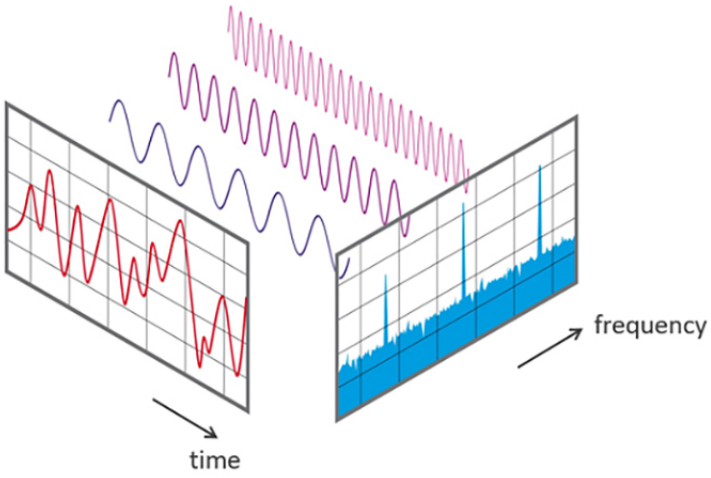
$$16 = 1(12) + 4$$

$$12 = 3(4) + 0$$

Now that the remainder is 0, we see that the greatest common factor between 28 and 100 is 4.

3.2 The Fourier Transform

Fourier Transforms allow us to take a convoluted waveform and decompose it into individual sinusoidal functions of different frequencies. It essentially takes a function that takes a time input (period based function) and turns it into a function of frequency input. [10] The image below does a great job at showing what the Fourier Transform does. You can see the convoluted waveform in terms of time below and how the Fourier Transform is able to map out the different sine waves of varying frequencies that when summed up give us the original waveform.



[11]

The question then is: why does Shor's Algorithm make use of a Fourier Transform? To explain, it is necessary to explain the concept of modular exponentiation, a critical field in cybersecurity and computer science. Take for example, two numbers a and I. If b is the remainder from dividing a by I, then we can write the following:

$$a \equiv b \pmod{I}$$

[6]

where mod is an operator that gives the remainder after a division and the $\equiv$ symbol stands for equivalence or congruence. Congruence simply means that $a \pmod{I}$ is equal to $b \pmod{I}$. As it turns out, when we raise a to increasing powers, the value of b will repeat at some point.

To give an example, let's assume a is 5, and I is 11. Let's write out the modular exponentiation.

$5^0 \equiv 1 \mod 11$	$5^5 \equiv 1 \mod 11$
$5^1 \equiv 5 \mod 11$	$5^6 \equiv 5 \mod 11$
$5^2 \equiv 3 \mod 11$	$5^7 \equiv 3 \mod 11$
$5^3 \equiv 4 \mod 11$	$5^8 \equiv 4 \mod 11$
$5^4 \equiv 9 \mod 11$	$5^9 \equiv 9 \mod 11$

As we can see, there is a pattern with the way the b values change as a is raised to sequential exponents. The sequence in this case is $1, 5, 3, 4, 9$ and it repeats for every increase of 5 in the exponent. We call this value the period of the modular exponentiation function. When we write out this modular exponentiation function, we can actually show why a Fourier Transform is necessary.

$$f(x) = a^x \mod I$$

$$a^x \mod I = a^{x+r} \mod I$$

where r is the period of the function. Since this is true, then we also know that

$$a^r = 1 \mod I$$

or more simply

$$a^r = 1$$

We can then move the 1 over to the other side and factor as the expression turns into a difference of squares.

$$a^r - 1 = (a^{\frac{r}{2}} + 1)(a^{\frac{r}{2}} - 1) = 0 \mod I$$

[6]

This expression essentially means that $a^{\frac{r}{2}} \pm 1$ could hold one or both of the prime factors that make up I if we run the Euclidean Algorithm on both of them with I. Even if they are not the prime factors, they provide great next guesses for the value of a that we choose allowing Shor's Algorithm to follow a pathway for eventually getting the right prime factors. The reasoning for this delves into number theory which is outside the scope of this paper but it remains a crucial part of the amazing speed and power of this algorithm.

Now this is all interesting but it still begs the question, why do we need a Fourier Transform to accomplish Shor's Algorithm. It all goes back to the fact that we are trying to find the period of the modular exponentiation function. We use this period to find possible factors or at least give us our next values to continue the algorithm. Fourier Transforms, specifically the inverse Fourier Transform are perfect for relating periodic time functions and frequency functions and we want to find the period of this modular exponentiation function so that we can get our prime factors.

3.2.1 The Discrete Fourier Transform

Before learning about the Quantum Fourier Transform, it is necessary to know about the Discrete Fourier Transform. The Discrete Fourier Transform takes an input vector that is filled with sampled points from the convoluted waveform graph. We do this because it is computationally easier to take a set of sampled points, as well as more realistic because the data we are dealing with is finite and not infinite. [12] It is also incredibly important to understand that the Fourier Transform essentially applies a matrix operation to our input vector to transform it. This transformation is commonly called a change of basis. [14] If we represent a convoluted waveform with a large vector of sampled points, as such:

$$\hat{f}(x) = \begin{pmatrix} . \\ . \\ . \\ f(0) \\ f(0.01) \\ f(0.02 \\ f(0.03) \\ . \\ . \\ . \end{pmatrix}$$

[12]

We can then create a new vector that instead of directly taking in x-values, takes the value n referring to the index of a number in the vector $\hat{f}(x)$ starting from 0 up to $N-1$, where N is the amount of sampled points, usually called cardinality in the context of computer science.

$$\hat{f}[n] \begin{pmatrix} \hat{f}[0] \\ \hat{f}[1] \\ \hat{f}[2] \\ \hat{f}[3] \\ \hat{f}[...] \\ \hat{f}[N-1] \end{pmatrix}$$

[12]

Then, we can apply the Discrete Fourier Transform formula as such:

$$\hat{f_k} = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} \hat{f}[n] e^{\frac{-2\pi i n k}{N}}$$

[12]

Where k represents the iterator for the summation, i is the imaginary unit, and N is the length of the original matrix. The variable k refers to the kth element of the new matrix, but it also refers to a certain frequency "bin". [13] To simplify the full expression, we will let

$$\omega = e^{\frac{2\pi i}{N}}$$

[14]

We do this simplification when we show the matrix representation of the Discrete Fourier Transform because it shows the pattern that exists in that matrix when we write it out.

3.2.2 Showing That ω Is the Nth Root of 1

We notice that ω actually represents the principal Nth root of 1. To show this, let's take N to equal 2. This would lead to the following equation with ω:

$$\omega = e^{\frac{2\pi i}{2}} = e^{\pi i} = cos(\pi) + isin(\pi) = -1$$

To show that this is the principal second root of 1, we can represent 1 as a complex number as such:

$$1 = 1 + 0i$$

We can then convert this Cartesian complex number into one of the form $re^{i\theta}$ by calculating the modulus (radius) and argument (angle that the complex vector makes with the x-axis) with the following equations.

Assume that:

$$z = a + bi$$

then:

$$r = \sqrt{(a)^2 + (b)^2}$$

$$\theta = \arctan\frac{b}{a}$$

Now with $z = 1 + 0i$, we know that

$$r = \sqrt{(1)^2 + (0)^2} = 1$$

$$\theta = \arctan\frac{0}{1} = 2\pi$$

We can now write 1 as:

$$1e^{2\pi i}$$

To solve for the expression of the Nth root of one, we get the following equations:

$$(re^{i\theta})^N = 1e^{2\pi i}$$

$$r^N = 1$$

$$e^{N\theta i} = e^{2\pi i}$$

$$\theta = \frac{2\pi}{N}$$

It is important to note that when taking the Nth root of a complex number, there exist N solutions separated by the angle $\frac{2\pi}{N}$. So when $N = 2$, there are technically two complex answers we could attain. However, the Discrete Fourier Transform only looks at the principal Nth root or the first one we attain. Therefore, we do not need to look further than the first Nth root that we find.

3.2.3 The DFT Matrix

As stated before, the Discrete Fourier Transform essentially performs a matrix operation on a vector of values sampled from a curve. This matrix, called the DFT Matrix, follows the pattern:

$$\frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega & \omega^2 & ... & \omega^N \\ 1 & \omega^2 & \omega^4 & ... & \omega^{2N} \\ . & . & . & ... & . \\ 1 & \omega^N & \omega^{2N} & ... & \omega^{N^2} \end{bmatrix}$$

[14]

Let's take for example N to equal 2 again. From the previous section, we know that $\omega = -1$ when N = 2. We can then form a NxN, or 2x2 matrix in this case using this value

$$\frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 \\ 1 & \omega \end{bmatrix} \rightarrow DFT_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

This should look familiar as this is actually the representation of the Hadamard Gate in the Bloch sphere!

3.2.4 The Quantum Fourier Transform

As it turns out, the Quantum Fourier Transform (QFT) has an identical definition as the DFT except it substitutes qubit vectors instead of the functions we used previously. Also this definition of the QFT technically relates to the inverse Fourier Transform [14] meaning that we are actually going from the frequency space to the period space as shown in the following transformation:

$$\sum_x a(x)|x\rangle \ \rightarrow \sum_x A(x)|x\rangle$$

[15]

Another difference from the DFT to the QFT is that instead of referring to N as the cardinality or length of the vector that we are transforming, we substitute it with 2^n. This is because the Quantum Fourier Transform works faster and more efficiently with values of N that can be represented as some 2^n where n is in the set of integers. [16] This is primarily because

of the symmetry explored in the last section with the DFT matrix as having a cardinality of some 2^n nicely cleans up the calculations for us. That being said, the following definition represents the Quantum Fourier Transform when $|k\rangle$ and $|j\rangle$ are basis states:

$$|k\rangle \rightarrow \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} \omega^{jk} |j\rangle$$

[14]

Similar to the DFT, when we take $N = 2$ with the QFT, we get the Hadamard Gate back. If we continue to grow N with increasing powers of 2, we end up attaining symmetry in the subsequent matrices as such.

$$QFT_N = \begin{bmatrix} QFT_{N/2} & A_{N/2} \cdot QFT_{N/2} \\ QFT_{N/2} & -A_{N/2} \cdot QFT_{N/2} \end{bmatrix}$$

[14]

The $A_{n/2}$ matrix is defined as a set of rotations around the Z-axis. To give a better example, let's write out the QFT matrix when N = 4, or better written as when $2^n = 4$. We can calculate our ω value using the formula given in 3.3.1 and then fill in the matrix with the DFT Matrix pattern in 3.3.3.

$$\omega = e^{\frac{2\pi i}{4}} = cos(\frac{\pi}{2}) + isin(\frac{\pi}{2}) = i$$

$$QFT_4 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

While this matrix may not look special right now, if we switch columns 2 and 3 with each other, we end up finding a pattern. We'll call this modified QFT matrix $\overline{QFT}$ such that

$$\overline{QFT} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & i & -i \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -i & i \end{bmatrix}$$

[14]

If we look at the matrix as 4 separate matrices where the top left 4 numbers are one matrix, the bottom left 4 numbers are another matrix, the top right 4 numbers are another matrix, and the bottom right 4 numbers are the last matrix–we can make some pretty neat simplifications. The top left and bottom left matrices are both just the matrix representations of the Hadamard gate! While the top right and bottom right matrices may not seem to match any gate we know of so far, if we recall multiplying matrices, then we can actually make the correlation that the top right matrix is the S-gate multiplied with

the Hadamard gate and the bottom right matrix is the same thing but negative! To show more clearly, let's do the matrix multiplication and then assign our four different matrices in our modified QFT matrix.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

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

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

$$\overline{QFT} = \frac{1}{\sqrt{2}} \begin{bmatrix} H & SH \\ H & -SH \end{bmatrix}$$

[14]

3.3 Putting It All Together

When given an input number I, the first step of Shor's Algorithm includes making a guess of a number a such that

$$1 < a < I$$

. The Euclidean Algorithm is then run to confirm that both numbers are coprime, meaning that GCD(a, N) is equal to 1.

We then create a quantum state. The number of qubits needed to factor the integer N is determined by squaring the number N and finding the nearest 2^n where 2^n is greater than or equal to N^2 and less than or equal to $2N^2$. [6] Then, the number of qubits needed to make this calculation is (n-1). For example, take N to be 200,000.

$$(200000)^2 \leq 2^{37} < 2(200000)^2$$

37 is the closest power of 2 that when calculated is bigger than or equal to $(200000)^2$ and less than $2(200000)^2$. The number of qubits then needed to factor 200,000 would be 37 - 1 or 36 qubits.

The formula for actually creating a quantum state is as follows:

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle |f(x)\rangle$$

Where $f(x)$ is our modular exponentiation function:

$$f(x) = a^x \mod I$$

Now that our qubits are aligned in the right state and we have essentially set up our problem, we can apply our Quantum

Fourier Transform with each of our qubits to find the period of the modular exponentiation function, r. The way this works is that the quantum computer will have its state measured after running the Quantum Fourier Transform. This is very likely to return an integer value that is a multiple of $\frac{2^n}{r}$. [6] From there, classical methods are used to extract the period from the final measured state of the quantum computer.

Once we have found the period, we check to see if it is even or odd. If we get an odd period, then we restart the process and choose a new guess for a. [6] If our period is even however, we check if $a^{\frac{r}{2}\pm 1}$ are inputs for the Euclidean Algorithm that will give us our prime factors. [17] If we run the Euclidean Algorithm with both the factors and the input number and we receive a result other than one, then those are the prime factors and we are done!

3.3.1 Example

Let's try an example of Shor's Algorithm to really see how it works in action. We will start with the following conditions: our input number I will be equal to 21. Our initial random guess, a will be 13. Upon observation, we can tell that our result should give us the prime factors of 3 and 7. So to start with, we are going to run the Euclidean Algorithm and run our initial test to see if 13 is already a prime factor of 21.

13 goes into 21 once with a remainder of 8 so our first equation will look like

$$21 = 1(13) + 8$$

Then, following the pattern of the Euclidean Algorithm (check 3.2 if you need a refresher), we can set up the rest of our equations.

$$13 = 1(8) + 5$$

$$8 = 1(5) + 3$$

$$5 = 1(3) + 2$$

$$3 = 1(2) + 1$$

$$2 = 2(1) + 0$$

Now that our remainder is zero, we see that our result from the Euclidean Algorithm is that the greatest common factor between 13 and 21 is 1 which is correct. However, that does mean that we need to continue on with our Quantum Fourier Transform to continue to search for the prime factors. Before starting with the Quantum Fourier Transform though, let's write out the modular exponentiation with our specific a value and see if we can find the period of our function so we can

confirm it with our QFT.

$$13^0 \equiv 1 \mod 21$$

$$13^1 \equiv 13 \mod 21$$

$$13^2 \equiv 1 \mod 21$$

$$13^3 \equiv 13 \mod 21$$

Well that was easy! It looks like our period, r, is 2 as our numbers start to repeat every increase of 2 on our exponent. That means that our best guesses for the Euclidean Algorithm are as follows:

$$13^{\frac{2}{2}} + 1 = 14$$

$$13^{\frac{2}{2}} - 1 = 12$$

Now let's try our Euclidean Algorithm with both of these values.

With 14 and 21:

$$21 = 1(14) + 7$$

$$14 = 2(7) + 0$$

So our greatest common factor is 7 between 14 and 21 which checks out!

With 12 and 21:

$$21 = 1(12) + 9$$

$$12 = 1(9) + 3$$

$$9 = 3(3) + 0$$

So our greatest common factor is 3 between 14 and 21.

Since both of our results from the Euclidean Algorithm are not one, we are done with the algorithm and we have gotten that our prime factors are 3 and 7, just as we predicted!

Now let's try this same method using the Quantum Fourier Transform! We need to set up our quantum state first as such so we need to calculate how many qubits we need for this operation.

$$(21)^2 \leq 2^9 < 2(21)^2$$

So this means that we need 9 - 1 or 8 qubits to perform this operation. Now we can follow the quantum state formula:

$$\frac{1}{\sqrt{2^8}} \sum_{x=0}^{2^n-1} |x\rangle|f(x)\rangle$$

$$f(x) = 13^x \mod 21$$

Even before running the QFT, we can see that the period is 2 when we set up the quantum state as we see the pattern 1, 13, 1, 13 emerge when we calculate $f(x)$ at increasing x-values.

Now let's construct our modified QFT matrix to see the matrix operation that we are applying to our qubits. Remember that we're constructing our modified QFT matrix because of the symmetry that it contains. Because this matrix would be a 2^8 by 2^8 matrix, we will write out the recursive formula rather than the full matrix.

$$\overline{QFT}_{256} = \begin{bmatrix} QFT_{128} & A_{128} \cdot QFT_{128} \\ QFT_{128} & -A_{128} \cdot QFT_{128} \end{bmatrix}$$

Remember that our QFT follows the formula:

$$\frac{1}{2^n} \sum_{k=0}^{2^n-1} \omega^{jk} |j\rangle$$

Now this total summation would be quite difficult to show all in this paper but we can calculate our ω value which represents the 2^n root of one and plays an important role in creating the pattern we find in our modified QFT matrix aiding in the speed of Shor's Algorithm:

$$\omega = e^{\frac{2\pi i}{2^8}} = e^{\frac{\pi i}{128}} = cos(\frac{\pi}{128}) + isin(\frac{\pi}{128}) = 0.9997 + 0.0245i$$

Now we can use this ω value to quickly compute our QFT matrix and use the Quantum Fourier Transform to find our period r by measuring the final quantum state. The process from here is the same as before where we use the Euclidean Algorithm with $a^{\frac{r}{2}} \pm 1$ to get our possible prime factors. If we end up getting a result of one for both of them, we repeat the process from step 1 with a new randomly generated integer between one and our input number.

3.3.2 Showing the Classical Approach for Solving Our Example

While I do not have the resources or skill set to properly implement the Quantum Fourier Transform, I was able to create a version of Shor's Algorithm that can handle relatively small semi-prime numbers that uses a Discrete Fourier Transform instead. I programmed this in Python using the libraries math, random, and cmath. The cmath library is especially important because it allows the ability to create complex variables which is necessary for implementing a Discrete Fourier Transform.

The steps the algorithm takes are essentially the same as the steps shown above. First we make a guess and run the

Euclidean Algorithm to see if the greatest common factor between our guess and the number we want to factorize is one or not. Then if it is one, we compute values of our modular exponentiation function with the guess that we made. I opted to compute 10 values to keep it simple. We take these values and plug them into the DFT which returns a list of complex values. The part where my code diverges from the actual quantum algorithm is when it comes to determining the period. In a quantum setting, the final state is measured and then the period can be deduced, but since I am not using quantum computing, I opted to go with magnitude analysis. Essentially, the function then finds the complex value with the largest magnitude and sets this equal to the period if it is even. This is because the complex value with the largest magnitude essentially tells us the "main" period in our modular exponentiation function. From there, it then uses the difference of two squares formula to find numbers that could be used to directly find the prime factors or serve as good next guesses to run the algorithm next. The algorithm recursively runs until the Euclidean Algorithm outputs a result that is not 1. This result is of course one of the prime factors, and the other can be found by just dividing the input number by the first prime factor.

Some important aspects of this program are the implementation of the DFT and the magnitude analysis after to determine the period of the modular exponentiation function.

The image below shows the code for the DFT.

```
def DFT(N : int, func_values : list[:int]):

    dft_values = []

    #summing for all k frequencies
    for k in range(0, 4):
        grand_sum = 0
        #for each k frequency, we have to take the sum from 0 to N and take the function value * the complex part (look at formula for DFT in section 3.2.1)
        for n in range(0, N):

            real_part_complex = math.cos((-2 * math.pi * n * k) / N)
            im_part_complex = math.sin((-2 * math.pi * n * k) / N)

            complex_num = complex(real_part_complex, im_part_complex)
            grand_sum += func_values[n] * complex_num

        dft_values.append((1 / math.sqrt(N)) * grand_sum)

    return dft_values
```

This code essentially just implements the DFT from formula (section 3.2.1) using a for loop as a tool for completing the summation part of the DFT. I limited the amount of values that the iterator k, which represents a certain "frequency" (remember that we're technically finding the period) bin as to not overwhelm my hardware.

After running the DFT, my program performs magnitude analysis. Essentially, it just assumed that the period r of the modular exponentiation function was the complex value computed by the DFT with the highest magnitude.

```
def calculateMagnitudes(the_values):

    #this just creates a list of the magnitudes of each of the complex numbers we get from running the DFT function
    new_values = []
    for value in the_values:
        new_values.append(math.sqrt(math.pow(value.real, 2) + math.pow(value.imag, 2)))
    return new_values

def getLargestMagnitude(magnitude_values):

    #just a simple function for getting our largest magnitude (we're going to use this to assume our period)
    largest_value = 1
    for value in range(2, len(magnitude_values)):
        if magnitude_values[value] > magnitude_values[largest_value]: largest_value = value
    return largest_value
```

This code shows the calculation of magnitudes by the formula shown in section 2.2 and the function for finding the largest magnitude out of all the complex values that the DFT computed.

All of the code I wrote can be found in Appendix A. Running the program with an initial guess of 13 for the semi-prime number 21 returns either 3 or 7. From there, all one has to do is divide the semi-prime number by the result to get the other prime number.

```
PS C:\Users\cleme> & C:/Users/cleme/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/cleme/Documents/Math IA/dftformathia.py"
3
Result: 3
```

This version of Shor's Algorithm (dubbed "Baby" Shor's Algorithm) can handle factoring semi-prime numbers with 6-7 digits. The following image shows the result from the program for the prime factorization of 107501, a semi-prime number made from the multiplication of 193 and 557.

```
PS C:\Users\cleme> & C:/Users/cleme/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/cleme/Documents/Math IA/dftformathia.py"
193
Result: 193
```

However, it's important to note that this classical method fails when the semi-prime is large enough. For example, when running "Baby" Shor's Algorithm with the semi-prime number 10,002,200,057, the program is unable to complete as it ends up recursively calling the algorithm so many times to the point where the compiler stops it completely.

```
RecursionError: maximum recursion depth exceeded while calling a Python object
PS C:\Users\cleme>
```

In fact, this error shows exactly why quantum computing is needed to carry out Shor's Algorithm. The computational advantage quantum computing carries out by using the laws of quantum physics far surpasses the capabilities of classical computing hardware.

4 Conclusion

In conclusion, Shor's Algorithm is an extremely powerful algorithm for calculating the prime factors of a very large semi-prime number. While we do not have the quantum computing power as of now to run it effectively for very large semi-prime numbers, we will eventually have this power in the future once we create quantum computers with higher numbers of qubits. The industry leaders for this technology at the moment are IBM, Google, Amazon, and Microsoft with IBM having the largest quantum computer with 1,121 qubits. This algorithm really questions our security and encryption systems on the Internet because as we advance technologically in terms of computing power, we will also need to advance our own security standards. The idea of using quantum computing to encrypt our data may very well become our reality in the future because it will become necessary in order to protect our data. While the RSA encryption system is a very trusted system, it will eventually need replacement as quantum computers will be able to quickly break through it.

It is interesting to note the dynamic between our classical mathematical systems and the quantum versions of them. Take for example the relationship between the Discrete Fourier Transform and the Quantum Fourier Transform. They are almost identical except for the fact that the Quantum Fourier Transform operates on qubit vectors rather than a function vector as the Discrete Fourier Transform does. Our framing of our problem also changes as we have to create qubit states in order to "create" our problem and then use Shor's Algorithm to actually solve it. This is similar to classical mathematics where we first have to define our problem with functions, graphs, tables, etc and then use our problem solving techniques to solve them except we are now using quantum circuits and quantum concepts in order to define them.

What I found the most interesting throughout my research was the use of the classical Euclidean Algorithm. The Euclidean Algorithm is a classical algorithm and the use of it for Shor's Algorithm shows that not everything can be quantum computed for maximum efficiency. For small values and parts of our quantum algorithm, we must use a classical algorithm as it actually becomes more efficient compared to establishing qubit states and then running a quantum circuit. This is similar to the algorithm QAOA (Quantum Approximate Optimization Algorithm) that I got to work with during my internship at TerraNexum as there is a classical optimizer and quantum optimizer. The merge between these two different computing worlds is quite unique because it combines the best of both worlds to create the most efficient algorithms.

Overall, Shor's Algorithm remains an incredibly powerful algorithm that will likely be used in the future to change the way we secure and encrypt our data. Researching this algorithm made me more aware of the possibilities that quantum computing opens us up to along with the possible dangers that it may present in our "outdated" classical systems. It is important that we all understand the risks and benefits of improved technology in the future so that we do not run the risk of breaking what we already have.

References

[1] E. Consulting, sep 2020. [Online]. Available: https://www.encryptionconsulting.com/education-center/what-is-rsa/

[2] I. for Quantum Computing. [Online]. Available: https://uwaterloo.ca/institute-for-quantum-computing/quantum-101/quantum-information-science-and-technology/what-qubit

[3] C. S. Exchange. [Online]. Available: http://scienceexchange.caltech.edu/topics/quantum-science-explained/entanglement

[4] IBM. [Online]. Available: https://www.ibm.com/topics/quantum-computing

[5] SoniaLopezBravo, "The qubit in quantum computing - azure quantum," Jul. 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/quantum/concepts-the-qubit

[6] S. Clarke, *Quantum Computing: A Mathematical Analysis of Shor's Algorithm*, 2019. [Online]. Available: https://digitalcommons.sacredheart.edu/cgi/viewcontent.cgi?article=1398&context=acadfest

[7] Taso. [Online]. Available: https://akyrillidis.github.io/notes/quant_post_7

[8] R. Flint, "The quantum gates everyone should know in quantum computing," dec 2022. [Online]. Available: https://quantumzeitgeist.com/the-quantum-gates-everyone-should-know-in-quantum-computing/

[9] QuTech, "S gate." [Online]. Available: https://www.quantum-inspire.com/kbase/s-gate/

[10] 3Blue1Brown. (2018). [Online]. Available: https://www.youtube.com/watch?v=spUNpyF58BY

[11] M. Brock, Jun. 2019. [Online]. Available: https://insightincmiami.org/data-visualization-using-the-fourier-transform/

[12] jnez71, "Can someone clearly explain the discrete fourier transform (dft)?" Jun. 2018. [Online]. Available: https://math.stackexchange.com/q/2809394

[13] Iain. [Online]. Available: https://www.youtube.com/watch?v=8V6Hi-kP9EE

[14] R. LaRose, *Shor's Algorithm Part 2: The Quantum Fourier Transform.* none. [Online]. Available: https://www.ryanlarose.com/uploads/1/1/5/8/115879647/shor2-qft.pdf

[15] F. Xi Lin, *Shor's Algorithm and the Quantum Fourier Transform*, California. [Online]. Available: https://www.math.mcgill.ca/darmon/courses/12-13/nt/projects/Fangxi-Lin.pdf

[16] R. O'Donnell and S. Mohanty, Oct. 2015. [Online]. Available: https://www.cs.cmu.edu/~odonnell/quantum15/lecture09.pdf

[17] QuTube. [Online]. Available: https://www.qutube.nl/quantum-algorithms/shors-algorithm

Appendices

A Code for Classical "Baby" Shor's Algorithm

A.1 Modular Exponentiation Function

```python
def exponentiation(a : int, I : int, x : int):
    #f(x) = a^x mod I where I is our semi-prime to factor
    return math.pow(a, x) % I


def computeValues(a, I):

    #compute an arbitrary list of function values, i just do 10 (easy on the hardware)
    function_values = []

    for x in range(0, 10):

        function_values.append(exponentiation(a, I, x))


    return function_values
```

A.2 Euclidean Algorithm

```python
def euclideanAlgo(a, b):

    #look at section 3.1 for the process on this one, not the most concise function i've ever written but it does the job
    remainder = a % b
    if remainder == 0: return b
    q = b
    w = remainder

    while remainder != 0:

        remainder = q % w
        if remainder == 0: break
        q = w
        w = remainder

    return w
```

A.3 Discrete Fourier Transform Implementation

```python
def DFT(N : int, func_values : list[:int]):

    dft_values = []

    #summing for all k frequencies
    for k in range(0, 4):
        grand_sum = 0
        #for each k frequency, we have to take the sum from 0 to N and take the function value * the complex part (look at formula for DFT in section 3.2.1)
        for n in range(0, N):

            real_part_complex = math.cos((-2 * math.pi * n * k) / N)
            im_part_complex = math.sin((-2 * math.pi * n * k) / N)

            complex_num = complex(real_part_complex, im_part_complex)
            grand_sum += func_values[n] * complex_num

        dft_values.append((1 / math.sqrt(N)) * grand_sum)

    return dft_values
```

A.4 Magnitude Analysis

```
def calculateMagnitudes(the_values):

    #this just creates a list of the magnitudes of each of the complex numbers we get from running the DFT function
    new_values = []
    for value in the_values:
        new_values.append(math.sqrt(math.pow(value.real, 2) + math.pow(value.imag, 2)))
    return new_values

def getLargestMagnitude(magnitude_values):

    #just a simple function for getting our largest magnitude (we're going to use this to assume our period)
    largest_value = 1
    for value in range(2, len(magnitude_values)):
        if magnitude_values[value] > magnitude_values[largest_value]: largest_value = value
    return largest_value
```

A.5 "Baby" Shor's Algorithm Function

```
def runBabyShor(initial_guess, semi_prime_to_factor):

    #run the Euclidean Algorithm first, if the only factor in common is one, go through with the rest of the process
    if(euclideanAlgo(semi_prime_to_factor, initial_guess) == 1):

        #sample some values of our modular exponentiation graph, f(x) = a^x mod I where a is our initial guess, I is our prime that want to factor
        #then run the DFT function and get back a list of complex numbers. then create a list of the magnitudes of those complex numbers
        #the period will be the index with the largest magnitude
        func_values = computeValues(initial_guess, semi_prime_to_factor)
        dft_values = DFT(len(func_values), func_values)
        magnitude_values = calculateMagnitudes(dft_values)

        period = getLargestMagnitude(magnitude_values)
        #recursion seemed like a good idea for this algo.
        #if we don't have a period of a multiple of 2, we'll end up with weird decimal values so just run the algo again with a different guess
        if period % 2 != 0:
            return runBabyShor(random.randrange(2, semi_prime_to_factor), semi_prime_to_factor)
        else:
            #otherwise, if we have our next best guess, run this algorithm again
            prime_factor_guess = int(math.pow(initial_guess, (period/2)) - 1)
            return runBabyShor(prime_factor_guess, semi_prime_to_factor)
    #if our euclideanAlgo gives us a result other than 1, that result is one of the factors of the semi-prime!
    elif euclideanAlgo(semi_prime_to_factor, initial_guess) != 1:
        print(euclideanAlgo(semi_prime_to_factor, initial_guess))
        return int(euclideanAlgo(semi_prime_to_factor, initial_guess))
```