Shor's Algorithm

Elliott Pryor, Benjamin Bushnell, Shengnan Zhou

13 November, 2019

1 The Problem

The problem of factoring large numbers has existed for centuries. Euclid's algorithm provides a very efficient way to determine the greatest common divisor of two numbers (GCD). Say we look for $GCD(a, b)$ then a is a factor of b if the $GCD(a, b) > 1$. If we have one of the factors, it is very easy to find the other factor as we can just divide the number by its factor. However, factoring very large numbers using Euclid's algorithm is very time consuming as we have to try every single number less than the value we are factoring. We can make a few improvements to our guesses; however, classically this problem cannot be solved in polynomial time.

The RSA encryption algorithm takes advantage of this fact. First, some background information on RSA encryption. RSA is what's called an asymmetrical encryption algorithm. To best illustrate what asymmetrical encryption is, here's an example: Client A and Server B both have public encryption keys and private decryption keys. Each entitie's keys are related such that their private and public keys are linked, and data encrypted with an entitie's public key can only be easily decrypted by their own decryption key. So let's say B wants to send an encrypted message to A, that no one else is allowed to see. B uses A's public key to encrypt the message, then sends A the encrypted message. Because A kept their decryption key private, they will be the only party able to use it to decrypt the message that B sent. If a nefarious party had intercepted the message, all they would have is the public encryption key, and the encrypted message, but without the decryption key they would have no way to read the message. This is how RSA works as an asymmetrical encryption algorithm.

So, how does RSA leverage the np time complexity of factorization in this assymetrical encryption model? MORE STUFF HERE (NOTE: discuss trap door making it easy to compute ϕ?)

Though it isn't impossible to break, the time required to crack RSA encryption is an effective deterrent for malicious individuals who wish to steal data. Because of it's efficacy, RSA encryption is widely used for encrypting sensitive data which is to be sent between two parties. For example, it is used for things like emails, chat messages and web browsers. Clearly, it is very important that RSA encryption remains uncracked so that the security of sensitive data sent using these services remains intact. On the other hand, quantum computing can make things easier and faster, but even so, quantum computing can only decrypt small numbers. It would still take years to break some standard RSA encryptions at current state. Until then, we are safe.

This brings us to Shor's algorithm. Shor's algorithm was invented in 1994 that cleverly uses properties of quantum computing to quickly solve prime factorization of even large numbers. This is exactly what RSA assumes is not possible. The short description of Shor's algorithm is that we can take any random guess g, run the guess through Shor's algorithm, and get back a better guess $g^{p/2} \pm 1$. For any large number N, when we take a random guess g, $mN = g^p + 1$ for some p and m. Break this down, we get $(g^{p/2}+1)(g^{p/2}-1) = mN$. Once we have the two better guesses, we can use Euclid's algorithm to find the shared factors, thus breaks the encryption. However, the most important part of this algorithm is to find p, and it takes very very long time without quantum computing. Quantum computing can take in multiple values as superpositions, and find p fast.

For sections 2 and 3 assume we are trying to find:

$$ab = C$$

where a and b are factors of n and $a, b, C \in \mathbf{N}$.

2 Classical Computation

The classical part of this algorithm is really simple. We just have to guess a random number as our factor. Then we use Eucild's algorithm to verify that we didn't get extremely lucky and guess a factor. If we guess a factor, then we don't need to use the quantum part of the algorithm. Then we feed our guess (which we know does NOT share factors with the number in question) into the quantum algorithm.

3 The Quantum Algorithm

The quantum part of this algorithm is what turns the random number that we guessed into the actual factor. In short, it does this by finding the period of some function that we can relate to the factors. Given that that is a gross oversimplification of what happens, we will cover some of the math supporting Shor's Algorithm.

First, we must define a few relations.

Theorem 1. *Given $a, b \in \mathbf{Z}$ and a, b share no common factors. Then*

$$a^p = mb + 1$$

for some $m, p \in \mathbf{Z}$.

Theorem 2. *Given $a^x = mN + r$ for $a, x, m, N, r \in \mathbf{Z}$. Then*

$$a^{x+yp} = kN + r$$

for some $y, p, k \in \mathbf{Z}$.

We know that the guess given to the quantum portion of this algorithm, g, does not share any factors with the number we are trying to factor, C. Then by theorem 1 we can express this as $r^p = mC + 1$. We can rearrange this to $(r^{p/2} + 1)(r^{p/2} - 1) = mC$. Then we know that our factors of C are related to $(r^{p/2} + 1)$ and $(r^{p/2} - 1)$. Once we find these numbers, we can use Euler's formula to calculate the factors.

In order to find p we use theorem 2 and some properties of quantum computers. By theorem 2, we know that $g^{x+yp} = kC + r \text{ mod(C)} = r \ \forall y \in \mathbf{Z}$. So we can raise our guess to integer powers and search for when the remainder repeats. Classically, we cannot do this efficiently. However, with quantum computers we have a superposition of states that we can exploit to efficiently compute this. If we poll a random remainder value from our modulus calculation we get a superposition of the states that result in this value: g^x, g^{x+p}, g^{x+2p}... This has a period of p which we can by taking the Fourier transform of this superposition. The Fourier transform returns the period of the function. Assuming that p is even, then we are done! We have now found the factors of C. If p is odd, then $r^{p/2}$ is not an integer, and we have to restart the process with a new guess.