Abstract

Factoring large numbers is both an interesting and very important problem. It is essential to encryption of data in the RSA cryptosystem. Currently, there is no known classical method that can factor a large enough number fast enough to compromise RSA. There do exist methods that can factor numbers quickly up to a limit, like the Pollard ρ algorithm and the quadratic sieve, which we discuss here. We go over the underlying math to build up the algorithms. However, with the rise of quantum computing, breakthroughs have been made. We briefly discuss quantum computing from a more mathematics point of view than physics, before looking at Shor's algorithm which factors a number much faster than classical methods.

1 Introduction

The fundamental theoreom of arithmetic states that any positive integer n has a unique prime factorization. While this is interesting on its own, it raises the question: Is it possible to factor n and find its prime factors? The short answer is yes, and it is not very difficult. As long as the numbers are small enough. Take, for example, the number 20. It can be factored into 4×5. This is not its prime factorization, however, since 4 is not prime. The prime factorization for 20 is $2 \times 2 \times 5$. Finding this prime factorization is almost trivial. What if the number has twenty digits? Hundreds of digits? With large numbers, factoring gets difficult. It is this difficulty which makes prime factorization extremely relevant in the field of cryptography.

Cryptography is the study of methods to create secure communications between two parties [8]. Plaintext refers to a message before encryption, and ciphertext refers to the same message after encryption. The word cryptosystem refers to an encryption scheme and its corresponding decryption scheme [8]. They are usually grouped into two camps: symmetric and asymmetric cryptosystems. In a symmetric system, the key used to encrypt and decrypt information are the exact same, whereas an asymmetric system has a different key for encrypting and decrypting. Asymmetric systems are also known as public key systems. In today's world, both types of cryptosystems are used, as most public key cryptosystems are computationally intensive. Thus, symmetric systems are used to send information, and asymmetric systems are used to send small, but important pieces of information [1].

Where does prime factorization come into play? It turns out there are a number of asymmetric cryptosystems which fully rely on the difficulty of factoring a large number [6]. One of which is the RSA cryptosystem created in 1978, whose name comes from the last names of its creators: Rivest, Shamir, and Adleman. RSA is still widely used today. Understanding how RSA works and the math that drives it reveals the importance of prime factorization.

Suppose Alice and Bob want to securely communicate using RSA. How does it

work? First, they must generate a key. Alice must choose two large prime numbers p and q. Ideally, they have about the same number of digits for better security. Alice then computes $n = pq$ and $\phi(n) = (p-1)(q-1)$, where ϕ is the Euler totient function.[1] If we know the prime factorization of n, finding its totient is easy. In this case, its prime factors are precisely p and q. Being able to quickly calculate $\phi(n)$ relies on the fact that the totient function is multiplicative[2] and that the totient of primes is relatively simple. Proofs for these can be found on pages 95-97 in [8]. Now Alice has n and $\phi(n)$. She must then choose some integer e such that $1 < e < \phi(n)$ and it is coprime to $\phi(n)$. Finally, she computes $d = e^{-1} \pmod{\phi(n)}$.[3] Alice and Bob now have everything they need to send messages to each other: d is Alice's private key and the numbers n and e form their public key.

If Bob wants to send a message m, which is a number modulo n, it should be coprime to n. He simply computes $c = m^e \pmod{n}$. Then Alice can decrypt the ciphertext by computing $m = c^d \pmod{n}$. This is RSA in a nutshell, and it does indeed work [8]. There are a number of nuances, some of which are chosen plaintext and ciphertext attacks against naive RSA, padding messages to increase security, and schemes to convert between letters and numbers to be able to send actual English [8]. However, the focus of this paper is not RSA. It is how a cryptosystem like RSA may be broken.

If Eve, the eavesdropper, can factor n, then she knows p and q, meaning she can easily compute Alice's private key d. Eve can then read all of the messages she intercepts. The usual approach to factoring n is to check whether it divides 2, 3, 5, 7,... $\sqrt{n}$ which takes too long on a large n to be practical. Therefore at the end of the day, whether or not RSA is secure relies soley on the ability to factor some large number n. It is believed that RSA is secure, but no one can know for sure [8]. It could be the case that there does exist a method of factoring a number in polynomial

[1] $\phi(n)$ returns the number of positive integers less than or equal to n and coprime to n, and two numbers a, b are coprime if their only common factor is 1.

[2] A function f is multiplicative if $f(ab) = f(a)f(b)$ whenever $\gcd(a, b) = 1$.

[3] $e^{-1} \pmod{\phi)(n}$ refers to the inverse of e in the multiplicative group modulo $\phi(n)$, which is a number modulo $\phi(n)$ such that $(e)(e^{-1}) \equiv 1 \pmod{\phi)(n}$.

time and no one has come across it yet. Polynomial time in this case refers to an algorithm's time complexity. In other words, in a reasonable amount of time.

Clearly, the factoring problem is crucial to the security of RSA and other systems like Rabin-Williams [6]. If there really does exist a way to factor a number in polynomial time, it would of course be better for someone with good intentions to find it first. This is precisely why there have been efforts to solve this problem, on both classical computers and quantum computers. In fact, while the attempts on classical computers have some caveats, the attempt on quantum computers has proven successful. Putting quantum computers aside for now, some attempts at factoring on a classical computer are the Pollard ρ algorithm and the quadratic sieve.

2 Factoring on Classical Computers

As mentioned before, the usual approach to factoring some number n is by simply checking whether n is divisible by the primes between 2 and $\sqrt{n}$. This method is impractical for large numbers, but how impractical? In other words, how slow is this algorithm really? Rather than checking the primes, the algorithm will simply check all numbers. This will not make much difference in the end. Consider the following pseudocode:

Algorithm 1: Brute force factoring algorithm

```
   input: Integer n to factor
1  factors ← empty list;
2  N ← n;
3  while i <= √n do
4  |  if N  mod i ≡ 0 then
5  |  |  append i to factors;
6  |  |  N ← N/i;
7  |  else
8  |  |  i ← i + 1;
9  end
10 return factors
```

Suppose checking whether n is divisible by i is one operation. In the worst case,

checking every number up to $\sqrt{n}$ results in precisely $\sqrt{n}$ operations. Said differently, its runtime is $O(\sqrt{n})$. It shouldn't be that slow right? Unfortunately, checking whether one number divides another is costly on a computer, on the bit level. If we write n in binary, it has d digits. Then:

$$2^{d-1} \leq n < 2^d$$
$$\sqrt{2^{d-1}} \leq \sqrt{n} < \sqrt{2^d}$$
$$2^{\frac{d-1}{2}} \leq \sqrt{n} < 2^{\frac{d}{2}}.$$

Therefore, the algorithm's runtime is really $O(\sqrt{n}) = O(2^{\frac{d}{2}})$. The runtime grows exponentionally, which is not ideal. Fortunately, there are a few clever approaches that can be taken to factor n, albeit with some caveats.

2.1 Pollard ρ Algorithm

The Pollard ρ algorithm gets its name from the diagram that is sometimes used to describe it, which looks like the letter ρ [8]. The idea behind the algorithm is closely related to the birthday problem, which is described later in this section.

The goal is to factor n. The algorithm relies on the fact that if two numbers x and y are conguent modulo m, then their difference $|x-y|$ must be divisible by m. If m is indeed a factor of n, then $|x-y|$ and n share a factor, namely m. Then we compute the greatest common divisor with the Euclidean algorithm, which is quick. Thus, the Pollard ρ algorithm finds one factor of n, and finding the other is now trivial.

Definition 1. Given integers $a_1, \ldots, a_n$ that are not all zero, the greatest common divisor (gcd) of these integers is the largest positive integer that divides all of them.

2.1.1 Brief Analysis of Euclidean Algorithm

The Euclidean algorithm employs the division algorithm to quickly find the greatest common divisor of two integers a and b.

Theorem 1. (The division algorithm) Let a, b be positive integers. Then there exists integers q, r such that $a = bq + r$ with $0 \leq r < b$.

The following algorithm is recreated from [8].

Algorithm 2: Euclidean algorithm

input: Integers a and b to find GCD of

$r_0 \leftarrow a$;
$r_1 \leftarrow b$;
$i \leftarrow 1$;
repeat
  $q_i \leftarrow \lfloor \frac{r_i}{r_{i-1}} \rfloor$;
  $r_{i+1} \leftarrow r_{i-1} - q_i r_i$;
  $r_{i-1} \leftarrow r_i$;
  $r_i \leftarrow r_{i+1}$;
until $r_i = 0$;
return r_{i-1}

Its speed depends on the size of the remainder r at each iteration. If $r < \frac{b}{2}$, then the algorithm halves the problem with each iteration, making it very quick.

Suppose $r_1 > \frac{b}{2}$, then it suffices to show $r_2 \leq \frac{b}{2}$ to show the algorithm is quick.

$$r_1 > \frac{b}{2}$$
$$b = r_1 q_2 + r_2 > \frac{b}{2} q_2 + r_2.$$

Since $r_1 < b$, q_2 cannot be 0. But q_2 cannot be 2 or greater, since that implies:

$$r_1 q_1 + r_2 > b + r + 2 \geq b.$$

Which is not possible. Since q_2 is an integer, then it must be 1. Then

$$r_2 = b - r_1 q_1 < b - \frac{b}{2} = \frac{b}{2}.$$

Therefore at every other iteration of the Euclidean algorithm, the subsequent inputs

will always be less than half of the last iteration. Then the algorithm runs in $O(log(b))$, which is quite fast.

2.1.2 The Birthday Problem

This problem is discussed in [8], but will also be restated here.

Question 1. *How many people must be in a room in order for it to be more likely than not for two people to share a birthday? Excluding February 29th and assuming all birthdays are equally likely.*

It turns out only 23 people are needed. The idea is to view each person entering the room as a sequence of events, and to compute the complement, i.e. compute the probability that no one shares a birthday. When 1 person is in the room, the probability of no one sharing birthdays is 1, or $\frac{365}{365}$; it is impossible to share. This person then occupies one day out of the year for their birthday. Now person 2 walks in. If they do not share a birthday, then person 2 occupies a different day for their birthday, of which there are only 364 left after person 1. The probability of this happening is

$$\frac{365}{365} \times \frac{364}{365}.$$

When the third person walks in, the probability becomes

$$\frac{365}{365} \times \frac{364}{365} \times \frac{363}{365}.$$

Suppose k people walk in. Then the probability none of them share a birthday must be

$$\frac{365}{365} \times \frac{364}{365} \times \cdots \times \frac{366-k}{365}, \text{ for } k \geq 1.$$

We want the smallest k such that this product decreases past $\frac{1}{2}$. The first value of k

for this to happen is the smallest k such that the probability of not sharing a birthday is less than half. In other words, the smallest k such that the probability of sharing a birthday is higher than not sharing a birthday. Let the probability of not sharing a birthday be $P(N)$ and sharing be $P(S) = 1 - P(N)$.

k	1	2	...	22	23	24
$P(S)$	1	.003	...	.476	.507	.538
$P(N)$	0	.997	...	.524	.493	.462

It turns out that in general, if there were n equally likely possible birthdays, it would require $\sqrt{2n \log(2)}$ people before expecting a collision [8]. It is also acceptable to say that it would require $O(\sqrt{n})$ people. This fact is key to the Pollard ρ algorithm.

2.1.3 The Algorithm

We can treat the m possible birthdays as the elements of the group $\mathbb{Z}/m\mathbb{Z}$, i.e. the possible values for $x \pmod m$. Then, by the birthday problem, about $\sqrt{m}$ random numbers should be tested before getting two with the same value modulo m. The algorithm takes care of the random part by grabbing random numbers in an organized way. Of course, it is possible a factor is found before the expected value. The algorithm goes like this:

Algorithm 3: Pollard ρ factoring algorithm

input: Integer n to factor, and a polynomial function $f(x) : \mathbb{Z}/n\mathbb{Z} \to \mathbb{Z}/n\mathbb{Z}$

```
x, y ← 2 ; // Note:  initial values for x and y can be anything
repeat
    x ← f(x);
    y ← f(f(y));
    d ← gcd(|x − y|, n) ; // Apply Euclidean algorithm
    if d = n then
        return −1; // Algorithm fails, choose new f(x)
until d ≠ 1;
return d ; // d should be a factor of n
```

For example, let $n = 325474481$ and $f(x) = x^3 + 9236x^2 + 84x + 17265 \pmod n$. Let $x, y = 2$. Then we get,

x	y	$\gcd(\lvert x-y\rvert, n)$
54385	235323699	1
235323699	21532251	1
17564425	201009351	1
$\vdots$	$\vdots$	$\vdots$
318863288	254594522	1
862569	99792644	21863

We get that one factor of n is 21863, and the other factor is 14887. This took 60 iterations.

The polynomial is just an easy way to grab random numbers modulo n. Let us be more rigorous about why the algorithm works. Let n be the positive integer to factor. Suppose x and y are distinct numbers modulo n. While they are values modulo n, we actually want them to collide in terms of m, the unknown factor of n, so that their difference is a multiple of m. Take their difference and compute the greatest common divisor of the difference and n with the Euclidean algorithm. If the GCD is $m \neq 1$, then m is a factor of n since $|x - y| = km$ and $n = jm$.

One of the main strengths of this algorithm is that it is fairly easy to program compared to others. Which means there must be some trade offs. It works best when one of the factors p is relatively small [8], since there would be less values in $\mathbb{Z}/p\mathbb{Z}$. Also, the algorithm is random, so it could be the case that sometimes it takes much longer, or multiple polynomials have to be tested before finding a factor. It is most certainly the case that in modern RSA, the numbers are so large that this algorithm fails, otherwise RSA would have been compromised. If the factor p is small enough, it should take about $\sqrt{p}$ steps to find it [8]. The Pollard ρ algorithm is still a popular choice for factoring, but there are others out there which are slightly better, and more complicated.

2.2 Quadratic Sieve

The quadratic sieve is one of the best algorithms for factoring [8]. The general idea is take advantage of smooth numbers, factoring differences of squares, and some linear algebra. If we can find two squares whose difference is n, or even a multiple of n, then we can factor n. Where smooth numbers come into play, is to help find such squares whose difference is divisible by n.

2.2.1 Smooth Numbers

Definition 2. A number n is k-smooth if all of its prime factors are less than k.

There are a few different ways to check whether n is k-smooth. The most obvious method is by using the brute force factoring algorithm. We first factor n and keep a list of its prime factors. Then check if the largest factor is less than k. This is of course very slow as n gets large.

A much better method requires having a list of primes up to k. Luckily, there is a relatively quick way to generate a list of primes: the sieve of Eratosthenes. The idea is to go through the list of numbers up to k, "marking" the multiples of each prime as composite. The algorithm terminates when all numbers have been marked prime or composite.

Algorithm 4: Sieve of Eratosthenes

```
   input: k, the number to check up till
 1 i ← 2;
 2 P ← empty list;
 3 while i < k do
 4 |   mark i as prime;
 5 |   append i to P;
 6 |   for j = 2i to k by i do
 7 |   |   mark j as composite;
 8 |   end
 9 |   i ← next unmarked number;
10 end
11 return P
```

Now that we have a list of primes up k, we can efficiently check whether a number

n is k-smooth. One way to do this is to take n and divide by 2 as many times as possible, then 3, then 5, and so on up to k. If all primes have been used and $n \neq 1$, then it must still have some prime factor greater than k, so it is not k-smooth. If $n = 1$ before reaching k, it must be k-smooth.

This whole process of using the sieve of Eratosthenes and then testing numbers $n_1, n_2, \ldots$ for smoothness can be done all at once, as described by [7]. Run through the sieve of Eratosthenes as usual, but rather than marking a number as composite, we replace that number by its quotient with the current prime and powers of the current prime. Then, finding k-smooth numbers is the same: if there is a 1 left in that number's location, it must be k-smooth. It turns out such an algorithm to find k-smooth numbers up to n has a time complexity of $k \log \log n$, so it is rather fast. The proof for this can be found in [7]. We now have a quick way to find smooth numbers given an upper bound n and a smoothness bound k.

2.2.2 The Algorithm

First, we must first discuss how squares, and their differences, come into play. Suppose the goal is to factor 899. First, notice that $899 = 900 - 1 = 30^2 - 1^2$, so 899 can be written as a difference of squares, and differences of squares can be factored. Then,

$$899 = 30^2 - 1^2 = (30 - 1)(30 + 1) = 29 \times 31.$$

899 factors into 29 and 31. In general, when trying to factor n, if we can find two squares whose difference is n then we can factor n. In fact, even if their difference is only multiple of n, things are still okay. If $x^2 - y^2 = kn$, then the $\gcd(x - y, n)$ should be a nontrivial factor of n. Now, how can we find two squares whose difference is a multiple of n? By using smooth numbers.

First, choose a smoothness bound B. Let $\pi(B)$ denote the number of primes up to B. Now, find $\pi(B) + 1$ numbers a_i such that $a_i^2 \pmod{n}$ is B-smooth. Let $b_i = a_i^2$

(mod n). Since all the a_i^2's are squares, any product of them is still a square. Then if the product of their corresponding b_i's is also a square, then we have found two squares that are congruent modulo n: the product of the b_i's and the product of the a_i^2's.

$$b_i \times \cdots \times b_j \equiv (a_i \times \cdots \times a_j)^2 \pmod{n}.$$

The goal was to find an x^2 and y^2 that are congruent modulo n, so:

$$x^2 = b_i \times \cdots \times b_j \qquad y^2 = (a_i \times \cdots \times a_j)^2$$
$$x = \sqrt{b_i \times \cdots \times b_j} \qquad y = a_i \times \cdots \times a_j.$$

Then all that is left to do is run the Euclidean algorithm on $\gcd(x - y, n)$, which should produce a factor of n.

Now, it seems there could be an issue: What if the number to factor is not nice like 899, and we have to search through lots of a_i^2's, like in the example used on page 70 of [7] where $n = 1649$. We would have to try multiplying a_1^2 with $a_2^2, a_3^2 \ldots$, and a_2^2 with $a_3^2, \ldots$, and so on. But then we have to try multiplying three a_i^2's together, then four together and so on, until we find a subset whose corresponding b_i's also multiply to a square. In other words, we go through every subset of our set of a_i^2's, and there are 2^n subsets. This would take much too long. Fortunately, this is where $b_i = a_i^2 \pmod{n}$ being smooth is very helpful.

Note that a positive integer x is a square if and only if all of its prime factors p_i divide it an even number of times. But by construction, b_i is B-smooth, so each b_i has at most $\pi(B)$ prime factors to choose from. Therefore it should be easy to find at least two b_i's such that in their product's factorization, all primes have even powers. In fact, grabbing $\pi(B) + 1$ numbers guarantees these b_i's can be found. A much more rigorous proof for this can be found in [7]. In short, by viewing the prime factorization of each b_i as a vector in the vector space $\mathbb{F}_2^{\pi(B)}$, then having $\pi(B) + 1$ vectors makes

the set linearly dependent. But to be linearly dependent in this space is the same as having some subset of the vectors add to the 0 vector modulo 2. In other words, there is some subset of the b_i's whose product is divisible an even number of times by all of its prime factors. Notice that b_i being B-smooth also makes it easy to factor. Because its prime factors are small, brute force factoring is not terribly slow, so it can be used.

Let us review our steps so far. To factor n, we first choose a smoothness bound B. Then we find $\pi(B) + 1$ numbers whose squares modulo n are B-smooth. Then, the final step is to find a subset of those squares whose product is also a square. This last step requires a bit of linear algebra, as hinted at by the proof mentioned before. Let the prime factorization of each b_i be:

$$b_i = \prod_{j=1}^{\pi(B)} p^{e_{ij}}.$$

Then, take all of the e_{ij}'s modulo 2 and place them into a matrix, where the row, i, corresponds to the b_i, and the column, j, corresponds to the number or index of the prime in the list of primes up to $\pi(B)$. To illustrate, take the number 2467^2. It's prime factorization is $5 \times 11^2 \times 13 \times 19$. If $B = 20$, the primes are $\{2, 3, 5, 7, 11, 13, 17, 19\}$. Then its row in the matrix would look like:

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

Once we have our matrix modulo 2 of the e_{ij}'s, we flip or transpose the matrix, then perform row reduction. Row reduction of a matrix is the combination of three different row operations to transform a matrix so that: All non-zero rows are above zero rows, the first 1 in row i is to the right of the first 1 in row $i-1$, and if the first 1 in row i is in column j, all other entries in column j are 0. These row operations include replacing row i with the sum of itself and the multiple of another row (technically two different operations) and switching two rows.

Once the matrix has been reduced, locating the good b_i's is simple. First look for a column that does not contain any leading 1's. There are two cases for such a column. If the column, say column j, has all 0 entries, we have gotten lucky and the corresponding b, which is b_j, is already a square. We take b_j and we are done. If column j has some 1's, simply look for all columns that have a leading 1 in the same row that column j has a 1. Then take all those new columns and column j, and multiply the corresponding b's and their corresponding a's to get two squares modulo n. Why does this work? Well, after transposing the matrix, each row corresponds to the power of a prime in each b_i, but finding a good subset at this point is not easy, so we reduce the matrix. Note that row reduction does not affect whether there is a good subset or not. After reduction, the rows now correspond to sums of exponents in the prime factorization [8].

Finally, compute the gcd of the difference of their square roots and n, which should produce a factor of n. The algorithm looks something like this:

Algorithm 5: Quadratic Sieve

input: n, the number to factor, and K, the smoothness bound

```
1  Primes ← list of primes up to K ; // Could use Eratosthenes sieve
2  A ← empty list;
3  B ← empty list;
4  i ← 0;
5  while length(A) < length(Primes) + 1 do // π(K) + 1
6  |  a ← ⌈n⌉ + i;
7  |  b ← a^2 (mod n);
8  |  if b is K-smooth then // repeated trial division using Primes
9  |  |  append a to A;
10 |  |  append b to B;
11 |  end
12 |  i ← i + 1;
13 end
14 M ← length(B) × length(Primes) matrix;
   // continues on next page
```

An example of the algorithm in use can be found in [8] on pages 133-135.

The quadratic sieve is one of the best factoring algorithms on classical computers. One important aspect of the quadratic sieve that we have brushed past is the choice

```
for i to length(B) do
    pf ← factor(B[i]) ; // pf will be prime factorization of b_i
    foreach p in pf do
        e ← power of p;
        e ← e (mod 2);
        j ← index of p in Primes;
        M[i, j] ← e;
    end
end
transpose(M);
rowReduction(M);
j ← 0;
foreach column in M do
    j ← current column index;
    if column does not contain leading 1 then
        break;
    end
end
if column j contains all 0 then
    x ← √B[j];
    y ← A[j];
else if column j contains 1 then
    Ones ← list of 1's row indices;
    x ← √(B[j] × ∏_{i∈Ones} B[i]);
    y ← A[j] × ∏_{i∈Ones} A[i];
end
return gcd(x − y, n);
```

of smoothness bound. It turns out it is a rather involved optimization problem, requiring some analytic number theory and a bit of probability. A thorough proof for the choice of smoothness bound as well as the running time for the sieve can be found in [7]. The runtime for the quadratic sieve is:

$$\exp\left((1+o(1))(\log n \log\log n)^{1/2}\right)$$

Another important point is the use of matrices. This could have some issues. In short, as K, the smoothness bound, gets larger, the matrix used will also need to grow. Moreover, the row reduction method described, called Gaussian reduction, is relatively slow compared to other methods. Therefore, the matrix problem could

influence the choice of smoothness bound. The matrix problem is more thoroughly discussed in [7].

That being said, it is the algorithm of choice for composites with around 20 to 120 digits that do not have small prime factors easily discoverable through other methods [7]. Another sieve, called the number field sieve, performs better on larger numbers, although there is no well defined cut off for when it starts to perform better. As technology evolves and computers get faster, it is possible more clever factoring algorithms will appear.

3 A Brief Introduction to Quantum Computing

We have mentioned before that there exists a factoring algorithm on quantum computers that runs in polynomial time. In order to understand the algorithm, this section will serve as a brief introduction to quantum computing. Note that we do not currently have powerful enough quantum computers to run such an algorithm, but the theory still holds. In addition, there are also quantum cryptosystems that are provably unbreakable [8], but we will focus on how quantum computing can break current cryptosystems.

In the early 1900's, physicists realized their current physics was not enough to explain some bizarre results, so they established the field of quantum mechanics. Quantum mechanics is a mathematical framework or set of rules used to construct physical theories. Quantum mechanics is to physical theories as an operating system is to application software [5]. Rules are set by the operating system, but the app has freedom to accomplish what it needs to within those rules. Classical computers, and even the mathematical concept of information, are both inherently dictated by the laws of classical physics. Once the field of quantum mechanics was established, it made its way into computing and information theory. To define quantum computing and information concretely, it is the result of using the physical reality described by quantum theory to perform a task a classical computer might not be able to do. For

a much more thorough look at the history, we refer the reader to Chapter 1 in both [4] and [5].

3.1 The Qubit

In the classical world, a bit is the smallest piece of information and it has two possible states, 0 or 1. Quantum computing and information have a slightly different smallest piece of information: the quantum bit, or qubit. Similar to a bit, it has two states $|0\rangle$ and $|1\rangle$. However, it can have a state other than these two. It is possible to form a linear combination of states, often referred to as a superposition. A qubit in a superposition is written like so:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle\,.$$

This '$|\ \rangle$' symbol is called a ket and is part of Dirac notation, which is the standard notation for states in quantum mechanics. The numbers a and b are complex numbers such that $|a|^2 + |b|^2 = 1$. This can be interpreted as saying this qubit has probability $|a|^2$ of being a $|0\rangle$ and probability $|b|^2$ of being a $|1\rangle$. In other words, the state of a qubit is a vector in a two-dimensional complex vector space [5]. The states $|0\rangle$ and $|1\rangle$ are called computational basis states, since they form an orthonormal basis for the vector space. Additionally, these complex vector spaces go by the name "Hilbert spaces", denoted $\mathcal{H}$, and we can represent kets as finite column vectors and operators as finite matrices [4].

Unlike a normal bit that must be 0 or 1, a qubit exists in a continuum of states until we measure it, meaning its state is somewhere between $|0\rangle$ and $|1\rangle$. Once a qubit has been measured, that superposition, that state between $|0\rangle$ and $|1\rangle$, is lost forever. When measured again, it will always return the same state. Then, if we measure $|\psi\rangle$ and it results in $|0\rangle$, then every measurement after will always return $|0\rangle$. That is to say a qubit's state, the values of a and b, are unobservable, but they do exist. All we know is the probability it will be $|0\rangle$ or $|1\rangle$ when measured.

However, basis states other than $|0\rangle$ and $|1\rangle$ can be used, allowing for a slightly different measurement of the qubit. Take the following states:

$$|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$
$$|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle .$$

Now we can express $|0\rangle$ and $|1\rangle$ in terms of $|+\rangle$ and $|-\rangle$ by adding or subtracting the two equations.

$$|0\rangle = \frac{1}{\sqrt{2}}|+\rangle + \frac{1}{\sqrt{2}}|-\rangle$$
$$|1\rangle = \frac{1}{\sqrt{2}}|+\rangle - \frac{1}{\sqrt{2}}|-\rangle .$$

All probabilities are still valid. It turns out that any superposition or state $|\psi\rangle$ of $|0\rangle$ and $|1\rangle$ can be rewritten in terms of $|+\rangle$ and $|-\rangle$ [8]. Given a valid state $|\psi\rangle = a|0\rangle + b|1\rangle$,

$$\begin{aligned}|\psi\rangle = a|0\rangle + b|1\rangle &= a\left(\frac{1}{\sqrt{2}}|+\rangle + \frac{1}{\sqrt{2}}|-\rangle\right) + b\left(\frac{1}{\sqrt{2}}|+\rangle - \frac{1}{\sqrt{2}}|-\rangle\right) \\ &= a\frac{1}{\sqrt{2}}|+\rangle + a\frac{1}{\sqrt{2}}|-\rangle + b\frac{1}{\sqrt{2}}|+\rangle - b\frac{1}{\sqrt{2}}|-\rangle \\ &= \left(\frac{a+b}{\sqrt{2}}\right)|+\rangle + \left(\frac{a-b}{\sqrt{2}}\right)|-\rangle .\end{aligned}$$

Why is this helpful? Since the probabilities of $|\psi\rangle$ in terms of the new basis states $|+\rangle$ and $|-\rangle$ are still valid, we can actually measure $|\psi\rangle$ with respect to this basis. In other words, when we measure the qubit, it will result in, or collapse to $|+\rangle$ or $|-\rangle$ with probabilities $(\frac{a+b}{\sqrt{2}})^2$ and $(\frac{a-b}{\sqrt{2}})^2$. Now, we can measure the qubit again in terms of $|0\rangle$ and $|1\rangle$, getting either one half the time. This is true regardless of the original state $|\psi\rangle$. But we could still measure the qubit again.

For example, suppose we have $|\psi\rangle$, a superposition of $|0\rangle$ and $|1\rangle$. When measured, the qubit collapses to $|1\rangle$. Now measure it in terms of $|+\rangle$ and $|-\rangle$, and the qubit

collapses to $|-\rangle$. Now measure it again, and it might collapse to $|0\rangle$. Consecutive measurements in the same basis always gets the same result, but measuring in a basis different from the last could result in something different. This is because measuring a qubit is a fundamental disturbance to it, meaning measuring it without it being modified is impossible.

We can also perform operations on a qubit like we would a bit. Basic operations like negating a bit or adding two bits are called logic gates. The quantum version are called quantum gates. It is possible to start with a qubit in one state and transform it so that it is in a superposition. The only rule is that it must convert to a "good" basis. In other words, quantum operations must be unitary transformations. Put simply, they must behave well with respect to superpositions.

This example is restated from [8]: Suppose we have an operation, or quantum gate, that sends $|0\rangle$ to $|\oplus\rangle = \frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle$, and $|1\rangle$ to $|\ominus\rangle = \frac{24}{25}|0\rangle + \frac{7}{25}|1\rangle$. These are valid states as $|a|^2 + |b|^2 = 1$. This gate is fine in terms of the states $|0\rangle$ and $|1\rangle$, but how about a superposition? What is $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ sent to? It must be sent to $\frac{1}{\sqrt{2}}|\oplus\rangle + \frac{1}{\sqrt{2}}|\ominus\rangle$. This is equal to

$$\frac{1}{\sqrt{2}}\left(\frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle\right) + \frac{1}{\sqrt{2}}\left(\frac{24}{25}|0\rangle + \frac{7}{25}|1\rangle\right) = \frac{1}{\sqrt{2}}\left(\frac{39}{25}|0\rangle + \frac{27}{25}|1\rangle\right).$$

If we check the probabilities,

$$|a|^2 + |b|^2 = \left(\frac{1}{\sqrt{2}}\frac{39}{25}\right)^2 + \left(\frac{1}{\sqrt{2}}\frac{27}{25}\right)^2 = \frac{9}{5} \neq 1.$$

Therefore, this quantum gate is not good.

A change of basis that behaves well on superpositions is one type of operation we can do. For example, the situation described previously that sent $|0\rangle$ and $|1\rangle$ to $|+\rangle$ and $|-\rangle$. In fact, this specific operation is very important, and is called the Hadamard transform. Its implementation is referred to as a Hadamard gate. Put another way, a Hadamard gate will send the two basis states $|0\rangle$ and $|1\rangle$ to a superposition of the

two states. In addition, the Hadamard gate is its own inverse [5]. Measuring a qubit with respect to the $|+\rangle\backslash|-\rangle$ basis means applying a Hadamard gate to a qubit, and then measuring it.

So far, we have discussed examples using a one qubit system with the basis states $|0\rangle$ and $|1\rangle$. When we add qubits to a quantum system, we double the dimension of the corresponding complex vector space. Therefore, an n-qubit system has dimension 2^n. The basis states of the n-qubit system would be all the kets composed of only 0s and 1s, i.e. the binary strings of length n. We can also refer to these basis states using the decimal representation of the binary string, denoting the number of qubits with a subscript [10]. To illustrate, suppose we have a 3-qubit system. Our 8 basis states will be

$$\begin{gathered}|000\rangle \quad |001\rangle \quad |010\rangle \quad |100\rangle \quad |011\rangle \quad |101\rangle \quad |110\rangle \quad |111\rangle, \\ |0\rangle_3 \quad |1\rangle_3 \quad |2\rangle_3 \quad |3\rangle_3 \quad |4\rangle_3 \quad |5\rangle_3 \quad |6\rangle_3 \quad |7\rangle_3.\end{gathered}$$

Then an arbitrary state in this system can be written nicely as a sum,

$$\begin{aligned}|\psi\rangle &= a_0 |0\rangle_3 + a_1 |1\rangle_3 + \cdots + a_7 |7\rangle_3 \\ &= \sum_{j=0}^{7} a_j |j\rangle_3 .\end{aligned}$$

with $|a_0|^2 + |a_1|^2 + \cdots + |a_7|^2 = 1$. For future notation purposes, we can also let $N = 2^n$. Then for an n-qubit system, an arbitrary state would be

$$\begin{aligned}|\psi\rangle &= a_0 |0\rangle_n + a_1 |1\rangle_n + \cdots + a_{2^n-1} |2^n - 1\rangle_n \\ &= \sum_{j=0}^{2^n-1} a_j |j\rangle_n = \sum_{j=0}^{N-1} a_j |j\rangle_n .\end{aligned}$$

The values a, b from the 1-qubit system and these values a_i are called probability amplitudes.

There is quite a bit of nuance to n-qubit systems with things like entanglement and what it means to "add a qubit to the system" [10], but that would be out of the

scope of this paper and we only really need this general understanding and notation to look at Shor's algorithm.

4 Shor's Algorithm

Shor's algorithm is a quantum factoring algorithm that runs in polynomial time. It takes advantage of the quantum Fourier transform and continued fractions. We will first discuss the general idea behind the algorithm.

First, assume that n is odd and not a prime power. There are efficient classical methods for an n that is even or a prime power [9]. Then, if we want to factor n, it is enough to find a number $x \not\equiv \pm 1 \pmod{n}$ such that $x^2 \equiv 1 \pmod{n}$. If we find this x, then $\gcd(x-1, n)$ and $\gcd(x+1, n)$ must be nontrivial factors of n.

Proof. If $x^2 \equiv 1 \pmod{n}$, then $(x+1)(x-1) \equiv 0 \pmod{n}$, so $n|(x+1)(x-1)$. But $x \not\equiv 1 \pmod{n}$, so $x - 1 \not\equiv 0 \pmod{n}$. Suppose for contradiction that $\gcd(x-1, n) = 1$. Then $(x+1)(1) \equiv 0 \pmod{n}$, which implies $x \equiv -1 \pmod{n}$, which is a contradiction. Hence, $\gcd(x-1, n)$ cannot be 1. It also cannot be n by the same analysis. Also by the same analysis, $\gcd(x+1, n)$ cannot be 1 or n.

□

In order to find such an x, we will have to look at the order of a number a between 1 and $n-1$ in $(\mathbb{Z}/n\mathbb{Z})^\times$. The order of a is a number r such that $a^r \equiv 1 \pmod{n}$. Let n be a product of two distinct odd primes p and q. Note that the order of a must divide $|(\mathbb{Z}/n\mathbb{Z})^\times| = \phi(n) = (p-1)(q-1)$ because of Lagrange's Theorem. Furthermore, the order of a in $(\mathbb{Z}/n\mathbb{Z})^\times$ is determined by its order in $(\mathbb{Z}/p\mathbb{Z})^\times$ and $(\mathbb{Z}/q\mathbb{Z})^\times$: the order of a modulo n is the least common multiple of its order modulo p and q. Now, suppose r is the order of a modulo n. If r is even, then $x = a^{r/2}$ satisfies that $x^2 = a^r \equiv 1 \pmod{n}$. In addition, $x \not\equiv 1 \pmod{n}$ because that would imply $\frac{r}{2} < r$ is the order of a, but r is least by construction. But could $x \equiv -1 \pmod{n}$? For this to happen, $x \equiv -1 \pmod{p}$ and $x \equiv -1 \pmod{q}$. Therefore, as long as x is

not congruent -1 modulo p or modulo q, x will not be congruent -1 modulo n. It turns out this is usually the case.

Now our problem is to figure out when $x \equiv -1 \pmod{p}$ and $x \equiv -1 \pmod{q}$ and to show that both happening is not common. $\mathbb{Z}_p^\times$ and $\mathbb{Z}_q^\times$ are cyclic groups, so let g, h be generators for each respectively. Then there must exist integers s, t such that $1 \leq s \leq p-2$, $1 \leq t \leq q-2$ and $a \equiv g^s \pmod{p}$ and $a \equiv h^t \pmod{q}$. Furthermore, the order of a in $\mathbb{Z}_p^\times$ is the smallest integer k such that sk is a multiple of $p-1$, because of Fermat's Little Theorem. Likewise, the order of a in $\mathbb{Z}_q^\times$ is the smallest integer l such that sl is a multiple of $q-1$. Then,

$$k = \frac{p-1}{\gcd(p-1, s)} \qquad l = \frac{q-1}{\gcd(q-1, t)}.$$

If r is the order of a in $\mathbb{Z}_n^\times$, then $r = \operatorname{lcm}(k, l)$. If the largest powers of 2 that divide k and l are equal, so $k = 2^i k'$ and $l = 2^i l'$, then our a will be bad and we need to pick a new a. If $i = 0$, then k and l are both odd, making r odd and $a^{r/2}$ is not an integer. If $i \geq 1$, we must check if $a^{r/2} \equiv -1 \pmod{n}$. This happens if and only if $a^{r/2} \equiv -1 \pmod{p}$ and $a^{r/2} \equiv -1 \pmod{q}$. Since k and l have the same amount of 2's, then r must have that same amount of 2's. Also, since k is the order of a modulo p, $a^{k/2} \equiv -1 \pmod{p}$. Then,

$$a^{r/2} = a^{\frac{k}{2}m} \equiv (-1)^m \equiv -1 \pmod{p}.$$

We can replace k with l and p with q to get a similar result, so this a is bad. Note that m is odd. If it was even, that would imply r does not have the same number of 2's as k and l.

Now, how often do the powers of 2 match? First, notice that $p-1$ is even, so let 2^α be the largest power of 2 in $p-1$. Then the number of 2's in k must be $d = \alpha - z$, where z is the number of 2's in s. Also by the Chinese Remainder Theorem, choosing a random $a \pmod{n}$ is equivalent to choosing a random s and t between 1 and $p-2$

and 1 and $q-2$, respectively. Let us look at the case where k is odd, so $d=0$. This happens when $2^\alpha | s$. This is the case because

$$k = \frac{p-1}{\gcd(p-1,s)} = \frac{2^\alpha m}{2^\alpha \gcd(m,s')}, \text{with } m,\ s' \text{ odd}$$

In other words, we can control d by looking at z. With $d=0$, s must be a multiple of 2^α. There are precisely $\frac{p-1}{2^\alpha}$ such numbers. Then the multiples of 2^α form $1/2^\alpha$ of the $p-1$ numbers. Now, suppose $d=1$. By the same analysis, the power of 2 in s must be $2^{\alpha-1}$, which means s is a multiple of $2^{\alpha-1}$. There are $1/2^{\alpha-1}$ such numbers, but this set includes the multiples of 2^α which we have to remove because if s is one of these numbers, k does not have the desired power of 2. Then the possible values for s are $1/2^\alpha$ of the whole set. We can continue with this process of iterating the power of 2 in k and making sure to remove bad values of s. This results in the following probabilities of choosing a valid s for any d:

power of 2 in k	0	1	2	...	d	...	α
chance of valid s	$\frac{1}{2^\alpha}$	$\frac{1}{2^\alpha}$	$\frac{1}{2^{\alpha-1}}$	...	$\frac{1}{2^{\alpha-d+1}}$	...	$\frac{1}{2}$

We can use the exact same analysis on l, q, and t that we did on k, p, and s to get the same probabilities. Suppose the largest power of 2 in q is β, and $\alpha \leq \beta$. Then it follows that the probability that the powers of 2 match in k and l is

$$\begin{aligned} P(match) &= \frac{1}{2^\alpha}\frac{1}{2^\beta} + \frac{1}{2^\alpha}\frac{1}{2^\beta} + \frac{1}{2^{\alpha-1}}\frac{1}{2^{\beta-1}} + \cdots + \frac{1}{2}\frac{1}{2^{\beta-\alpha+1}} \\ &= \frac{1}{2^\alpha 2^\beta} + \frac{1}{2^\alpha 2^\beta}\left(1 + \frac{1}{4} + \cdots + \frac{1}{4^{\alpha-1}}\right) \\ &= \frac{7}{3 \cdot 2^\alpha \cdot 2^\beta} - \frac{1}{3 \cdot 4^{\alpha-1} \cdot 2^\alpha \cdot 2^\beta} \end{aligned}$$

The largest this can be is when $\alpha = \beta = 1$, which gives us $7/12 - 1/12 = 1/2$. In other words, we have at most a $1/2$ chance of matching, so at least a $1 - 1/2$ chance of not matching. This all shows that we have at least $1/2$ chance of choosing a good a, where the powers of 2 in k and l do not match.

We have shown the case for $n = pq$, but this result can be shown for any product

of odd primes, which can be seen in Appendix 4 of [5]. In fact, the probability of choosing a good a increases with the number of prime factors in n. In general, if n has w prime factors, the probability of choosing a good a is $1 - \frac{1}{2^{w-1}}$. In other words, when running Shor's algorithm, we can choose a good a fairly quickly.

Suppose k has more 2s than l does in its factorization. Then

$$a^{r/2} \equiv 1 \pmod{p} \qquad a^{r/2} \equiv -1 \pmod{q}$$

and vice versa. Hence, $x = a^{r/2}$ satisfies that $x^2 \equiv 1 \pmod{n}$ and $x \not\equiv \pm 1 \pmod{n}$. All of this work proves that if we can compute the order of an element in $\mathbb{Z}_n^\times$, then we can factor n.

Shor's algorithm uses a quantum computer to find the order of some element a, and it does so from a slightly different angle. We let Q be a number, usually a power of 2, and let $f : \mathbb{Z}/Q\mathbb{Z}$ be a periodic function. Then $f(x) = f(x+k)$ for some positive minimal integer k. Then f is completely determined by $f(0), f(1), \ldots, f(k-1)$. We assume that we know how to compute f and that $f(0), f(1), \ldots, f(k-1)$ are distinct. However, we do not know what k is. The goal is now to find the period k with a quantum computer.

For factoring n (finding the order of a), we will use the function $f(x) = a^x \pmod{n}$. Then the period k of this function is the smallest positive integer such that $a^x \equiv a^{x+k} \pmod{n}$. Then $1 \equiv a^k \pmod{n}$. Hence, k is the order of a in $\mathbb{Z}_n^\times$. We also define Q to be the smallest power of 2 greater than n. Being a power of 2 will make manipulating the qubits easier. Moreover, this makes Q large enough to ensure we get a good result after running the algorithm [9]. This will be made more clear in a later section.

Shor's algorithm requires two key ingredients: the quantum Fourier transform and continued fractions.

4.1 Quantum Fourier Transform

To help get a better understanding of what the Quantum Fourier Transform (QFT) does, it may help to briefly look at the classical Fourier transform. Any function $f(x)$ can be broken down or rewritten into a sum sines and cosines regardless of whether it is a periodic function or not [3]. The Fourier transforms use this idea to take any input function and return that function's frequency domain. Functions are usually looked at with respect to time, but it is possible to also look at them with respect to frequency. Put very simply, the Fourier transform will return the spectrum of a function, which reveals the various frequencies that exist in the function. These frequencies are part of how any function can be rewritten using sines and cosines. There are a number of Fourier transforms, but they all have a common form: they take a function $g(t)$ that depends on some time t, and return a function $G(f)$, where f is frequency [2]. This means $G(f)$ should have peaks on its graph at the frequencies that make up $g(t)$.

The classical Fourier transform most closely related to the quantum version is the Discrete Fourier Transform (DFT). Put simply, the DFT will take a finite vector of complex numbers and transforms each component of the vector in a specific way. The QFT performs the exact same transformation on the amplitudes in a given state [5]. The following definition is written using [10] and [8].

Definition 3. The Quantum Fourier Transform ($\mathcal{F}$) of a state $|\psi\rangle$ in an n-qubit system that has $2^n = N$ basis states is

$$\mathcal{F}(|\psi\rangle) = \mathcal{F}\left(\sum_{j=0}^{N-1} a_j \left|j\right\rangle_n\right) \rightarrow \sum_{j=0}^{N-1} b_j \left|j\right\rangle_n,$$

where

$$b_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} a_k e^{\frac{2\pi i j k}{N}}.$$

This can be written together like so

$$\mathcal{F}(|\psi\rangle) = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} a_k e^{\frac{2\pi i j k}{N}} |j\rangle_n .$$

Note that the QFT can also be applied to a basis state $|j\rangle$:

$$\mathcal{F}(|j\rangle) = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{\frac{2\pi i j k}{N}} |k\rangle .$$

It is also valid to simplify $e^{\frac{2\pi i}{N}}$ as ω, which in this case is specifically a primitive N-th root of unity.

Definition 4. Let $N \in \mathbb{N}$. An N-th root of unity is a complex number z such that $z^N = 1$. It is also primitive if all other N-th roots of unity can be written as a power of z. When N is prime, all N-th roots of unity are primitive, except 1 which is only primitive for $N = 1$. There are always N N-th roots of unity and 1 is always one of them [10].

Another way to think about N-th roots of unity is to think about taking steps around the unit circle starting from $(0, 1)$ and using Euler's formula: $e^{\theta i} = \cos(\theta) + \sin(\theta)i$. When $\theta = 2\pi$, we go around the whole circle. If we want the three 3rd roots of unity, we want to take 3 steps or go one third of the way every step. To do so, we let $\theta = \frac{2\pi}{3}n$, for $n = 0, 1, 2$.

There is a lot going on here. To go into all of the intricacies of the QFT is out of the scope of this paper. Instead, we will try to get an intuitive understanding of it and what it does to the input state. For example, take a 1-qubit system. We have $n = 1$ and $N = 2$ which means $e^{\frac{2\pi i}{N}} = \omega = -1$ since -1 is a primitive 2nd root of

unity. Then the QFT on a state $|\psi\rangle = a_0 |0\rangle + a_1 |1\rangle$ would look like

$$\begin{aligned}
\mathcal{F}(|\psi\rangle) &= \frac{1}{\sqrt{2}} \sum_{j=0}^{1} \sum_{k=0}^{1} a_k (-1)^{jk} |j\rangle_1 \\
&= \frac{1}{\sqrt{2}} \sum_{j=0}^{1} a_0 (-1)^{j(0)} + a_1 (-1)^{j(1)} |j\rangle_1 \\
&= \frac{1}{\sqrt{2}} [(a_0 (-1)^{(0)(0)} + a_1 (-1)^{(0)(1)}) |0\rangle_1 + (a_0 (-1)^{(1)(0)} + a_1 (-1)^{(1)(1)}) |1\rangle_1] \\
&= \frac{1}{\sqrt{2}} [(a_0 + a_1) |0\rangle_1 + (a_0 - a_1) |1\rangle_1].
\end{aligned}$$

Suppose $|\psi\rangle$ is one of the basis states. For $|0\rangle_1 = |0\rangle$, $a_0 = 1$ and $a_1 = 0$. Likewise, for $|1\rangle_1 = |1\rangle$, $a_0 = 0$ and $a_1 = 1$. Then

$$\mathcal{F}(|0\rangle) = \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle , \qquad \mathcal{F}(|1\rangle) = \frac{1}{\sqrt{2}} |0\rangle - \frac{1}{\sqrt{2}} |1\rangle .$$

The important thing to note is that the QFT placed both basis states into a superposition whose probability amplitudes are all equal, namely 1/2. It turns out this is the case in general for an n-qubit system. The QFT of any one of the N basis states is sent to a superposition with equal probability for each basis state. Looking back at the last formula of the QFT definition, it is easy to see why this is the case. The formula implies that the probability for each basis state after the QFT is

$$\left| \frac{e^{\frac{2\pi i j k}{N}}}{\sqrt{N}} \right|^2 = \frac{|e^{\frac{2\pi i j k}{N}}|^2}{|\sqrt{N}|^2} = \frac{|e^{\frac{2\pi i j k}{N}}|^2}{N}.$$

Recall that $e^{\frac{2\pi i j k}{N}}$ is a complex number that we can rewrite as $\cos(2\pi jk/N)+\sin(2\pi jk/N)i$ using Euler's formula. Then the absolute value of $e^{\frac{2\pi i j k}{N}}$ is just the magnitude of the complex number it refers to. But $e^{\frac{2\pi i j k}{N}}$ is an N-th root of unity, so that complex number lies on the unit circle. Therefore its magnitude, or absolute value, must be

1. Then

$$\frac{|e^{\frac{2\pi ijk}{N}}|^2}{N} = \frac{1^2}{N} = \frac{1}{N}.$$

There are precisely N basis states in the n-qubit system, so they have equal probability.

Why is this important? In an arbitrary state $|\psi\rangle$, the probability amplitudes of each basis state will not just be 1 for one and 0 for all others like a basis state $|j\rangle$. These different probability amplitudes will clearly end up affecting the new superposition that the state is placed in, and the state will not just be evenly spread over the basis states. In other words, the QFT will spread out a quantum state over all possible quantum states, and the initial probability amplitudes will affect how the state gets spread out [8]. This spreading out reveals the periodicity of the state's wavefunction, much like a classical Fourier transform would reveal the periodicity of an input function [11]. A wavefunction, put simply, is another way of describing the state of a qubit. We have been describing the state as a vector.

This is key to Shor's algorithm. If we construct our qubit nicely by somehow using our function $f(x) = a^x \pmod{n}$, the QFT of that qubit will have terms that cancel in a way that is helpful to us. Moreover, implementing the QFT on a quantum computer takes only $O(e^2)$ quantum gates, where e is the number of qubits we are using [8].

4.2 Continued Fractions

The other main piece of Shor's algorithm is continued fractions. A continued fraction is a number of the form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ddots}}}$$

where a_i is an integer and $a_i \geq 1$ for $i \geq 1$. Rational numbers have finite continued fraction expansions. The continued fraction expansion of a real number is easy to compute. If we truncate the expansion we can get a good approximation of that real number. In fact, if we truncate and get some number s/t, this is the best rational approximation with denominator $\leq t$ [8].

Suppose we want to approximate α. Let $\alpha_0 = \alpha$. Then $a_0 = \lfloor \alpha_0 \rfloor$. The left over is then $\alpha_0 - a_0 < 1$. If we take the reciprocal, we get $\alpha_1 = \frac{1}{\alpha_0 - a_0} > 1$. Now take the floor of this to get a_1. Then calculate the leftover again. This process is repeated until the reciprocal of the leftover is an integer, which means it is equal to its floor and we cannot get another α. Note that this is an entirely classical algorithm.

Algorithm 6: Continued fraction expansion

input: α, the rational number to approximate

$A \leftarrow$ empty list;
$\alpha_0 \leftarrow \alpha$;
$i \leftarrow 0$;
repeat
  $a_i = \lfloor \alpha_i \rfloor$;
  $\alpha_{i+1} = \lfloor \frac{1}{\alpha_i - a_1} \rfloor$;
  append a_i to A;
until $\lfloor \alpha_i \rfloor = \alpha_i$;
return A

This algorithm will terminate for rationals, but not for irrationals. The algorithm for irrationals is identical, except we take an additional input n, which indicates when

the loop stops since irrationals have infinite expansions. Fortunately, for Shor's algorithm, we will only have to compute expansions for rationals. Then, to approximate α, we take the a_i's up to some $i \geq 0$ and use them to write out the expansion. We now have a nice way to approximate a real number.

4.3 Algorithm Specifics

With the Quantum Fourier Transform and a way to approximate real numbers using continued fractions, we can now discuss how Shor's algorithm finds the period of a function, thus factoring n. The function of interest is $f(x) = a^x \pmod{n}$, with $1 \leq a \leq n-1$ and $\gcd(a, n) = 1$. We get to choose a. We also define Q as the smallest power of 2 greater than n. The following algorithm is written using [9] and [8].

First, initialize a qubit called $|y\rangle$. It will be in a superposition with all probability amplitudes equal:

$$|y\rangle = \frac{1}{\sqrt{Q}} \sum_{j=0}^{Q-1} |j\rangle \,.$$

This takes $e = \log_2(Q)$ qubits. Then use another e qubits to express $f(j)$, so we now have $2e$ qubits. We label the whole qubit as $|y, f(y)\rangle$:

$$|y, f(y)\rangle = \frac{1}{\sqrt{Q}} \sum_{j=0}^{Q-1} |j, f(j)\rangle \,.$$

Now we apply the QFT on the first e qubits, the ones in a superposition with equal amplitudes. This gives us the state

$$\begin{aligned} |\mathcal{F}y, f(y)\rangle &= \frac{1}{\sqrt{Q}} \sum_{j=0}^{Q-1} \frac{1}{\sqrt{Q}} \sum_{\ell=0}^{Q-1} e^{\frac{2\pi i j \ell}{Q}} |\ell, f(j)\rangle \\ &= \frac{1}{Q} \sum_{j=0}^{Q-1} \sum_{\ell=0}^{Q-1} e^{\frac{2\pi i j \ell}{Q}} |\ell, f(j)\rangle \,. \end{aligned}$$

From here, it is not so hard to see the associated probability amplitude, i.e. the probability of collapsing, to a state $|\ell, f(j)\rangle$. However, we can rewrite this to make it a little easier to see how the order of a can be determined from the probability amplitudes. The expression can be rewritten as

$$\frac{1}{Q}\sum_{m=0}^{Q-1}\sum_{\ell=0}^{Q-1}|\ell, m\rangle \sum_{j:f(j)=m} e^{\frac{2\pi i j\ell}{Q}}.$$

Remember that the values of $f(j)$ will repeat depending on our choice of a, as the values of $f(j)$ are just the subgroup generated by a in the multiplicative group modulo n.

What the above expression tells us is that there will be multiple ways to reach the states $|\ell, m\rangle$. But they will have different probability amplitudes based on our function, as the amplitude depends on $m = f(j)$. Depending on which values of j give us $f(j) = m$, the amplitudes will change.

Now, we measure these $2e$ qubits. We end up with $2e$ qubits collapsed to $|\ell, m\rangle$. The probability of getting this state is

$$\left|\frac{1}{Q}\sum_{j:f(j)=m} e^{\frac{2\pi i j\ell}{Q}}\right|^2 = \frac{1}{Q^2}\left|\sum_{b=0}^{(Q-j_0-1)/r} e^{\frac{2\pi i \ell r b}{Q}}\right|^2.$$

where j_0 is the smallest value such that $f(j_0) = m$ and r is the period which is what we want to find. We are summing over $0 \leq j < Q$ such that $a^j \equiv m \equiv a^{j_0}$. In other words, we are summing over all j such that $j \equiv j_0 \pmod{r}$. Then we can write $j = br + j_0$, which is how we get the right side of the above equation [9].

If we measure and end up with $|\ell, m\rangle$, then it was likely the case that this state had a high probability. Why is this a useful observation? Well, the above probability is only large when $e^{\frac{2\pi i \ell r}{Q}}$ is close to 1. In other words, when $\ell r/Q$ is close to an integer, which we can call c.

To see why this is the case, recall that $e^{2\pi i \ldots}$ is a primitive N-th root of unity, so we are summing over all N N-th roots of unity. This sum will always be 0, as many

of the terms will just cancel each other out. Another way to think about this is that we are just going around the entire unit circle, starting and ending at $(1, 0)$. This means some of the possible states we could have collapsed to will have lots of terms cancel out. The only time we do not get a lot of cancellation, is if the second term of the sum, $b = 1$, ends up close to the positive real axis. In other words, it goes around the entire circle. Then every term of the sum after will also end up going around the circle, and these terms will not cancel out and we end up with a large sum. The $b = 1$ term being close to the positive real axis is the same as $e^{\frac{2\pi i \ell r}{Q}}$ being close to 1.

We can be fairly sure that after measurement, we end up with $\frac{\ell r}{Q} \approx c$. Since we know ℓ and Q, we rearrange and say $\frac{\ell}{Q} \approx \frac{c}{r}$, with $r < Q$. We are interested in r, the period of our function. Now we just use the continued fractions algorithm. Given a real number α and a bound Q, we can find all rational numbers $\frac{s}{t}$ such that $t < Q$ and $|\alpha - \frac{s}{t}| < \frac{1}{tQ}$ [8]. A detailed proof for this can be found in subsection 10.5.3 of [10]. In other words, if $\frac{s}{t}$, a convergent of $\frac{\ell}{Q}$, is close enough to $\frac{c}{r}$, then $\frac{s}{t}$ should also be a convergent of $\frac{c}{r}$. This is where we need Q to be large. The larger Q is, the tighter our bound is, which ensures we can get a good convergent $\frac{s}{t}$.

Then we compute the continued fraction expansion of $\frac{\ell}{Q}$, and truncate it at every possible point in the list to get a different convergent $\frac{s}{t}$ for $\frac{\ell}{Q}$. One of these convergents should be $\frac{c}{r}$.

We keep track of the denominators t, which are candidates for r. Since this list is not large, it is easy to check each one. Pick some x and test if $f(x + t) = f(x)$. If such a t is found, we have found the period r. If no good t is found, redo the algorithm with a new value of a. If r is odd, redo the algorithm with a new a until we get an even r. Recall the probability of having a good a is at least $1 - \frac{1}{2^{w-1}}$, so usually we find r quickly. Now we know that $x^2 = a^r \equiv 1 \pmod{n}$, which implies $(x-1)(x+1) = (a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \pmod{n}$. Finally, compute $\gcd(x-1, n)$ and $\gcd(x+1, n)$ to get factors of n. And that is Shor's algorithm!

By running this algorithm about $O(\log \log r)$ times, we have a high probability of finding r. The proof for this can be found in Shor's original paper [9]. Morever,

if we are clever enough, we can make changes to the algorithm to do less quantum computing and more postprocessing to cut down the number of trials by a constant factor [9]. It must be said that this is all only a surface level look at Shor's algorithm (and quantum computing in general). It brushes past some of the specific quantum computing ideas like how order finding is based on the phase estimation algorithm [5] and the specifics of the qubit manipulation going on.

Before wrapping up, we will go through an example of the algorithm. Let us try to factor $n = 15$. We randomly choose $a = 7$, so $\gcd(7, 15) = 1$. Our function is $f(x) = 7^x \pmod{15}$. We also define $Q = 2^4 = 16$, so we will be using 8 qubits total. Now, we can initialize our qubit:

$$\begin{aligned} |y\rangle &= \frac{1}{\sqrt{16}} \sum_{j=0}^{15} |j\rangle \\ &= \frac{1}{\sqrt{16}} \left[|0\rangle + |1\rangle + \cdots + |15\rangle\right]. \end{aligned}$$

Compute $f(j)$:

$$\begin{aligned} |y, f(y)\rangle &= \frac{1}{\sqrt{16}} \sum_{j=0}^{15} |j, f(j)\rangle \\ &= \frac{1}{\sqrt{16}} \left[|0, 1\rangle + |1, 7\rangle + \cdots + |15, 7^{15} \bmod 15\rangle\right]. \end{aligned}$$

Now take the QFT of the first 4 qubits.

$$\begin{aligned} |\mathcal{F}y, f(y)\rangle &= \frac{1}{\sqrt{16}} \sum_{j=0}^{15} \frac{1}{\sqrt{16}} \sum_{\ell=0}^{15} e^{\frac{2\pi i j \ell}{16}} |\ell, f(j)\rangle \\ &= \frac{1}{16} \sum_{j=0}^{15} \sum_{\ell=0}^{15} e^{\frac{2\pi i j \ell}{16}} |\ell, f(j)\rangle \\ &= \frac{1}{16} \sum_{m=0}^{15} \sum_{\ell=0}^{15} |\ell, m\rangle \sum_{j:f(j)=m} e^{\frac{2\pi i j \ell}{16}}. \end{aligned}$$

To illustrate what is happening, we write out a few terms of the second expression

above. Notice how states begin to repeat after $j = 4$, so we can collect like terms.

$$\frac{1}{16}\Big[e^{\frac{2\pi i(0)(0)}{16}}|0,1\rangle + e^{\frac{2\pi i(0)(1)}{16}}|1,1\rangle + \cdots + e^{\frac{2\pi i(0)(15)}{16}}|15,1\rangle\Big\} j=0$$
$$+e^{\frac{2\pi i(1)(0)}{16}}|0,7\rangle + e^{\frac{2\pi i(1)(1)}{16}}|1,7\rangle + \cdots + e^{\frac{2\pi i(1)(15)}{16}}|15,7\rangle\Big\} j=1$$
$$+e^{\frac{2\pi i(2)(0)}{16}}|0,4\rangle + e^{\frac{2\pi i(2)(1)}{16}}|1,4\rangle + \cdots + e^{\frac{2\pi i(2)(15)}{16}}|15,4\rangle\Big\} j=2$$
$$+e^{\frac{2\pi i(3)(0)}{16}}|0,13\rangle + e^{\frac{2\pi i(3)(1)}{16}}|1,13\rangle + \cdots + e^{\frac{2\pi i(3)(15)}{16}}|15,13\rangle\Big\} j=3$$
$$+e^{\frac{2\pi i(4)(0)}{16}}|0,1\rangle + e^{\frac{2\pi i(4)(1)}{16}}|1,1\rangle + \cdots + e^{\frac{2\pi i(4)(15)}{16}}|15,1\rangle\Big\} j=4$$
$$+\ldots\Big].$$

Now, we measure the qubit. Suppose it collapses to $|12, 13\rangle$. So $\ell = 12$ and $m = 13$. From the sum above, we can see the smallest j such that $f(j) = 13$ is $j = 3$. Therefore, $j_0 = 3$. Then, the probability of collapsing to this state is:

$$\frac{1}{16^2}\left|\sum_{b=0}^{(16-3-1)/r} e^{\frac{2\pi i(12)rb}{16}}\right|^2.$$

We care about the $b = 1$ term: $e^{\frac{2\pi i(12)r(1)}{16}} \approx 1$. Then we have that $12r/16 \approx c \in \mathbb{Z}$, or $3/4 \approx c/r$. We compute the continued fraction expansion of $3/4$, which gives us the list $[0; 1, 3]$. Then our convergents are $0/1$, $1/1$, and $3/4$. We test if $f(x) = f(x + t)$ for $t = 0, 1, 4$ and find that 4 works! We now have our period of f or order of $a = 7$, which is $r = 4$. Since 4 is even and $a^{r/2} = 7^2 \pmod{15} = 4 \neq -1 \pmod{15}$, we can set $x^2 = a^r = 7^4$. Then $x \pm 1 = a^{r/2} \pm 1 = 7^2 \pm 1$. Now, compute $\gcd(48, 15) = 3$ and $\gcd(50, 15) = 5$. We have successfully factored $n = 15$.

5 Closing Thoughts

One other important part about factoring large numbers is knowing when we have actually obtained the prime factorization. This requires being able to quickly determine if a number is prime or not. There are a number of methods like the

Miller-Rabin and Agrawal-Kayal-Saxena primality tests, which are discussed in [8]. The challenge of factoring is a two step process of breaking down the number and checking if we have the prime factorization. Primality testing is generally easier than doing the full factorization of a number. While primality testing has its uses in factorization, it is also important to make cryptosystems like RSA work. We need lots of different prime numbers, so we need to be able to quickly check if the numbers we generate are prime [8].

Quantum computing offers a lot of potential growth for cryptography and cybersecurity. To talk about the quantum cryptosystems and related algorithms, including all of the math that goes on, would take another paper or two. Quantum computing is still a growing field, and there is still a lot of progress to made until we can utilize its full potential.

There is some very clever and complicated math going on in these factorization algorithms, and these algorithms are important as they help prove that the encryption methods we use today are still sound. As quantum computing makes more and more advances, it will be interesting to see how cryptography and cybersecurity will evolve along with it.

References

[1] Lynn Margaret Batten. *Public key cryptography: applications and attacks*. John Wiley & Sons, 2013.

[2] Robert M Gray and Joseph W Goodman. *Fourier transforms: an introduction for engineers*, volume 322. Springer Science & Business Media, 2012.

[3] John Francis James. *A student's guide to Fourier transforms: with applications in physics and engineering*. Cambridge university press, 2011.

[4] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An introduction to quantum computing*. OUP Oxford, 2006.

[5] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*, volume 2. Cambridge university press Cambridge, 2001.

[6] Ray A Perlner and David A Cooper. Quantum resistant public key cryptography: a survey. In *Proceedings of the 8th Symposium on Identity and Trust on the Internet*, pages 85–93, 2009.

[7] Carl Pomerance. Smooth numbers and the quadratic sieve. In *Proc. of an MSRI workshop, J. Buhler and P. Stevenhagen, eds.(to appear)*, pages 1–10, 2008.

[8] Simon Rubinstein-Salzedo. *Cryptography*, volume 260. Springer, 2018.

[9] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[10] Robert S Sutor. *Dancing with Qubits: How quantum computing works and how it can change the world*. Packt Publishing Ltd, 2019.

[11] Yaakov S Weinstein, MA Pravia, EM Fortunato, Seth Lloyd, and David G Cory. Implementation of the quantum fourier transform. *Physical review letters*, 86(9):1889, 2001.