Redefining Complexity to Consider Quantum Computers and Exploring Quantum Cryptoanalysis

Jonathon D'Arcy

Year 4 Project
School of Mathematics
University of Edinburgh
16/4/2020

Abstract

In this paper we look to understand the fall of one the most globally recognised modern day cryptosystems, RSA. To do so we will first investigate one of the major classes of cryptography, public key cryptography, and define its parts. We will then proceed to define the RSA cryptosystem and show how it conforms to the class of a public key cryptosystem. Finally, we will investigate how Peter Shor broke the RSA system with the use of a quantum computer.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Jonathon D'Arcy*)

To my loving family, who make me prouder more than I could ever hope to say.

Contents

Chapter 1

Introduction

"Cryptography is a most unusual science. Most professional scientists aim to be the first to publish their work, because it is through dissemination that the work realises its value. In contrast, the fullest value of cryptography is realized by minimising the information available to potential advesaries"

-James H Ellis (Ellis [1]).

Cryptography, the study of securing communication, and Cryptanalysis, the study of cracking these secure communications, have existed in parallel for thousands of years. In fact, the oldest example of intentional cryptography comes from ancient Mesopotamia around 1500BC, when a potter encoded a stone tablet with secret directions on how to create coloured glazes (Khan [2]). Since then, Cryptanalysts and Cryptographers have been pushing each other's fields to higher and higher limits. One of these major pushes came in 1976 when cryptographers Whitfield Diffie and Martin Hellman published a paper titled *"New Directions in Cryptography"* (Diffie and Hellman [3]), which formulated the concept of a public key cryptosystem, or PKC.

A public key cryptosystem is, in short, a method which would allow people who have never met before to securely send each other information, even if eavesdroppers had heard them fully set up the system. This idea rattled the world and shaped how information security was conducted in the newly forming Communication Age. However, unbeknownst to Diffie and Hellman, they were not the first to produce this idea. In 1997, the British Government declassified work done in 1970 by Mathematician James H. Ellis (Ellis [1]; Wayne [4]). His work encapsulated the idea of a public key cryptosystem, under the title Non-Secret Encryption (NSE). As far as we know, Ellis was the first to formally discuss the idea; however, interestingly, he was not the first to implement it. To that we must credit, Walter Koenig Jr..

In 1944, Walter Koenig Jr. worked for AT&T Bell Telephone Laboratories, or Bell Labs for short. Bell labs has long been one of the great American think tanks of the modern age, responsible for the likes of Information Theory, Lasers, Unix, C, and C++. At the time however Bell Labs was working on methods to secure long range communications for the war in the Pacific theatre. Koenig specifically was tasked to help prepare a classified report called Project C-43 for

the Communications Division of the US Armed Forces[†] (Koenig [6]). On page 23 of that report, Koenig described a type of radio masking that would distort and reform radio communications in such a way which reflected all the properties of a public key cryptosystem. Sadly, Koenig dismissed the idea as interesting yet irrelevant as it was not suitable for the long-range communications the American Military required at the time.

Despite Koenig's dismissal, his passage about radio masking has become a source for great speculation. The reason for this is in 1942, the Americans and British began sharing their cryptographic methods, and in 1945 the countries signed the BRUSA Agreement formalizing their collaboration (Johnson [7]). This collaboration was then expanded to include Canada, Australia, and New Zealand under the UKUSA Agreement, which still runs today (National Archives [8]). As we know, the UK's knowledge of public key cryptosystems originated from Ellis, verifiable from his many letters and reports (Ellis [1]). Strangely, even though the Americans should also root their cryptographic origins to Ellis under the UKUSA agreement, they have never formally confirmed this. In fact, they have actively denied all attempts to secure declassification through the Freedom of Information Act, even though it should only match what the British have already declassified (Wayne [4]; Young [9]).

Some believe that the Americans discovered public key cryptography years earlier than Ellis, and kept the knowledge hidden in flagrant disregard of the UKUSA agreement (Wayne [4]). This is largely speculation however, as the only evidence of public key cryptosystems pre-dating Ellis (aside from Koenig's report) comes from a document called the National Security Action Memorandum 160 (Wiesner [10]). This Memorandum was sent in 1962 directly to President Kennedy by Jerome Wiesner, the chair of President Kennedy's Science Advisory Committee. Wiesner was asking President Kennedy in the Memorandum for clarification about the security controls for Nuclear Warheads both home and abroad. Many years later, in 1993, a retired high-ranking official of the NSA called Dr. Lowell K. Frazer, gave a talk at the ACM Computer and Communications Security Conference. During the talk titled, *"The Early Days in Nuclear Command and Control"*, Dr.Frazer stated that public key cryptography was the solution implemented to solve the problems posed in Memorandum 160, implying that public key cryptography had been a known concept since the 60's (Bellovin [11]). Unfortunately, apart from a scrubbed version of the memorandum outlining parts of Wiesner's concerns, no evidence has been released by the NSA.

This is a single example behind the fascinating and at times enigmatic history of modern day Cryptography[††]. In studying its history, one lesson is truly clear; we can never know what resources or abilities an adversary may have when they try to break into our systems. To combat this, we must assume the worst, that one's adversaries have access to the best-known systems and methods when attempting to break our systems. In this report we will explore the

[†]It is interesting to note that C-43 ran parallel to the much better known SIGSALY project that Alan Turing and Claude Shannon worked on for the United States high command (Weadon [5]).

[††]If the reader has the time to read about the history of modern cryptography, reading the NSA's Declassified Cryptologs is truly fascinating (NSA [12]).

work of Peter Shor, whose quantum algorithm in 1994 cracked the world's most popular public key cryptosystem: RSA (Shor [13]). Peter Shor's work invokes the assumption that the enemy can now break RSA. While this may seem to be a harsh assumption, as the algorithm requires the use of a large functioning quantum computer, we must assume for the most confidential information that potential adversaries have this resource. Cryptography today stands again on the edge of major changes spurred on by Shor's quantum cryptanalysis, and again the fields propel themselves forward.

Chapter 2

Cryptography

2.1 Public Key Cryptography

Initially, the problem which public key cryptography claims to solve sounds impossible. In essence, it claims a method which allows any pair of people who have never met before to share information without fear of being understood by anybody else, even if an eavesdropper had access to the same information as all participants. Though, how this works is surprisingly straight forward and relies on two concepts: One-Way Functions and Trap Door Functions.

Definition 2.1.1 (One-way Functions)**.** A one-way function is an invertible function, f, which is computationally easy to calculate in one direction, but computationally hard to calculate in the other (Diffie and Hellman [3]; Hoffstein, Pipher, and Silverman [14]).

Definition 2.1.2 (Trapdoor Function)**.** A trapdoor function is one which is computationally difficult to compute, unless some piece of information about the function, called the trapdoor information, is included, in which case it becomes computationally easy to compute (Diffie and Hellman [3]; Hoffstein, Pipher, and Silverman [14]).

In a public key cryptosystem, we want to have a function which is both one-way and has a trapdoor. This trapdoor will affect the invertibility of the function, meaning it will be computationally difficult to invert the function unless somebody has the trapdoor information.

With this function in place we can demonstrate how a public key cryptosystem works. To do this let us say that we have two parties, Alice, and Bob, who wish to communicate without a third party, Eve, being able to understand them. To make Alice and Bob's job more difficult, we will say that Eve can easily hear all correspondence between Alice and Bob. For Alice to receive a communication from Bob, Alice begins by setting up a one-way trapdoor function, as described earlier. Alice then says this function to both Bob and Eve but keeps the trapdoor information for herself. With the function in hand Bob inputs his message and relays the scrambled output to Alice and Eve. Alice has no problem inverting this function to retrieve Bob's original message as she can use the trapdoor information. However, for Eve it is much harder as she does not have the trapdoor

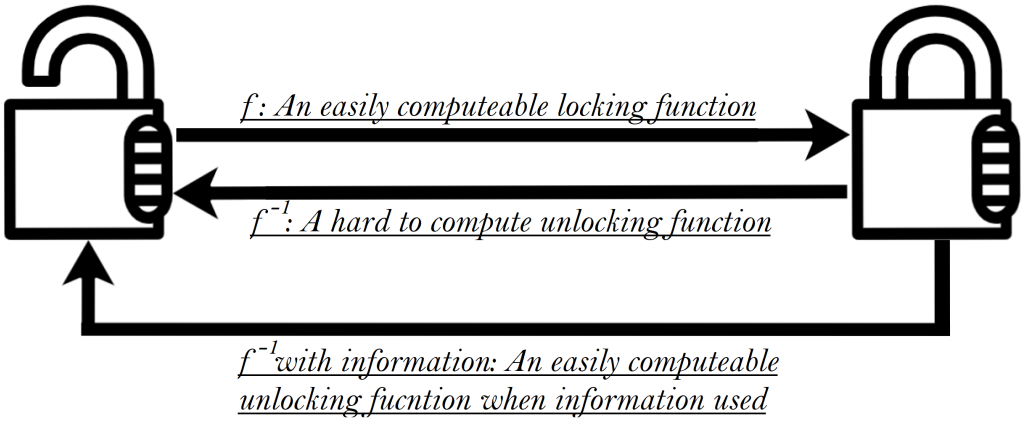

Figure 2.1: The Idea of Public Key Cryptography

information, leaving her with the computationally difficult inverse of Alice's function. If Alice then wanted to speak to Bob, Bob would set up his own one-way trapdoor function and proceed as Alice had originally.

This idea of a public key cryptosystem is represented in Figure 2.1 and can be thought of as the information security equivalent of a combination lock; anybody can lock a combination lock, but it is very difficult to unlock it without the combination. Typically, when talking about public key cryptosystems, we refer to the one-way trapdoor function (and any auxiliary information required to operate it, which does not infringe on the trapdoor information) as the public key, $\mathcal{K}_{pub}$, and the trapdoor information as the private key, $\mathcal{K}_{priv}$ (Hoffstein, Pipher, and Silverman [14]).

2.2 How Difficult is Difficult

When creating the idea of a public key cryptosystem, Diffie and Hellman established that five properties were necessary:

Definition 2.2.1 (Properties of a Public Key Cryptosystem)**.**

1. The encryption function must be inversely related to the decryption function.
2. The creation of the public and private keys must be easy to compute.
3. The encryption function with the public key must be easy to compute.
4. The decryption function with the private key must be easy to compute.
5. The decryption function without the private key must be hard to compute.

(Diffie and Hellman [3]).

The meaning of these properties differs depending on the reader's definition of easy and hard computations. When Diffie and Hellman originally wrote their paper, they began by defining hard to compute functions as those whose fastest algorithms required over 10^{100} operations over a given threshold (Diffie and Hellman [3]). This definition, as they knew at the time, was substandard, and later in the paper, they switched definitions to the more generally accepted Cobham's Thesis (Cobham [15]).

Cobham's Thesis deals with the expansive topic of complexity classes, which groups problems into sets describing their difficulty to compute (Rothe [16]). To fully communicate the intricacies of complexity classes however, would require its own paper. In lieu of this, we will provide the true name of the classes for readers who understand them; as well as a definition which, while not rigorously defined, aims to capture the personality of the class. For example, we would define Cobham's Thesis as such,

Definition 2.2.2 (Cobham's Thesis)**.** A function is defined as computationally easy if there exists an algorithm for the function which runs in the complexity class $\mathcal{P}$. This class is one where algorithms run in polynomial time, are always correct, and only requires the use of a classical computer (Rothe [16]). If no such algorithm exists, we call the function computationally hard (Cobham [15]).

An algorithm running in polynomial time means that the number of operations an algorithm must execute for each input grows in a way that is bounded by some polynomial function (Hoffstein, Pipher, and Silverman [14]; Rothe [16]).

Cobham's Thesis, as we said earlier, is a generally accepted definition for easy and difficult computations. However, we are not by any means required to use it, as it is more of a dogma rather than some piece of rigorous reasoning (Rothe [16]). We instead choose to use the following definition for the purpose of this paper, as it is better suited for cryptographic standards,

Definition 2.2.3. A function is defined as computationally easy if there exists an algorithm for the function which runs in the complexity class $\mathcal{BPP}$. This class is one where an algorithm runs in polynomial time, is correct at least half of the time, and only requires the use of a classical computer and a random number generator (Rothe [16]). If no such algorithm exists, we call the function computationally hard.

We hope this change of definition will not be too difficult for Cobham loyalists to digest as $\mathcal{P}$ is not only a subset of $\mathcal{BPP}$, but it is conjectured that $\mathcal{P}$ is equivalent to $\mathcal{BPP}$, making both definitions identical (Rothe [16]). Though, while this conjecture is still unproven, we want to include protection from the full class of $\mathcal{BPP}$. This is because an expansion to $\mathcal{BPP}$ protects us from algorithms that will either probably or certainly decrypt us in polynomial time, as opposed to Cobham's Thesis which will only protect us against those that will certainly decrypt us in polynomial time. It is our opinion that to accept Cobham's Thesis in a cryptographic sense, is equivalent to being fine with your lock working only half of the time.

In any case however, we need to be careful as both definitions have their flaws (Cook [17]). Most importantly, we are cautious with the fact that the computations of some inputs of a computationally difficult function are not necessarily impossible to compute, nor is every input of an easily computable function necessarily possible to compute. This confusion is due to the fact that we define a functions difficulty by the rate at which their operation count grows, rather than the actual operation requirements.

The problem with looking at this rate is that we do not consider the practical measures of these operation values. For example, an algorithm in polynomial time can be scaled to include any additional amount of finite operations everywhere, so much so that there would be realistically no possible way to calculate the function at any point; however, this function would still be in Polynomial time and called easily computable. Similarly, we can find non-polynomial functions that grow with no rate, using very few operations before exploding exponentially at input values larger than what is realistically practical to consider. Solving these issues however is not simple as they bring in more philosophical questions such as defining some upper bound of useful input numbers when comparing functions, or the choice of threshold on how many operations is too many to realistically compute. To avoid answering these questions we instead draw the line in the sand by looking at the rate and tell anybody who wishes to apply these functions to be careful.

2.3 RSA

In 1977, three mathematicians, Ron Rivest, Adi Shamir, and Leonard Adleman, published the RSA public key cryptosystem (Hoffstein, Pipher, and Silverman [14]; Rivest, Shamir, and Adleman [18]). This system was not only the first publicly available public key cryptosystem but became the most popular in information security. This is because the system was touted as unbreakable due to the powerful one-way trapdoor function that it relied on (Hoffstein, Pipher, and Silverman [14]). In fact, the system was thought to be so strong it was used as the juggernaut that carried cryptographic keys for weaker, yet faster systems (Menezes, et al. [19]).

To provide an overview of how RSA works, we return to our characters Alice and Bob. Suppose Bob would like to send a secure message to Alice, to do this we require Alice to set up the RSA scheme. Alice begins by choosing two large, distinct prime numbers, p and q. These values will make up Alice's private key, and she will never reveal them as they will be used to easily decrypt Bob's message later. Alice then has the task of creating the public key, consisting of two values. The first called the RSA number, N, is equal to the product of p and q. We want N to be massive, in fact to give an idea of the sheer scale we want N to be, the US Department of Commerce requires banks and financial services in the US to have their semi-prime RSA number be a minimum size of 10^{160} (Barker and Roginsky [20]). The second value of the public key is the encryption exponent, e, Alice wants e to be a positive integer such that e and $(p-1)(q-1)$ are coprime, for reasons we will divulge shortly.

For now, we will take this tuple of $\mathcal{K}_{pub} = (N, e)$ and send it to Bob. Bob's task is simple, all Bob must do is take his message and encode it into the integers using an encoding scheme[†]. The purpose of the encoding scheme is not for security, but to package the message into a form that allows us to conduct our computations. Bob then takes his encoded message, m, and feeds it into an equation of the form,

$$c \equiv m^e \pmod{N} \tag{2.1}$$

This resulting value, c, is Bob's now encrypted message which he sends to Alice, along with the name of the encoding scheme he used.

To decrypt Bob's newly obfuscated message, Alice first computes the multiplicative inverse of e in $\pmod{((p-1)(q-1))}$, which we call e^*. Alice then takes the encrypted message and calculates,

$$m' \equiv c^{e^*} \pmod{N} \tag{2.2}$$

This returned value, m', is equivalent to the encoded message, m, that Bob had originally. Alice then simply decodes the message using the encoding scheme Bob provided her, thus completing the RSA Scheme. We summarize the RSA scheme in Table 2.1 below.

Alice	**Bob**
Set Up	
Choose large odd primes p and q. Choose encryption exponent e with $\text{GCD}(e, (p-1)(q-1)) = 1$. Publish $N = pq$ and e.	
Encryption	
	Encode message using $\mathfrak{S}$ giving m. Use published key (N, e) to compute $c \equiv m^e \pmod{N}$. Send c and $\mathfrak{S}$ to Alice.
Decryption	
Compute e^* satisfying $ee^* \equiv 1 \pmod{(p-1)(q-1)}$. Compute $m' \equiv c^{e^*} \pmod{N}$ Decode m' using $\mathfrak{S}$ to get message	

Table 2.1: An RSA Walkthrough

[†]An example of such an encoding scheme would be Unicode or ASCII (Unicode Consortium [21]; Mackenzie [22]).

2.4 Why RSA is a PKC

To show that RSA is in fact a public key cryptosystem, it is necessary to show that it conforms to all the properties given in Definition 2.2.1. We will explore each individually in the subsequent sections.

2.4.1 Property 1: Inverse Relationship

The first condition that a public key cryptosystem must suffice is that the encryption function must be inversely related to the decryption function. In the case of RSA, we can write this as the following proposition.

Proposition 2.4.1. *Let there be large, distinct primes p and q, and let $e \in \mathbb{Z}$ such that* $\text{GCD}(e,(p-1)(q-1)) = 1$. *Then the congruence $x^e \equiv c \pmod{pq}$ has a unique solution $x \equiv c^{(e^*)} \pmod{pq}$ where e^*, is the multiplicative inverse of $e \in \mathbb{Z}_{(p-1)(q-1)}$.*

Proof. To prove this proposition, we need to show three properties. The first is that e has a multiplicative inverse, e^*, in $\mathbb{Z}_{(p-1)(q-1)}$. To prove this, we begin by taking the fact that $\text{GCD}(e,(p-1)(q-1)) = 1$ and apply the Extended Euclidean algorithm (Hoffstein, Pipher, and Silverman [14]; Silverman [23]) to find integers e^* and v such that

$$ee^* + (p-1)(q-1)v = 1. \tag{2.3}$$

Simply rearranging this gives us that:

$$ee^* - 1 = -(p-1)(q-1)v \implies ee^* \equiv 1 \pmod{(p-1)(q-1).} \tag{2.4}$$

This proves that e does in fact have a multiplicative inverse element in $\mathbb{Z}_{(p-1)(q-1)}$.

The second property we must prove is that $x^e \equiv c \pmod{pq}$ has solution $x \equiv c^{(e^*)} \pmod{pq}$. To show this we split the property into two cases. In the first case will have the assumption that $\text{GCD}(c,pq) = 1$ and the second case will have that $\text{GCD}(c,pq) \neq 1$.

(I) We start the first case with the assumption that $\text{GCD}(c,pq) = 1$, and that c^{e^*} is in fact a solution to $x^e \equiv c \pmod{pq}$. Then we have that there exists some $k \in \mathbb{Z}$ such that

$$(c^{e^*})^e \equiv c^{1+k(p-1)(q-1)} \pmod{pq} \equiv c \cdot c^{(p-1)(q-1)^k} \pmod{pq}. \tag{2.5}$$

Euler's Theorem for pq (Hoffstein, Pipher, and Silverman [14]; Silverman [23]) gives us the reduction that,

$$c \cdot c^{(p-1)(q-1)^k} \equiv c \cdot 1^k \pmod{pq} \equiv c \pmod{pq}. \tag{2.6}$$

(II) In the second case we have that $\text{GCD}(c,pq) \neq 1$. Here we begin by using the Chinese Remainder Theorem (Hoffstein, Pipher, and Silverman [14]; Silverman [23]; Conrad [24]) which states given $\text{GCD}(p,q) = 1$ the following

holds,

$$x \equiv b \pmod{p} \wedge x \equiv b \pmod{q} \implies x \equiv b \pmod{pq}. \tag{2.7}$$

As p and q are primes they clearly will be coprime allowing us to use this theorem to say that,

$$(c^{e^*})^e \equiv c \pmod{q} \wedge (c^{e^*})^e \equiv c \pmod{p} \tag{2.8}$$

$$\implies (c^{e^*})^e \equiv c \pmod{pq}. \tag{2.9}$$

To show the sub-cases in (2.8) we can also use the fact that pq is a semi-prime. When we couple this fact with our assumption that $\text{GCD}(c, pq) \neq 1$ we have that either $\text{GCD}(c, pq) = p$ or $\text{GCD}(c, pq) = q$. Since these are symmetric cases, we will assume $\text{GCD}(c, pq) = p$. Doing this allow us to show that $(c^{e^*})^e \equiv c \pmod{p}$, as

$$\text{GCD}(c, pq) = p \implies c \pmod{p} \equiv 0 \tag{2.10}$$

To show $(c^{e^*})^e \equiv c \pmod{q}$ we use Euler's Formula for pq again. To do this we recognize that as p and q are distinct, $\text{GCD}(c, q) = 1$, giving us the following

$$\begin{aligned}(c^{e^*})^e &\equiv c^{ee^*-1}c \pmod{q} \\ &\equiv c^{h(p-1)(q-1)}c \pmod{q}, \text{ for some integer } h \\ &\equiv 1^{h(p-1)}c \pmod{q} \\ &\equiv c \pmod{q}\end{aligned}$$

Thus as $(c^{e^*})^e \equiv c \pmod{q}$ and $(c^{e^*})^e \equiv c \pmod{p}$ we have from (2.8) and (2.9) that $(c^{e^*})^e \equiv c \pmod{pq}$ when $\text{GCD}(c, pq) \neq 1$.

The third and final property that we must show is that the solution $x \equiv c^{(e^*)} \pmod{pq}$ found was unique. To show uniqueness we assume that there exists two solutions, c^{e^*} and y, such that $y \not\equiv c^{e^*} \pmod{pq}$. As we have that $e^*e = 1 + v(p-1)(q-1)$ for some integer v from (2.3), we get that,

$$y \equiv y^{e^*e - k(p-1)(q-1)} \pmod{pq}. \tag{2.11}$$

Rearranging this reveals that,

$$y \equiv y^{ee^*}(y^{(p-1)(q-1)})^{-k} \pmod{pq} \equiv c^{e^*} \pmod{pq}. \tag{2.12}$$

This gives us a contradiction of the assumption $y \not\equiv c^{e^*} \pmod{pq}$, thus proving uniqueness, and completing the proof of the proposition. □

2.4.2 Property 2: Computing Keys

To show that the second condition of Definition 2.2.1 holds for RSA, we need to be able to show that we can easily compute the public and private keys. This

entails finding three easily computable algorithms. The first algorithm should create the encryption exponent, e. The second algorithm must be responsible for creating the large and distinct primes, p, and q. The third and final algorithm is responsible for computing N from p and q. If found, we can consider the union of all three algorithms as one overarching, easily computable algorithm for computing the keys, due to the below theorem,

Theorem 2.4.2. *Complexity Class, $\mathcal{BPP}$, is closed under conjunction (Rothe [16]).*

First, we can reduce the task of creating the encryption exponent to that of creating another large prime (Menezes, et al. [19]). To perform this reduction, we will choose an e which is a prime greater than that of both p and q. This will therefore force the condition that $\text{GCD}(e, (p-1)(q-1)) = 1$. The second task, also one of generating primes, is done by generating an odd number, checking if it is a prime, and repeating the process until one has been found. As generating odd numbers is trivial, the problem moves to verifying if a number is prime in polynomial time, which we can do using the Miller-Rabin Test (Conrad [25]; Miller [26]).

This test tells us, on any given run, whether a number is composite or probably prime. Furthermore, if the number is composite, a run will confirm it is composite at least 75% of the time. Since each run is independent[†] we can assess numbers to any desired tolerance. This algorithm for primality is our first real example of a $\mathcal{BPP}$ class algorithm and why we must protect against them. Interestingly, if the Generalized Riemann Hypothesis is proven, Miller-Rabin becomes a deterministic test, and runs in the complexity class $\mathcal{P}$ (Conrad [25]). The final algorithm is trivial as we only need to multiply our values of p, and q, however to be exhaustive in supplying methods we will say we use the Furer's algorithm (Furer [27]).

2.4.3 Property 3: Encrypting a Message

To show that the third condition of Definition 2.2.1 holds for RSA, we need to present a method for easily calculating the encryption function. Recall the encryption function is of the form,

$$E(m) \equiv m^e \pmod{N}, \tag{2.13}$$

where e, m, and N are known and in $\mathbb{Z}$. This problem is commonly referred to as the modular exponentiation problem and there are many algorithms which solve this in $\mathcal{P}$, let alone in $\mathcal{BPP}$ (Menezes, et al. [19]). One of the simplest in $\mathcal{P}$ is the Fast Powering Algorithm (Hoffstein, Pipher, and Silverman [14]).

[†]Given that different supplementary values are used.

Algorithm 2.4.3 (Fast Powering Algorithm (Hoffstein, Pipher, and Silverman [14]))**.**

1. *Compute the binary expansion of* e *as*

$$e = e_0 + e_1 \cdot 2 + e_2 \cdot 2^2 + \cdots + e_r \cdot 2^r \text{ with } e_0, \ldots, e_r \in \{0, 1\}.$$

2. *Compute the powers* $m^{2^i} \pmod N$ *for* $i \leq i \leq r$.

3. *Compute* $m^e \pmod N$ *using the formula*

$$m^e \equiv m^{e_0} \cdot m^{2 \cdot e_1} \cdot m^{2^2 \cdot e_2} \cdot \ \ldots \ \cdot m^{2^r \cdot e_r}$$

This is not the most efficient algorithm for this problem but is among the most digestible as we just want to show that class $\mathcal{P}$ algorithms do exist for modular exponentiation problems. The best implementation will differ on a situational basis (Menezes, et al. [19]).

2.4.4 Property 4: Decrypting a Message with the Key

The fourth condition is similar to the third, as we just have to show existence of an easily computable algorithm for decryption. Decryption in RSA has two parts, the first of which is creating e^*. To calculate e^* we can use the Extended Euclidean algorithm (Hoffstein, Pipher, and Silverman [14]; Silverman [23]) to solve the following Diophantine equation (Silverman [23]) in $\mathcal{P}$,

$$ex + (p-1)(q-1)y = 1, \tag{2.14}$$

where x and y are unknown. The reason this is important is because $x = e^*$ as seen by below,

$$1 = ex + (p-1)(q-1)y \tag{2.15}$$
$$ex = 1 - (p-1)(q-1)y \tag{2.16}$$
$$ex \equiv 1 \pmod{(p-1)(q-1)}. \tag{2.17}$$

The second part of decryption is the decryption function,

$$D(c) \equiv c^{e^*} \pmod N \tag{2.18}$$

where e^*, c, and N are known and in $\mathbb{Z}$. The second part should seem remarkably familiar, as it is also a modular exponentiation problem, and hence can easily computed by algorithm 2.4.3. Together we can combine these algorithms to provide one which is easily computable for all of decryption from Theorem 2.4.2.

2.4.5 Property 5: Decrypting a Message without the Key

The fifth, and final, point of Definition 2.2.1 is much more difficult to show than any prior. This is because it requires a proof showing that no algorithm can

run in $\mathcal{BPP}$ which decrypts RSA without the private key; this is a proof that has never been made. This is not a problem that just RSA has, but all public key cryptosystems, as lower bounds in complexity theory are incredibly difficult to show (Hoffstein, Pipher, and Silverman [14]; Rothe [16]). In fact, it is not yet known if a true one-way function exists, and if an individual could prove its existence they would have consequently proved the famous $\mathcal{P} = \mathcal{NP}$ millennium prize problem (Hoffstein, Pipher, and Silverman [14]). To subvert this issue, we say the problem is quasi-one way and we attack the problem repeatedly to see if an algorithm can be made which runs in $\mathcal{BPP}$ (Diffie and Hellman [3]). The longer the problem can withstand these attacks, the more confidence we gain in it. This is the main strategy used by the NSA, who consistently assess cryptographic schemes sent in until they break them (NSA [12]). RSA has long been under this method of testing, and its assumption has long been held.

2.5 The Strength of RSA

The strength of a public key cryptosystem comes from the difficulty an attacker would have in deriving the private key information from the public key information. This is because if one could create an easily computable algorithm which derived the private key from the public key anybody could read the messages between participants and the security of the system instantly falls apart (Hoffstein, Pipher, and Silverman [14]; Menezes, et al. [19]). This line of attack is a typical route for Cryptanalysts to take when attempting to breaking public key cryptosystems. This is because there is a necessity that somewhere in the public key information is a clue about the private key (Menezes, et al. [19]). The reason for this is because the public key is essentially the instructions of how to build the lock that will protect the information, and in that must be the instructions of how to make the private key fit that lock. For RSA, these instructions are kept in N, and the entire strength of the system relies on the assumption that it cannot be factorized back into p and q.

This at first sounds strange, as surely the factorization of a number is simple to compute, after all we first learn of the concept in grade school. Though aside from a few basic tricks, such as checking if a number is odd or even to tell if its divisible by 2, or summing the digits to check if a number is divisible by 9, we largely skim over how we actually factorize numbers. That is because generally this process is exceedingly difficult (Hoffstein, Pipher, and Silverman [14]; Woll [28]). In fact, this is a problem that has been worked on for hundreds of years by some of the most influential mathematicians; Gauss even stated in Disquisitiones Arithmeticæ, that "The problem of distinguishing prime numbers from composites, and of resolving composite numbers into their prime factors, is one of the most important and useful in all of arithmetic" (Gauss [29]). Despite these many efforts, however, we have not been able to find an algorithm which factors a given number in polynomial time, and a lot of people did not think it was possible until Peter Shor unexpectedly announced he had.

Chapter 3

Cryptanalysis of RSA

3.1 A Change in Difficulty

At its core, a computer is a device which processes information according to a set of instructions. To do this computers encode the information they are working with into some physical property; which they are able to both read and alter (Adamatzky [30]; Perry [31]). For example, the typical household computer operates by encoding binary information into positive and negative voltages. This encoding, however, does not have to be electrically based, nor does the mapping have to be binary. This method only became standard for logistical reasons as electricity is easy to manipulate and using two states provides a greater accuracy during measurement (Glusker [32]). In theory any physical property can be used given it is observable and alterable, for example, Fluidics, is the application of fluid mechanics to perform computations (McKetta [33]).

This idea of being able to use different properties is not some new revelation, in fact Fluidics has been used since the 1930's in everything from controlling medical devices to sprinklers (Weathers [34]). Obviously, these unconventional methods of computing never truly took off to the scale of electrical computing. This is because computationally there was no difference between the machines, meaning that anything you could compute using one physical encoding scheme could be computed on all other physical encoding schemes (Perry [31]). This meant that the values of the different schemes were derived from the logistical benefits they possessed, such as reliability or durability. Of this group of different computers, which we now call classical computers, none could compete with the cost, ease, and tailorable nature of the electrical computer.

Then a computer became known with great prospects of breaking this mould. In 1980, physicist Paul Benioff published *"The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines"* (Benioff [35]). This paper was the first published description of a computer whose physical encoding was based on the revolutionary results from quantum mechanics. This idea of a quantum computer led quickly to much excitement, as the rules which govern the quantum world are fundamentally different from that of the classical world[†] (Mermin [36]). If harnessable this meant

[†]Assuming the worlds in question stay on their side of Bohr's correspondence limit.

that a quantum computer could be able to compute in ways that were not possible on a classical computer. These new abilities could incur massive changes[†] in what we had assumed possible, opening both new avenues of calculations and the ability for massive algorithmic speed-up. The quantum computer is in fact so different from the classical computer it requires its own form of complexity classes to be made. The most important of which is the class $\mathcal{BQP}$ (Rothe [16]).

Definition 3.1.1 ($\mathcal{BQP}$)**.** The complexity class $\mathcal{BQP}$, otherwise known as bounded-error quantum polynomial time, is the quantum computing analogue to $\mathcal{BPP}$. In particular, this class is one where on any given run we output a result in polynomial time, which is correct more than half of the time, and has access to the use of a quantum computer (Rothe [16]).

This new complexity class calls into question our pre-existing definitions of easy and hard calculations. These definitions had been made with classical computers in mind, however now we have a machine which can calculate in new and previously impossible ways. It would be irresponsible to not include protections from this type of machine, as it is irrelevant to us what machine an adversary is using. Rather, what we genuinely care about is if an adversary can likely decrypt our messages in polynomial time. This prompts us to make a new definition for what we consider an easy or hard calculation to be.

Definition 3.1.2. A function is defined as computationally easy if there exists an algorithm for the function which runs in the complexity class $\mathcal{BQP}$. If no such algorithm exists, it is computationally hard.

This new definition does not mean any computations once called easy will now be called hard. This is because $\mathcal{BPP} \subseteq \mathcal{BQP}$, since every computation which can be calculated on a classical computer can also be calculated on a quantum computer (Rothe [16]; Mermin [36]). This new definition does, however, give new routes to declare hard computations easy. This redefining was ultimately the fate of RSA's factorization problem.

In 1994, a computer scientist named Peter Shor was working at Bell Labs, where Koenig had implemented public key cryptography decades earlier. Shor had become enamoured with the prospects of quantum computing after watching a lecture on the subject (Horgan [37]). He soon found himself at night secretly attempting to devise $\mathcal{BQP}$ algorithms for problems that nobody had found $\mathcal{BPP}$ equivalents for. Then suddenly he emerged with an algorithm for general factorization which ran in $\mathcal{BQP}$. Shor's algorithm consists of three major parts. The first is a classical reduction of a factorization problem to that of a period finding problem via Miller's reduction. The second is a quantum algorithm which Shor used to approximate the frequency of the period finding problem. Third and finally is the use of the continued fractions algorithm to gather the period, and hence the factorization from Millers.

[†]Intentionally we avoid the word improvements here. This is because while quantum computers are with no doubt revolutionary machines, they do not improve the classical computer. To say this would be comparable to saying planes are an improvement on cars because they go faster. While true, few would trade their hatchback for a light aircraft.

3.2 Input Requirements

Shor's algorithm is typically referred to as a prime factorization algorithm for any given integer. This, however, is an incorrect assessment of Shor's for two reasons. The first is that Shor's algorithm does not provide every prime factor, rather on success it returns two general divisors of the inputted number. The second problem is that Shor's does not work for any given integer. In particular, Shor's algorithm factorizes integer inputs, N, which suffice the following requirements.

1. N must be composite.
2. N must be odd.
3. N cannot be a prime power.

So why is Shor's so typically referred to as a prime factorization algorithm, when it clearly is not?

The reason that Shor's algorithm is referenced this way is that there are already $\mathcal{BPP}$ class algorithms which resolve the cases that Shor's algorithm cannot perform. More specifically, to verify that an inputted number is in fact composite, we can use the aforementioned Miller-Rabin test (Conrad [25]; Miller [26]). To resolve if an inputted number is even, we simply divide the inputted number by 2 and identify if we are returned an integer value or not. Finally, the third problem case for Shor's is about N being a prime power. To resolve this we repeatedly execute root finding algorithms on the function $f(x) = \sqrt[k]{N} - b$, where $b \in \mathbb{Z}$ iterating over integer k such that $2 \leq k \leq \log_2 N$ (Menezes, et al. [19]). While this is one of the more arduous checks, these root-finding algorithms run in $\mathcal{P}$ (Menezes, et al. [19]). Later, Daniel Bernstein gave an algorithm in *"Detecting Perfect Powers in Essentially Linear Time"* which performed this task efficiently and firmly in $\mathcal{P}$ (Bernstein [38]).

With all these cases covered, we can create a method to gather the prime factorization of any integer value, by creating the General Divisor Sub-function. This sub-function will be able to take any general integer and divide it into two smaller factors of the inputted number. We then execute this sub-function repeatedly on a larger factorization function which identifies which divisors must themselves be sub-divided. Since, on every execution of the sub-function we always split N or terminate, without losing factors, eventually we will be able to achieve the prime factorization of any number. Figures 3.1 and 3.2 display this property graphically, structuring the sub-function beginning with the most efficient algorithms.

When discussing Shor's algorithm regarding the factorization of RSA numbers this becomes simpler. RSA numbers as we recall, are semi-primes whose factors are large and unique, because of this we know that all RSA numbers must trivially satisfy all three conditions. Further, as we have that these are semi-primes, on Shor's success the result must be the prime factorization of N, p and q, as there are no other factorizations to give.

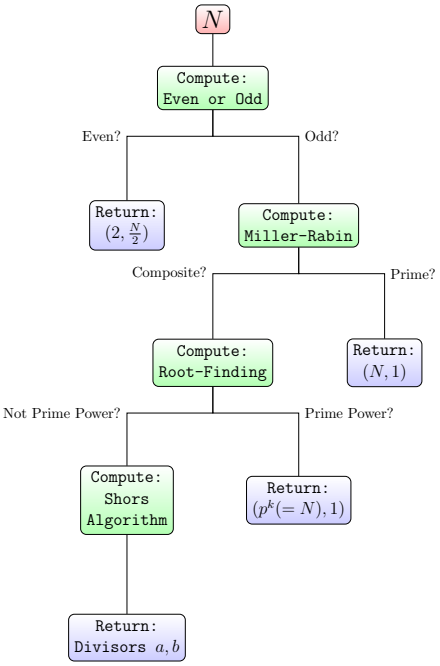

Figure 3.1: General Divisor Sub-function

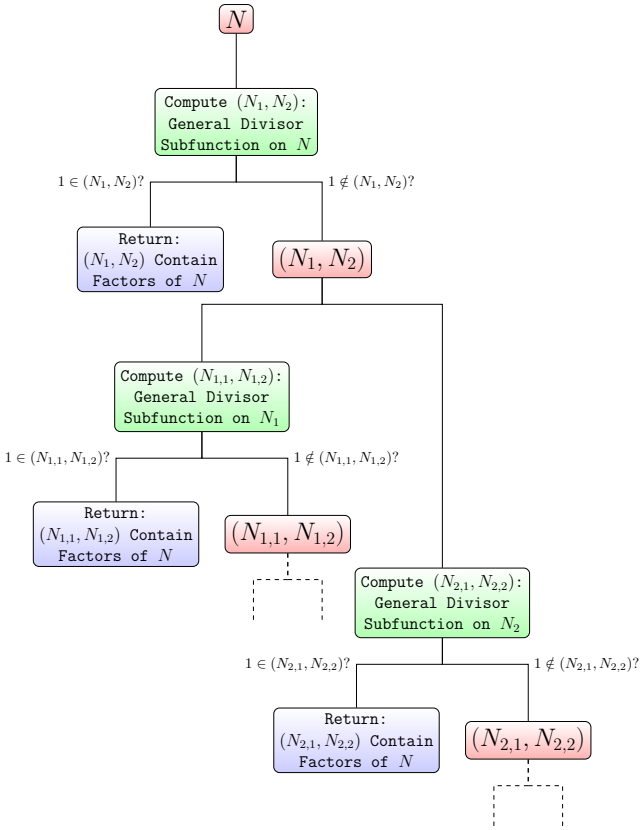

Figure 3.2: Prime Factorization Function

3.3 Millers Reduction

Shor's algorithm begin with a reduction of the factorization problem to one of the period finding problem. Shor himself did not create this reduction, as it was actually discovered[†] by Gary Miller[†], who first gave the reduction in 1976 in a paper titled *"Riemann's Hypothesis and Tests for Primality"* (Miller [26]). Today, Shor states that he is unsure if he re-discovered this reduction himself or if Miller had informed it in conversation (White [39]). In any case however, the reduction is performed as such.

We begin taking our number to be factorized, N, and choosing a random $x \in \mathbb{N}$, such that $x < N$. Our hope for x is that it will be an integer multiple of a factor of N. If this is the case then we will have immediately divided N, as if $x = k \times p$ for some integers p and k such that $p \mid N$ and $k < \frac{N}{p}$ we have through the Euclidean algorithm that,

$$\mathrm{GCD}(x, N) = p. \tag{3.1}$$

This use of the Euclidean algorithm rather than directly attempting to guess a factor works significantly to our benefit as it greatly increases the number of guesses which lead us to a division of N. More specifically, for general N with

[†]Or invented depending on the readers philosophical subscriptions.

[†]The same Miller as that in the Miller-Rabin Test

a prime factorization of $N = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \cdots \cdot p_n^{\alpha_n}$ the number of possible correct guesses increases approximately in the range of $\sqrt{N}$ to $N - \sqrt{N}$.

Proof. Suppose we have a general composite number $N \in \mathbb{Z}$. Since we know that N will have a unique prime factorization, we can say that,

$$N = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \cdots \cdot p_n^{\alpha_n}. \tag{3.2}$$

From this we can discard the repeated values to get the set of prime values with divide N, well call this set $\mathbf{P} = \{p_1, p_2, \ldots, p_n\}$. If we directly wanted to guess a factor there are clearly n the possible correct guesses, as $|\mathbf{P}| = n$.

To calculate the number of unique integer multiples, we can use the fact that Euler's Totient function (Silverman [23]), $\varphi(n)$, gives a closed form calculation for the number of co-prime elements less than n. Clearly we then have that $N - \varphi(N)$ gives us the number of unique integer multiples inclusive of N, which we discount as we said our guess, x, was less than N. This gives the number of correct guesses possible using the Euclidean algorithm is equivalent to,

$$N - \varphi(N) - 1 = (N - 1) - N \prod_{i=1}^{n} (1 - \frac{1}{p_i}). \tag{3.3}$$

To see the difference we have lost between the two guesses we use the property that $\sqrt{N} \leq \varphi(N)$ (Kendall and Osborn [40]; Mitrinović and Sándor [41]) and $\varphi(N) \leq N - \sqrt{N}$ for composite n (Sierpiński and Schinzel [42]; Mitrinović and Sándor [41]). With this we can bound our gained number of guesses, D, from above as,

$$D = N - \varphi(N) - 1 - n \leq N - \sqrt{N} - 1 - n, \tag{3.4}$$

and below as,

$$D = N - \varphi(N) - 1 - n \geq \sqrt{N} - 1 - n. \tag{3.5}$$

This can be further simplified using the property that the unique number of prime factors for a given number, N, is bounded above by $\frac{\log(N)}{\log(\log(N))}$, and below by 1 (Hardy and Wright (Hardy and Wright [43])). Implementing these for n gives us the bounds,

$$\sqrt{N} - 2 \leq D \leq N - \sqrt{N} - 1 - \frac{\log(N)}{\log(\log(N))} \tag{3.6}$$

As N becomes large we can approximate the bounds as,

$$\sqrt{N} \leq D \leq N - \sqrt{N}. \tag{3.7}$$

□

In the chance that our guessed x is not a multiple of a factor of N[†]. We can use x to produce a new and much better guess. To do this we first need to find the order of our chosen x in modulo N, recalling we define this as,

[†]Which is very likely for RSA numbers as (3.3) gives us that there is only $\frac{p+q-2}{N}\%$ chance a given number is an integer multiple of a factor.

Definition 3.3.1 (Order of Element Modulo N)**.** The order of an element x modulo N is the smallest positive integer, r, satisfying $x^r \equiv 1 \pmod{N}$. If no such r exists we say x has infinite order Modulo N (Hoffstein, Pipher, and Silverman [14]).

The reason that we want to find this order, r, is that with it we can perform the following manipulation,

$$x^r = 1 \pmod{N} \tag{3.8}$$
$$x^r - 1 = 0 \pmod{N} \tag{3.9}$$
$$\implies x^r - 1 = kN \quad \text{for some } k \in \mathbb{N} \tag{3.10}$$
$$(x^{\frac{r}{2}} + 1)(x^{\frac{r}{2}} - 1) = kN. \tag{3.11}$$

This resulted splitting looks very promising as given we can achieve it and the prime factorization of N not wholly contained in either $(x^{\frac{r}{2}} + 1)$ or $(x^{\frac{r}{2}} - 1)$ it is a necessity that both $\mathrm{GCD}(x^{\frac{r}{2}} - 1, N)$ and $\mathrm{GCD}(x^{\frac{r}{2}} + 1, N)$ reveal factors of N. This begs the question of how often does this splitting not work.

The first problem point is obviously when x has infinite order modulo N. Luckily, we know that this will never happen as since we have that $\mathrm{GCD}(x, N) = 1$, by Euler's Theorem we have that

Theorem 3.3.2 (Euler's Theorem (Hoffstein, Pipher, and Silverman [14]))**.** *For $N \in \mathbb{N}$ such that $2 \leq N$, and any $x \in \mathbb{N}$ such that $\mathrm{GCD}(x, N) = 1$, we have that*

$$x^{\varphi(N)} \equiv 1 \pmod{N}. \tag{3.12}$$

Where $\varphi()$ is the Euler totient function.

This implies that the order of x exists and bounded above by the value $\varphi(N)$. The second and third problems are respectively that r isn't even, and one of the factors fully contains the factorization of N. At first these two problems might not seem connected, but at their core they are, and we can show that either of these problems occur with a probability of $(\frac{1}{2})^{k-1}$ where k is the number of prime factors of N (Mermin [36]).

Proof. Suppose that $N = \prod_{i=1}^{k} p_i^{a_i}$. Since $x^r \equiv 1 \pmod{N}$ we have that $x^r \equiv 1 \pmod{p_i^{a_i}}$. r is not necessarily the order of x for these problems however, just that $r = \mathrm{LCM}(r_1, r_2, \ldots, r_k)$, where r_i denotes the order of x modulo $p_i^{a_i}$. Consider the largest power of two which then divides each of these r_i, we will call this value n_i. The only way for r to be odd is if all these $n_i = 0$. Further, the only way to contain the entire factorization of N in either $(x^{\frac{r}{2}} + 1)$ or $(x^{\frac{r}{2}} - 1)$, is to have $(x^{\frac{r}{2}} + 1) \equiv 0 \pmod{N}$, as $(x^{\frac{r}{2}} - 1) \equiv 0 \pmod{N}$ would contradict the fact that r is the order of x modulo N. For $(x^{\frac{r}{2}} + 1) \equiv 0 \pmod{N}$ we require $(x^{\frac{r}{2}}) \equiv -1 \pmod{p_i^{a_i}}$ for all i. This occurs when n_i is equivalent for all i and greater than 0. Thus, we find that our reduction only fails in general when n_i agrees for all i.

We have from the Chinese remainder Theorem that choosing $x \pmod{N}$ at random is equivalent to choosing $x_i \pmod{p_i^{a_i}}$ and taking $x = \prod_{i=1}^{k} x_i$. Since we have that $2 \nmid N$, we can use the fact that the multiplicative group for any

odd prime power, $p_i^{a_i}$, is cyclic (Mermin [36]) thus the probability of choosing any particular power of two to be the largest that divides r_i is $\frac{1}{2}$. Thus, the probability that the all these largest powers of two agree is $(\frac{1}{2})^{k-1}$. This is also the reason we do not let N be a prime power as if we did the probability of failure certain. □

In the case that this x fails in either of these ways, we simply begin again choosing a new x-guess. Amazingly, this reduction implies that even on the most secure RSA number, by taking 10 guesses of x we would have had a 99.9% chance of cracking the scheme.

Algorithm 3.3.3 (Miller's Reduction (Miller [26])).

Input: $N \in \mathbb{N}$ *such that* N *is composite, odd, and not a prime power.*

1. *Choose a random* $x \in \mathbb{N}_{<N}$ *and calculate* $\mathrm{GCD}(x, N)$ *if this is does not equal 1, then we have found a factor of* N *through* x.
2. *Find the order* r *in the problem* $x^r \equiv 1 \pmod{N}$.
3. *If* r *is odd discount this* x *and start from step 1.*
4. *If* $x^{\frac{r}{2}} \equiv -1 \pmod{N}$ *discount this* x *and start from step 1.*
5. *Gather factors* $p, q = \mathrm{GCD}(x^{\frac{r}{2}} \pm 1 \pmod{N}, N)$.

This shifts the problem of factorization to that of calculating the order of x modulo N. This problem is actually a specific case of the discrete logarithm problem as we are trying to identify $r \equiv \log_x 1 \pmod{N}$ for the smallest r. There are some $\mathcal{P}$ complexity algorithms for solving certain classes of this problem, such as Shanks Giant Baby Step algorithm (Hoffstein, Pipher, and Silverman [14]). These, however, typically depend on either N being prime or knowing the factorization of N. These are not things that we can obviously provide, in fact the fastest known method is one produced by Leonard Adelman in a paper called *"A subexponential Algorithm for discrete logarithms over all finite fields"* (Adleman and Demarrais [44]). This algorithm runs outside of $\mathcal{BQP}$ as it is not polynomially bounded. In-fact this is in the same complexity class as the current most efficient classical factorization algorithm, the General Number Field Sieve (Hoffstein, Pipher, and Silverman [14]).

3.4 Finding the Period

To solve the problem of finding the order of an integer number x modulo N, Shor used the property that $f(y) = x^y \pmod{N}$ is periodic. Moreover, Shor noticed that finding the length of the period modulo N is equivalent to finding the order modulo N. Shor realised that this periodicity could help us to find the order, r, because if we can find a way to produce a large amount of values of the function $f(y) = x^y \pmod{N}$ we can use a special type of Fourier Transformation called the Discrete Fourier Transform (Walker [45]) which runs in $\mathcal{P}$, to approximate the period.

There is an issue however. While it is indeed true that the Discrete Fourier runs in $\mathcal{P}$, we have the issue of generation enough values of $f(y) = x^y \pmod N$ to gather an accurate enough approximation (Perry [31]). To even run the transform would require at least an entire period worth of samples, at which point if we could calculate that why not just choose the first value congruent to 1. This finally is what Shor's quantum computation is about, massive value calculation (Perry [31]). Shor's uses the quantum computer to calculate not just a full period of values, but N full periods. Why Shor can do this requires the explanation of a quantum computer.

3.5 Quantum Computers

Quantum computers, as previously stated, are fundamentally different to classical computers. The reasons for this are plentiful, however there is one difference that is the most imperative to communicate to the reader, that is the principle of Superposition. Superposition is a fundamental property in quantum mechanics which allows a system to be in more than one state at a given time, this property follows into quantum computations (Mermin [36]). To explain, on a classical computer a bit can either be in the state 1 or 0. On a quantum computer however a qubit, the quantum analogue to a bit, does not have to be in either state 1 or 0. It can also be in Superposition between the states, essentially meaning that the qubit is in both states at once. Crucially however, when we observe a qubit in superposition, it will then suddenly collapse to either the state of 1 or 0. How the qubit collapses depends on the superposition, this is because (with great simplification) there is a corresponding probability that can be manipulated for how the qubit collapses[†].

Importantly, Superposition is also scalable, meaning that on a classical computer if there were 3 binary bits being used in conjunction, they could only be in one of the 8 states,

$$\boxed{000}, \boxed{001}, \boxed{010}, \boxed{011}, \boxed{100}, \boxed{101}, \boxed{110}, \boxed{111}.$$

Which could be used to denote the integer values,

$$\boxed{0}, \boxed{1}, \boxed{2}, \boxed{3}, \boxed{4}, \boxed{5}, \boxed{6}, \boxed{7}.$$

The quantum computer however can have each of their three qubits be in superposition. Since, each qubit will have its own manipulable probabilities, we can use basic probability theory to say that globally the collections of qubits are in superposition of the eight integer values. We would represent this as,

$$\alpha_0\boxed{0} + \alpha_1\boxed{1} + \alpha_2\boxed{2} + \alpha_3\boxed{3} + \alpha_4\boxed{4} + \alpha_5\boxed{5} + \alpha_6\boxed{6} + \alpha_7\boxed{7},$$

[†]It's important to make the distinction here that the qubit wasn't "actually" in a state with some probability, rather the qubit has some probability of being in a given state once actualized (Mermin [36]).

where α_i is the probability[†] that upon observation the qubit will collapse to the corresponding state.

This property is useful because while qubits are under superposition they still can be calculated on. This is incredibly handy as if we wanted to apply a function to all possible configurations of the integers 0 to 7, on a classical computer we would have to interact with 3 bits, 8 different times. With a quantum computer however we only must do it with 3 qubits which are under superposition one time. We do have to remember however that if we then wanted to observe these configurations, the quantum computer must collapse giving us only one state back, while the rest of the information is lost forever (Mermin [36]).

This at first seems quite frustrating, and truthfully it is, though it can still be extremely useful. This is because if we can smartly manipulate these probabilities, it is possible to coerce the qubits to result back interesting answers (Mermin [36]). This is essentially what Shor did, he used this property of superposition on a large collection of k qubits which allowed him to simultaneously calculate huge amounts of values for the function $f(y) = x^y \pmod N$. Shor then manipulated the probabilities of the superposition so that when it was collapsed it returned with high probability an integer λ which is within $\frac{1}{2}$ of an integer multiple of $\frac{2^k}{r}$, where k still denotes the number of qubits used and r denotes the period of the function (Mermin [36]). Finally, Shor used the continued fractions algorithm to extract this value of r, slaying the factorization problem.

3.6 Discrete Fourier Transform

Typically, when we want to find the period of a given set of discrete data points, we find the frequency and extrapolate the period from it. To do this in classical problems we tend to use the Discrete Fourier Transformation (Walker [45]). There is a computational speed-up of the discrete Fourier transformation called the Fast Fourier Transformation, but for our purposes we shall keep to the more straightforward predecessor. In essence, the Discrete Fourier Transform maps a periodic sequence of k values, $\{x_0, x_1, \ldots, x_{k-1}\}$, to another set of k values, $\{c_0, c_1, \ldots, c_{k-1}\}$, which represent how well each x_i conforms to the frequency of that the discrete sequence repeats (Perry [31]). More formally, we define the Discrete Fourier Transform as follows,

Definition 3.6.1 (Discrete Fourier Transform)**.** Given a set of k complex numbers $x_0, \ldots, x_{k-1}$ we can transform them to a set of k complex numbers $c_0, \ldots, c_{k-1}$, representing their frequency, where

$$c_j = \sum_{n=0}^{k-1} x_n e^{-2\pi i \frac{nj}{k}} \qquad j = 0, \ldots, k-1.$$

We gather this transformation directly from using left-Riemann sum approximations on the standard Fourier Series (Walker [45]).

[†]such that $\sum_{i=0}^{7} \alpha_i = 1$

In Shor's we use the quantum analogue of this to manipulate the probabilities of the qubits (Shor [13]). This is done by first creating a list of input values in superposition spanning from 1 to N^2. We then calculate these values under $f(y) = x^y \pmod N$ and partially collapse them[†], resulting a superpositioned list where all the input values that output a specific value have some equal probability, and all other input values have no probability. We then apply the Discrete Fourier Transformation's analogue on these probabilities, resulting a superposition of possible frequency values, where high probabilities lie on values which accurately described the rate at which identical outputs occurred (Mermin [36]).

This may be hard to grasp, so we will run through an example. Suppose we had arrived at this point with $N = 21$ and $x = 11$. We begin by calculation a list of input values at least the size of 512, as it is the largest power of 2 which fully contains N^2, we call this register 1, R_1.

R_1	0	1	2	3	4	5	6	7	8	...	511

We then apply register 1 to the function $f(i) = 11^i \pmod{21}$, and place the values in a second register, R_2. These registers have been set up in a way so that they are actually part of one larger list but for clarity's sake it is simpler to think of them as two separate lists. This does however mean that under superposition they have the same associated probability under observation, we will write the probability for each state as P,

R_1	0	1	2	3	4	5	6	7	8	...	511
R_2	1	11	16	8	4	2	1	11	16	...	8
P	0.19%	0.19%	0.19%	0.19%	0.19%	0.19%	0.19%	0.19%	0.19%	...	0.19%

We then look at just R_2, this collapses it transforming us to the following.

R_1	0	1	2	3	4	5	6	7	8	...	511
R_2	1	11	16	8	4	2	1	11	16	...	8
P	0%	1.17%	0%	0%	0%	0%	0%	1.17%	0%	...	0%

It is these P values that we preform out discrete Fourier Transformation on outputting us a superpositioned list of possible frequencies with an associated probability. We represent this in our example as Figure 3.3.

The peaks that we identify in this figure are the locations of frequencies which conformed well to the earlier superposition. This itself is in superposition however, meaning that once we try to see what these peaks are, it will collapse. This collapse will most likely occur however at or around the frequencies and if we wanted to improve the chances it does we simply have to increase the number of input values that we originally had done. The reason there are multiple peaks is because we are also being resulted the multiples of the true frequency of the data as well. The problem thus only become ones of extracting the period from a decimal representation of the frequency or integer multiple of the frequency. This is done completely classically using the Continued Fractions algorithm

[†]This can be done by only observing a subset of the qubits

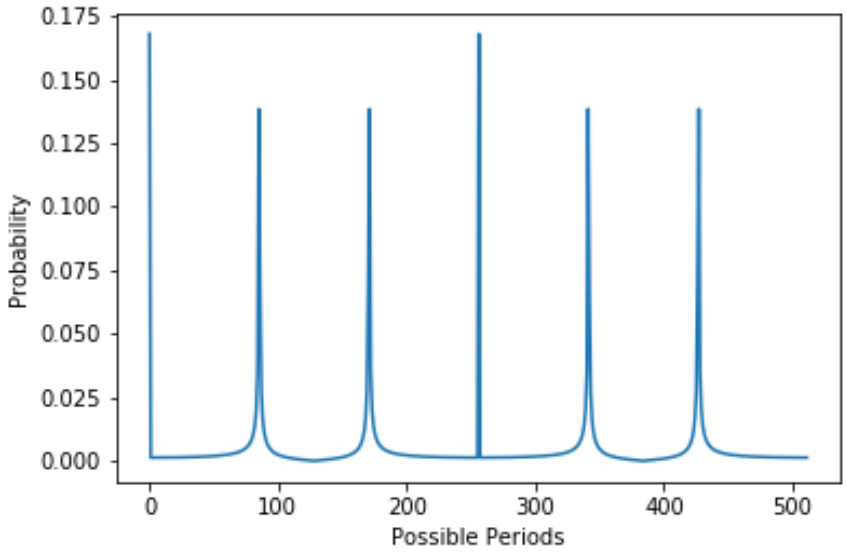

Figure 3.3: Qubit Probability Representation after Fourier Transform

3.7 Continued Fractions

The continued fractions algorithm (Silverman [23]; Hardy and Wright [43]) is a method of representing any value in $\mathbb{R}$, as a sequence of integers. This algorithm as it turns out is exceedingly good at finding fractional approximations for numbers, such as π's famous approximate of $\frac{22}{7}$. For our purposes however, we will focus on this algorithm's representation of rational elements of $\mathbb{R}$. To perform the continued fractions algorithm on rational values we simply expand the element into its mixed fractional form, take the reciprocal of the remaining proper fraction and repeat until there is no proper fraction to take the reciprocal of. For clarity, say we wanted to compute the continued fraction of $\frac{11}{9}$, this would appear as:

$$\frac{11}{9} = 1 + \frac{2}{9} = 1 + \frac{1}{(\frac{9}{2})} = 1 + \frac{1}{4 + \frac{1}{2}} = 1 + \frac{1}{4 + \frac{1}{(\frac{2}{1})}} = 1 + \frac{1}{4 + \frac{1}{2+0}} \tag{3.13}$$

As all the numerators of this representation are 1, it is common to denote the continued fraction as an array. In this array, we separate all elements by commas except the first which is separated by a semi-colon. If we wished to display (3.13) in this form we would have $\frac{11}{9} = [1; 4, 2]$ (Silverman [23]; Hardy and Wright [43]). The reason that we need to understand the continued fractions algorithm is that not only is it good at making rational approximations, but it can also tell us what simple rational values lie near more complex ones. More specifically, we have by theorem that certain values are represented in the continued fractions of their neighbours (Hardy and Wright [43]).

Theorem 3.7.1 ((Hardy and Wright [43])). *If x is an estimate for a fraction $\frac{j}{k}$ that differs from it by less than $\frac{1}{2k^2}$ then $\frac{j}{k}$ will appear in the continued fraction algorithm for x.*

This can be used to find the true value of r as if we have our approximation from the quantum Fourier transformation,λ be within $\frac{1}{2}$ of an integer multiple of the true frequency, we get that,

$$\left|\lambda - \frac{m2^k}{r}\right| < \frac{1}{2} \tag{3.14}$$

for some integer m. From this we can derive that,

$$\left|\frac{\lambda}{2^k} - \frac{m}{r}\right| < \frac{1}{2^{k+1}}. \tag{3.15}$$

Since earlier we made sure to choose k to be the largest power of 2 which is greater than N^2 (3.15) must suffice Theorem 3.7.1, and the continued fraction representation of $\frac{m}{r}$ must be represented in the continued fraction of $\frac{\lambda}{2^k}$. This value can then be extracted in the computational class $\mathcal{P}$, by simply discarding the end most value of the continued fraction and testing if it results the period value r. With this value r now shown to be obtainable in the computational class $\mathcal{BQP}$ we have shown that RSA is computationally easy for Eve to decrypt without the private key.

Chapter 4

Conclusion

Peter Shor and the fall of RSA denote a clear shift in cryptography similar to that of Alan Turing and the fall of the enigma machine. In both cases a method of encryption fell primarily due to revolutionary changes in computing abilities, raising the bar on what it means for a cryptographic scheme to be unbreakable. It is now the duty of the cryptographers to defend our information against this quantum revolution. Thankfully however, cryptographers have long seen what was to come on the horizon and have been preparing for the change.

In the 1970's the unpublished work of Jerome Wiesner's son, Steven Wiesner, began circulating while he finished his graduate program at Columbia University. This work would go on to lay the groundwork of quantum resistant cryptography (Khan [2]). Though while we have made great strides in protecting ourselves in most cryptographic areas, via the creation of new methods or the bolstering of old, today there still is a large gap in our defences (Rieffel, Eleanor, and Wolfgang [46]). There is currently no provably quantum resistant public key cryptosystem at our disposal. Methods have been proposed such as using Latticed based systems, and while these have appeared to be resistant so far, this is only conjecture (Hoffstein, Pipher, and Silverman [14]). In truth the road ahead in cryptography is once again uncertain as we march into this quantum world, with the exception that we know we cannot go back.

Bibliography

[1] J. H. Ellis *The Story of Non-Secret Encryption.* FOIA Case # 19136, http://cryptome.org/nsa-nse/nsa-nse-03.pdf

[2] David Khan. *The Codebreakers: A Comprehensive History of Secret Communication from Ancient Times to the Internet, Revised and Updated.*, Scribner, 1996.

[3] W. Diffie and M. Hellman. *New directions in cryptography.* IEEE Transactions on Information Theory, vol. 22, no. 6, pp. 644-654, November 1976.

[4] P. Wayne. *British Documents Outlines Early Encryption Discovery.* New York Times, December, 24, 1997. https://archive.nytimes.com/www.nytimes.com/library/cyber/week/122497encrypt.html

[5] Weadon, Patrick D. *National Security Agency Central Security Service, About Us, Cryptologic Heritage, Historical Figures and Publications, Publications, WWII.* NSA. www.nsa.gov/about/cryptologic-heritage/historical-figures-publications/publications/wwii/sigsaly-story/.

[6] W. Koenig Jr. *Final Report on Project C-43: Continuation of Decoding Speech Codes Part 1 - Speech Privacy Systems - Interception, Diagnosis, Decoding, Evaluation.* National Defense Research Committee Office of Scientific Research and Development, Bell Telephone Laboratories. https://apps.dtic.mil/dtic/tr/fulltext/u2/a800206.pdf

[7] T. R. Johnson. *United States Cryptologic History: (U) American Cryptology During the Cold War, 1945-1989. Book 1: The Struggle for Centralization 1945-1960.* Center for Cryptologic History, National Security Agency, 1998.

[8] *Newly released GCHQ files: UKUSA Agreement* The National Archives, 13 June 2008, www.nationalarchives.gov.uk/ukusa/.

[9] J. Young. *Series of Letters to NSA.* http://cryptome.org/nsa-foia-recap.htm

[10] J. B. Wiesner. *National Security Action Memorandum No.160.* The White House. https://www.cs.columbia.edu/~smb/nsam-160/nsam-160.pdf

[11] S. M. Bellovin. *A Pre-History of Public Key Cryptography.* Columbia University. https://www.cs.columbia.edu/~smb/nsam-160/

[12] Various. *Cryptolog.* National Security Agency Central Security Service.

[13] Peter Shor. *Algorithms for quantum computation: discrete logarithms and factoring.* Proceedings 35th Annual Symposium on Foundations of Computer Science. IEEE Comput. Soc. Press: 124–134. doi:10.1109/sfcs.1994.365700. ISBN 0818665807.

[14] Jeffrey Hoffstein, Jill Pipher & Joseph H. Silverman. *An Introduction to Mathematical Cryptography*, Springer, 2008.

[15] A. Cobham. *The intrinsic computational difficulty of functions.* I.B.M. Research Center, Yorktown Heights, N.Y., USA.

[16] J. Rothe. *Complexity Theory and Cryptology: an Introduction to Cryptocomplexity: Foundations of Complexity Theory.* Springer, 2005.

[17] Stephen A. Cook. *An overview of computational complexity.* Commun. ACM 26, 6 (June 1983), 400–408. DOI:https://doi.org/10.1145/358141.358144

[18] R. Rivest, A. Shamir, L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Communications of the ACM. 21 (2): 120–126. CiteSeerX 10.1.1.607.2677. doi:10.1145/359340.359342

[19] Menezes, A. J., et al. *Handbook of Applied Cryptography.* CRC, 2001.

[20] E. Barker, A. Roginsky. *Transitioning the Use of Cryptographic Algorithms and Key Lengths.* NIST Special Publication 800-131A Revision 2. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf

[21] *"What Is Unicode?"*. Unicode. www.unicode.org/standard/WhatIsUnicode.html.

[22] Mackenzie, Charles E. (1980). *Coded Character Sets, History and Development.* The Systems Programming Series (1 ed.). Addison-Wesley Publishing Company, Inc.

[23] Silverman, Joseph H. *A Friendly Introduction to Number Theory.* Pearson, 2018.

[24] K. Conrad. *The Chinese Remainder Theorem.* Expository Papers, https://kconrad.math.uconn.edu/blurbs/ugradnumthy/crt.pdf.

[25] K. Conrad. *The Miller-Rabin Test.* Expository Papers, https://kconrad.math.uconn.edu/blurbs/ugradnumthy/millerrabin.pdf

[26] Miller, Gary L. (1976), *Riemann's Hypothesis and Tests for Primality*, Journal of Computer and System Sciences,

[27] M. Fürer (2007). *Faster Integer Multiplication* Proceedings of the 39th annual ACM Symposium on Theory of Computing, and SIAM Journal on Computing, Vol. 39 Issue 3, 979–1005, 2009.

[28] Heather Woll. *Reductions among number theoretic problems.*, Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195 USA doi:10.1016/0890-5401(87)90030-7.

[29] Carl Friedrich Gauss, *Disquisitiones Arithmeticae*, Yale University Press, 1965

[30] Adamatzky, Andrew. *Unconventional Computing.* International Journal of General Systems, vol. 43, no. 7, 2014, pp. 671–672., doi:10.1080/03081079.2014.927149.

[31] Perry, Riley Tipton *Quantum Computing from the Ground Up.*, World Scientific, 2012.

[32] Glusker, Mark. *The Ternary Calculating Machine of Thomas Fowler.* 37th International Symposium on Multiple-Valued Logic (ISMVL'07), 2007, doi:10.1109/ismvl.2007.56.

[33] McKetta, John *Encyclopedia of Chemical Processing and Design: Volume 23 – Fluid Flow.* CRC Press. p. 28. ISBN 9780824724733.

[34] Terry M. Weathers. *NASA CONTRIBUTIONS TO FLUIDIC SYSTEMS.* NASA SP-5112 https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19730002533.pdf

[35] Benioff, Paul (1980). *The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines.* Journal of Statistical Physics.

[36] Mermin, N. David *Quantum Computer Science: an Introduction.*, Cambridge University Press, 2016.

[37] Horgan, John. *Quantum Computing for English Majors.* Scientific American Blog Network, Scientific American, 20 June 2019, blogs.scientificamerican.com/cross-check/quantum-computing-for-english-majors/.

[38] Bernstein, Daniel. *"Detecting perfect powers in essentially linear time."* Mathematics of Computation of the American Mathematical Society 67.223 (1998): 1253-1283

[39] White, Philip. *Was the Reduction in Shor's Algorithm Originally Discovered by Shor?* Theoretical Computer Science Stack Exchange, 25 July 2015, cstheory.stackexchange.com/questions/25512/was-the-reduction-in-shors-algorithm-originally-discovered-by-shor/25513#25513.

[40] Kendall, D. G. and Osborn, R. *Two Simple Lower Bounds for Euler's Function.* Texas J. Sci. 17, 1965.

[41] Mitrinović, D. S. and Sándor, J. *Handbook of Number Theory.* Dordrecht, Netherlands: Kluwer, 1995.

[42] Sierpiński, W. and Schinzel, A. *Elementary Theory of Numbers, 2nd Eng. ed.* Amsterdam, Netherlands: North-Holland, 1988

[43] Hardy, Thomas, and E. M. Wright *An Introduction to the Theory of Numbers.*, Oxford University Press, 1960.

[44] Adleman, Leonard M., and Jonathan Demarrais. *A Subexponential Algorithm for Discrete Logarithms over All Finite Fields.* Advances in Cryptology — CRYPTO' 93 Lecture Notes in Computer Science, pp. 147–158., doi:10.1007/3-540-48329-2_13.

[45] Walker, James S. *Fast Fourier Transforms: Second Edition.* CRC, 1996.

[46] Rieffel, Eleanor, and Wolfgang Polak. *Quantum Computing: a Gentle Introduction.* The MIT Press, 2014.

Appendices

Appendix A

Code for RSA and Shor's

In this appendix we have attached the relevant code to allow the reader to both create their own RSA protected messages and then decrypt them with a classical formulation of Shor's decryption algorithm.

RSAEncryptionDecryption.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 25 17:45:16 2020

@author: jonny
"""

#Preamble:

import random

#Finding GDC with Euclids algorithm:

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

```
1
2  #Finding the multiplicative inverse with Euclids extended ...
       algorithm:
3
4  def multiplicative_inverse(e, phi):
5      d = 0
6      x1 = 0
7      x2 = 1
8      y1 = 1
9      temp_phi = phi
10     while e > 0:
11         temp1 = temp_phi//e
12         temp2 = temp_phi-(temp1*e)
13         temp_phi = e
14         e = temp2
15         x = x2-(temp1*x1)
16         y = d-(temp1*y1)
17         x2 = x1
18         x1 = x
19         d = y1
20         y1 = y
21         if temp_phi == 1:
22             return d + phi
23
24 def is_prime(num):
25     if num == 2:
26         return True
27     if num < 2 or num % 2 == 0:
28         return False
29     for n in range(3, int(pow(num,0.5))+2, 2):
30         if num % n == 0:
31             return False
32     return True
33
34
35 def generate_keypair(p, q):
36     if not (is_prime(p) and is_prime(q)):
37         raise ValueError('Both numbers must be prime.')
38     elif p == q:
39         raise ValueError('p and q cannot be equal')
40     n = p * q
41     phi = (p-1) * (q-1)
42     e = random.randrange(1, phi)
43     g = gcd(e, phi)
44     while g != 1:
45         e = random.randrange(1, phi)
46         g = gcd(e, phi)
47     d = multiplicative_inverse(e, phi)
48     return ((e, n), (d, n))
```

```
1
2  def encrypt(pk, plaintext):
3      key, n = pk
4      cipher = [pow(ord(char), key, n) for char in plaintext]
5      return cipher
6
7  def decrypt(pk, ciphertext):
8      key, n = pk
9      plain = [chr(pow(char, key, n)) for char in ciphertext]
10     return ''.join(plain)
11
12
13 p = int(input("Enter a prime number: "))
14 q = int(input("Enter another, larger prime number: "))
15 print("Generating your public/private keypairs now . . . ")
16 private, public = generate_keypair(p, q)
17 print("Your public key is ", public ," and your private key ...
       is ", private)
18 message = input("Enter a message to encrypt with your public ...
       key: ")
19 encrypted_msg = encrypt(public, message)
20 print("Your encrypted message is: ")
21 print(''.join(map(lambda x: str(x), encrypted_msg)))
22 PrivDecrypt=input("Would you like to see the decryption using ...
       the Private Key? (Y/N)")
23 if PrivDecrypt=="Y":
24     print("\nDecrypting message with private key ", private ...
           ," . . . ")
25     print("Your message is:")
26     print(decrypt(private, encrypted_msg))
27 elif PrivDecrypt=="N":
28     pass
29 else:
30     print("\nThis was not a valid response, assuming private ...
           key decryption is wanted.")
31     print("Decrypting message with private key ", private ," ...
           . . . ")
32     print("Your message is:")
33     print(decrypt(private, encrypted_msg))
34
35 #Adapted from https://gist.github.com/JonCooperWorks/5314103
```

QuantumAnalouge.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb  9 13:19:48 2020

@author: jonny
"""

import numpy as np, matplotlib.pyplot as plt, math

def order(x,N,c,t):
    A=range(t)
    B=[]
    for i in range(t):
        b=x**A[i]%N
        if b == c:
            B.append(1)
        else:
            B.append(0)
    fourier=np.fft.fft(B)
    fourier1 = [abs(ele) for ele in fourier]
    fourier1 = [float(ele)/sum(fourier1) for ele in fourier1]
    fourier2=sum(fourier1)
    c=np.random.choice(A, p=fourier1)
    fig,ax=plt.subplots()
    ax.plot(np.arange(len(B)), (np.abs(fourier))/t)
    ax.set_ylabel('Probability')
    ax.set_xlabel('Possible Periods')
    return B,fourier, plt,c,fourier1,A,fourier2
B,fourier,plot,c,fourier1,A,f2=order(11,21,11,512)
plt.savefig('This_fig')
```

ContinuedFrac.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb  9 14:47:51 2020

@author: jonny
"""
import math

def contFrac(x, k):
    cf = []
    q = math.floor(x)
    cf.append(q)
    x = x - q
    i = 0
    while x != 0 and i < k:
        q = math.floor(1 / x)
        cf.append(q)
        x = ((1/x)-q)
        i = i + 1
    return cf

A=contFrac(4915/8192,9)
print(A)
```