An Interactive Model To Show How A Quantum Computer Can Break RSA Encryption

Stanley Hitchcock
Supervisor: Dr. Steve Vickers

Submitted in conformity with the requirements
for the degree of MSc Computer Science
School of Computer Science
University of Birmingham

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Signed ....

Abstract

**An Interactive Model To Show How A
Quantum Computer Can Break RSA Encryption**

Stanley Hitchcock

Due to the complicated nature of a quantum computer, it is often difficult to understand how one works. This project attempts to address this issue by simulating one. This interactive model shows the individual steps of breaking RSA using a quantum period finding algorithm. The model can also factor the multiple of two prime numbers.

Acknowledgements

I would like to thank my supervisor, Dr. Steve Vickers, for his support and advice throughout the project.

Contents

List of Figures

Chapter 1

Introduction

When I speak of this project, the idea of quantum computing baffles and confuses most people. When I started my research, I learnt about all the physical properties of quantum computers, the type of operations that can be applied and many other things. Still I could not see how a quantum computer could practically achieve something of use, nor did I understand its sheer speed. This was the inspiration for my project. My aim was to design a way to visualise a quantum computer and to do this I chose a specific process to model, breaking RSA.

RSA is one of the first useful public key encryptions and is still used today. As I explain in my background research, the security of RSA relies on the difficulty of factorising a semiprime, the product of two prime numbers. The problem is that if the semiprime gets larger, the difficulty of factorisation increases exponentially. For example, in 2010, with the help of "many hundreds of machines", factoring RSA-768 took around two years to break[1]. In the same article it stated that RSA-1024 is a thousand times harder to break and RSA-2048 is about 4 billion times harder than RSA-1024.

This is where quantum computers come in. They can work exponentially faster than our 'classical' computers, meaning that an increase in the size of RSA will not affect a quantum computer the way it would a classical one. So the idea is to visualise how a quantum computer would break RSA and to find a way to 'view' the quantum computer at all times and see what is happening.

A warning though, the software is designed to be a teaching aid. With the help of this and a craving to research this topic, I believe that this software could really help in finding that final moment of understanding. I hope that the making of this software will be as educational for the people who use it in the future as it has been for me.

Chapter 2

Background Information and Research

2.1 RSA

2.1.1 History

Up until 1976, all publicly used cryptography was symmetric. Symmetric cryptography uses the same key to both encrypt and decrypt messages. This form of encryption has two flaws. The first is that key distribution is difficult. Two users must establish a key before they can communicate, and if an eavesdropper reads the initial key exchange, any cipher is useless. The second is that every user involved has to keep that key secure and to do this keys must be changed often, which can lead to key management issues.

In 1976, W. Diffie and M. Hellman published an article describing a new type of cryptography, called public key cryptography[2]. In this scenario all users have their own public and private key pair. This would allow users to publish their private key to the public and other users could encrypt their messages to send with their public key. However, Diffie and Hellman could not find a one-way function, a function that is computationally easy to apply but difficult to find the inverse. It was not until 1978 that this function was discovered and published by R. L. Rivest, A. Shamir and L. Adleman[3].

2.1.2 Encryption / Decryption

RSA uses the mathematical principle that factoring the product of two large primes is much more difficult than multiplying two primes. The encryption and decryption for two people, Alice and Bob, proceeds as follows.

First a private / public key pair is produced by Alice:
Take two large random primes p and q and take the product

$$N = p \cdot q \tag{2.1}$$

Find Eulers totient function of N

$$\phi(N) = (p-1)(q-1) \tag{2.2}$$

Pick a large number that is coprime to $\phi(N)$, such that

$$gcd(e, \phi(N)) = 1 \tag{2.3}$$

Find the inverse of $e \pmod{\phi(N)}$

$$e \cdot d = 1 \pmod{\phi(N)} \tag{2.4}$$

The public key is (e, n) the private key is d.
For Bob to encrypt a message M to send to Alice

$$M^e \equiv C \pmod{N} \tag{2.5}$$

For Alice to decrypt the ciphertext C

$$C^d \equiv M \pmod{N} \tag{2.6}$$

This works due to the fact that $C^d = (M^e)^d = M \pmod{n}$ and is proven as follows.
From (2.4), we know

$$ed - k\phi(N) = 1 \text{ for some } k \in \mathbb{N} \tag{2.7}$$

And that if N and M are coprime then by Euler's totient theorem

$$M^{\phi(N)} \equiv 1 \pmod{N} \tag{2.8}$$

Then

$$M^{ed} = M^{1+k\phi(N)} = M \cdot M^{k\phi(N)} = M \cdot (M^{\phi(N)})^k \equiv M \pmod{N} \tag{2.9}$$

2.1.3 Breaking

Method 1: Factoring

If N can be factored into its two primes, this means that $\phi(N)$ and then d can be calculated. There are many different ways of doing this from testing random primes to the number field sieve. One of the most efficient ways of factoring is the number field sieve, however the order is still exponential[4].

Method 2: Order-finding

If we can find the order of an encrypted message C modulo N, we can work out M without factoring N.

$$C^r \equiv 1 \pmod{N} \tag{2.10}$$

(If the order does not exist, then C and N share a common factor and N can be factorised instantly.)
Therefore $r|\phi(N)$ and using the fact that $gcd(e, \phi(N)) = 1$ then

$$gcd(e, r) = 1 \tag{2.11}$$

So e has an inverse

$$e \cdot d' \equiv 1 \pmod{r} \tag{2.12}$$

Which means that

$$e \cdot d' = 1 + kr \qquad \text{for some } k \in \mathbb{N} \tag{2.13}$$

Therefore

$$\begin{aligned} C^{d'} &= (M^e)^{d'} && (2.14) \\ &= M^{1+kr} && (2.15) \\ &= M \cdot M^{kr} && (2.16) \\ &\equiv M \pmod{N} && (2.17) \end{aligned}$$

The solution d' may not be the private key d, but it can be proven that $d \equiv d' \pmod{r}$

If we are lucky, we can also factor N into it's two primes, p and q.
The first condition we need is

$$r \equiv 0 \pmod{2} \tag{2.18}$$

This allows us to introduce x such that

$$x = C^{\frac{r}{2}} \tag{2.19}$$

$$x^2 - 1 \equiv (x+1)(x-1) \equiv 0 \pmod{N} \tag{2.20}$$

The second condition is that

$$x + 1 \not\equiv 0 \pmod{N} \tag{2.21}$$

If both conditions hold, then

$$gcd(x-1,N)=p \tag{2.22}$$

$$gcd(x+1,N)=q \tag{2.23}$$

So how does one find the order efficiently?

2.2 Fourier Transform

"Used to represent a non-periodic function by a spectrum of sinusoidal functions."[5] Without going into too much mathematical detail, using a Fourier transform on a periodic function will reveal the period and multiples of it as the largest amplitudes. By using this technique we can apply method 2 in the previous section and break RSA. If we model the function

$$f(k)=C^k \pmod N \qquad \text{for } k\in\mathbb{N} \tag{2.24}$$

for enough values of k and take a Fourier Transform of this, we can work out the order. The details of this will be explained later.

(This does seem pointless on a classical computer, as eventually calculating values of k will lead us to $f(k)=1$, but again this method is essential for quantum computers.)

2.2.1 Discrete Fourier Transform

The specific type of Fourier transform we will be using is the discrete Fourier transform as the above function is not continuous. The equation for this is

$$X_k=\sum_{j=0}^{N-1} x_j e^{-2\pi ijk/N} \tag{2.25}$$

With inverse

$$x_j=\frac{1}{N}\sum_{k=0}^{N-1} X_k e^{2\pi ijk/N} \tag{2.26}$$

2.3 Quantum Computers

The subject of quantum computing is a vast and complicated one. The purpose of the project is to give an insight into what they are doing, not how they are doing it. So instead of a complete explanation of quantum computing, only the most essential features are explained as follows.

2.3.1 Superposition

In classical computing, computers are built with transistors. These transistors can either be off or on, 0 or 1.

In quantum computing, computers are built with quantum particles. Instead of being in a state of 0 or 1, these quantum bits - qbits - can be in a combination of both, called a superposition. This means that 2 qbits can be in a combination of all 4 different possible states at once. This continues for larger numbers of qbits, where n qbits can be in 2^n states at once.

This allows us to achieve computation that is exponentially faster than classical computation. However these computations are limited and they have some downsides.

2.3.2 Measurements

In classical computing, measuring the state seems obvious as each bit is a definite 0 or 1. In quantum computing this is not the case. By making a measurement of a quantum state, every qbit is forced to make a decision, 0 or 1. By doing this the quantum state is destroyed. With this limitation, the state has to be prepared correctly to make a useful measurement.

2.3.3 Useful Operations

Modular Exponentiation

Modular exponentiation on a quantum computer is necessary for the quantum Fourier transform explained below. This is the slowest algorithm needed for this project, however in 2012 the bottleneck was reduced right down to only $O(n^3)$[6].

Quantum Fourier Transform

The quantum Fourier transform is a way to achieve a discrete Fourier transform on a quantum computer. Currently the fastest known quantum algorithm is $O(n \log n)$, with n being the number of qbits[7]. Compared to the fastest classical algorithm with $O(n2^n)$ this is an incredible improvement.

2.4 Finding the Order

With help from N. D. Mermin[8], we can explain how a measurement can lead to the order.

From taking a Fourier Transform of $f(x) = C^x \pmod{N}$, the points with the largest amplitudes relate to the period.

$$y \approx \frac{jx_{max}}{r} \tag{2.27}$$

y is a measurement of one of the large amplitudes, x_{max} is the total number of points, j is an integer and r is the period. The smallest x_{max} we can use, without knowing r and to ensure that a full period is made with the modular exponentiation, is 2^n with n being the smallest amount of classical bits needed to represent N.

If we use a stronger x_{max} of $2^n > N^2$, this will ensure that our measurement can be bounded by a very small number.

$$\left| \frac{y}{2^n} - \frac{j}{r} \right| \leq \frac{1}{2^{n+1}} \tag{2.28}$$

From here we now have an overly accurate measurement for the period, as the maximum difference between two measurements $y_1/2^n$ and $y_2/2^n$ is now at most $1/N^2$. This means that we can find a unique value for j/r.

As $y/2^n$ has a much larger denominator than j/r, we can simplify the fraction down to a more reasonable approximation and get our r.

To do this we use continued fractions.

$$\frac{y}{2^{2n}} = \cfrac{1}{a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \cfrac{1}{a_m}}}} \tag{2.29}$$

By limiting our m we can find an approximation. If at every stage we turn our continued fraction back into a simplified fraction and check our denominator, we can limit the denominator from getting too large. The limit here would be n as $r < n$.

This will finally find our j_0/r_0. Note that j/r was not used, as j can share a common factor with r, leaving us with an overly simplified fraction. There are two possibilities here. The first is that several multiples of r_0 can be checked, or another measurement can be made and the lowest common multiple of the two measurements can be made. Since there is a reasonably low chance that the greatest common divisor of j_0 and r_0 will be high, this final calculation is usually easy.

Once r is found, we can use the methods shown above to break RSA.

2.4.1 Probability

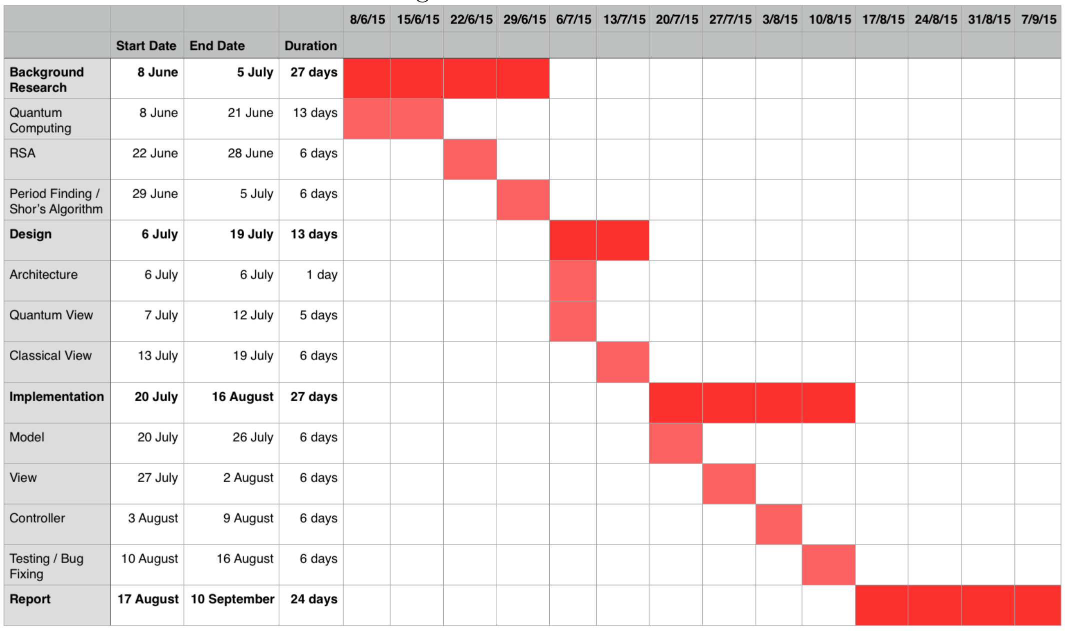
Mermin also made a note that the probability of measuring a useful value is bounded below by $4/\pi^2 = 0.4053$. This lower bound encouraged me to test every possible measurement for a specific semiprime and message, and then seeing if all the successful measurements added up to more than 40%.

2.5 Planning

This background research left me with two clear sections of work. The first is the quantum section, which includes the modular exponentiation, the quantum Fourier transform and the measurements. The second is the classical section, moving from a measurement to either factorisation or decryption. A final section on probability described on page 13 adds another interesting element.

A Gantt chart was created whilst completing my background research to guide me through the timings of the project.

Figure 2.1: Gantt Chart



				8/6/15	15/6/15	22/6/15	29/6/15	6/7/15	13/7/15	20/7/15	27/7/15	3/8/15	10/8/15	17/8/15	24/8/15	31/8/15	7/9/15
	Start Date	**End Date**	**Duration**														
Background Research	**8 June**	**5 July**	**27 days**	🟥	🟥	🟥	🟥										
Quantum Computing	8 June	21 June	13 days	🟥	🟥												
RSA	22 June	28 June	6 days			🟥											
Period Finding / Shor's Algorithm	29 June	5 July	6 days				🟥										
Design	**6 July**	**19 July**	**13 days**					🟥	🟥								
Architecture	6 July	6 July	1 day					🟥									
Quantum View	7 July	12 July	5 days					🟥									
Classical View	13 July	19 July	6 days						🟥								
Implementation	**20 July**	**16 August**	**27 days**							🟥	🟥	🟥	🟥				
Model	20 July	26 July	6 days							🟥							
View	27 July	2 August	6 days								🟥						
Controller	3 August	9 August	6 days									🟥					
Testing / Bug Fixing	10 August	16 August	6 days										🟥				
Report	**17 August**	**10 September**	**24 days**											🟥	🟥	🟥	🟥

2.5.1 RSA or Factorisation

During my background research, many of the documents that I read did not mention RSA, instead they were only factorisation algorithms. It seemed as if I had to choose between the two but this was not the case. Since both methods relied on finding the order of a semiprime, I could just use a random ciphertext if only factorisation was needed. Therefore both methods were implemented.

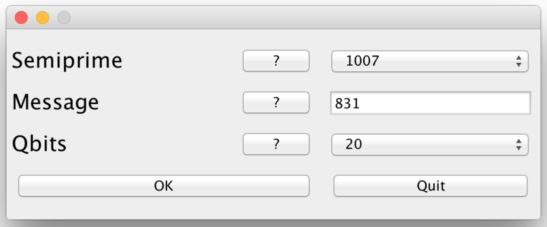
Chapter 3

Design

As the program is interactive and educational, the design of the program is important. The user needs to be clearly shown what is happening at each stage in detail, along with descriptions when necessary. To design this program, the task was split into three sections: the menu, the quantum state and the classical calculations.

3.1 Menu

Figure 3.1: Menu



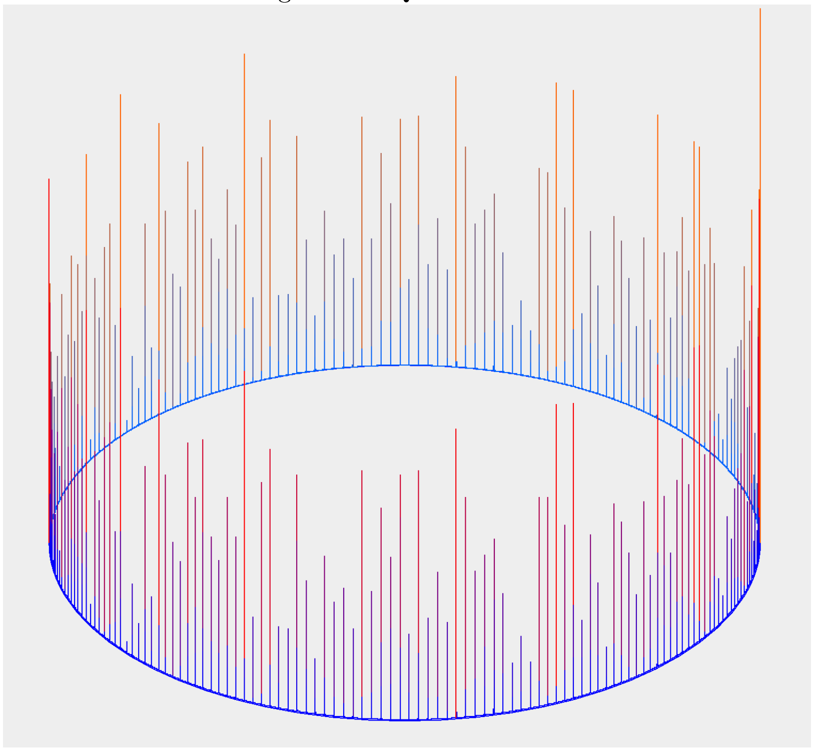
Semiprime ? 1007
Message ? 831
Qbits ? 20

In this part of the program three key pieces of data need to be collected, the semiprime N, a ciphertext C and the number of qbits n. To decrypt the ciphertext in RSA, the other part of the public key e is needed, but as N can be factorised without e it is not asked for in the menu. This also allows a random C to be chosen just to factorise N.

To help the user understand what each of these parts are, a small description for each part is needed, denoted by a button with a question mark.

3.2 Quantum State

Figure 3.2: Quantum State



In designing a useful graphical user interface (GUI) for the quantum state, Steve Vickers' quantum Fourier transform applet[9] was a useful tool. In this program, it is shown that every possible measurement could be shown as an individual line around a circle, with their heights showing the probability of measurement. For n qbits, this would show 2^n possible measurements. As the number of qbits increases, the state seems to be exponentially more complex, but limiting the state to a circle is an elegant solution.

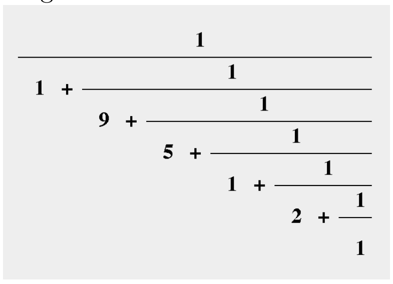
Looking at the state in this way, it is impossible to show in a real quantum computer because the probabilities of the state can never be measured. However, it is very informative for the user to see the state in this way.

Dr. Vickers sent me the code for this program to help with designing the GUI, but the code had to be completely rewritten to be used effectively. This will be explained in the implementation section.

For the quantum state, both quantum methods need to be shown, the modular exponentiation and the quantum Fourier transform. Buttons are needed to let the user see each stage individually.

3.3 Classical Calculations

Figure 3.3: Continued Fraction



$$\cfrac{1}{1+\cfrac{1}{9+\cfrac{1}{5+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1}}}}}}$$

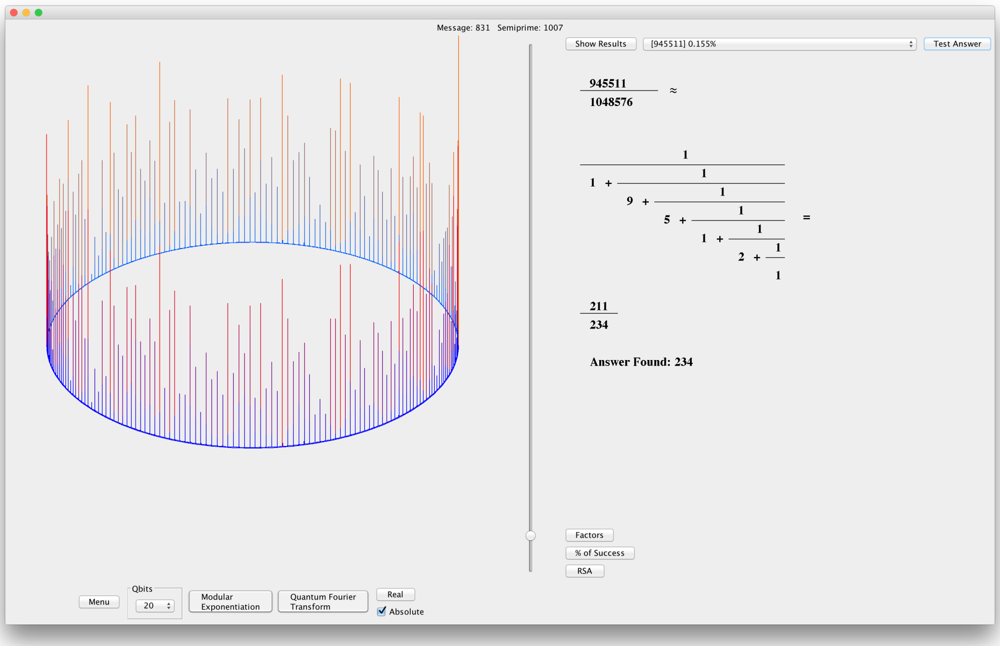
Designing the classical side was less obvious as the quantum side. The most interesting part of the classical calculations are the continued fractions. Showing a continued fraction in the GUI would also be interesting and allow the user to see how the measurement is simplified into its final fraction, showing the period.

Showing the eventual RSA decryption needed to be here too, as well as factorisation if possible. As these need their own separate sections it was decided to implement buttons with their own dialogs.

Chapter 4

Implementation

Figure 4.1: Main View



Message: 831 Semiprime: 1007

[945511] 0.155%

$$\frac{945511}{1048576} \approx$$

$$\cfrac{1}{1 + \cfrac{1}{9 + \cfrac{1}{5 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1}}}}}} =$$

$$\frac{211}{234}$$

Answer Found: 234

Qbits: 20

[✓ Absolute]

This section of the report will follow the main stages of the architecture used for completing the program, Model View Controller. As the controller has deep links with the view, the explanation will combine the two.

4.1 Model

4.1.1 State

This is the main class. Here the constructor needed the number of qbits n, the semiprime and the message. From that an array of size 2^{n+1} is created, which shows the probabilities of every possible measurement. This is twice the size expected as a Fourier transform needs both real and imaginary parts. A brief list of important methods is shown below:

initPeriodFind: the modular exponentiation. This is applied to the real parts of the large array.

fft: the Fourier transform. Steve Vickers had originally written a method for this, but it was not efficient $O(2^{n^2})$. With an external library named JTransforms[10], a Fast Fourier Transform (FFT) is now implemented to apply the same operation with a greater efficiency $O(n2^n)$.

scale: a scaling method. At all times, the state has to have a total probability of measurement of 100%. With methods such as the modular exponentiation, this method is applied at the end to keep this condition true.

sort: a sorting method. When a measurement is taken from a quantum computer, the likelihood is based on probability. Therefore the most likely measurements should be displayed first and to do this we need a sorting method. This method creates a new class for each measurement and sorts using its own compareTo method.

factorise: a factorisation method. This method will factorise N if it can, throwing relevant exceptions if not.

unencrypt: an RSA decryption. This requires another number e, part of the public key. This will successfully decrypt the ciphertext every time.

percentage: a statistical tool. This was added later on in development as it is not essential. This method works through every possible measurement and calculates if it is a useful one. Then every useful measurement, along with its likelihood of measurement is added up, giving a total percentage. As explained in section 2.4.1 this should be at least 40% at all times and this baseline was useful in testing.

continuedFraction: This method takes the measurement $y/2^n$ and converts it into a continued fraction. The return is an ArrayList of integers, in the form of

$$\cfrac{1}{a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \cfrac{1}{a_m}}}} \tag{4.1}$$

Where each a_k is a member of the ArrayList.

4.1.2 Fraction

When moving from a measurement to finding the period, many methods required working with fractions and continued fractions. To support this a fraction class was created. Several methods exist in this class that convert a fraction into a complete continued fraction, find the correct approximation and return the period and the continued fraction in different forms. Some of the important methods are explained below.

continuedFractionEval: conversion from a continued fraction to a fraction. This will take an ArrayList of integers as explained in the previous method and convert it into a single fraction.

multiContinuedFractionEval: This method takes every single possible approximation and returns an ArrayList of type Fraction. This was useful when manual testing the approximation of j/r, explained in the section 2.4.

4.2 View & Controller

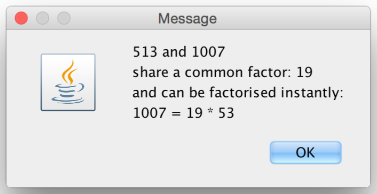
MainFrame is the class that contains the main method. Here the menu (MenuPanel) is deployed, along with the main screen (StateGUIPane). CardLayout is used to switch between the menu and the main screen when needed. StateGUIPane then proceeds to deploy two more key classes, StateView for the quantum parts and EvaluationPanel for the classical parts. StateGUIPane also initialises two more classes, StateButtonPanel for buttons and InfoPanel to show the data entered at the menu. Now that a rough layout of Java swing classes has been established, details will now be explained.

4.2.1 Menu

The menu uses a GridBagLayout. The GUI only requires a text field, 2 dropdown menus, 5 buttons and some text. The difficulty in implementation is to make sure that no data is entered incorrectly, by sufficiently limiting the options available to the user. This is important partly for the protection of the program, but even more so for the protection of the user as error will lead to confusion. If any numbers are entered incorrectly, the entirety of the program will not work. Every possible user input has the possibility of error, the issues and their resolutions are explained below.

The semiprime field. As the semiprime has to be the product of two primes, a dropdown menu was used instead of a text field. A small program was used separately to generate semiprimes up to a limit, which were added to the dropdown menu.

Figure 4.2: Common Factor Warning Dialog

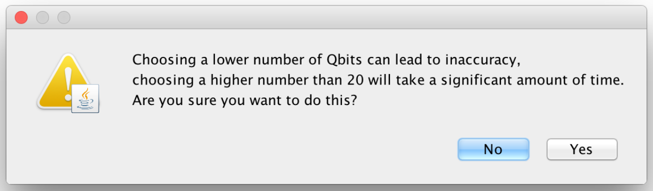


513 and 1007
share a common factor: 19
and can be factorised instantly:
1007 = 19 * 53

The message field. A text field had to be used here but this is dangerous for three reasons. The first is that anything can be entered, including symbols or letters. The solution was to use a formatted text field, only allowing integers. The second is that the message must be smaller than the semiprime and larger than one. Pressing the OK button initialised a check before continuing. The third was only noticed when testing began, the message cannot share a common factor with the semiprime as explained in my background research. Again this was checked at the OK press with the extended Euclidean algorithm. When this occurred a dialog factorising the semiprime was displayed, as the extended Euclidean algorithm can work extremely fast, even with very large numbers.

The qbits. In the background research section it was specified that for accurate numbers, double the amount of qbits were needed than compared to the amount of classical bits needed to represent the semiprime. Due to this the qbits dropdown menu automatically changes to keep this condition true, but allows the user to change as needed. However, a warning dialog shows when the number of qbits is changed, urging the user that it will lead to inefficiency or inaccuracy. A warning is also issued for higher numbers of qbits, as the program takes exponentially longer to run the higher the number is. This also links with the semiprime and the warning also appears when a large semiprime is chosen.

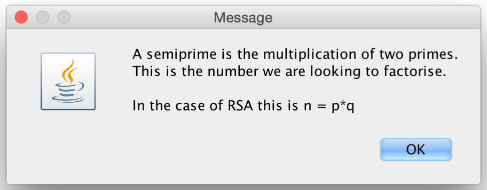
Figure 4.3: Qbit Warning Dialog



Choosing a lower number of Qbits can lead to inaccuracy,
choosing a higher number than 20 will take a significant amount of time.
Are you sure you want to do this?

Later on in implementation, help buttons were added. Each button leads to a dialog explaining what each part does and is very useful to new users of the program to help understand what to enter.

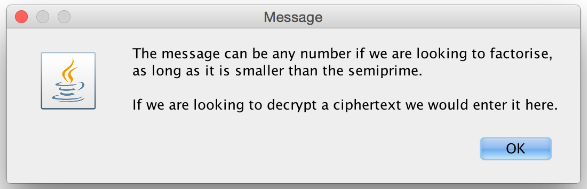
Figure 4.4: Semiprime Help Dialog



A semiprime is the multiplication of two primes.
This is the number we are looking to factorise.

In the case of RSA this is n = p*q

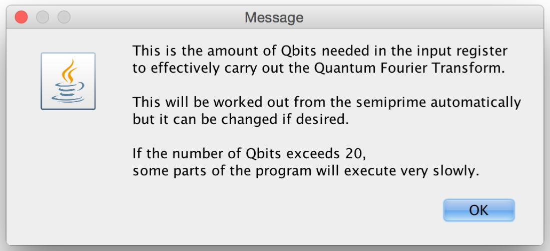
Figure 4.5: Message Help Dialog



The message can be any number if we are looking to factorise,
as long as it is smaller than the semiprime.

If we are looking to decrypt a ciphertext we would enter it here.

Figure 4.6: Qbit Help Dialog



This is the amount of Qbits needed in the input register
to effectively carry out the Quantum Fourier Transform.

This will be worked out from the semiprime automatically
but it can be changed if desired.

If the number of Qbits exceeds 20,
some parts of the program will execute very slowly.

4.2.2 Quantum Parts

StateView is the quantum display and is shown in Figure 3.2. This uses the Graphics class to paint a series of lines. An oval is split into 2^n separate parts and each part contains a line. From the model, every pair (real and imaginary parts) is taken from the large array in the model and their absolute value is displayed as a line of that magnitude. In the

StateButtonPanel there are ways of just displaying the real or imaginary parts of the array and everything is scaled accordingly. There is a colour gradient from blue to red depending on how long each line is. There is also a slider to the right, which works on a quadratic scale, to display lengths from very small to very large.

When the modular exponentiation and the quantum Fourier transform buttons are pressed, they trigger the methods in the model. The model then updates the view here and the image is updated. When each button is pressed, a dialog appears explaining what is happening.

Figure 4.7: Button Panel

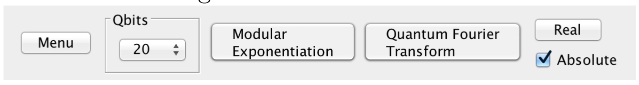
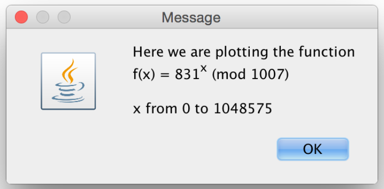


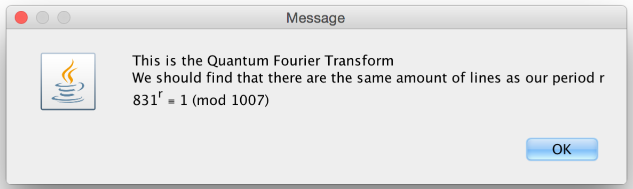
Figure 4.8: Modular Exponentiation Dialog



Here we are plotting the function
f(x) = 831^x (mod 1007)

x from 0 to 1048575

Figure 4.9: Quantum Fourier Transform Dialog



This is the Quantum Fourier Transform
We should find that there are the same amount of lines as our period r
831^r = 1 (mod 1007)

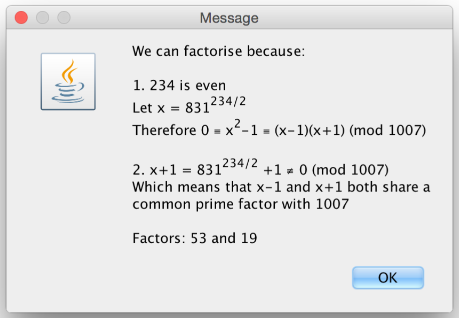
4.2.3 Classical Parts

This is where the rest of the mathematics is done, from measurements to factorisation. In the EvaluationPanel, after completing the quantum Fourier transform, the button ‘Show Results’ will trigger the sort method in the model and list every single measurement, sorted from most likely to least. The user can then pick any measurement and attempt to test the answer. This displays the fraction as a continued fraction, approximates it accordingly and checks if it is a useful measurement.

A useful measurement is one that divides the order. For a measurement that is useful, the fraction approximated will be j/r. If by chance that j and r share a common factor, r will be a fraction of what it should be. As the numbers are not extremely large, every time a measurement is taken the denominator is multiplied up to a maximum of N to see if the order is found.

If the order is found at any point, three more buttons are of use. The first, factors, will attempt to factorise the semiprime. If the two conditions described in my background research are met, the number is factorised, with explanation. If not the reason will be shown.

Figure 4.10: Factorisation Dialog



We can factorise because:

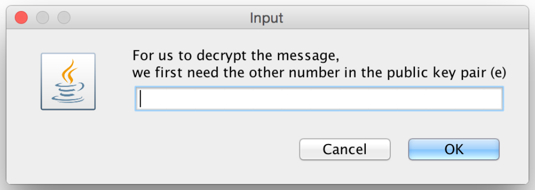
1. 234 is even
Let x = $831^{234/2}$
Therefore 0 ≡ x^2−1 ≡ (x−1)(x+1) (mod 1007)

2. x+1 = $831^{234/2}$ +1 ≢ 0 (mod 1007)
Which means that x−1 and x+1 both share a common prime factor with 1007

Factors: 53 and 19

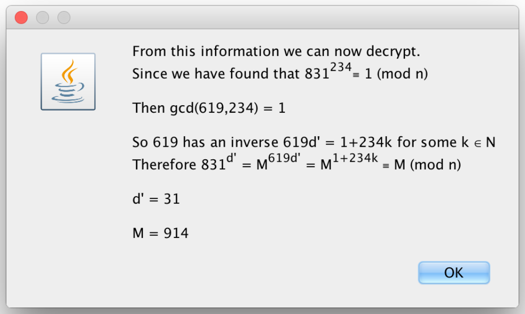
The second is RSA. This will take you through two dialogs. The first will ask for the missing part of the public key e and the second will show the decryption process.

Figure 4.11: RSA: Requiring e



For us to decrypt the message,
we first need the other number in the public key pair (e)

Figure 4.12: RSA: Decryption Dialog


From this information we can now decrypt.
Since we have found that $831^{234} = 1 \pmod{n}$

Then gcd(619,234) = 1

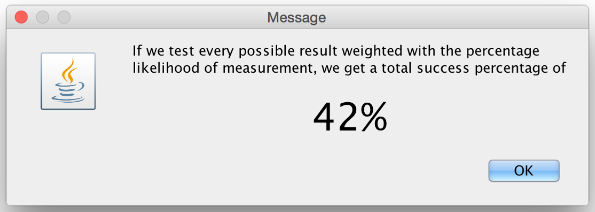
So 619 has an inverse 619d' = 1+234k for some $k \in N$
Therefore $831^{d'} = M^{619d'} = M^{1+234k} = M \pmod{n}$

d' = 31

M = 914

The final button is % of success. This calculates the percentage of selecting a useful measurement as explained in the model. This is shown in a dialog.

Figure 4.13: % Of Success Dialog


Message

If we test every possible result weighted with the percentage likelihood of measurement, we get a total success percentage of

42%

4.3 Testing

The program was mostly tested manually. Once the mathematical tests were completed in JUnit, the program just needed to be run through several times, trying all possible scenarios and pressing all buttons. Manual testing did clear up several problems and it was done relentlessly throughout development. For the JUnit testing, two classes were made.

4.3.1 StateTest

This class tested the State class. Four methods were tested here, modExp, extendedEuclid, factorise and unencrypt.

4.3.2 FractionTest

This class tested the Fraction class. Four methods were tested here, simplify, add, continuedFractionEval and multiContinuedFractionEval.

Chapter 5

Usability

This software is designed for educational purposes and there are some good ways of getting the most out of it. In this section a few scenarios are explained with specific numbers and results. These can be used in lectures or for personal use.

5.1 Small Number of Qbits

It is difficult to see that modular exponentiation is a periodic function. For higher numbers of qbits the circle starts to look like a blur, individual lines cannot be made out as seen in figure 5.1. So choosing a low semiprime of 21 and a message of 11 makes it easy to see that the function is periodic. This is due to only using 10 qbits, which is automatically chosen by the software. It is even possible to count the number of lines between each period. This is shown in figure 5.2.

5.2 Changing the Number of Qbits

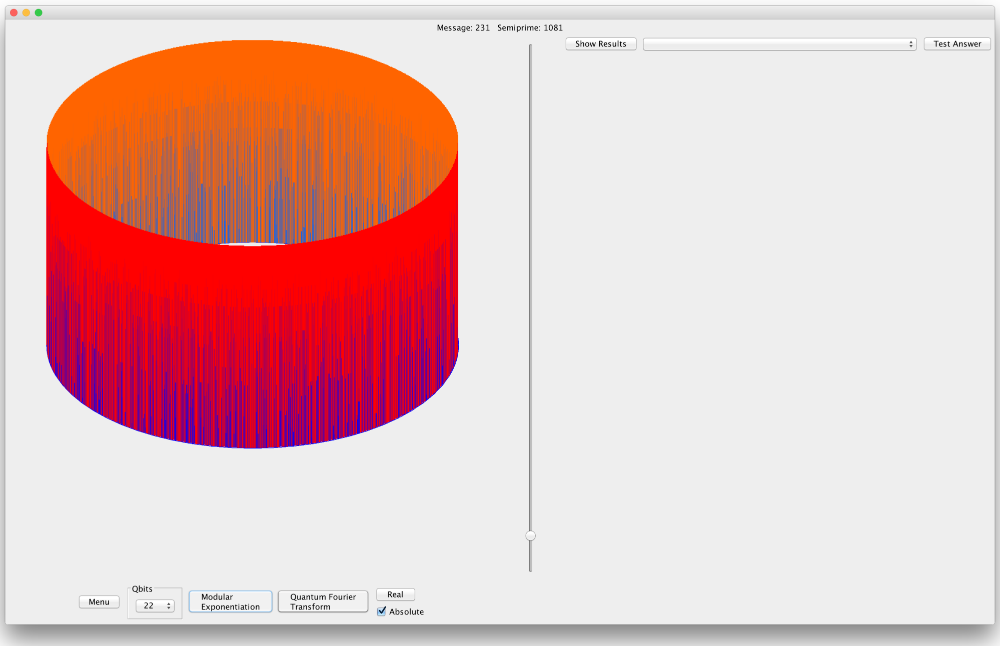
From the same example as above, changing the number of qbits and repeating the quantum Fourier transform can yield some interesting results. The first is that raising the number makes each point much more accurate, with little noise surrounding the points.

Lowering the number of qbits, even to as little as 6, still shows the same shape, but is much less accurate. Figures 5.3, 5.4 and 5.5 on the next pages show the difference.

5.3 Easier Factorisations

The factorisation of 15 has been a tremendous step forward for quantum computers but there is a reason it will take so much longer to factorise anything else. If 15 is entered as

Figure 5.1: Modular Exponentiation With a Large Number of Qbits


Message: 231 Semiprime: 1081

Qbits: 22

[✓ Absolute]

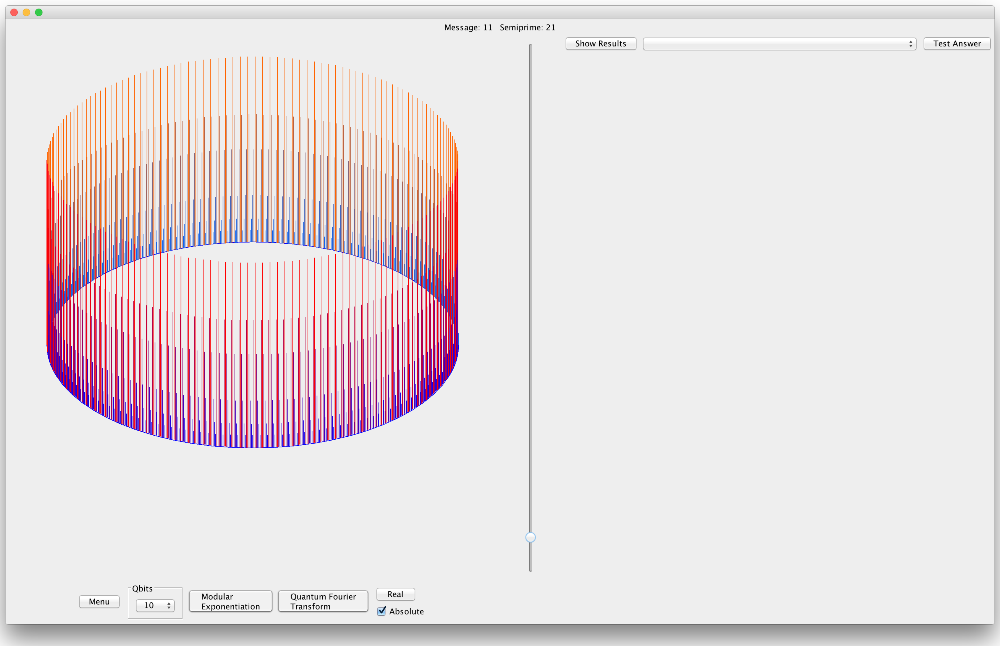
the semiprime and 7 the message, the program will suggest using 8 qbits. Moving through to the quantum Fourier transform it will prove obvious that the period is 4.

If the same process is tried for a much smaller number of qbits than suggested, for example 3, the exact same results will occur. This is due to the fact that the factors of 15 are in the form of $2^{2^n} + 1$, known as Fermat numbers. This leads to a $\phi(15) = 8$, which is a power of 2 and r must divide this. If we substitute r for 2^k in equation 2.28, we get an exact answer for our measurements, as long as $n \geq k$.

$$\frac{y}{2^n} - \frac{j}{2^k} = 0 \tag{5.1}$$

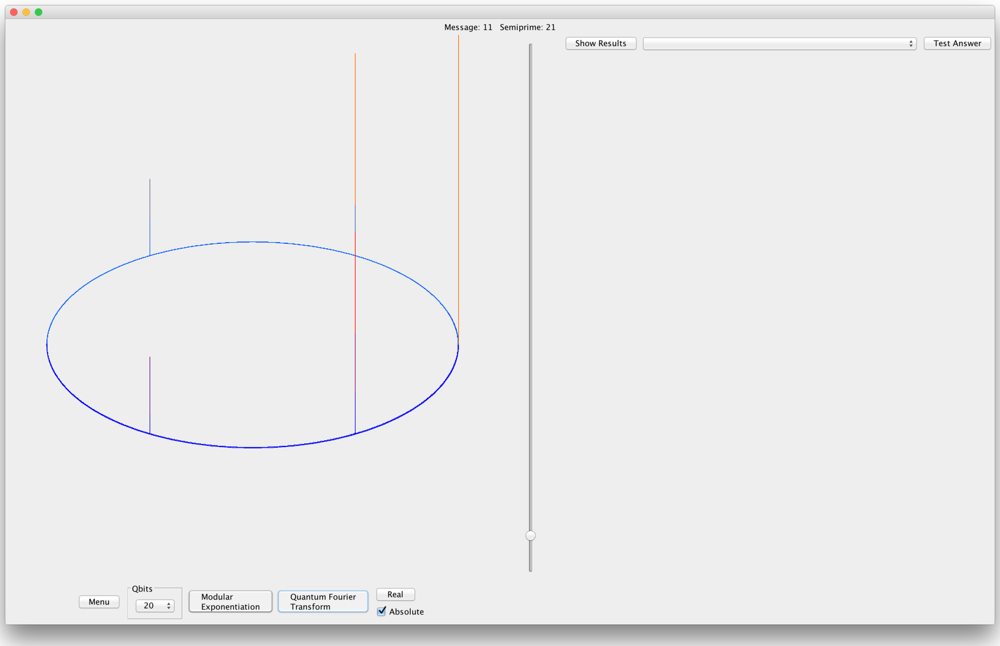
Five other numbers in this program exist that are in the form of two prime Fermat numbers multiplied: 51, 75, 771, 1285 and 4369. These can all be factorised using much fewer qbits than suggested.

Figure 5.2: Modular Exponentiation With a Small Number of Qbits


Message: 11 Semiprime: 21

Qbits 10

Figure 5.3: Higher Number of Qbits


Message: 11 Semiprime: 21

Qbits 20

Figure 5.4: Recommended Number of Qbits

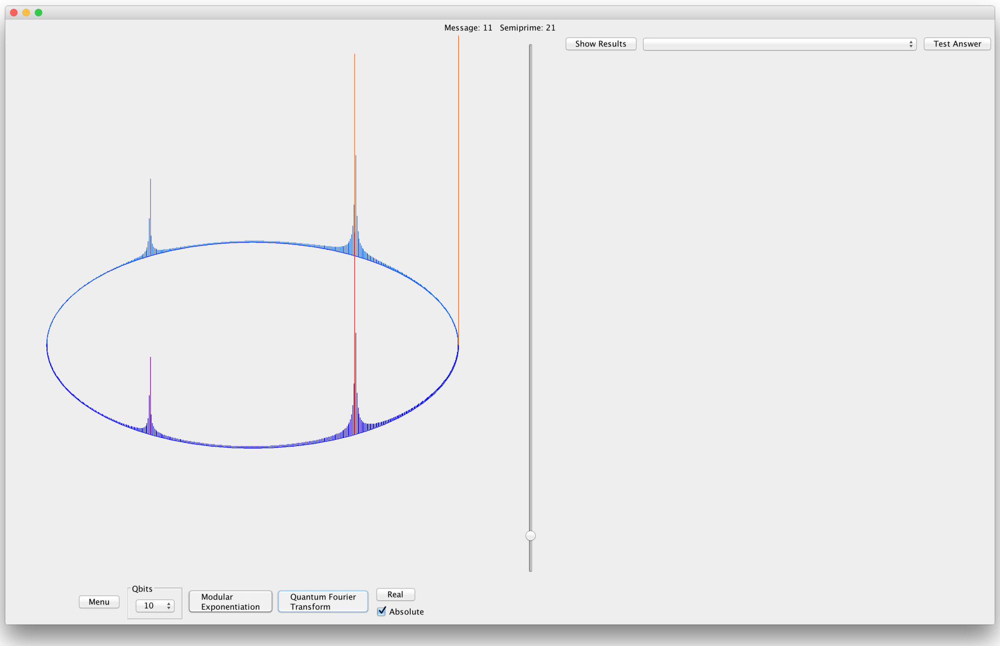
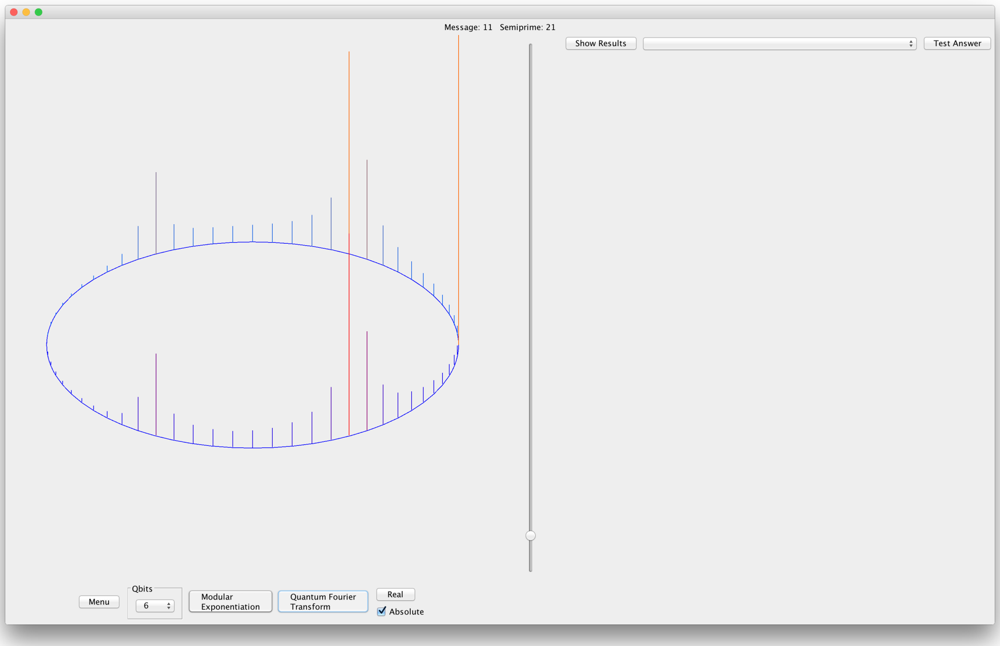


Figure 5.5: Lower Number of Qbits



Chapter 6

Evaluation

In this chapter, specific sections of the project are reviewed that have a particular importance, from challenges overcome to problems that still remain.

6.1 Resolved Problems

6.1.1 Quantum Fourier Transform Speed

When the program was in its early stages, the speed of the QFT was extremely slow. As this was the slowest part of the program, any improvements would speed the entire program up drastically. Luckily an external library was found that supported multithreading called JTransforms, which was mentioned in the implementation.

6.1.2 Transferring Measurements

When the exchange of data between the quantum and classical sides was first developed, every single possible measurement was transferred into a JComboBox. With the smaller semiprimes this was fine, even for 2^{16} different measurements. Eventually this did bottleneck the system and it was for no purpose. The user was only going to try the first few answers, maybe the first hundred at a stretch but the millionth? The solution was a simple fix but it was not as obvious as it is now.

6.1.3 Warnings

When early prototypes were implemented, there were many ways of running into an issue, be it one of speed, entering incorrect data or pressing a button at the wrong time. So several warning dialogs were implemented throughout the program to stop the user from entering incorrect data. It was one of the last things added as it is not seen as a high priority at first, but for any program - especially an educational one - the user needs an experience that is clear and free of error.

6.2 Remaining Problems

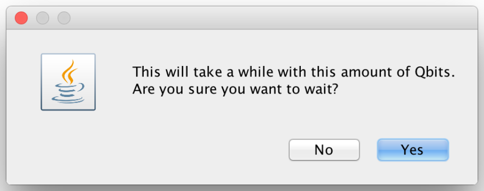
6.2.1 Large numbers

Any number over 8192 (2^{13}) runs out of Java heap space. I know that the heap space can be increased but the calculations are struggling with numbers this large anyway. There was always bound to be a limit and I am happy that it is this high.

6.2.2 % Success

The % Success button is the longest calculation in the entire program. Trying to press this with the number of qbits over 20 issues a warning. It is not needed to achieve an answer and it would not be possible to do on a quantum computer anyway. It was added after completing the rest of the program as it is useful to see that the program works as expected.

Figure 6.1: % Success Warning Dialog



This will take a while with this amount of Qbits.
Are you sure you want to wait?

Before my presentation, my code was slightly wrong, giving me a few values below 40% when using large numbers. This issue has now been fixed.

6.2.3 Speed

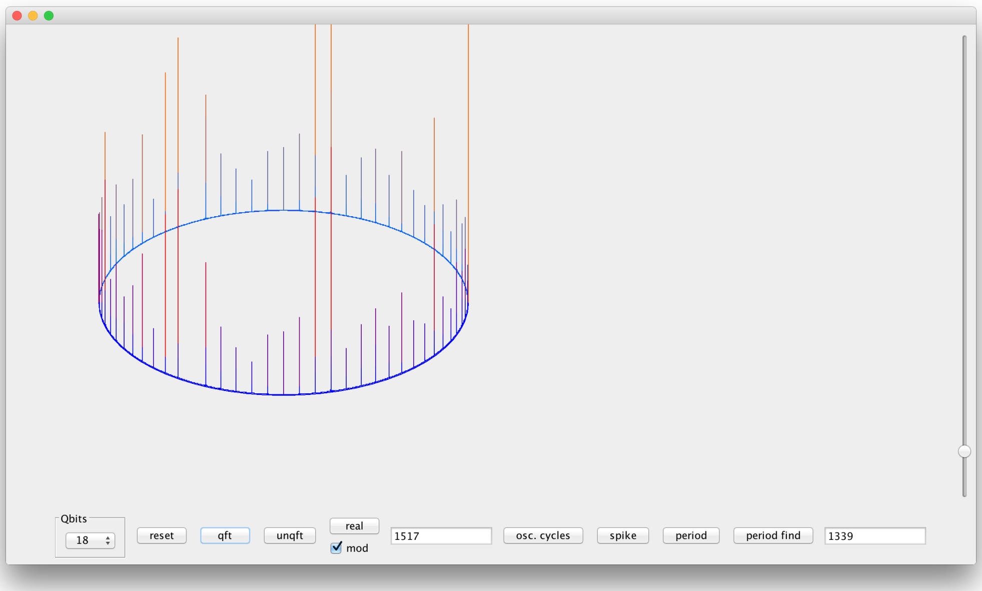
A lot of work was put in to make sure the program is efficient and quick to use, however larger numbers still take a long time to process. The problem lies in the speed of painting the graphics on the quantum side. An effective way to solve this problem without losing accuracy in the image seems impossible. There could be better libraries to paint such complex images but in the time frame of implementation this could not be achieved.

6.3 Removed Parts

6.3.1 Quantum Functions

When designing the quantum state, there were other functions added. The functions included displaying sine waves and specific periodic functions. The optional functions were all removed as they would have led to confusion in the final product, especially as variables cannot be altered after the menu.

Figure 6.2: Old Prototype



Qbits 18

[✓ mod]

1517

1339

6.3.2 Multithreaded Modular Exponentiation

The class ModExp was created to achieve this and it does work. The problem is that the speed of the original method was actually much faster than expected. The original method could finish within a few seconds, no matter how large the semiprime. The new class was in its early stages but after noticing the speed of the original method, deploying a large number of threads did not make an improvement.

6.4 What I Would Have Done Differently

There is one thing you can never do enough of when designing a piece of software, and that is planning. The entire classical side was added after the rest of the program was built, and I did not even think about displaying a continued fraction when I started creating the software. I think that continuously building upon software did have its advantages and I was actually working out the mathematics and testing my estimations by making the software. The GUI element really did help my understanding.

6.4.1 Time Frame

The Gantt chart that was created in the initial planning stage in figure 2.1 was difficult to keep up with. The initial setback was the background research. Before even thinking about the software, the mathematics behind the process had to be fully understood. Little material exists on the subject, especially on simulating quantum computers on classical ones. This meant that I had to spend more time on research than expected before moving on to the next stage.

Chapter 7

Conclusion

As far as the software is concerned, it is a success. Every semiprime that has been tested has lead to successful factorisation. Every RSA public key and ciphertext has been decrypted too. There is no way to cause an error in the program and the program is efficient with numbers that are not too large. It also accurately simulates how a quantum computer would carry out order finding, the core of this algorithm. In terms of visualisation the GUI is clear and the program is easy to use, as it offers explanation for every operation and provides sufficient warning and limitation too.

However, the reason for the program was not just to factorise large numbers, the largest number in this program can be factorised by a pocket calculator in under a second. The problem was the educational aspect, to help users understand what is happening to a quantum computer as it goes through the stages of factorisation and RSA decryption. Chapter 5 explains some of the many different ways to visualise a quantum computer that can lead to a better understanding of one.

This application would be a very useful tool to those with an interest in quantum computing, specifically the process it follows to break RSA.

Bibliography

[1] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thom, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, "Factorization of a 768-Bit RSA Modulus," in *Advances in Cryptology CRYPTO 2010* (T. Rabin, ed.), vol. 6223 of *Lecture Notes in Computer Science*, pp. 333–350, Springer Berlin Heidelberg, 2010.

[2] W. Diffie and M. Hellman, "New Directions in Cryptography," *Information Theory, IEEE Transactions on*, vol. 22, pp. 644–654, Nov 1976.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb 1978.

[4] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, "The Number Field Sieve," in *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, (New York, NY, USA), pp. 564–572, ACM, 1990.

[5] n. Fourier, *Oxford English Dictionary.* Oxford University Press, 1989.

[6] A. Pavlidis and D. Gizopoulos, "Fast Quantum Modular Exponentiation Architecture for Shor's Factorization Algorithm," *ArXiv e-prints*, July 2012.

[7] L. Hales and S. Hallgren, "An Improved Quantum Fourier Transform Algorithm and Applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, (Washington, DC, USA), pp. 515–, IEEE Computer Society, 2000.

[8] N. Mermin, *Quantum Computer Science: An Introduction.* Cambridge University Press, 2007.

[9] S. Vickers, "QFT Applet." `http://www.cs.bham.ac.uk/~sjv/teaching/quantumComp/QFTApplet.html`.

[10] P. Wendykier, "JTransforms." `https://sites.google.com/site/piotrwendykier/software/jtransforms`.