



Quantum inspired Factoring

Peter Elgar

Delft University of Technology



TUDelft

Quantum inspired Factoring

by

Peter Elgar

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday December 20, 2022 at 02:00 PM.

Student number:	5396328
Project duration:	December 1, 2021 – December 20, 2022
Thesis committee:	Dr. ir. S. Verwer, TU Delft, advisor Dr. S. Feld, TU Delft, supervisor

Preface

On the cover of this thesis we have a picture of a remake of the Bombe machine at Bletchley Park. A machine designed by the computer scientist Alan Turing. Its aim was to break the Nazi's correspondences that were encrypted using the Enigma machine. The Bombe can be seen as one of the devices that brought into motion our modern understanding of cryptography and computing. The story of how this machine got made has fascinated me ever since I visited Bletchley park back when I was still at school. We can only imagine what the excitement was like when the machine was first plugged in and was able to decipher the first encrypted message.

Also on the Bletchley Park campus is the British computer museum. Inside it there is a plethora of early computers build and designed during the 1940's to 1960's. Most notably a remake of the Colossus computer used to break messages encrypted using the Lorenz machine. Another reason why the museum is fascinating, is that it contains a compendium of funny and faulty computer designs. Many of which never made it past the prototype phase or the first generation. Proof to show that the progress in computing was not always as linear as what we now like to think in our age where Moore's law rules the roost.

Now in 2022 we are living through a new epoch in computing, the dawn of the quantum age. Here too we are not seeing a clear linear advancement, but a progression of peaks and troughs, paired with many new designs that may never leave the drawing board. Where at the end of this epoch only a few designs will rule the roost of quantum computing.

Just like in the 1940's a new generation of cryptography will come to fruition. New encryption standards will be designed as older methods will be thrown into the dustbin of history.

One encryption standard that will eventually be thrown in the dustbin of history is RSA. It is a simple mathematical problem that has fascinated me since the start of my master degree in September 2020. Even though the power and ability of modern quantum computing is laughable and disappointing, the research of it and around it is far from disappointing or laughable. During my research I have thoroughly enjoyed learning about this new area of study. Therefore I very much hope that this thesis is a testament to the great importance of this area of research, and also hope that for people who have little or no knowledge of this research area, that this will be a good entry point into what we can use the quantum computing paradigm for. I would therefore very much like to thank my thesis advisor Dr. Ir. Sicco Verwer and thesis supervisor Dr. Sebastian Feld for allowing me to research this field.

Furthermore, I would like to thank the people at the Feld group, namely Aritra Sarkar, Dhruv Bhatnagar, Luise Prelinger, Matt Steinberg, Matti Dreef, Medina Bandic and Sacha Szkudlarek; for making this experience very enjoyable. I would also like to say that a lot of my knowledge of this subject is thanks to the brilliant lectures given by Sridhar Tayur et al. at Carnegie Mellon University [8]. I would like to thank all fellow master's students that have had to put up with me, with special thanks to Adriaan de Vos. Last but not least I would like to thank my girlfriend Tamara for putting up with me during this period.

*Peter Elgar
Rotterdam, December 2022*

Abstract

RSA encryption standard is a vital component of everyday internet communication. It is currently seen as being unbreakable as the problem that it is based on, semiprime factorisation, is an NP problem. Therefore, to try and break RSA using the current state of the art factoring method will take thousands of years.

However, thanks to the advent of quantum computers we do know that RSA can theoretically be broken in seconds in two main ways. Firstly, by using Shor's algorithm. Secondly, by formulating it as a binary optimisation problem and solving the formation with the quantum approximate optimization algorithm or solving the formulation by using quantum annealing.

Unfortunately, there does not exist a quantum computer today that is accurate nor large enough to be able to break RSA encryption any time soon. For example, the largest semiprime number that has been factored by a quantum computer was a 41-bit number, as opposed to 2048-bit numbers that get used for RSA encryption.

Fortunately, what is possible is to split the binary optimisation problem formulation up into subproblems, solve them individually and combine the solutions to get the answer, as shown in Wang et al. [50]. Unfortunately, how the Wang et al. approach splits the problem up is not scalable, as when the problem gets larger so do the subproblems. Therefore, in the end the subproblems will expand to such a size that even they cannot be solved.

The problem of ever expanding subproblems is the main focus of this thesis. We present a new approach that splits the problem into subproblems that are of constant size. Consequently, no matter how large the problem gets, all components can still be solved. Using our new method, we have been able to vastly outperform previous records set by quantum computers. However, our approach does not outperform current state of the art classical factoring methods.

However, we also show that our new approach has limits. The increase in the amount of subproblems means an exponential increase in the spatial complexity when combining the solutions.

Fortunately, we also present ways in which we can reduce the spatial complexity. These methods have a mixed success. However, they have meant that we can factor numbers that have more than triple the bit length than if we were not to use them.

Finally, our techniques in reducing the spatial complexity have led us to discover a new weakness in RSA encryption. Therefore, potentially wreaking havoc on the security of the internet.

Contents

Preface	i
Abstract	ii
Nomenclature	vii
1 Introduction	1
1.1 Research Question	2
1.2 Our Contribution	2
1.3 Overview	2
2 Preliminaries	3
2.1 Ising Model	3
2.2 Quadratic unconstrained binary optimization	4
2.3 RSA	5
2.4 Factoring methods	6
3 Related Work	7
3.1 Factoring Methods	8
3.1.1 Direct Method	8
3.1.2 Table Method	8
3.2 Reduction Methods	12
3.2.1 Gröbner Basis	12
3.2.2 Energy Landscape Manipulation	13
3.2.3 Removing Carry variables	14
3.2.4 Deduction Reduction	14
3.2.5 Excludable Local Configuration	15
3.2.6 Split Reduction	15
3.2.7 Graver Basis	15
3.3 Running Methods	17
3.3.1 As one problem	17
3.3.2 Parallel	17
3.3.3 Sequential	18
3.3.4 Meet in the middle	19
3.3.5 Divide and Conquer	19
4 Factoring Method	21
4.1 Cell Method Formulation	22
4.2 Solving Cell Method Terms	24
4.3 Reducing the number of variables	25
4.4 Merging the solutions together	27
4.5 Viewing the problem as a tree	29
4.5.1 Depth-first Search Tree Factoring	30
4.5.2 Breadth-First Search Tree factoring	31
4.5.3 Priority Search Factoring	31
4.5.4 Weaknesses in RSA	32
4.6 Filtering	33
4.6.1 Min-Max filtration	33
4.6.2 Prime Filtration	34
4.6.3 Dynamic Case Filtration	36

5	Evaluation	38
5.1	Theoretical Evaluation	38
5.1.1	Generating cells	38
5.1.2	Solving cell method terms	39
5.1.3	Reduction of the number of variables	39
5.1.4	Merging the solutions	39
5.1.5	Filtering the solutions	40
5.1.6	Summary of Theoretical Analysis	40
5.2	Practical Evaluation	40
5.2.1	Time Comparison	41
5.2.2	Iteration Comparison	42
5.2.3	Space Complexity Comparison	43
5.2.4	Comparing Different Filter methods	44
5.2.5	Reduction method	46
5.2.6	Comparison with other factoring methods	49
5.2.7	Summary	49
6	Conclusion	51
6.1	Discussion	51
6.2	Limitations	52
6.3	Future Work	52
6.4	Concluding Remarks	53
	References	54
A	Cell Method example	57
A.1	RSA Numbers	59

List of Figures

2.1	Graphical representation of Ising Model	4
3.1	Visual overview of chapter	7
3.2	Representation of normal case vs the distributed case	17
3.3	Graphical representation of parallel process	18
3.4	Graphical representation of sequential process	19
3.5	Graphical representation of meet in the middle process	19
3.6	Graphical representation of divide and conquer method taken Guerreschi 2021 [27]	20
4.1	Number of rows during the merging process	29
4.2	Time in seconds for every merge	29
4.3	Tree view of the factoring problem	30
4.4	DFS method of factoring	31
4.5	BFS method of factoring	31
4.6	Priority search method	32
4.7	Graphical representation of DFS and BFS factoring methods	32
4.8	What areas the different filtration methods try to reduce.	33
4.9	Min-Max filtration method	34
4.10	Graphical representation of prime number filter	35
4.11	Graphical representation of the dynamic filter	36
5.1	Total Factoring time	41
5.2	Total Factoring time broken down into different components	41
5.3	Box plot of time it takes to factor numbers	42
5.4	Mean total number of iterations it takes to factor numbers	42
5.5	Box plot of number of iterations it takes to factor numbers	43
5.6	Mean data size at the end of factoring N	44
5.7	Box plot of data size at the end of factoring N	44
5.8	Time in seconds till solution is found, when running filtering methods on their own	46
5.9	Number of iterations till solution is found, when running filtering methods on their own	46
5.10	Data size at then end of the algorithm, when running filtering methods on their own	46
5.11	Line plot of mean number of rows for every number between 20 - 40 bits in length.	47
5.12	Box plot of both reduction methods, looking at the percentage of the reduction it has managed to make.	47
5.13	Time in seconds till solution is found, when running different reduction methods	47
5.14	Number of iterations till solution is found, when running different reduction methods	47
5.15	Data size at then end of the algorithm, when running different reduction methods	48
5.16	Box plot of time it takes to factor numbers using normal reduction method	48
5.17	Box plot of time it takes to factor numbers using ruthless reduction method	48
5.18	Box plot of time it takes to factor numbers	49
5.19	GMP-ECM factoring all the same numbers	49

List of Tables

3.1	Generic binary table method example	8
3.2	Generic representation of the cell method, assuming that p and q are not necessarily odd and not having used the substitutions of 3.5	9
3.3	This is a formulation of the column method, where we have not substituted the most and least significant bits of p and q	10
3.4	Generic representation of the block method, without having substituted the values for n and p_0 and q_0	11
4.1	List of symbols used in the Factoring Method chapter	22
4.2	Generic representation of the cell method 4.2	23
4.3	General truth table of a term, all terms that are given here are all the possible correct terms. The left most column represents the row number.	25
4.4	Example of two tables before reduction	26
4.5	Same tables after the reduction has been made	26
5.1	List of symbols used in the Evaluation chapter	38
5.2	Algorithmic complexity of the all sections of the algorithm	40

Nomenclature

Abbreviations

Abbreviation	Definition
QUBO	Quadratic Unconstrained Binary Optimization
qubits	quantum bits
GNFS	General Number Field Sieve, currently the best factoring method

Symbols

Symbol	Definition
N	The RSA number we wish to factor
n_i	bit i of N
p, q	the two factors of N , where $p > q$
p_i	bit i of factor p
q_i	bit i of factor q
S_i	Sum bit i
z_i	carry bit i
L_N, L_p, L_q	bit length of N , p and q
$\pi(x) = \frac{x}{\ln x}$	prime counting function π , it approximately gives the position of the prime relative to other prime numbers
M	number of cells in the problem
L	number of variables in the problem
K	number of rows in the final result Dataset
R	number of ranges

1

Introduction

Our current methods of online communication are based on the fact that we can correspond with a large degree of confidentiality, integrity and availability. Partly thanks to this triad, it has allowed world wide internet communication, e-commerce and many software services to become ubiquitous in the last 30 years. Supplanting businesses and services that were once the sole preserve of bricks and mortar companies. One such form of encryption that has allowed internet communication to become safe is asymmetric (or also called public key) encryption. One of the oldest and most well known asymmetric encryption schemes is called RSA encryption. Despite RSA becoming less popular over the years due to elliptic curve encryption, RSA is still widely used. One metric to see how wide the adoption of RSA is, is by looking at the number of root certificates that use RSA [1]. About 3/4 of the root certificates that are recognised by Mozilla use RSA. We can therefore say that everyone using the internet to visit a website or sending an email is reliant on RSA in some way or form.

The security of RSA is based on a simple mathematical problem. Given two prime numbers p and q , the product will be N , a semiprime number. It is easy to calculate N with p and q , however it is hard to calculate p and q if only N is known. Obviously for small numbers such as 143 this is a straightforward task, however for a number N with 617 decimal digits this would take a long time. The current state of the art factoring method is the general number field sieve (GNFS) [11]. The GNFS runs sub exponentially in time, and the largest RSA number that has been factored is RSA-250 which is a semiprime number with 250 decimal digits or 829 binary digits, where it took 2700 CPU core-years to factor [28].

Although RSA is currently secure, it does face a risk from a new paradigm of computing, namely quantum computing. We currently know of two ways quantum computers can factor numbers, firstly through Shor's algorithm on a universal quantum computer [45], and secondly through making a discrete optimisation problem first devised by Schaller et al. which can be run on a universal quantum computer and on quantum annealing computers [44]. Both factoring methods when run on a quantum computer that has enough quantum bits (qubits), connectivity and lack of noise, will be able to factor large RSA numbers in seconds or minutes, instead of days or years.

Despite the grand prospects of quantum computing, in the medium term quantum computing faces many difficulties. The most well known problems are the lack of quality in terms of the production of qubits, lack of connectivity between qubits, and the amount of noise during calculations [41]. All these problems have meant that we so far have not been able to do calculations where quantum computers have either been faster or more accurate than classical computers. The largest numbers to be factored using these methods are currently 21 for Shor's algorithm [33] and 1099551473989 using the optimisation method [31].

What would make a significant difference, would be to be able to split the problem into sections and solve those parts individually, then combine the answers together to find the solution. Consequently, this would mean that instead of having to build a computer that is 10 times more powerful, it could be possible to just split the problem up into 10 sections and solve it with 10 computers that are already available.

Fortunately, what we do know to be possible, is that both quantum factoring methods can be run distributively, as shown by Gidney et al. for Shor's algorithm [24], and by Wang et al. for the discrete

optimisation problem [49]. The problem in the Wang et al. approach is that the size of the sections is proportional to the size of the problem instance. Consequently, this means that the Wang et al. method is not scalable. However, in this thesis we shall introduce a scalable distributive method of factoring that firstly splits the problem instance into sections that are constant in size, then solves the individual sections, and finally merges the individual solutions together to get the answer. Furthermore, we shall look at how effective this method is in terms of speed in solving and merging the individual sections together, and see how accurate it is in finding the two factors of the semiprime number. Finally, we look at what the pitfalls of this method are and see how we can overcome these problems.

1.1. Research Question

Therefore this thesis will be about making a new factoring algorithm that uses a discrete optimisation method whereby we can split the formulations into sections that are constant in size. Then solve all the constituent sections and finally combine the solved sections to get the factors of the semiprime number.

Therefore our research question is:

How can we build a scalable distributive factoring algorithm that is based on a discrete optimisation problem?

1.2. Our Contribution

Our contributions are listed as follows:

1. A thorough analysis of the intersection between the factoring problem and quantum computing.
2. An analysis of the pitfalls of the current quantum based factoring methods.
3. A proposal to overcome the different pitfalls of past implementations.
4. An analysis of the effectiveness and efficiency of our new algorithm.
5. Based on our research into our new approach, we discuss a potential new weakness of RSA encryption.

1.3. Overview

The thesis will be divided up into the following chapters. Firstly we shall go over some preliminary subjects in chapter 2, secondly we shall go over related work about different factoring methods, optimisation and how the problem can be split into subproblems in chapter 3. Before going on to explain how our new factoring method works in chapter 4. After having explained the factoring method we shall evaluate the different parts of the algorithm theoretically and practically in chapter 5. This thesis will finish with a conclusion in chapter 6 where we shall answer our research question, and list the different limitations and future work.

2

Preliminaries

Before going into the related work of how to factor semi-prime numbers as a discrete optimisation problem, we must first explain a number of concepts before the different methods can be understood. This thesis shall not explain how quantum or quantum annealing computers work, for a general introduction to quantum annealing and its applications we suggest reading *Quantum Integer Programming* lecture notes by Tayur et al. [8]. However, it will explain how one can formulate an optimisation problem that a quantum (annealing) computer can solve. For this we use the Ising model, which allows us to view our problem as a lattice of qubits that in the end of the annealing phase will be +1 or -1. Unfortunately, in most optimisation problems we don't want our variables in our solution to be either +1 or -1, but to be either 1 or 0. Therefore we formulate the problem as a quadratic unconstrained optimisation (QUBO) problem and then later convert to the Ising model. After having explained both concepts an example will be given to show how a QUBO problem can be formed, and then turned into a problem so that a quantum annealer can solve it.

After having explained these concepts, we shall briefly discuss RSA encryption and different factoring algorithms.

2.1. Ising Model

The Ising model is a mathematical way of formulating an optimisation problem for a quantum annealing computer, where we describe the energy of a system where we hope to minimise the energy. In our case we wish to minimise the energy to 0, when that happens then we have found our solution. Its general formulation can be seen in equation 2.1. The Ising model describes a lattice where the σ are the quantum bits (qubits) or magnets which have a spin which is between -1 and +1 with their bias h . For the h value a negative value means that the qubit is more inclined to be -1, and positive value more likely to be +1, and 0 meaning no bias. Then we also have the coupling strength between the qubits denoted by the letter J , where positive values mean ferromagnetic momentum i.e attracted to the other qubit, negative values mean anti-ferromagnetic momentum, and 0 no momentum. Finally μ which means magnetic momentum, this can be largely ignored as on a quantum annealing computer it is a constant.

$$H(\sigma) = - \sum_{(ij) \in E(G)} J_{ij} \sigma_i \sigma_j - \mu \sum_{i \in V(G)} h_i \sigma_i \quad (2.1)$$

As a general approach we can therefore visualise this as a graph like in figure 2.1. Here we have arrows which can be seen as the qubits or as magnets, that are either up or down, thereby either +1 or -1, or north or south. Therefore, we can say that the J values can be seen as the edges with a given weight, and h values being the vertices also with a given weight.

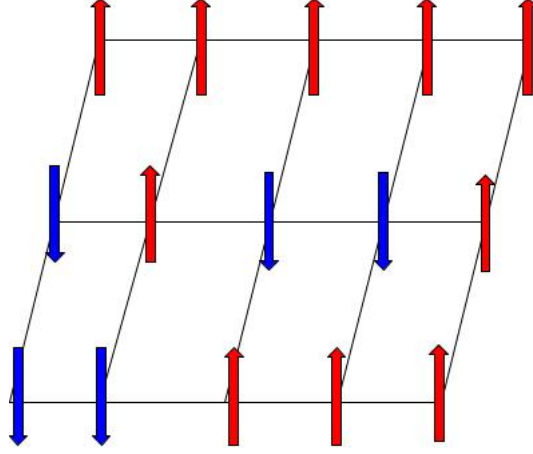


Figure 2.1: Graphical representation of Ising Model

2.2. Quadratic unconstrained binary optimization

As computers and programming languages work with the concept that a value is 1 or 0, not -1 or +1. Therefore, we use the quadratic unconstrained binary optimization (QUBO) formalisation as seen in equation 2.2. This formalisation can be converted into the Ising model using equation 2.4. The QUBO form can be described as having a vector \mathbf{x} , which has all the variables described as x in equation 2.2, also we have c which is our constant. Then a matrix \mathbf{Q} which contains all the weights i.e. h and J values like in the Ising model. The goal of the QUBO is to minimize the energy, so in a factoring case that would mean getting a value that is 0.

$$\min_{x \in \{0,1\}^n} \sum_{(ij) \in E(G)} x_i Q_{ij} x_j - \sum_{i \in V(G)} Q_{ii} x_i + c \quad (2.2)$$

$$\min_{x \in \{0,1\}^n} \mathbf{x}^T \mathbf{Q} \mathbf{x} + c \quad (2.3)$$

$$x_i = (1 - \sigma_i)/2 \quad (2.4)$$

One aspect that is important to note is that relations can maximally be quadratic. If the binary formulation has higher order terms then we call it a HUBO (higher order unconstrained binary optimization). There are several methods of converting a higher order term into a quadratic term. Boros et al. and Dattani et al. have produced reference books on how to do that [10] [18].

Formulating a QUBO problem instance

We shall now take a look at how we can formulate a problem from an integer cost function to a QUBO and then to an Ising model. We shall take an easy example; finding the prime factors of 15.

The formulation starts off by writing the problem for integers as in equation 2.5. We square the equation as we are dealing with an optimisation problem, therefore we want the answer to be 0 not negative infinity. Therefore the correct answers can be $p = 5$ and $q = 3$ as that would result in $(15 - (5 \times 3))^2 = 0$. On the other hand if we were to have $p = 2$ and $q = 7$ then our answer would be $(15 - (7 \times 2))^2 = 1$.

$$0 = (N - p \times q)^2 \quad (2.5)$$

Now that we have formulated the problem in decimal form, we shall now need to formulate p and q in binary, giving every binary its corresponding weight, where q_0 is the least significant bit and q_2 the most significant:

$$= [15 - (p_1 \times 2^1 + p_0 \times 2^0)(q_2 \times 2^2 + q_1 \times 2^1 + q_0 \times 2^0)]^2 \quad (2.6)$$

As we are doing prime factorisation we can make an assumption. As all prime numbers are odd with the exception of 2, and we can easily check if 15 is divisible by 2, so therefore we can substitute the least significant bit with 1.

$$= [15 - (p_1 \times 2^1 + 1)(q_2 \times 2^2 + q_1 \times 2^1 + 1)]^2$$

Now that we have turned the problem into binary representation we need to expand the problem.

$$= 16p_1^2q_1^2 + 64p_1^2q_1q_2 + 16p_1^2q_1 + 64p_1^2q_2^2 + 32p_1^2q_2 + 4p_1^2 + 16p_1q_1^2 + 64p_1q_1q_2 \\ - 104p_1q_1 + 64p_1q_2^2 - 208p_1q_2 - 56p_1 + 4q_1^2 + 16q_1p_2 - 56q_1 + 16q_2^2 - 112q_2 + 196$$

As it is the case that squaring a binary number gives you the same binary number, this means we can substitute all the squares and simplify the formulation.

$$= 16p_1q_1 + 64p_1q_1q_2 + 16p_1q_1 + 64p_1q_2 + 32p_1q_2 + 4p_1 + 16p_1q_1 + 64p_1q_1q_2 - 104p_1q_1 \\ + 64p_1q_2 - 208p_1q_2 - 56p_1 + 4q_1 + 16q_1p_2 - 56q_1 + 16q_2 - 112q_2 + 196$$

We now have a simplified formulation of the problem, yet we do not have a QUBO. As we have a cubic term, this therefore means that we need to substitute it. We shall substitute it by adding a new variable using the following formulation: $p_1q_2q_3 = xq_2 + 2(p_1q_1 - 2p_1x - 2q_1x + 3x)$ as shown in [10].

$$= 128p_1q_1q_2 - 56p_1q_2 - 48p_1q_2 + 16q_1q_2 - 52p_1 - 52q_1 - 96q_2 + 196$$

$$= 128(q_2x + 2(p_1q_1 - 2p_1x - 2q_1x + 3x)) - 56p_1q_2 - 48p_1q_2 + 16q_1q_2 - 52p_1 - 52q_1 - 96q_2 + 196$$

We now have have our QUBO.

$$= 200p_1q_1 - 48p_1q_2 - 512p_1x + 16q_1q_2 - 512q_1x + 128q_2x - 52p_1 - 52q_1 - 96q_2 + 768x + 196$$

In order to convert the problem into an Ising model we shall need to substitute the binary variables with equation 2.4.

$$= 200 \frac{1-s_1}{2} \frac{1-s_2}{2} - 48 \frac{1-s_1}{2} \frac{1-s_3}{2} - 512 \frac{1-s_1}{2} \frac{1-s_4}{2} + 16 \frac{1-s_2}{2} \frac{1-s_3}{2} - 512 \frac{1-s_2}{2} \frac{1-s_4}{2} \\ + 128 \frac{1-s_3}{2} \frac{1-s_4}{2} - 52 \frac{1-s_1}{2} - 52 \frac{1-s_2}{2} - 96 \frac{1-s_3}{2} + 768 \frac{1-s_4}{2} + 196$$

$$= 116s_1 + 100s_2 + 24s_3 - 160s_4 + 50s_1s_2 - 12s_1s_3 - 128s_1s_4 + 4s_2s_3 - 128s_2s_4 + 32s_3s_4 + 298$$

After substituting we can reduce the values by dividing everything by 2.

$$= 58s_1 + 50s_2 + 12s_3 - 80s_4 + 25s_1s_2 - 6s_1s_3 - 64s_1s_4 + 2s_2s_3 - 64s_2s_4 + 16s_3s_4 + 149$$

Finally, we get our coefficients for the linear terms h , and the quadratic terms the J values which we can put into matrices.

$$h^T = (58, 50, 12, -80), J = \begin{matrix} & s_2 & s_3 & s_4 \\ \begin{matrix} s_1 \\ s_2 \\ s_3 \end{matrix} & \begin{pmatrix} 25 & -6 & -64 \\ & 2 & -64 \\ & & 16 \end{pmatrix} \end{matrix}$$

2.3. RSA

The RSA encryption standard is based on the prime factorisation problem for semiprime numbers i.e numbers that are solely the product of two prime numbers [43]. Where our semiprime number we call N and its two factors are p and q . It is an NP problem where it is easy to calculate N from p and q , but hard to calculate p and q when you only have N . There is however a difference between the general problem and what most RSA encryption libraries tend to use, that is that RSA always has an N that has a specific bit length. Currently, this is usually a bit length of 1024, 2048 or 4096, where the bit length of p and q is typically half of N 's bit length, this does not necessarily need to be the case, however for practical reasons it is the case that for a number with an even bit length the bit lengths of p and q

are both half the bit length of N . On the other hand for an odd bit length of N are $L_p = \lceil L_n/2 \rceil$ and $L_q = \lfloor L_n/2 \rfloor$, where L stands for the bit length of a number.

The reason for both prime numbers being the same length is down to how one usually generates prime numbers. This is done by picking a number at random that is of a certain bit length, then testing if it is prime. Testing whether it is prime or not is easy to do through the Miller–Rabin primality test. If not prime then we select another number at random that is of a certain bit length. We do the same for the other factor. Thereby we have our two factors. As it may have already been noticed it is more convenient to simply have a fixed bit length for both factors so that we have a fixed bit length for N , than first randomly choosing a bit length for p and then choosing another bit length for q to get the correct bit length for N .

Therefore, in this thesis we assume that all semiprime numbers are of the RSA form, i.e. the two factors have the same bit length.

2.4. Factoring methods

There are several factoring methods, however we can split the factoring methods into two groups. One being general-purpose, i.e. those that are able to factor all numbers, and special-purpose, i.e. those that are able to factor a small subset of all numbers.

Both classes of factoring methods are important. Especially the special-purpose factoring methods, as they can often factor large numbers in a matter of seconds. Therefore, if one can break 1% or 0.1% or even less of all internet certificates using RSA, then it is still possible to cause a lot of harm.

The current best performing general-purpose factoring method in time is the general number field sieve (GNFS) [11]. This has been the method that has been used to factor all the recent large RSA numbers [28]. There are several other factoring methods, these are generally considered less efficient in time in factoring numbers. GNFS and other general-purpose methods will not be explored in this thesis, as they require a deep understanding of number theory.

What we will mention a couple of times is one special-purpose factoring method, called Fermat's factoring method [7]. It is an important method, as some quantum factoring papers such as by Karamlou et al. have factored numbers that are Fermat factorable [31]. A number N is Fermat factorable if the two factors vary little in their binary form. This means that no matter how large a number is, if the two factors have a small Hamming distance then it can be factored in milliseconds.

The use of numbers that can be factored by Fermat's factoring method in previous quantum factoring papers is problematic, as it undermines the progress made in quantum computing. There is only one reason why these numbers get used, it is because as Dattani et al. show, the QUBO formulation for these numbers can be reduced to a small size [19], unlike most semiprime numbers. Therefore, in this thesis, all numbers that have been factored are not Fermat factorable.

3

Related Work

Quadratic unconstrained binary optimisation (QUBO) problems and the QUBO method of factoring semiprime numbers have been studied extensively in the past decade to fifteen years [14] [25] [32]. In this chapter, we discuss the different research that has been done, and that has made it possible for us to design a new method of factoring semiprime numbers.

The related work chapter is split into three sections; methods of factoring in section 3.1, methods of reducing variables in section 3.2, and methods of running the optimisation problem in section 3.3. All these sections can be seen as doing the following; the methods of factoring are ways in which we can initially formulate the problem; the methods of reduction are ways in which we can make the problem formulation smaller; the methods of running are ways in which we can either improve the accuracy or the efficiency of solving the problem through splitting the problem into sub-problems. We can view the three sections as three overlapping circles or sets that overlap each other as shown in figure 3.1. The reason for them overlapping is that we can argue that certain methods of reduction can change the method so much that we get a new method entirely, and other reductions can be seen as changing the process of solving the problem entirely as it aims to calculate or solve a large chunk of the problem. Finally, as this thesis will not go into detail about quantum annealers or simulated annealers we therefore do not get into a fourth or possibly fifth section on how to best perform the annealing process on a quantum annealer or simulated annealer. Furthermore, there are different devices that can solve Ising or QUBO problem e.g. Digital Annealers and Ising machines [5] [35]. Moreover, other software that is specifically designed to solve optimisation software such as Gurobi and Google OR [42]. Both devices and software we do not talk about in this thesis.

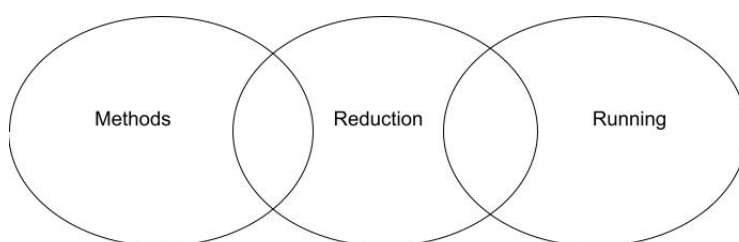


Figure 3.1: Visual overview of chapter

What must also be noted is that not all methods for running and reduction can be combined together, as will become apparent in all corresponding sections. Furthermore, certain methods of reduction may even have no use for this problem. However, we have added them as we felt that they are interesting concepts, and the process of inventing a new method of factorisation was also a process of knowing which methods and techniques could not be applied to this problem.

3.1. Factoring Methods

In this section we shall look at the different factoring methods that have been proposed to factor semiprimes that can be translated into a quadratic unconstrained binary optimization problem (QUBO) or an Ising model, meaning it can be run on a quantum annealing computer. Not all the papers were initially intended for quantum annealing computers. Moreover, the initial paper does not even mention quantum computers at all [13]. Several methods have originally been used to run on a nuclear magnetic resonance quantum computer (NMR) [19] [55], and another was used on quantum gate computer using the Quantum Approximate Optimization Algorithm (QAOA) [4] [38].

3.1.1. Direct Method

The direct method can be seen as the most simple way possible to try and formulate the problem, this was also the method used to give a toy example of how to formulate a QUBO problem in section 2.2. With N being the product and p and q being the factors, the optimisation problem can be seen in integer form in equation 3.1. This method gets used in two papers by Warren [51] and Jiang et al. [30]. Warren looks at how to simply factor small semiprime numbers up to 1000, and Jiang et al. uses it as an example to compare it to another factoring method.

$$(N - p \times q)^2 = 0 \quad (3.1)$$

With the formulation of equation 3.1 it is currently in an integer form which would not be able to run on a quantum annealing computer, therefore we would need to make p and q into a Boolean form, as shown in section 2.2. Here we have to assign p and q with a specific bit length. For example if we were to say that p and q were of bit length 4 we can formulate it as equation 3.2. As all primes with the exception of 2 are odd, we can substitute p_0 and q_0 (the least significant bit) with 1. That means that we can formulate the problem as equation 3.3, where L_p and L_q are the binary lengths of p and q .

$$2^0 \times p_0 + 2^1 \times p_1 + 2^2 \times p_2 + 2^3 \times p_3 = p \quad (3.2)$$

$$(N - (1 + \sum_{i=1}^{L_p} 2^i \times p_i) \times (1 + \sum_{i=1}^{L_q} 2^i \times q_i))^2 = 0, \quad (3.3)$$

The problem with this formulation is that when expanding this problem for larger numbers one will not only end up getting many variables, one will also get many cubic or larger terms as has been shown in section 2.2. This means that not only will one need to use more logical variables as the problem expands. One will also need many more ancillary variables to substitute all the cubic and larger terms.

3.1.2. Table Method

We can now look at the second class of methods called the table method. This was mentioned back by Burges in 2002 as a way of describing the problem as an optimisation problem for simulated annealers [13]. The table formulation is essentially a binary form of long multiplication, where the product N is at the bottom, and the factors p and q are on top. Only in this case do we know the product, but not the factors. Between the factors and the product, every row p gets multiplied by a bit of q .

				p_3	p_2	p_1	p_0
				q_3	q_2	q_1	q_0
				p_3q_0	p_2q_0	p_1q_0	p_0q_0
			p_3q_1	p_2q_1	p_1q_1	p_0q_1	
		p_3q_2	p_2q_2	p_1q_2	p_0q_2		
	p_3q_3	p_2q_3	p_1q_3	p_0q_3			
n_7	n_6	n_5	n_4	n_3	n_2	n_1	n_0

Table 3.1: Generic binary table method example

The formulation in table 3.1 is not complete yet, as it is necessary to add in the carry bits so that it is possible to calculate the variables between columns. As mentioned before there are three versions of this factoring method; the cell, column and block method. The initial version that was applied to a quantum computer was done by Schaller et al. and was first published in 2007 [44].

Cell Method

The initial implementation of the cell method was done as a QUBO in 2007 [44]. Here the method is formulated so that every product of p and q has a corresponding carry (z) and sum (S) variable. This version is called the cell method and has two other implementations, one by D-wave systems using a constraint satisfaction formulation [3]. The second method uses Gröbner basis to reduce the amount of variables [21]. We shall go into more detail on how the Gröbner basis can be used to reduce the size of the QUBO problems in section 3.2.

As can be seen in the table 3.1.2, the table is split into cells, where every product includes a sum value S above the product and a carry variable z below the product. The number of cells is equal to the product of the two binary variables $p \times q$, in other words, $4 \times 4 = 16$. All of these cells remain the same size, unlike the columns in the column method and blocks in the block method. However, the number of cells expands greatly as the number that has to be factored becomes larger.

				p_3 q_3	p_2 q_2	p_1 q_1	p_0 q_0
				0	0	0	0
				p_3q_0 0	p_2q_0 0	p_1q_0 0	p_0q_0 0
		0		S_{21}	S_{11}	S_{01}	
		p_3q_1 z_{31}		p_2q_1 z_{21}	p_1q_1 z_{11}	p_0q_1 0	
	S_{32}	S_{22}		S_{12}	S_{02}		
	p_3q_2 z_{32}	p_2q_2 z_{22}		p_1q_2 z_{12}	p_0q_2 0		
S_{33}	S_{23}	S_{13}		S_{03}			
p_3q_3 z_{33}	p_2q_3 z_{23}	p_1q_3 z_{13}		p_0q_3 0			
n_7	n_6	n_5	n_4	n_3	n_2	n_1	n_0

Table 3.2: Generic representation of the cell method, assuming that p and q are not necessarily odd and not having used the substitutions of 3.5

The cell method can be described as the minimum sum of all the cells, where the cells have to be squared, as can be seen in equation 3.4. Furthermore, as we have the quadratic term p_iq_j , when squaring the cell we shall get cubic terms such as $p_iq_jz_{i,j}$. This means that in order to transform the problem into a QUBO the quadratic term p_iq_j will have to be substituted. This means that for every cell an ancillary variable will have to be added.

$$f = \min \sum_{ij} H_{ij}^2$$

$$H_{ij} = q_i p_j + S_{i,j} + z_{i,j} - S_{i+1,j-1} - 2z_{i,j+1} \quad (3.4)$$

To make the formulation complete, this means that certain variables have to be substituted. These are listed in equation 3.5. These are the edge substitutions, for the sum and carry variables to be

$$\begin{aligned}
z_{0,j} &= S_{1,j-1}, \\
S_{i,0} &= n_i, \\
S_{k+1,j-1} &= n_{k+j}, \\
S_{i,n-k} &= z_{i,n-k} = 0, \\
z_{k,j} &= 0, \\
S_{1,n-k-1} &= 0 \\
\text{Where } k &= L_p
\end{aligned} \quad (3.5)$$

The general benefits of the cell method can be seen that it needs less ancillary variables in comparison to the direct method, this is because there are fewer higher-order terms than in the direct method. Every cell stays the same size unlike the column and block method, and some of the cells could even be easily solved with a binary solver and therefore it may be possible to reduce the number of variables for the entire problem. These advantages shall be discussed in more detail in the section 3.2. As the size of every cell remains the same, it is, therefore, possible to split the problem up into separate parts that are equal in size and run the problem in parallel. When the two other table methods do not divide as neatly into equal-sized sections.

The issue with the cell method is that for every product of two binary variables we need sum and carry variables, and then an ancillary variable to make the formulation a QUBO. In general, this means that we greatly increase the number of variables as the problem gets larger. As opposed to the column and block method (which will be discussed in the next two subsections) there are many extra variables that have to be introduced. Therefore, running the problem as one would be less scalable.

Column Method

To reduce the amount of variables that are needed, one can then formulate the problem in terms of columns, thereby reducing the number of carry variables needed compared to the cell method. This was initially thought up by Burges to factor numbers for simulated annealers [13]. This was then done on a nuclear magnetic resonance quantum computer by Xu et al. [55]. There have been a number of other implementations that use this method and that have tried to improve on these methods. Firstly Dattani et al. where they managed to factor the number 56153 on an NMR quantum computer using preprocessing [19]. Then there is Pal et al., where they used a hybrid method to reduce the number of carries, and then to factor, the number 551 [38]. It was also possible to improve the factoring method using the Gröbner basis in the same paper as that used the same approach to reduce the cell method [21]. There was one method that also did pre-processing and ran the problem on a quantum gate computer, however this can in theory also be run on a quantum annealing computer as it was applied using the QAOA algorithm [4]. All these improved methods are discussed in other subsections.

As can be seen in table 3.3 there are only carry variables and no sum variables. Moreover, in comparison to the cell method example in table , there are fewer carry and sum variables, 9 carries in table 3.3 verses 14 in table . This means that opposed to the cell methods we have to introduce significantly fewer variables. The number of carries per column (as can be deduced from equation 3.6) is dependent on the number of products per column and how far to the left the column is. Furthermore, the middle column, where n_3 is, will have as many products of $p_i \times q_i$ as the bit length of p .

				p_3 q_3	p_2 q_2	p_1 q_1	p_0 q_0
			p_3q_1 p_2q_2 p_1q_3	p_3q_0 p_2q_1 p_1q_2 p_0q_3	p_2q_0 p_1q_1 p_0q_2	p_1q_0 p_0q_1	p_0q_0
	p_3q_3	p_2q_3					
z_{67}	z_{56}	z_{45}	z_{34}	z_{23}	z_{12}		
z_{57}	z_{46}	z_{35}	z_{24}				
n_7	n_6	n_5	n_4	n_3	n_2	n_1	n_0

Table 3.3: This is a formulation of the column method, where we have not substituted the most and least significant bits of p and q

The formulation of the column method can be seen in equation 3.6, where every column needs to be squared in order to form a minimisation problem. Because every column has to be squared, this means that there will be cubic and quartic terms. Therefore more variables will need to be introduced in order to make all terms quadratic.

$$f = \min \sum_{1 \leq i \leq (L_p + L_q + 1)} H_i^2$$

$$H_i = \sum_{j=0}^{L_q} q_j \times p_{i-j} + \sum_{j=1}^i z_{j,i} - n_i - \sum_{j=1}^{L_q+1+i-n_i} 2^{j-i} z_{i,i+j} \quad (3.6)$$

The general advantage of the column method is that it significantly reduces the amount of carry variables needed for the whole problem. This means that larger numbers can be run in one go with a greater probability of success as there are fewer variables as shown in Peng et al. [39], and also larger numbers can be embedded on the quantum annealing computer for instance. Furthermore, the column method is also able split the problem up and run in parallel.

The disadvantage of the column method is that it sits between the block method and the cell method. In that it does not have the same advantage as the block method, as in that it has more carries. Similarly it does not have the advantage that the cell method has, in that the columns do expand in size, in comparison to the cell method where the constituent cells do stay the same size.

Block Method

The final improvement that can be made, is done by combining a number of columns to form a block in order to reduce the amount of carries needed even more. The block method was initially introduced by Jiang et al. [30]. It has also been used by Wang et al. with a different method of turning the cost function into a QUBO [50]. Furthermore, there is a hybrid method of doing the factorisation by Peng et al. that reduces the amount of carries even more [39]. Finally, there is an approach where the columns are solved individually and then the intersection of all the results are used to find the final result by Wang et al. [49].

The general approach of the block method is to combine the columns together, in previous papers this has ranged between 2 to 4 columns; usually dependent on the size of the problem, with exception of the first column (which is left out as we are assuming that both prime factors are odd) and the last column as the most significant bit of n is dependent on carries. We can see that the amount of carries has been reduced even more in table 3.4, from 14 carries in the cell method to 9 carries in the column method to 4 in the block method.

				p_3	p_2	p_1	p_0
				q_3	q_2	q_1	q_0
				p_3q_0	p_2q_0	p_1q_0	p_0q_0
				p_3q_1	p_2q_1	p_1q_1	p_0q_1
				p_3q_2	p_2q_2	p_1q_2	p_0q_2
				p_3q_3	p_2q_3	p_1q_3	p_0q_3
z_{56}	z_{45}	z_{34}	z_{23}				
n_7	n_6	n_5	n_4	n_3	n_2	n_1	n_0

Table 3.4: Generic representation of the block method, without having substituted the values for n and p_0 and q_0

The formulation of the method is given in two parts in equations 3.7 and 3.8, where L_n and L_p are the number of binaries in n and p , and z_k stands for a carry variable for block k . In formula 3.8 we have three cases for how to calculate the blocks, the initial case is for the first block, the last case is for the last block and all the other blocks correspond to the second case. As in the other methods, it is also here that in order to formulate it as a QUBO it will be necessary to add ancillary variables for p_iq_j in order for it to remain quadratic.

$$g_k = \begin{cases} \sum_{i=0}^k p_i q_{k-i} & k < L_p \\ \sum_{i=0}^{L_p-1} p_i q_{k-i}, & k \geq L_p \end{cases}$$

where $1 \leq k \leq L_n, k \in \mathbb{Z}^+$

(3.7)

$$f = \min \left(\left(\sum_{k=0}^{col_1-col_0-1} 2^k g_k - \sum_{k=1}^{col_1-col_0} 2^{k+col_1+col_0} z_k \right)^2 + \dots + \right. \\ \left(\sum_{k=0}^{col_{i-1}-col_{i-1}-1} 2^k g_k + \sum_{k=1}^{col_{i-2}-col_{i-3}} 2^{k-1} z_k - \sum_{k=1}^{col_{i-2}-col_{i-3}} 2^{k+col_{i-1}+col_{i-2}} z_k \right)^2 + \dots + \\ \left. \left(\sum_{k=0}^{col_i-col_{i-1}} 2^k g_k + \sum_{k=1}^{col_{i-1}-col_{i-2}} 2^{k-1} z_k - \sum_{k=1}^{col_{i-1}-col_{i-2}} 2^{k+col_i+col_{i-1}} z_k \right)^2 \right) \quad (3.8)$$

The general advantage of this method can be seen that it has the least amount of carries compared to all other methods. Furthermore it has been shown that it is possible to successfully run the problem in parallel [49].

The general disadvantage can be seen that the constituent parts are the largest out of the table methods. This could be a disadvantage when wanting to split the problem up and run it in sections. As there are a limited amount of qubits available it will become a disadvantage to have such large sections.

3.2. Reduction Methods

In this section we go over the various ways in which the problem can be reduced so that we have less variables. Gröbner basis and split reduction can be seen as overlapping with the methods section as it can transform the binary problem so much that it can be seen as another formulation. Furthermore, Graver basis overlaps with the running section as how it is implemented changes the way it gets run.

3.2.1. Gröbner Basis

Gröbner basis can be seen as the most simple way of writing a set of polynomial equations, potentially splitting a complicated equation into smaller and more easily understood equations ¹.

In our case it can be seen as a useful tool to solve a set of polynomial equations, or at least to simplify the set. There are generally two methods of getting the Gröbner basis. First one is called the Conti and Traverso method (CT Method) [16]. The second method, which we use in the examples, is called the Bertsimas, Perakis and Tayur method (BPT) [9]. Furthermore, there are a number of algorithms that can be used to calculate the Gröbner basis. The most famous and original algorithm is called the Buchberger algorithm [12]. There are a number of other new and improved implementations such as the F4 and F5 algorithm [23] [22]. However, in general they all have the same issue, in that they all take at least exponential time to calculate the Gröbner basis. The reason, however, for writing about Gröbner basis in this thesis is that it was used in one of the factoring papers [21]. Furthermore, the lecture series by Tayur et al. [8], which was used extensively in this research, spent a lot of time on the Gröbner basis.

As a general example we can look at equation 3.9 where we have six equations, the first two can be seen as constraint equations to give the constraints of our problem, the last four are used to say that all the variables are binary. After having given the Buchberger algorithm the polynomial equations we get the Gröbner basis, as can be seen in equation 3.10. Here on out we know that x is equal to 1 and y is the opposite value of z . Therefore, we can reduce the problem to equation 3.11, from here we can calculate all the possible combinations that adhere to the constraints, as can be seen in equation 3.12.

$$\{xyzw + xz + yw - z = 0, x + xy + z - 2 = 0, w(w - 1) = 0, x(x - 1) = 0, y(y - 1) = 0, z(z - 1) = 0\} \quad (3.9)$$

$$\{wz - w = 0, y + z - 1 = 0, x - 1 = 0\} \quad (3.10)$$

$$wz - w = 0 \quad (3.11)$$

$$(x, y, z, w) = \{(1, 0, 1, 0), (1, 0, 1, 1), (1, 1, 0, 0)\} \quad (3.12)$$

Gröbner basis can be seen as being a method to either reduce the number of variables in a set of equations, or to be used as a way to rewrite the equations into a set of smaller equations. For the initial case this can be quite easy to see as that is what we have just done in the example. What has to be remembered is that for most cases we shall need to have a cutoff point in terms of the maximal size of an equation that it can take, as all algorithms that calculate the Gröbner basis do not scale well in time. This is what was done for the cell method in the paper by Dridi and Alghassi [21]. They went on to use the Gröbner basis again to further reduce the amount of variables. Finally one could go even as far to say that Gröbner basis changes the formulation of the method so much so that it is a whole new class by itself, as if it is given a general formula such as in equations 3.4, 3.6 and 3.8 it will rewrite it in a different way.

Unfortunately, this reduction method is rather useless if one were to use it every time to factor a new number, as it is only fast for small equations. However, it may be better for a larger general formulation

¹The formal definition of the Gröbner basis involves using concepts that go beyond the scope of this thesis.

as then you would only need to run it once. However, the same operations can probably be done just as well or better using other algorithms or binary optimisation software.

3.2.2. Energy Landscape Manipulation

Energy landscape manipulation (ELM) can be seen as a way to reduce or increase the energy range of the optimisation problem i.e. lowering or increasing the energy of unwanted solutions, but also as a way to reduce the amount of cubic or larger terms [47]. In other words, ELM can provide a new optimisation function whose minimum still occurs at the same place, but is easier to find.

ELM can work in two parts. Firstly if we are able to deduce that $f = g$ for some polynomial f, g we can manipulate the optimisation problem by adding $\lambda(f - g)$ for some $\lambda > 0$. This part is the deduction ELM, similar to how deduction reduction works in subsection 3.2.4. From equations 3.13 till 3.19 we have the different columns to factor a number using the column method. Equation 3.20 is the final optimisation function that would get sent to the quantum annealer.

$$2p_1 + p_2 + q_2 = 2z_{23} + 4z_{24} \quad (3.13)$$

$$p_1p_2 + p_1q_2 + p_3 + q_3 + z_{23} = 2z_{34} + 4z_{35} + 1 \quad (3.14)$$

$$p_1p_3 + p_1q_3 + p_2q - 2 + z_{24} + z_{34} + 2 = 2z_{45} + 4z_{46} \quad (3.15)$$

$$2p_1 + p_2q_3 + p_3q_2 + z_{35} + z_{45} = 2z_{56} + 4z_{57} \quad (3.16)$$

$$p_2 + p_3q_3 + q_2 + z_{46} + z_{56} = 2z_{67} + 4z_{68} + 1 \quad (3.17)$$

$$p_3 + q_3 + z_{57} + z_{67} = 2z_{68} + 4z_{79} \quad (3.18)$$

$$z_{68} + z_{79} = 1 \quad (3.19)$$

$$H_0 = (2p_1 + p_2 + q_2 - 2z_{23} - 4z_{24})^2 + \dots + (z_{68} + z_{79} - 1)^2 \quad (3.20)$$

However, in equation 3.13, if $z_{24} = 1$ then we must have $p_1 = p_2 = q_2 = 1$, this can be encoded by doing the following in equation 3.21.

$$z_{24}(1 - p_1), z_{24}(1 - p_2), \text{ and } z_{24}(1 - q_2) \quad (3.21)$$

This can then in turn be used to rewrite the equation as H_1 in 3.22.

$$H_1 = H_0 + z_{24}(3 - p_1 - p_2 - q_2) \quad (3.22)$$

We can finally do a similar deduction reduction for z_{79} where we get H_2 .

$$H_2 = H_1 + z_{79}(4 - p_3 - q_3 - z_{57} - z_{67}) \quad (3.23)$$

The second part of ELM can be seen as giving every part of the optimisation problem or cost function a weight λ . For example given an optimisation problem that is made out of two parts like equation 3.24, we wish to find two weights λ in order to reduce the energy range. This can be done in two ways. Firstly, we find what the max energy for every block is, e.g. $(0 + 0 - 1 - 1)^2 = 4$ and $(0 + 0 - 2 - 1 - 1)^2 = 16$. Therefore we can set the $\lambda_2 = 4\lambda_1$.

$$H_0 = (x_1 + x_2 - x_3 - 1)^2 + (x_1 + x_1x_2 - 2x_2x_3 - x_2 - 1)^2 \quad (3.24)$$

$$H_1 = \lambda_1(x_1 + x_2 - x_3 - 1)^2 + \lambda_2(x_1 + x_1x_2 - 2x_2x_3 - x_2 - 1)^2 \quad (3.25)$$

This would be the naive method as this does not always decrease the energy landscape and can even increase it. Therefore, as a general refinement it is best to keep to the rules in equation 3.26. Where E_i is the max energy for equation i , and E_{max} maximum energy of all the equations.

$$\lambda_i = \begin{cases} 1, E_i = E_{max} \\ 2, E_i \neq E_{max} \end{cases} \quad (3.26)$$

The advantage of this technique is that it can be relatively easy to implement this method, and can be quite an effective way of reducing a cubic function or a part of a function that is cubic.

Unfortunately, it may not always be possible to use this deduction, and searching for the possible deductions may be harder than as given in the example as it certain sections of the optimisation problem may be large.

3.2.3. Removing Carry variables

Removing the carry variables has been done by Pal et al. for the column method, and Peng et al. for the block method [38] [39]. In general, the concept is uncomplicated, as not all the corresponding product bits for every column or block are equal to 1. Therefore, the maximum number of carries, that were initially assigned, are not all necessary.

The advantage of using this method is that it is uncomplicated to implement and a time efficient way to reduce the number of variables. Therefore making the problem less complex.

Even though the calculation is seen as being simple, in terms of reducing the total amount of variables it is negligible. Furthermore, the methods used do not always seem clear nor applicable for all numbers. Therefore, it can be seen as a potentially easy improvement, yet on the grand scale of reducing or simplifying the formulation it does little.

3.2.4. Deduction Reduction

The deduction reduction method is a way in which one tries to reduce the amount of variables by looking at certain parts of the problem and asserting whether they are equal to 1 or 0 [48]. This is a suitable method for reducing the size of a factoring method as it is made out of smaller blocks, cells or columns. Even though, the paper's methods do not explicitly say they use this method, both Anschuetz et al. and Dattani et al. use a similar method to reduce the number of variables [4] [19]. Anschuetz et al. uses a similar technique where they remove all the linear terms to further decrease the amount of variables in the problem.

We can see this in the example of equations 3.27 till 3.29. From this formulation we get the optimisation problem in equation 3.30 where we have cubic and quartic terms.

$$x_1 + x_2 + x_3 = 1 \quad (3.27)$$

$$x_1x_4 + x_2x_5 = x_3 \quad (3.28)$$

$$x_1 + 2x_2 = x_3 + 2x_4 \quad (3.29)$$

$$\begin{aligned} H &= (x_1 + x_2 + x_3 - 1)^2 + (x_1x_4 + x_2x_5 - x_3)^2 + (x_1 + 2x_2 - x_3 - 2x_4)^2 \\ &= 2x_1x_2x_4x_5 - 2x_1x_3x_4 - 2x_2x_3x_5 - 2x_2x_3 + 6x_1x_2 - 3x_1x_4 - 8x_2x_4 + x_2x_5 + 3x_2 + 4x_3x_4 + x_3 + 4x_4 + 1 \end{aligned} \quad (3.30)$$

However, with the equations 3.27 till 3.29 we can deduce that certain products must be equal to 0 and we can therefore replace them with 0.

$$x_1x_2 = x_2x_3 = x_3x_1 = 0 \quad (3.31)$$

However, if we were to naively substitute the values with 0 we would get equation 3.32. on the surface this may be alright, however the minimum value is not 0 anymore but -3. Therefore, the quantum annealer may not reach the right answer.

$$H = -3x_1x_4 - 8x_2x_4 + x_2x_5 + 3x_2 + 4x_3x_4 + x_3 + 4x_4 + 1 \quad (3.32)$$

Therefore the reduction has to be done in a different way. We want an error term so that it is equal to 0 if $x_1x_2 = 0$ and strictly greater than 0 if $x_1x_2 \neq 0$. We can see all the following reductions:

1. Solution can be either 0 or 2: $2x_1x_2x_4x_5 \rightarrow 2x_1x_2$
2. Solution can be either 0 or -2: $-2x_1x_3x_4 \rightarrow 0$
3. Solution can be either 0 or -2: $-2x_2x_3x_5 \rightarrow 0$

Therefore we finally get a different optimisation problem where the minimum possible energy is 0 in equation 3.33.

$$H = 8x_1x_2 - 2x_2x_3 - 3x_1x_4 + x_2x_5 + 3x_2 + 4x_3x_4 + x_3 + 4x_4 + 1 \quad (3.33)$$

The advantage is that we are able to reduce the amount of cubic terms quite easily. The deductions however are only easy to find for small equations like in equation 3.29. For larger equations this will be more complicated and there will be more brute forcing necessary.

3.2.5. Excludable Local Configuration

An Excludable Local Configuration (ELC) is a partial assignment of variables that make it impossible to achieve the minimum [29].

$$H_{3-local} = b_1b_2 + b_2b_3 + b_3b_4 - 4b_1b_2b_3 \quad (3.34)$$

In equation 3.34 if $4b_1b_2b_3 = 0$, no other assignment of our variables will be able to reach a lower energy than if $4b_1b_2b_3 = 1$. Hence this gives us twelve ELCs, and one example is $(b_1, b_2, b_3) = (1, 0, 0)$ which we can use to form the polynomial:

$$H_{2-local} = H_{3-local} + 4b_1(1 - b_2)(1 - b_3) \quad (3.35)$$

$$= b_1b_2 + b_2b_3 + b_3b_4 + 4b_1 - 4b_1b_2 - 4b_1b_3 \quad (3.36)$$

The advantage can be seen in that we have removed a cubic term, or in other cases quadratic or single terms.

The disadvantage is that we cannot easily find the ELC's except through brute force, therefore when applying this method it can only really be done on smaller chunks. Furthermore, we do not necessarily have a guarantee that they exist for every cubic or quartic term.

3.2.6. Split Reduction

The split reduction method is a way of reducing the amount of cubic or greater terms in an equation by substituting a variable with a 1 and a 0, thereby creating two smaller equations that are easier to calculate. It was introduced 2015 by Okada et al. [36].

This technique can be seen as being useful in two ways. Firstly as a way to reduce the cubic or higher terms. Secondly, as a way to split the terms into smaller chunks so that Gröbner basis calculation or similar calculations can become easier.

To give an example we can first look at equation 3.37, where we have three cubic terms and one quartic term.

$$H = 1 + b_1b_2b_5 + b_1b_6b_7b_8 + b_3b_4b_8 - b_1b_3b_4 \quad (3.37)$$

We can split the equation in two parts, by substituting b_1 with 1 and 0, so we end up getting two equations. Both still have cubic terms, but H_0 can be easily solved by introducing an auxiliary variable. However, H_1 needs to be reduced even further.

$$H_0 = 1 + b_3b_4b_8 \quad (3.38)$$

$$H_1 = 1 + b_2b_5 + b_6b_7b_8 + b_3b_4b_8 - b_3b_4 \quad (3.39)$$

We can make H_1 quadratic by substituting b_8 with 1 and 0, we get two quadratic terms.

$$H_{1,0} = 1 + b_2b_5 - b_3b_4 \quad (3.40)$$

$$H_{1,1} = 1 + b_2b_5 + b_6b_7 \quad (3.41)$$

The benefit of using this method is that it is easy to implement and one can implement it on all higher order binary optimisation problems so that we can turn them into QUBO form. Moreover, this means that large equations can be split up into smaller equations that can possibly be solved on their own.

The problem, however, is that we cannot be sure this makes all equations less complicated, and may also make it even more complex as we may split the initial large equation into a large number of equations. Furthermore, we do not know how many split reduce steps we must take.

3.2.7. Graver Basis

Graver Basis is similar to Gröbner basis in that we end up with what can be called as test sets, i.e. a simplified set of equations to describe the problem. The difference lies in how we apply the Graver basis. As instead of using the test sets to have a simplified solution, one can use the Graver basis for augmentation in order to get to the optimal solution. Graver basis was first described in the 70's, however only later was there an algorithm to calculate it [26] [40]. Furthermore, its application for linear and integer programming only became popular in 2010 [37]. More recently a method has been invented that uses a hybrid of quantum annealing and classical computation, their implementation we shall use to show how Graver basis can be used for these kinds of problems [2].

The use of Graver basis is made out of two parts, the initial one is calculating the kernel (the test set), the second part is calculating an initial solution and then using the kernel to get to the optimal solution.

The idea of the kernel or the test set is that for every non-optimal but feasible solution (i.e. a solution that adheres to all the constraints) x_0 there exists $t \in S$ and $\lambda \in \mathbb{Z}$ such that $x_0 + \lambda t$ is feasible and $f(x_0 + \lambda t) < f(x_0)$. The vector t is called the improving or augmenting direction, and S is the set of test sets.

For the kernel to work, all vectors inside the kernel need to be conformal. With conformal we mean that for vectors $x, y \in \mathbb{R}$, where $x \sqsubseteq y$, when $x_i y_i \geq 0$ and $|x_i| \leq |y_i|$ for $i = 1, \dots, n$. Additionally, a sum $u = \sum_i v_i$ is called conformal, if $v_i \sqsubseteq u$ for all i (u majorizes all v_i)

Also before we get into every step of how we can use graver basis we need to define the lattice integer kernel of A , where A is the matrix for the values $Ax = b$.

$$\mathcal{L}(A) = \{x \mid Ax = 0, x \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}\} \setminus \{0\} \quad (3.42)$$

We can say that the Graver basis of A is:

$$\mathcal{G}(A) \subset \mathcal{L}(A) \subset \mathbb{Z}^n \quad (3.43)$$

Such that $\mathcal{G}(A)$ is \sqsubseteq - minimal i.e. $\nexists x, y \in \mathcal{G}(A)$ s.t. $x \sqsubseteq y$
Finally our encoding is as follows:

$$x^T = [x_1 x_2 \dots x_n] \quad x_i \in \mathbb{Z} \quad (3.44)$$

$$x_i = e_i^T X_i \quad (3.45)$$

$$x_i^T = [X_{i,1} X_{i,2} \dots X_{i,k_i}] \quad (3.46)$$

$$e_i^T = [2^0 2^1 \dots 2^{k_i}] \quad (3.47)$$

Finally to set a lower bound to allow for negative kernel values and reformulate:

$$x = L + EX = \begin{bmatrix} L_{x_1} \\ L_{x_2} \\ \vdots \\ L_{x_n} \end{bmatrix} + \begin{bmatrix} e_1^T & 0 & \dots & 0 \\ 0 & e_2^T & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & e_n^T \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix} \quad (3.48)$$

Now that the definitions have been given, we can now go through the steps on how to get the Graver basis:

1. Given a constraint matrix (i.e. all constraint equations in the form of a matrix) we turn it into a quadratic unconstrained integer optimisation matrix i.e. given matrix A as in $Ax = 0$ we turn it into the formulation $\min x^T Q_I x$ where $Q_I = A^T A$.
2. We use a two bit encoding as $e = [2^0 2^1]$, we then get the encoding matrix E . We shall also shift one step to the left with $L = -1$ to cover negative values. Then encode vector e to matrix E
3. Encode equation to $x = L + EX$
4. Reformulate it to QUBO form as follows $\min (L + EX)^T Q_I (L + EX)$
5. Then map it to Ising variables and reformulate as an Ising problem.
6. Solve the Ising problem and convert back to binary variables.
7. Decode and recover the kernel
8. convert kernel to Graver Basis, do classical \sqsubseteq - minimal filtration to get $\mathcal{G}(A)$

Then we find an initial solution, and we can augment to find our final optimal solution.

The benefit of this technique is that the annealer or solver does not initially have to find the optimal solution. Through getting the initial solution you can get to the optimal solution.

The disadvantages may be that calculating the kernel may simply be too difficult or take an unfeasible amount of time and that it may just be better to try and solve the original problem. Furthermore, the kernel may be incredibly large.

3.3. Running Methods

Running the problem can be seen as the way in which we can divide the problem into different chunks and how we feed the problem to an annealer or QUBO solver. Here we shall view the annealer or solver as being an oracle or black box that can only solve a problem that is smaller than a certain size. The approaches can be split into two groups. The first as being run as one problem, the other as being split into sections.

Both have advantages and disadvantages. The advantage for solving it as one problem is that if it can be solved as one, then it is the fastest way possible. The disadvantage is that as the problem gets bigger it either does not fit on the annealer or the accuracy becomes so low that we need to do so many anneals that being able to get the answer takes too much time. Therefore after a certain size we must split the problem into sub-problems in order to calculate the solution. The problem is that we firstly will need to split the problem into subsections, this may either not be possible nor may it be possible to split the problem into small enough problems for the anneal/solver to solve it. Secondly, when solving the sub-problem we get noise in the form of non-valid solutions with an energy of 0 as can be seen in figure 3.2. In other words, these are solutions that are correct for the subproblem, but not for the entire problem. For example, variable c_1 could be both 1 and 0 in subproblem 1, yet in subproblem 2 it may be the case that c_1 can only be 0. Thirdly, the number of calls to the annealer or solver may even grow exponentially with the increase in bit length of N , as the size of the entire problem increases exponentially so do the number of subproblems. Furthermore, it may also take a long time to merge all the solutions together, either because of the number of subproblems, or also because of the number of incorrect solutions with an energy of zero. This can mean an increase in spatial complexity when having to merge the solutions to the subproblems together. The problem of the increase in spatial complexity will be dealt with in section 4.4 in the next chapter.

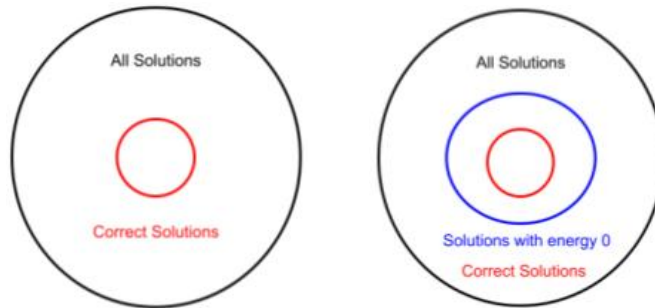


Figure 3.2: Representation of normal case vs the distributed case

In this section, we shall briefly go over methods that have been done before, and present a couple of concepts of our own.

3.3.1. As one problem

Most of the methods are run as one problem formulation, this is quite obvious as it is the easiest way to solve the problem. Furthermore, it may even be advantageous as all the needed qubits are present to solve it.

The issue however arises when the formulation gets too large. Firstly, the problem formulation may become too large for the annealer/solver, therefore not able to even run the problem. Secondly, Peng et al. shows that the accuracy can decrease as the QUBO gets larger [39]. Even using reduction methods as described in section 3.2, it is not always possible to reduce the QUBO to such a small enough size that it can be quickly solved.

3.3.2. Parallel

To try and avoid the problem becoming too large so that a quantum computer cannot solve it, or to avoid losing accuracy as the problem gets greater. We can try and split the problem up into smaller chunks and to try and run them individually, i.e. parallel or distributed as shown in 3.3. This was proposed by Wang et al. [49]. As can be seen in equation 3.49, all sections of the whole optimisation problem are to be squared, therefore the individual correct answers can also be solved by solving all functions

individually in equation 3.50. Here every block gets run individually and then the answers of every block get intersected together, where in the end you should get the final correct answer(s).

$$f = \min \sum H_i^2 \quad (3.49)$$

$$\begin{aligned} f_0 &= \min(H_1^2) \\ f_1 &= \min(H_2^2) \\ &\dots \\ f_n &= \min(H_n^2) \end{aligned} \quad (3.50)$$

The disadvantage is firstly that often one cannot run the problem in parallel, as one may not always have access to multiple quantum annealing computers. In that case, it may not be more advantageous to run the problem sequentially as opposed to running the problem longer or more times on a quantum computer. Furthermore, for the block and column methods, the central block/column will increase in size as N 's bit length gets larger, therefore there is still an upper limit. This will however not be the case for the cell method, as its cells do not increase in size

Blocks	III			II		I	
p				1	p_2	p_1	1
q				1	q_2	q_1	1
$p \cdot q$				1	p_2	p_1	1
				q_1	$p_2 q_1$	$p_1 q_1$	q_1
		q_2	$p_2 q_2$	$p_1 q_2$	q_2		
	1	p_2	p_1	1			
Carries	c_4	c_3	c_2	c_1			
N	1	0	0	0	1	1	1




Figure 3.3: Graphical representation of parallel process

3.3.3. Sequential

Running sequentially can be seen as a derivative of running the problem in parallel. As we know what the possible correct solution or solutions are for certain variables after having run the problem for one part of the problem, we could then substitute in the values for those variables for the other parts of the problem that have not been solved yet, thereby reducing the complexity of the parts individually. A graphical representation is given in figure 3.4, where the leftmost column is initially solved, its answers are fed into the next column which is then solved, and the process continues till the leftmost column has been reached.

This can be seen as advantageous, as it is the case that for every table method the most left or right-hand block/cell/column has the least amount of variables. Therefore, easier to calculate and find the corresponding answers for those variables.

The problem, however, is knowing whether you have found the correct answer for a variable. It may be the case that the wrong answer has been found for a variable, this will then likely mean it is impossible to find the solution to the entire problem. Furthermore, it may also be the case that no definite answers are found for variables in subproblems.

Blocks	III			II		I		
p				1	p_2	p_1	1	
q				1	q_2	q_1	1	
$p \cdot q$				1	p_2	p_1	1	
				q_1	$p_2 q_1$	$p_1 q_1$	q_1	
		q_2	$p_2 q_2$	$p_1 q_2$	q_2			
	1	p_2	p_1	1				
Carries	c_4	c_3		c_2	c_1			
N	1	0	0	0	1	1	1	1

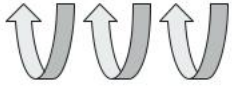


Figure 3.4: Graphical representation of sequential process

3.3.4. Meet in the middle

This can be seen as a corollary of the sequential way of running the problem. Instead of solving the problem from right to left or vice versa, we could solve the problem from both ends, as seen in figure 3.5.

As mentioned in the sequential subsection, as it is the case that both extremities have the least amount of variables, they are therefore the easiest parts to solve. Going from both extremities inwards would mean less computing time, and potentially will mean a greater rate of success as along the way the larger parts of the problem get reduced in size due to variables being solved.

However, this problem suffers from the same issues as parallel and sequential way of running the problem.

Blocks	III			II		I		
p				1	p_2	p_1	1	
q				1	q_2	q_1	1	
$p \cdot q$				1	p_2	p_1	1	
				q_1	$p_2 q_1$	$p_1 q_1$	q_1	
		q_2	$p_2 q_2$	$p_1 q_2$	q_2			
	1	p_2	p_1	1				
Carries	c_4	c_3		c_2	c_1			
N	1	0	0	0	1	1	1	1

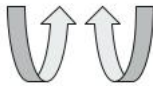


Figure 3.5: Graphical representation of meet in the middle process

3.3.5. Divide and Conquer

Divide and conquer approach in general is well known in classical computing. For QUBO problems it is less commonly used. The general idea is to split the problem into smaller parts and to solve those parts, and then to stick these smaller solved parts back together to get the solution. The main question is how one should split it up.

Guerreschi in 2021 proposes it for the max cut problem, here he ran it on a universal quantum computer [27]. However, as it was a QUBO the approach can be also be applied to an annealer or QUBO solver. What the paper found was that for a quantum computer the rate of success increased, albeit not for a simulated annealer.

The approach the paper takes is by drawing the problem as a graph like in figure 3.6 and doing community detection on the graph, thereby splitting the problem into smaller chunks.

The problem with divide-and-conquer approach is the case that you have to merge the solutions together. Which again may have issues in terms of large spatial complexity as mentioned in the other distributive methods. Besides the general problem, it may not be useful for all methods. As the block and column methods can have large columns, it may not be useful as you may just be grouping the largest columns and blocks together, which are too large for the annealer to solve anyway. Moreover, there are several ways in which one can split a graph into smaller sections, there currently is not a

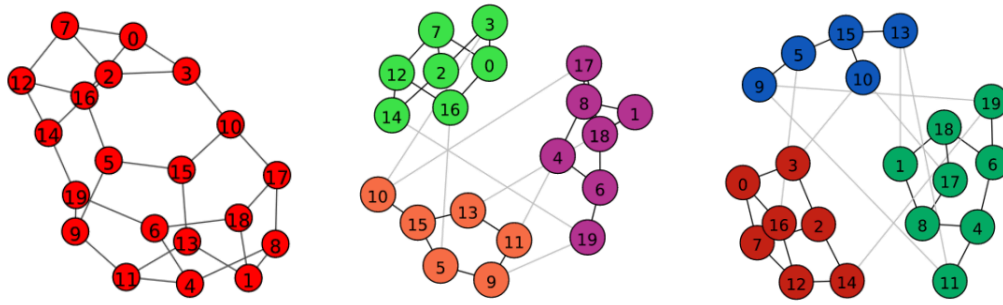


Figure 3.6: Graphical representation of divide and conquer method taken Guerreschi 2021 [27]

definitive answer on how to best split these problems into smaller chunks. Finally, we do not know how well the community detection algorithm scales in time.

4

Factoring Method

In summary of what we have shown in the related work chapter is that the only method that is able to be split up into sections that have a constant size, is the cell method. We then saw that several of the reduction methods were not relevant for this specific problem. Furthermore, we also noticed that no single or combination of the reduction methods could reduce the size of the problem to a significantly smaller problem all the time.

Therefore, the formulation of the problem is not scalable, as either the size of the problem would mean that it was too large for a quantum computer to take, or too large so that the chance of finding the solution would be zero. That is why it became important to look at methods of splitting the problem up into sub-problems and run these individually. As we know that a quantum annealer or a simulated annealer can only solve problems of a certain size, it was therefore important that as the size of the problem grew, the size of the sub-problems would stay constant. The only method that would have equal sizes for all sub-problems no matter how large the problem would get was the cell method.

The best approach to solving the cell method distributively is the approach of solving the problems in parallel and then combining the solutions together. This is because the cells are so small that they could be easily solved in parallel. Furthermore, there was little guarantee that in the stage of merging or intersecting the answers together the sequential or meet in the middle methods would have any advantage over the parallel method. Moreover, with the divide and conquer method there is also little guarantee that splitting the cells using the community detection algorithm would create a better split in the problem nor did it seem that this algorithm would be more efficient than splitting the cell method into its constituent cells.

In this chapter we shall therefore present our design of how to solve the cell method by solving every cell individually and then combine these cells together to get the final answer.

Our implementation was written in Python, and our implementation was based on the code that had been written initially by HPQC labs [46]. Furthermore, all equations and expressions were written with the help of Sympy the symbolic computation library [34].

The chapter is made out of the following sections; we shall revisit the cell method in section 4.1; then we shall go over how we can solve every cell individually in section 4.2; before looking at ways of reducing the size of the problem in section 4.3; then we shall look at how to merge the solutions together and note that the bottleneck is that we get an exponential rise in the spatial complexity in section 4.4. To try and solve this issue we look at ways of reducing the spatial complexity. We initially start with a section on how to view this problem as a search problem in section 4.5. This is used to find three different techniques of reducing the spatial complexity problem which we present in section 4.6. All symbols used in this chapter are listed in table 4.1.

Symbol	Definition
N	The RSA number we wish to factor
n_i	bit i of N
p, q	the two factors of N , where $p > q$
p_i	bit i of factor p

Symbol	Definition
q_i	bit i of factor q
S_i	Sum bit i
z_i	carry bit i
L_N, L_p, L_q	bit length of N, p and q
$\pi(x) = \frac{x}{\ln x}$	prime counting function π , it approximately gives the position of the prime relative to other prime numbers

Table 4.1: List of symbols used in the Factoring Method chapter

4.1. Cell Method Formulation

Before we start looking at our new method of factoring, we first need to recap on what the cell method does and how it can be split into smaller sections.

To be able to start factoring we first need to formulate the different equations. This method of formulating was first introduced by Schaller and Schützhold in 2007 [44]. In the original case it was meant to be described as a minimisation problem where every term C_{ij} is squared and added together, as can be seen in the formula 4.1. Where the S variables are the sum values and the z variables are the carry variables. Finally, as there are certain edge cases where one variable is equal to another or equal to zero. Then we must add these substitutions as can be seen in equations 4.2. Finally, the most significant bit is the most right bit in table 4.2 i.e p_0 and q_0 , they are all set to 1 as we know that both p and q are of the same length. Furthermore, as said in previous sections we know that the least significant bit must also be equal to 1. i.e. p_3 and q_3 in table 4.2.

$$f = \min \sum_{ij} H_{ij}^2$$

$$H_{ij} = q_i p_j + S_{i,j} + z_{i,j} - S_{i+1,j-1} - 2z_{i,j+1} \quad (4.1)$$

$$\begin{aligned} z_{0,j} &= S_{1,j-1}, \\ S_{i,0} &= n_i, \\ S_{k+1,j-1} &= n_{k+j}, \\ S_{i,n-k} &= z_{i,n-k} = 0, \\ z_{k,j} &= 0, \\ S_{1,n-k-1} &= 0, \end{aligned}$$

$$\text{Where } k = L_p \quad (4.2)$$

If one were to implement this representation the result would look like table 4.2. Where the number of cells is equal to the product of the two binary variables $L_p \times L_q$, in other words in this example $4 \times 4 = 16$. What should also be noted is that all the cells on the edges will have less than 6 variables due to the substitutions that have been made using equations 4.2. The cells not on the edges will all have six variables. Also what can also be observed is that the q variables go horizontal, and the p variables go diagonally.

The terms of the cell method for factoring 143, where p and q both have a bit length of 4, can be seen in terms 4.3.

				1	p_2	p_1	1
				1	q_2	q_1	1
				1	p_2	p_1	1
				0	0	0	0
		0		S_{21}	S_{11}	S_{01}	
		q_1		$p_2 q_1$	$p_1 q_1$	q_1	
		z_{31}		z_{21}	z_{11}	0	
		S_{32}	S_{22}	S_{12}	S_{02}		
		q_2	$p_2 q_2$	$p_1 q_2$	q_2		
		z_{32}	z_{22}	z_{12}	0		
	S_{33}	S_{23}	S_{13}	S_{03}			
	1	p_2	p_1	1			
	z_{33}	z_{23}	z_{13}	0			
n_7	n_6	n_5	n_4	n_3	n_2	n_1	n_0

Table 4.2: Generic representation of the cell method 4.2

$$\begin{aligned}
& -2 * S_{10} + S_{11} + z_{11} + 1 \\
& -2 * S_{11} + S_{12} - S_{21} + q_1 + z_{12} \\
& -2 * S_{12} - S_{22} + q_2 + z_{13} \\
& -2 * S_{13} - S_{23} + 1 \\
& S_{21} + p_1 - 2 * z_{11} + z_{21} \\
& S_{22} - S_{31} + p_1 * q_1 - 2 * z_{12} + z_{22} \\
& S_{23} - S_{32} + p_1 * q_2 - 2 * z_{13} + z_{23} \\
& \quad - S_{33} + p_1 \\
& S_{31} + p_2 - 2 * z_{21} + z_{31} \\
& S_{32} - S_{41} + p_2 * q_1 - 2 * z_{22} + z_{32} \\
& S_{33} - S_{42} + p_2 * q_2 - 2 * z_{23} + z_{33} \\
& \quad - S_{43} + p_2 \\
& S_{41} - 2 * z_{31} \\
& S_{42} + q_1 - 2 * z_{32} - 1 \\
& S_{43} + q_2 - 2 * z_{33} - 1
\end{aligned}$$

(4.3)

Generating the cells takes approximately the least amount of time compared to all other sections in this algorithm. The code that we implemented was based on the code from HPQC lab's implementation of the cell method [46], where we made some changes to make the code run more efficiently. The psuedocode of the implementation can be seen in algorithm 1. Even though this part of the algorithm takes the least amount of time, there are still improvements that can be made. Firstly by doing the substitution inside the first loop, and secondly by making the algorithm run in parallel.


```

Data:  $L_p, L_q, N$ 
Result: equations
 $L_N \leftarrow L_p + L_q$ ;
 $k \leftarrow L_p$ ;
edgeSubs = [ $p_1$ : 1,  $q_1$ : 1,  $p_{L_p}$ : 1,  $q_{L_q}$ : 1];
equations  $\leftarrow$  [];
targetDigits = ToBinaryArray( $N$ );
for  $i = 1$  to  $(k + 1)$  do
    for  $j = 1$  to  $(L_N - k + 1)$  do
        equation =  $p_i * q_j + S_{ij} + z_{ij} - S_{(i+1)(j-1)} - 2 * z_{(i-1)j}$ ;
        equations  $\leftarrow$  equations + [equation];
        if  $i == 1$  then
            edgeSubs[ $z_{0j}$ ] =  $S_{1(j-1)}$ ;
            edgeSubs[ $S_{1(product-k-1)}$ ] = 0;
        end
        if  $j == 1$  then
            edgeSubs[ $S_{0j}$ ] = targetDigits[ $i-1$ ];
        end
        if  $i == k$  then
            edgeSubs[ $S + (k + 1)(j - 1)$ ] = targetDigits[ $k + j - 1$ ];
            edgeSubs[ $z_{kj}$ ] = 0;
        end
        if  $j == product - k$  then
            edgeSubs[ $S_{i(product-k)}$ ] = 0;
            edgeSubs[ $z_{i(product-k)}$ ] = 0;
        end
    end
end
for  $x = 0$  to  $length(equations)$  do
    for  $var$  in  $variables(equations[x])$  do
        if  $var$  in  $keys(edgeSubs)$  then
            equations[ $i$ ]  $\leftarrow$  SubstituteVariable(equations[ $x$ ],  $var$ , edgeSubs[ $var$ ]);
        end
    end
end

```

Algorithm 1: Generating equations

4.2. Solving Cell Method Terms

Due to the cells at most only having 6 variables, this means that all cells do not have to be solved by a quantum annealer or any other annealer or special solver. Instead, it can simply be solved outright using Sympy's own solve function, or any other symbolic or binary solver. The Sympy solver returns a list of all the possible correct solutions. The correct solution is represented as a dictionary, in order to make processing of these results easier we transform the list of dictionaries into a Panda's data frame, which is essentially a table, which we can see as a truth table, as can be seen in table 4.3.

In the code of algorithm 2 we can see how the solver works. The maximum time observed to solve an equation was 0.01 seconds on a computer with a 2,2GHz 6-Core Intel i7 processor. Furthermore, the process of solving all the equations can be sped up through parallisation as none of the operations need to be done sequentially. There can be even more improvements done to this algorithm through only solving the edge cells. As all the inner cells are of the form given in equation 4.1, therefore all the truth tables will look like table 4.3. Therefore we could simply generate these tables without having to solve them.

	S2_2	S3_1	p2	q2	z1_2	z2_2
0	0	0	0	0	0	0
1	0	0	0	1	0	0
2	0	0	1	0	0	0
3	0	0	1	1	1	1
4	0	1	0	0	0	1
5	0	1	0	1	0	1
6	0	1	1	0	0	1
7	0	1	1	1	0	0
8	1	0	0	0	1	1
9	1	0	0	1	1	1
10	1	0	1	0	1	1
11	1	0	1	1	1	0
12	1	1	0	0	0	0
13	1	1	0	1	0	0
14	1	1	1	0	0	0
15	1	1	1	1	1	1

Table 4.3: General truth table of a term, all terms that are given here are all the possible correct terms. The left most column represents the row number.

Data: Equations

Result: SolvedEquations

SolvedEquations $\leftarrow []$;

for $i = 0$ to $\text{length}(\text{Equations})$ **do**

 SolvedEquations[i] $\leftarrow \text{BinarySolve}(\text{Equations}[i])$;

end

Algorithm 2: Solving equations

4.3. Reducing the number of variables

The reduction step can be seen as an optional step as the merging phase can work without it. However, the merging step will probably take longer, as without the reduction step it will mean that the number of unnecessary variables and rows in the tables will be greater. Therefore, a greater increase in data complexity.

The reduction section is made out of two parts, one is finding the different constraints per table, and the second part is enforcing these constraints so to reduce the size of the tables.

There are several different constraints one could find per table, but to keep it simple we only looked at cases for either one variable, or the relationship between two variables. The six deductions that we used were:

- 0 x is equal to a constant (0 or 1)
- 1 x is equal to variable y
- 2 x is the opposite of y
- 3 x is greater than y
- 4 x is less than y
- 5 x and y are never both 1 (NAND)

The wrapper function that calls the deduction and reduction functions can be seen in algorithm 3, and works as follows; it starts by getting all the truth tables from the `solveEquations` function. Then it creates a dictionary for every option we are dealing with; 0 for constants, 1 for equals, 2 for not equals, etc. . The `option` variable is initialised to 0 and we enter a while loop where the condition is that the `option` variable is less than 6.

In the while loop we assign the last deductions of that option to a new variable, and reset the deductions of that option. We then loop through every truth table, to find a deduction and add it to the deductions array.

After the deductions have been found, we check for the set difference between the new deductions and the last deductions. If there is a set difference, then we can reduce all the results. After that, we can assign the option to 0 to see if new constant deductions can be found. If however, it is not the case that there were new deductions found, then we increment the option variable and start looking at new methods of deduction.

Data: solvedEquations

Result: solvedEquations

option \leftarrow 0;

deductionFunctions \leftarrow [FindConstants(), FindEquals(), FindOpposites(), FindGreaterThanOr(), FindLessThanOr(), FindNand()];

reducerFunctions \leftarrow [EnforceConstants(), EnforceEquals(), EnforceOpposites(), EnforceGreaterThanOr(), EnforceLessThanOr(), EnforceNand()];

deductions \leftarrow [[],[],[],[],[],[]];

while option < 6 **do**

 lastDeductions \leftarrow length(deductions[option]);

for $i = 0$ to length(solvedEquations) **do**

 deductions[option] \leftarrow deductions[option] +
 deductionFunctions[option](solvedEquations[i]);

end

if lastDeductions < length(deductions[option]) **then**

for $i = 0$ to length(solvedEquations) **do**

 solvedEquations[i] \leftarrow
 reducerFunctions[option](solvedEquations[i],deductions[option]);

end

 option \leftarrow 0;

else

 option \leftarrow option + 1;

end

end

Algorithm 3: Wrapper function that calls the deduction and reduction functions

We managed to speed up the algorithm by making the deductions and reductions run in parallel. There are probably other areas of improvement for this algorithm, either in the number of reductions that can be made e.g. doing comparisons between three or more variables or in efficiency. However, this section of the factoring method takes relatively little time in comparison to the merge algorithm, therefore it did not become our main focus.

A simple example of how the reduction and deduction can be done is where we know from one table a variable is constant e.g. in table 4.4 where we can see that both p_4 and q_4 are equal to 1, and another table where we have p_4 where it can be 1 or 0. After doing this reduction we also know that S_{41} is equal to 0. Therefore, being able to reduce another table as seen in table 4.5.

There are two modes of doing the reductions, one is the “standard” one as shown in tables 4.4 and 4.5, here the variables do not get replaced. Then we can also have a “ruthless” version, where for example if S_{11} is equal to p_1 in one cell, then for all other cells where S_{11} is present, we replace S_{11} with p_1 .

p4	q4	S4_1	p4	z3_1
0	1	1	0	0
			1	0

Table 4.4: Example of two tables before reduction

p4	q4	S4_1	p4	z3_1
0	1	1	0	0

Table 4.5: Same tables after the reduction has been made

4.4. Merging the solutions together

The merging phase is the final part of the algorithm. Here all the truth tables get merged together so that we know what the values are of the two factors. The algorithm can be seen in algorithm 4. The method starts off by selecting the truth table that has both the variables p_1 and q_1 , then all the following truth tables get selected in order of most significant p or q variable to the least significant p or q variable. In this case we chose to first select q and then select p variables. The joining of the two tables is done through a hash join, which can be seen in algorithm 5. During this process filtering gets done which will be explained in sections 4.5 and 4.6, this has the function of reducing the size of the data and also trying to find a factor early. When the method has gone through all the variables of q or p , the program goes through all the rows of the final answer and looks at which answer for q or p could be a factor of N .

Data: solvedEquations

Result: finalResult

finalResult, solvedEquations \leftarrow GetFirstEquation(solvedEquations);

while length(solvedEquations) > 0 **do**

 variable \leftarrow GetMostSignificantVar(); // Get next most significant variable of p or q

if variable == null **then**

 // If there are no more variables then we can check all rows to see if it contains one of the factors

return checkAnswers(finalResult);

end

 result, solvedEquations \leftarrow GetNextResult(finalResult,solvedEquations,variable);

 finalResult \leftarrow HashJoin(finalResult,result);

 finalResult, answer \leftarrow doFilters(finalResult);

if answer != null **then**

return answer;

end

end

Algorithm 4: Merge results

The hash join as seen in algorithm 5 is done firstly by making a hash table and hashing the variables that are both in the finalResult and the current table on one end, and adding the corresponding variables on the other. The second part of the algorithm works by initialising a new dictionary, and then looping through all the results in the finalResult table, and also looping through the corresponding values inside the hash table. Here the joining happens, it is initially done by calculating the maximum possible values for p and q . The values are first checked if they are factors of N , then check if pmax is larger than qmax, if that is the case then the new value is inserted into the newResults dictionary, with the product of the pmax and qmax as the key and the joined values as the value. After the looping has been done, the newResults dictionary gets sorted and the dictionary gets converted into an array.

```

Data: result, finalResult
Result: finalResult, factor
hashTable  $\leftarrow$  MakeHashTable();
intersect  $\leftarrow$  keys(result)  $\cap$  keys(finalResult);
factor = null;
for  $i = 0$  to length(result) do
    variables, values = getVariablesValue(result[i],intersect);
    hashTable[variables] = values;
end
newResults = dictionary();
for row = 0 to length(finalResult) do
    for  $h = 0$  to length(hashTable[row]) do
        pmax, qmax = CalculateMaxResults(join(finalResult[row],hashTable[row][h]));
        if checkIfFactor(pmax) == True then
            factor = pmax;
        end
        if checkIfFactor(qmax) == True then
            factor = qmax;
        end
        if pmax > qmax then
            newResults[pmax*qmax] = join(finalResult[row],hashTable[row][h]);
        end
    end
end
newResults  $\leftarrow$  Sort(newResults);
finalResult  $\leftarrow$  GetValues(newResults);

```

Algorithm 5: HashJoin tables

Overall as will be discussed in chapter 5, this section takes the longest to run and is the largest bottleneck. There are a number of ways in which this algorithm can be improved, the most notable one would be making sure that the finalResult table needs to be sorted and is kept sorted throughout the process.

Bottleneck of merging

The merge phase is the final step, and as the size of the number gets larger this part of the algorithm starts to take the most amount of time. There are many methods of how one can make this process faster through parallelization. The naive fastest method would be a divide-and-conquer merge construction. However, the order of how the merging happens affects the speed of the merging process greatly as the spatial complexity increases greatly.

Our initial strategy was to start with the truth table with the least amount of rows and then consistently choose the next smallest truth table which could be merged. The merging would continue till all tables had been merged as can be seen in graphs 4.1, the number of the rows increases sharply and then plateaus till the very end when the number of rows decreases rapidly. The rapid decline happens as the final results start to filter out the answers.

We decided to change this method to the one that we have now, i.e. start with the cells with the most significant bits of p and q and continue downwards to the least significant bits of p and q , as we noticed that by the time the table had reached its peak in size it would contain at least one of the factors, or even both. Therefore, it was unnecessary to merge all the answers.

Furthermore, when testing the initial method we noticed two things. Firstly, the number of rows during the merging can be dependent on the merge strategy. Secondly, the number of rows approximately doubles every time the length of the bit length increases by 2. As we can see for figures 4.1 where 829201 is a 20 bit number and maximally has 128 rows, then for 13911017 which has a bit length of 24 bits we can see that it has at its peak 512 rows. Looking at other results we can come up with the formula that the peak number of rows will be 2^x , where $x = (\log_2(N) - 4)/2$. This will therefore mean that if we were to factor a 100-bit semiprime we would be dealing with

11692013098647223345629478661730264157247460343808 rows. A data complexity too large for any computer to work with.

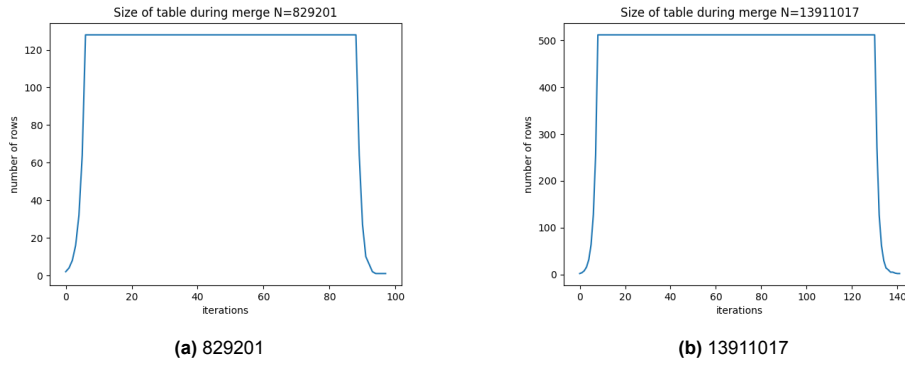


Figure 4.1: Number of rows during the merging process

This would not necessarily be a problem if it were not the case that our merging operation were running in polynomial or linear time, which will be discussed in chapter 5. As can be seen in figures 4.2 we see that the time it takes to merge is usually proportional to the number of rows in practice. Therefore slowing the process of factoring down.

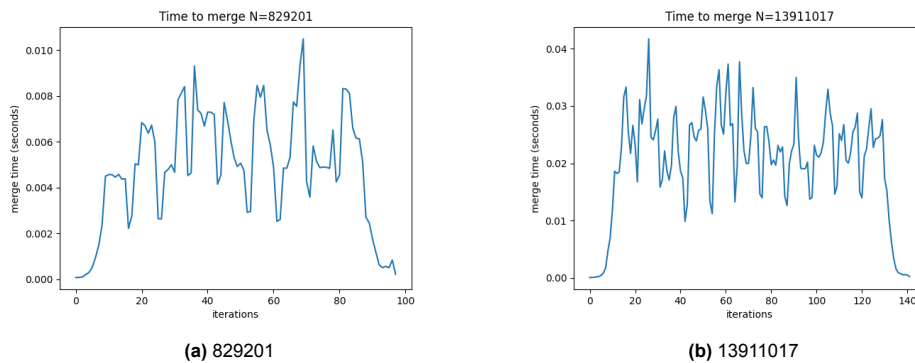


Figure 4.2: Time in seconds for every merge

As we can see from the two graphs 4.1 and 4.2 we have a problem with data complexity, where at the current pace we would not be able to factor any significant numbers that are any larger than the current best methods, nor will we be able to factor any numbers any faster. However, we can do two things; firstly, we can try and find one of the factors before the end of the merging process; secondly, we can try and filter out bad candidate answers. In sections 4.5 and 4.6 we shall show how certain methods of filtration can be made.

4.5. Viewing the problem as a tree

We shall introduce three methods of factoring that are not used in the final method. However, they can be seen as being useful to understand what we have done to reduce the size of the problem in section 4.6.

As we can see in figure 4.3 every variable for p and q can also be seen as a node in a binary tree. Where p_1 and q_1 are the most significant bits of both numbers. We also can observe that the left node is always 0 and the right node is 1. Every child node will be the opposite factor of the current node, so if the current node is a p both child nodes will be qs . The alternation has been chosen because it makes it possible to easily calculate the maximal and minimal possible product of p and q . At the bottom of the tree are the leaf nodes, where we can calculate the values of p and q , one of the leaf node's product of p and q will be equal to N . In terms of the properties of the tree, we can see that the size of the tree doubles every time we go down one level.

There are several ways of traversing trees to find an answer. However, there are two basic ways, breadth first search and depth first search. We shall look at both ways and how we can use these methods to factor numbers.

What should be noted is that using a trivial searching algorithm is not efficient for two reasons. Firstly, because of the potential size of the numbers that are to be factored are so large, as the size of the tree will double every time you go one level deeper, thereby taking an unreasonable amount of time to factor. Secondly, we are specifically doing prime factorisation. Therefore, after a certain depth there may not be primes between the maximal possible leaf (The rightmost leaf where all downwards nodes are 1) and the minimal possible leaf (The leftmost leaf where all downwards nodes are 0). Furthermore, as will be discussed later it is hard to discern approximately where N should be at the bottom of the tree.

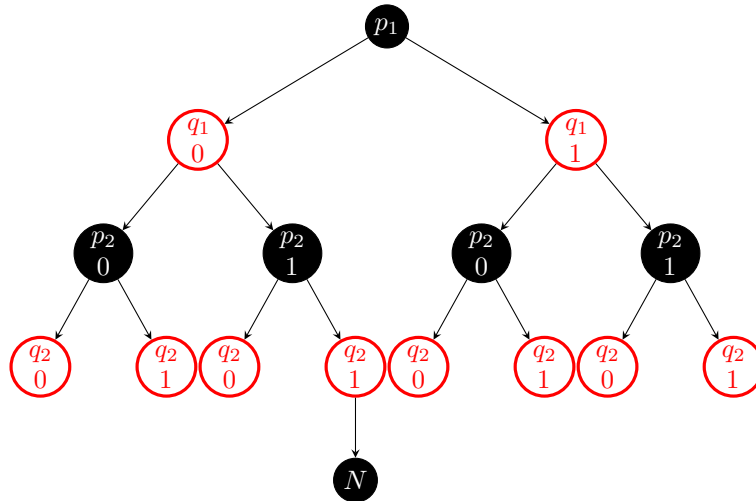


Figure 4.3: Tree view of the factoring problem

4.5.1. Depth-first Search Tree Factoring

The depth-first search (DFS) approach works by doing a normal depth-first search, either by looking first at the smaller numbers, meaning inclining to move to the child node with a zero as value instead of one, or prioritising larger numbers, therefore moving in the opposite direction. The example given in figure 4.4 shows all the steps of the DFS algorithm to get to N . In this case, the algorithm goes from left to right.

What we can do during every move is check if the leaves below the node have a possibility of being equal to N . This can be done by calculating the minimum and the maximum possible answers. If it is the case that the minimum possible answer is larger than N or that the maximum possible answer is smaller than N , then we can remove the node and subtree, and search in a different direction.

This process has its obvious pros and cons, it is advantageous if the answer is on the far left or right of the tree, less so if more in the middle. Furthermore, it is also good in that there is little overhead in terms of data that needs to be stored, as the search path does not need to be held in memory.

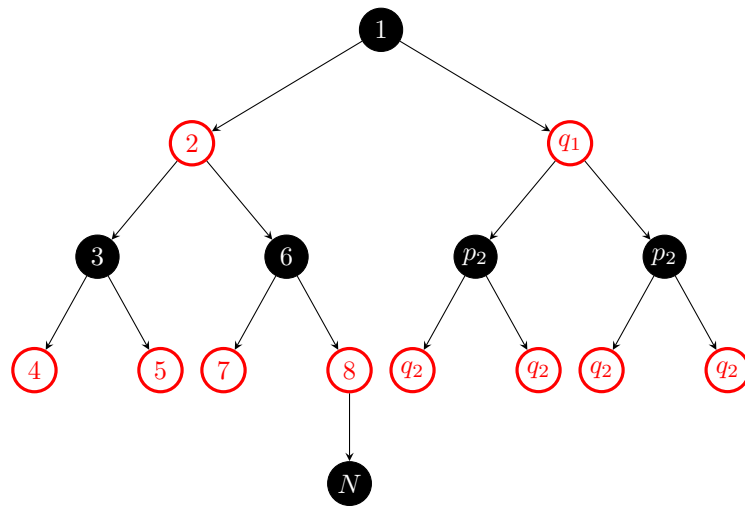


Figure 4.4: DFS method of factoring

4.5.2. Breadth-First Search Tree factoring

The breadth-first search (BFS) approach works in the opposite way. BFS tries all the nodes at the present depth, checking if they have viable answers by checking the maximal and minimal answers, after having checked all the nodes at the current depth it moves down to the next depth. This method won't be as fast or slow as the DFS version but will have the same average because it takes longer to get to the leaf nodes as opposed to DFS. An example of how BFS works can be seen in 4.5

Just like in the DFS version we can improve on this method by checking what the maximum and minimum possible answers are, and only continue in the cases where N is between those two extremes.

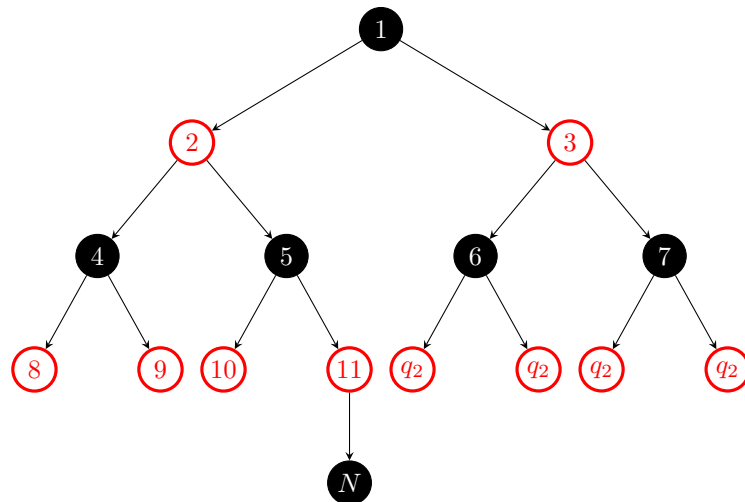


Figure 4.5: BFS method of factoring

4.5.3. Priority Search Factoring

The priority factoring method is a combination of both methods. Here we use a priority queue, where we calculate the priority of the node as being the difference between N and the average of the maximal and minimal numbers, if the N is between the maximum and minimum possible answer. The node with the smallest difference will be at the start of the queue, and the largest difference will be at the end of the queue. An example of how this method would work can be seen in figure 4.6.

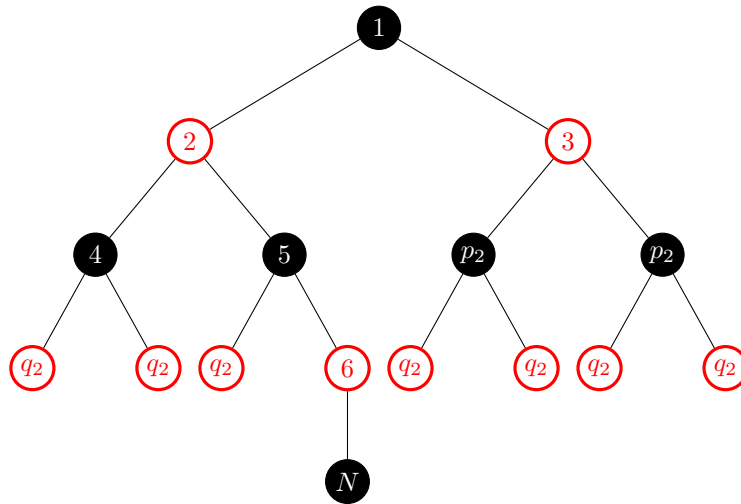
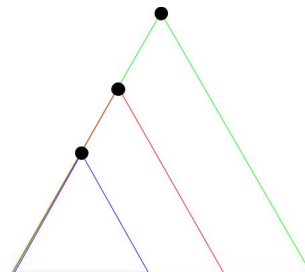


Figure 4.6: Priority search method

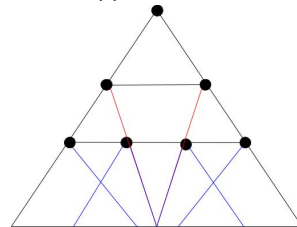
4.5.4. Weaknesses in RSA

Despite the fact that all tree methods in the worst case have to visit all vertices, they all have the advantage of being able to factor certain semiprime numbers quickly. Furthermore, these are not trivial cases as we can exploit the fact that we can check the min and max cases, where we also go and find the maximum and minimum prime numbers for both p and q . If at least p or q are factors of N then we can stop our search. This means that through searching the maximum and minimum primes we can come closer to finding the answer.

We can view the entire binary tree as a triangle where we start at the top and work our way down, as shown in figure 4.7. Somewhere at the bottom of the triangle is our answer N . Every time we visit a new node going down the tree, we check what the maximum and minimum semiprimes are, if N is between those numbers then we can continue down this path. Every time we are on a new node, we therefore are finding new primes. Only one of these primes needs to be a factor of N , if done properly, then we can find four new primes every iteration, therefore four possible factors of N .



(a) DFS method



(b) BFS method

Figure 4.7: Graphical representation of DFS and BFS factoring methods

4.6. Filtering

As for every merge step we get in the worst case a doubling of data every iteration, i.e. an exponential increase in spatial complexity. To reduce the spatial complexity we have invented three methods with which we can reduce the space; Min-Max filtration, Prime Filtration and Dynamic Case filtration. All three are meant to reduce different sections of the search space. As can be seen in figure 4.8 we can view the space of candidate answers as a triangle, which represents the area of a binary search tree, where the top contains the root node and the bottom the leaf nodes. Here we can see that the Min-max method (red) reduces the answers in the far left and right from start to finish; the Dynamic Case Filtration (yellow) starts a little later and filters the centre of the triangle. Finally, the prime filter (turquoise) filters the bottom of the triangle.

We shall explain how these methods work, what their strengths and weaknesses are.

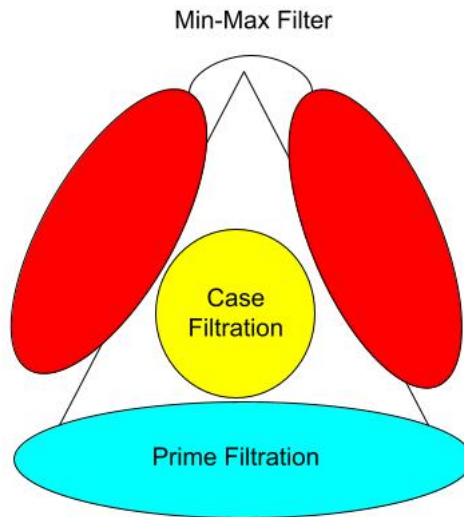


Figure 4.8: What areas the different filtration methods try to reduce.

4.6.1. Min-Max filtration

If we were to view the space we are dealing with as a binary search tree, and the number N being the product of one of the leaf node's p and q , like what has been shown in section 4.5. We can traverse the nodes to a certain depth and see what their maximum and minimum possible answers are. Where the maximum is where all the following bits are 1 and the minimum is where all the following bits are 0. If N is between the minimum and maximum, then we can proceed going down this path, but if it is not, then we can remove the node, as shown in figure 4.9.

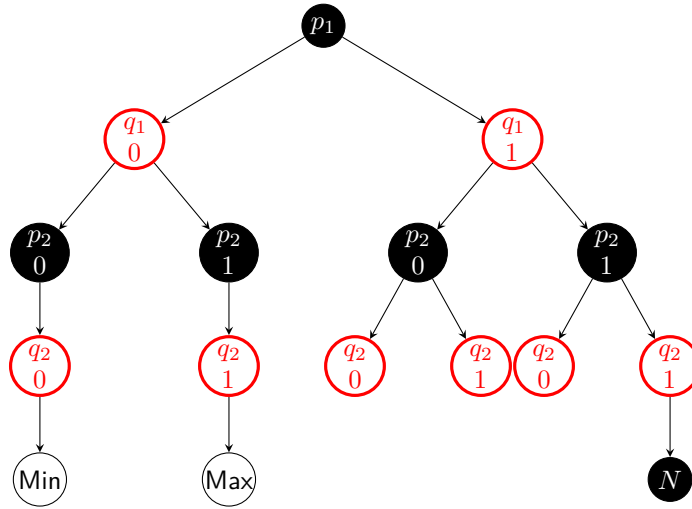


Figure 4.9: Min-Max filtration method

In our case we are not dealing with a tree, but an array of dictionaries. However, as we have the array ordered by the maximum possible answer for every case, then we can easily do a binary search on all the options for both extremities, i.e. checking for the maximal end and the minimal end, as can be seen in algorithm 6 where the maximum answers get removed.

```

Data: finalResult, N
Result: finalResult
low  $\leftarrow$  0;
high  $\leftarrow$  length(finalResult) - 1;
while low  $\neq$  high do
    mid  $\leftarrow$  (low + high)/2;
    pmin, qmin  $\leftarrow$  GetMinimumAnswer(finalResult[mid]); checkIfFactor(pmin)
    checkIfFactor(qmin) //Both pmin and qmin must be prime
    if pmin  $\times$  qmin  $>$  N then
        high  $\leftarrow$  mid - 1;
    else
        low  $\leftarrow$  mid + 1;
    end
end
return finalResult[0:high]

```

Algorithm 6: Filter Max Function

We improved this method in two ways. The improvements can be seen as closing the difference between the max and min possible answer. Firstly, we can say that we want the max and min possible ps and qs to be prime numbers. Secondly, we can reduce the maximum answer, by subtracting the maximum product with the minimum q , as we know that it is not possible to have an answer between max and max - qmin. Furthermore, as shown in algorithm 6 we can check if the q or p is a factor of N .

There are three issues with this method. One of them is that it only filters the extremities and not from the middle. As it is extremely probable that somewhere in the centre the factors for N will lie, therefore, it will probably take a long time to whittle down the answers till we get our solution. The second is that the range does not necessarily reflect the possible answers between the maximum and minimum answers. Meaning that even though N may be between the maximal and minimal answers, it is unlikely that N is below this node. Finally, this method is reliant on the data being ordered.

4.6.2. Prime Filtration

There are two facts we know for certain about prime numbers. Firstly, there are infinite many primes [15]. Secondly, the frequency of primes becomes sparser and sparser as our numbers get larger [20]. This means that the further we are in the merging process the less likely there will be prime numbers in every case.

For this filtration we can use Prime Number Theorem (PNT) and Cramer's or Wolf's conjecture [17] [54]. These theories can be used to estimate if there is a prime number between two numbers. It is generally accepted that there are infinitely many twin primes, i.e. two prime numbers with a gap of 2 [53]. However, there are several conjectures for the maximal gap between two prime numbers, but the most notable would be Cramer's and Wolf's conjectures [17] [54].

Cramer's conjecture is about the maximal gap between two primes, it estimates that the maximal possible gap between two primes is:

$$p_{n+1} - p_n = O(\ln(p_n)^2) \quad (4.4)$$

PNT introduces a function called the prime-counting function that tells us what the n th prime number is, denoted as $\pi(p_n) = x$ where p_n denotes the n th prime. This is largely an estimation but gets better as the prime numbers get bigger. The prime-counting function's definition is:

$$\pi(x) = \frac{x}{\ln(x)} \quad (4.5)$$

PNT also says that the maximal gap between a composite number is $n < p < 2 \times n$, where p is the prime number and n is the composite number and $n > 1$.

Finally, Wolf's conjecture estimates the maximum gap to be smaller, which is currently assumed to be closer to the actual max possible gap [54]. Its definition is:

$$\frac{x}{\pi(x)} (2 \ln(\pi(x)) - \ln(x) + c_0) \quad (4.6)$$

, where $c_0 = \log(C_2) = 0.2778769...$ and $C_2 = 1.320326...$, which is twice the twin primes constant.

We can use these formulas as a way to estimate if there is a possibility that between the maximum and minimum values of a subtree there is the possibility of having no primes, as seen in figure 4.10. Therefore, if we get to that point it makes sense to brute force that range to see if there are no prime numbers and if there are to check if they are factors of N . This can be used in tandem with the dynamic case filtration method. As it is likely that when the cases have a distance that is smaller than Wolf's or Cramer's conjecture, we can assume that the space complexity is large. Therefore it would be a good idea to do several subtrees at the same time, where the cases are either spaced in the centre or spread out. Our implementation is shown in algorithm 7.

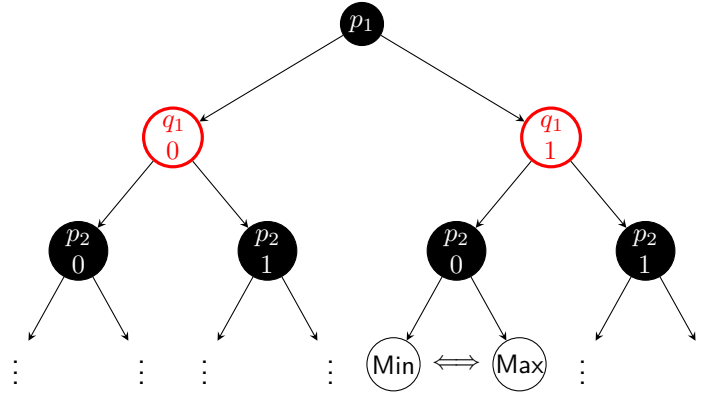


Figure 4.10: Graphical representation of prime number filter

Data: finalResult, ranges, N
Result: finalResult, ranges, factor
 budget \leftarrow maxDistance;
while budget > 0 **do**
 $i \leftarrow$ pickRandomRow(length(finalResult));
 qmin \leftarrow calculateQminPrime(finalResult[i]);
 qmax \leftarrow calculateQmaxPrime(finalResult[i]);
 //get q's maximum and minimum possible primes
 distance \leftarrow calculateDistance(qmax, qmin, ranges);
 //calculate distance by checking if parts of the range have already been scanned
 if calcWolfGap(qmin) > distance && budget > distance **then**
 budget \leftarrow budget - distance;
 factor \leftarrow ScanDistance(qmax, qmin, N, ranges);
 //go through all odd numbers between qmax and qmin and check if one the numbers is a factor of N
 ranges \leftarrow UpdateRanges(qmax, qmin, ranges);
 if factor != null **then**
 return finalResult, ranges, factor; //We have found our answer
 end
 finalResult \leftarrow finalResult[0:i-1] + finalResult[i+1:-1];
 else
 budget \leftarrow budget - distance;
 end
end
return finalResult, ranges, null;

Algorithm 7: Prime Filtration

4.6.3. Dynamic Case Filtration

The problem with the prime filter approach is that early on in the merge process it has no use as the gaps are probably so big that there is likely to be at least a prime number in that gap. Moreover, towards the end there are probably several cases where the gap is likely to have no primes, but because of the expansion in size it may by then become less effective, only being able to remove needles in a haystack.

Building on the prime filtration method is that we can simply try and see if between two numbers of p or q there is a number that is a factor of N . To avoid checking all gaps, we give it a budget that it can only check a certain distance every turn. Furthermore, we also keep track of the ranges that we have scanned, keeping them in memory in order to avoid scanning that area again. After having scanned the ranges, we can remove candidates that fall inside the scanned ranges. Therefore, reducing the space complexity of the problem, and is represented in figure 4.11.

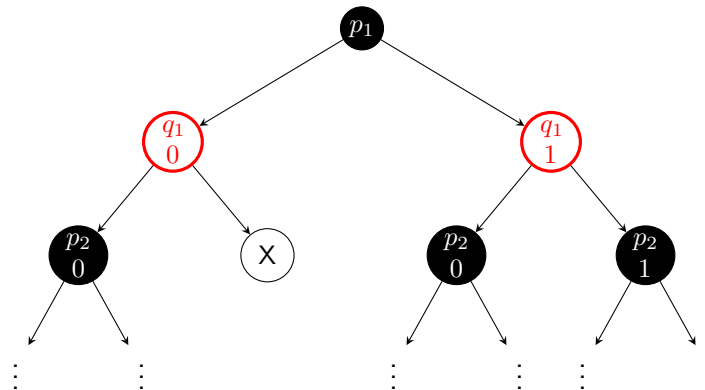


Figure 4.11: Graphical representation of the dynamic filter

The advantage of this method of filtration is that it can be implemented on the centre of the array of cases, not just on the maximum and the minimum cases. Furthermore, it can also be applied earlier

in the process of merging than prime filtration, and can be used in conjunction with the other filtration methods.

Our implementation is made up of two parts. The first part as shown in algorithm 8 is where we scan the candidate in the centre of the finalResult. The algorithm also takes two other parameters, ranges that have already been scanned and the maximum distance the algorithm is allowed to scan. The algorithm calculates the max and min possible q answers and scans that distance and checks if between these two numbers there is a number that factors N . After having scanned that distance the range gets added to the array of ranges.

```

Data: finalResult, ranges, maxDistance
Result: finalResult, ranges, factor
budget  $\leftarrow$  maxDistance;
i  $\leftarrow$  length(finalResult) / 2;
qmin  $\leftarrow$  calculateQminPrime(finalResult[i]);
qmax  $\leftarrow$  calculateQmaxPrime(finalResult[i]);
distance  $\leftarrow$  calculateDistance(qmax,qmin,ranges);
budget  $\leftarrow$  budget - distance ;
factor  $\leftarrow$  null;
while budget > 0 do
    factor  $\leftarrow$  ScanDistance(qmax,qmin,N,ranges);
    ranges  $\leftarrow$  UpdateRanges(qmax, qmin, ranges);
    finalResult  $\leftarrow$  finalResult[0:i-1] + finalResult[i+1:-1];
    if factor  $\neq$  null then
        | return finalResult, ranges, factor;
    end
    i  $\leftarrow$  length(finalResult) / 2;
    qmin  $\leftarrow$  calculateQminPrime(finalResult[i]);
    qmax  $\leftarrow$  calculateQmaxPrime(finalResult[i]);
    distance  $\leftarrow$  calculateDistance(qmax,qmin,ranges);
    budget  $\leftarrow$  budget - distance ;
end
return finalResult, ranges, factor

```

Algorithm 8: Dynamic search

The second part of the dynamic case filtration method is the “welding” algorithm as shown in algorithm 9, where if the distance between two ranges is smaller than the maxDistance given then the two ranges can be “welded” together by scanning the distance between the two ranges and making the two ranges one.

```

Data: ranges, maxDistance
Result: ranges, factor
factor  $\leftarrow$  null;
i  $\leftarrow$  1;
while i < length(ranges) do
    if ranges[i-1][1] - ranges[i][0] < maxDistance then
        | factor  $\leftarrow$  scanDistance(ranges[i-1][1],ranges[i][0]);
        | ranges[i-1]  $\leftarrow$  [ranges[i-1][0],ranges[i][0]];
        | ranges  $\leftarrow$  ranges[:i-1] + ranges[i+1:];
    else
        | i  $\leftarrow$  i + 1;
    end
end
return ranges, factor

```

Algorithm 9: “Weld” ranges together

5

Evaluation

In this chapter we evaluate the new factoring method that we have designed. The evaluation is given in two sections. The first part is the theoretical evaluation of every part of the factoring method, where we evaluate the algorithmic complexity of the algorithm. In the second part we look at how the method runs in practice for RSA numbers of several sizes. The list of symbols used in this chapter are given in table 5.1.

Symbol	Definition
N	The RSA number we wish to factor
n_i	bit i of N
p, q	the two factors of N , where $p > q$
p_i	bit i of factor p
q_i	bit i of factor q
S_i	sum bit i
z_i	carry bit i
L_N, L_p, L_q	bit length of N , p and q
M	number of cells in the problem
L	number of variables in the problem
K	number of rows in the final result Dataset
R	number of ranges

Table 5.1: List of symbols used in the Evaluation chapter

5.1. Theoretical Evaluation

This section will go over all the sections of the algorithm to see what the algorithmic complexity is. To avoid confusion we shall **not** use the letter N for the big O notation but will use the letter M and other letters.

5.1.1. Generating cells

Generating the cells is made out of two sections, the first section by generating the cells and the second by doing all the substitutions.

The first section is made by doing two for loops, from 1 going up to $k + 1$ and a second for loop from 1 that goes up to $L_N - k + 1$, where k is L_p . Every iteration generates a new cell. As both for loops generate all the cells, that means that $M = (k + 1) \times (L_N - k + 1)$, where M stands for all the cells that have been generated. Therefore we can conclude that the first section is $O(M)$.

In the case where $L_p = L_q$, it will mean that $L_N - L_p = L_p$. Therefore, $M = (k + 1) \times (L_N - k + 1) = (k + 1) \times (k + 1) \approx k^2 < L_N^2$. All cases where $L_p \neq L_q$, we shall get the case that $L_N \leq M < L_N^2$.

The second part is where variables in the cells get substituted. Here again the for loop goes over all the cells, which takes M iterations. Then loops over all the variables inside the equation, to find if a substitution has to be made. As mentioned all the cells are of the form $p_i \times q_j + S_{ij} + z_{ij} - S_{(i+1)(j-1)} -$

$2 \times z_{(i-1)j}$, therefore it only has to check 6 variables which is a constant, therefore we can conclude that the algorithmic complexity is also $O(M)$. Therefore, we can say that the entire algorithmic complexity is $O(M)$.

5.1.2. Solving cell method terms

For solving the cells, if we say that the length of the array of cells is M . As shown in section 4.2, in order to solve all the cells we need to loop over the entire array of cells.

Furthermore, we know that the time it takes to solve the cell is dependent on the number of variables in the cell, and as the number of variables is either 6 or less, we can say that the solving of a cell is of constant time. Therefore, we can conclude that the algorithmic complexity of this section of the algorithm takes $O(M)$.

5.1.3. Reduction of the number of variables

Again we denote the length of the array of the solved cells as being M . As can be seen in algorithm 3 after having initialised the variables, the algorithm enters a while loop that only breaks till the option variable is greater than 6. To determine the maximum number of iterations that can be taken we need to first look at the for loops inside the while loop.

The first nested loop is where the deductions are made, it is simply going over every solved cell and running a deduction function on it. We can safely say that the running of the deduction function can be seen as a constant, as it is dependent on the size of the solved cell, as we know that the maximal size is constant no matter how large the problem gets we will be dealing with the same size. Therefore, we can say that the algorithmic complexity of this nested loop is $O(M)$.

In the second nested loop we have a similar situation to the previous nested loop. Here the loop goes over all the solved cells, and checks if the variables inside the solved cell are in the current deduction dictionary, if so then the reduction gets made by the reducer function. As we also know that the number of variables per solved cell is 6 or less we can also say that the reduction function is a constant. Therefore the algorithmic complexity is $O(M)$.

Finally now that we know what the algorithmic complexity of the two nested loops are, we must now look at what the worst case scenario would be for the while loop. The worst case would be that all variables get solved, and at the end of running of this algorithm we would have solved the entire problem. The worst possible case for this would be that per iteration of the outer loop only one variable gets solved. Therefore, we can say that the outer loop takes $O(L)$ iterations, where L stands for the number of variables in the entire problem, where L 's relation to M is $L = M + (L_N/2 - 2)$.

In conclusion, we can therefore say that the algorithmic complexity of this entire algorithm is $O(L \times M)$.

5.1.4. Merging the solutions

The merging part of the algorithm is firstly made out of a while loop. The number of iterations that have to be taken for the while loop to run are at most equal to the number of variables in both factors p and q , thus the number of iterations for the initial loop that have to be made is $O(L_p + L_q)$.

Every iteration the algorithm finds a new result for it to merge into, at most this takes $O(M)$ iterations, where M denotes the number of results. Then the algorithm does a hash join and filters the solved cells.

The hash join initially goes through all the answers of the result to make a hash map. As the number of rows per result can only maximally be 16 rows, we can therefore say that this is constant in time. Next, the algorithm loops over all the rows in the current final table, this can be seen as being denoted as K , where K are the number of candidate solutions or rows in the final result. After the merge has been done, the new results are sorted, as this is Python the sorting algorithm is Tim Sort where the worst case performance would be $O(K \log(K))$ [6].

The filtering part of the algorithm will be discussed in subsection 5.1.5, all filtering methods either run in constant time (prime filtration and dynamic case filtration) or are more efficient (min-max filtration) than $O(K \log(K))$. Only the dynamic case filtration's weld method takes $O(R)$, where R are the number of ranges.

Therefore we can conclude that the merging algorithm takes $O((L_p + L_q) \times (M + K \log(K) + R))$

5.1.5. Filtering the solutions

All filtering functions are focused on reducing the size of the array of the solutions, we shall refer to the number of (candidate) solutions as being K .

Min-max filtration

Both max and min filtration work the same in that both run an adapted version of binary search. Inside both filtration methods there is a check of what the maximum and minimum possible answers are, this is a rather simple operation and can be seen as taking constant time. As we know that binary search is logarithmic we can therefore conclude that the algorithmic complexity is $O(\log K)$.

Prime Filtration

For prime filtration we have an outer while loop which breaks if the budget exceeds the budget that has been set. As the budget is constant no matter how large the size of the problem gets, therefore we can say that this section runs in the worst case in constant time, $O(1)$.

Dynamic Case Filtration

For the dynamic filtration case there are two sections. The first section is like the prime filtration in that we have a while loop which breaks if the budget has been exceeded. Again the size of the budget does not change. The second part deals with the welding of the ranges together. The algorithm loops over all the ranges and checks if the distance between the two ranges is not greater or smaller than a constant, if it is smaller than a constant then the algorithm searches between the two ranges and checks if it contains one of the factors of N . If we were to denote the length of the array of ranges as R , then we can say that the worst case is equal to the best case, as the algorithm always needs to loop over all the ranges. Therefore, we can say that the algorithmic complexity is $O(R)$.

5.1.6. Summary of Theoretical Analysis

In summary we can see that all parts of the algorithm run at most in linearithmic time in regard to K , which is in the merging part. Therefore, we can say that the main bottleneck of this algorithm comes in the merging section where we are heavily dependent on the size of K (number of rows in the final result) being small or having a manageable size. All the algorithmic complexities are listed in table 5.2.

Section	Complexity
Generating Equations	$O(M)$
Solving cells	$O(M)$
Reduction	$O(L \times M)$
Merging	$O((L_p + L_q) \times (M + K \log(K) + R))$
Min-max filtration	$O(\log K)$
Prime filtration	$O(1)$
Dynamic case filtration	$O(R)$

Table 5.2: Algorithmic complexity of the all sections of the algorithm

5.2. Practical Evaluation

The practical evaluation was done by running the algorithm on RSA numbers that have an even bit length and where it is not possible to factor these numbers by using Fermat's factoring method. The choice to choose even bit length numbers was because all RSA keys that get used for encryption have an even bit length. Secondly, All Fermat factorable numbers can be factored in milliseconds anyway. Therefore, it is of no interest to try and factor these numbers.

The numbers ranged from having a bit length of 20 to 74, and per bit length 10 numbers were chosen at random, all numbers are in the appendix section A.1. We felt that 10 numbers was a good amount in terms of variance and the time it would take to factor all numbers. The reason for choosing 74 bits as the upper limit was that anything longer, the variance in the time it took to factor these numbers would increase ever greater. We have managed to factor numbers that have a larger bit length, however, certain numbers would take far too long to factor, whilst others would be factored in

a matter of seconds. All numbers were factored on a computer with a 6 core CPU with a speed of 2.1 GHz and 96 GB of memory. Both the solving and the reduction parts of the algorithm were made to run in parallel with 10 threads. The budget for filtering for prime and dynamic filtration was set at 1000000000 decimal numbers.

The practical evaluation section is split into the following parts. Firstly we evaluate the performance in time it takes to solve the problem, number of iterations and spatial complexity. We then look at how the method performs with and without the different filtering methods to see which filtration method has the most effect. This was done by testing all numbers that have a bit length of 20 to 26. After that we shall take a look at how much the reduction-deduction method is able to reduce on its own, and to see what effect it has on the merge process. Then finally we compare our results to the results of a general purpose factoring method.

5.2.1. Time Comparison

We can firstly look at the mean total time taken in figure 5.1. We can see a gradual increase in time it takes to factor numbers, till it gets to around the 60 bit mark, after that there is an even greater increase in the amount of time taken. Looking at the total time in terms of a stacked graph in figure 5.2 we can see that from 20 bits till just before the 50 bit mark there seems to be an approximate 50/50 split between solving the equations and merging the solutions to find the factors. After the 50 bit mark the merging takes a larger proportion of the time, and after the 60 bit mark solving takes a minuscule amount of the total time.

From these two graphs we can infer that before the 50 bit mark, the filtering methods are able to either keep the spatial complexity small or are able to find the final answer before the spatial complexity becomes incredibly large.

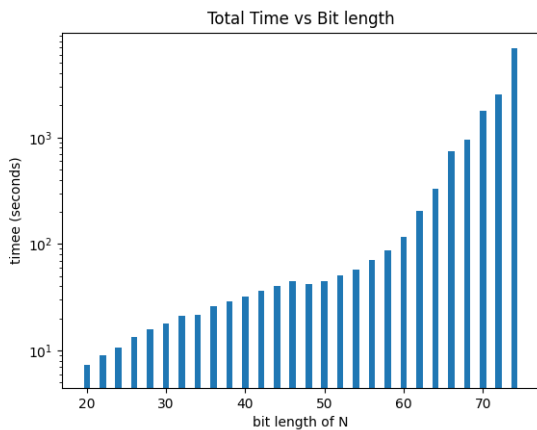


Figure 5.1: Total Factoring time

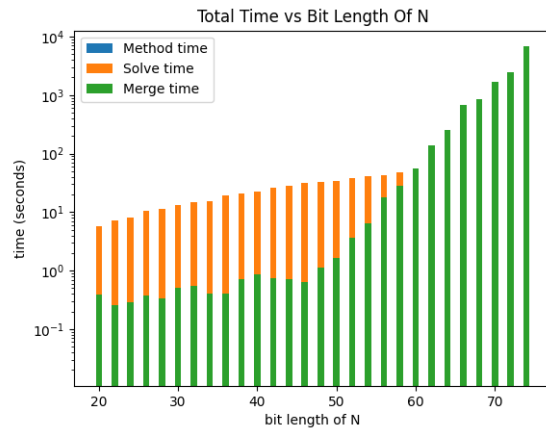


Figure 5.2: Total Factoring time broken down into different components

Furthermore, if we were to look at figure 5.3 we can see that for every bit length we have a box plot of the merge time. As we can see from the plots the average time often does increase, however, there is often a large disparity in terms of the time it takes to factor these numbers. Furthermore, we can see that many numbers can be factored faster than other numbers with a smaller bit length. Therefore, there is a large variance and if more numbers were to be tested this would become more apparent. What this means is that our factoring method is better at solving certain number than other numbers.

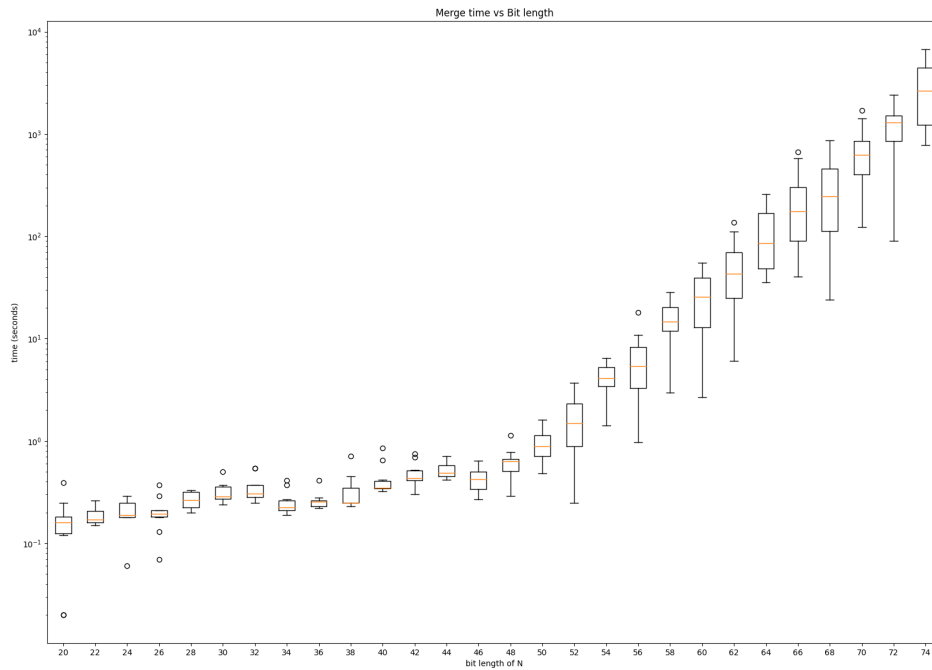


Figure 5.3: Box plot of time it takes to factor numbers

5.2.2. Iteration Comparison

Looking at the mean number of iterations during the merge phase in figure 5.2.4 we can see that the number of iterations stays relatively constant and is not dependent on the size of the number till the bit length of 70, when the number of iterations increases greatly.

Like in figure 5.2, we can infer that from the 20 to 70 bit length mark the filtration methods are able to find the answer within 15 iterations on average. Therefore, before all the solved cells have been merged. After the 70 bit length mark, the filtration methods are no longer able to find the answers within 15 iterations and the data complexity increases greatly, therefore also the time it takes to find the answer.

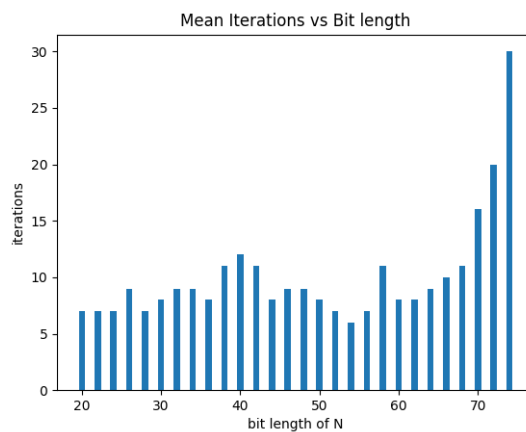


Figure 5.4: Mean total number of iterations it takes to factor numbers

Furthermore, when we look at the box plots for how many iterations it takes to factor the numbers in

figure 5.5, we can again see that the number of iterations stays constant till the bit length of 66 where after that it greatly increases. What we can see in this box plot as opposed to figure 5.3 is that early on there do seem to be more cases of outliers, however the median seems to be constant. This is probably because of the large outliers such as with bit length 40. This is probably the reason why the mean number of iterations varies between the 20 to 66 bit length in figure 5.2.4. What we can also see is that after bit length 66 we still get cases where it is not unusual for a factoring method to take less iterations than a number that has a bit length that is two bits shorter. Again, this suggests that our method is better at factoring certain numbers than others.

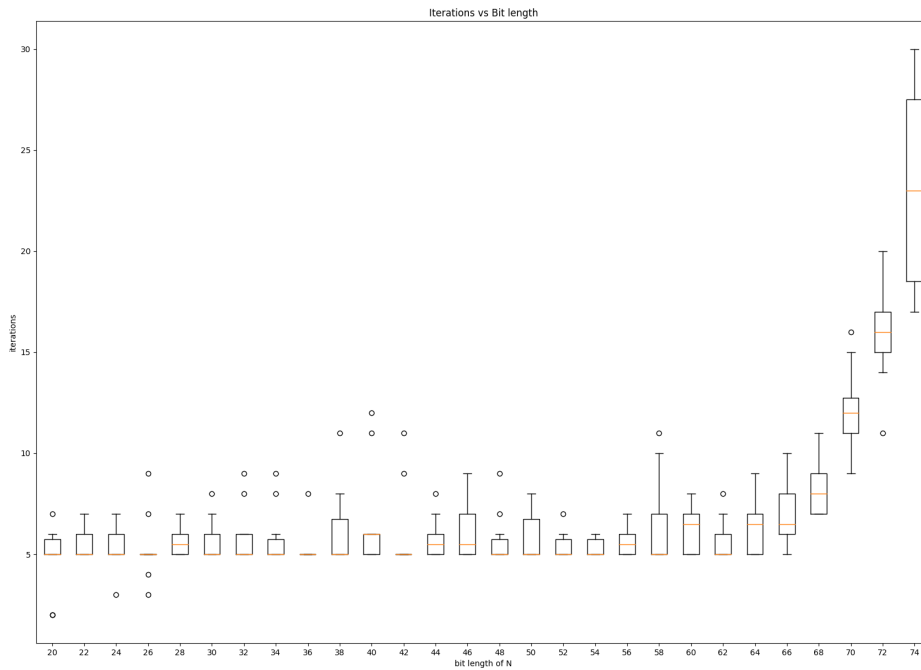


Figure 5.5: Box plot of number of iterations it takes to factor numbers

5.2.3. Space Complexity Comparison

Here we look at the space complexity, in which we mean the number of rows in the final results table at the end of merging. Here we have a similar situation like in the number of iterations. In figure 5.6 we can see a plateau from 20 to 66 bits, after which there is a great expansion in the number of rows.

This graph has a similar trajectory to the figure 5.2.4, this is what we expect as every iteration the data complexity doubles without filtration. In the 60 - 70 bit mark area, we can see that the filtration methods start to struggle to find the answer early, then struggle to keep the data size small.

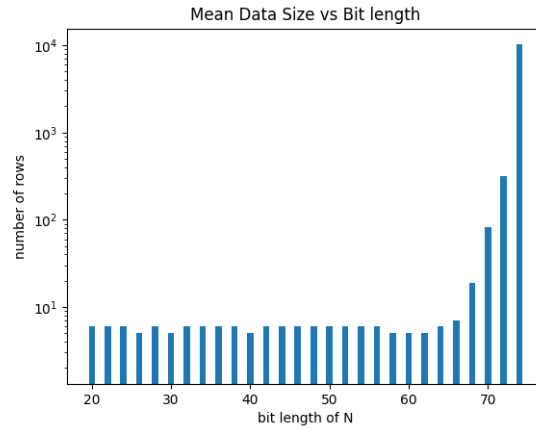


Figure 5.6: Mean data size at the end of factoring N

Furthermore, we can see in figure 5.7 that again the pattern is much the same as in figure 5.5. What is different however, is that there are less outliers than in the previous two box plots of time 5.3 and number of iterations 5.5. Again this suggests what we have mentioned for the time comparison and the iteration comparison, where we believe that our factoring method is better at factoring certain numbers than others.

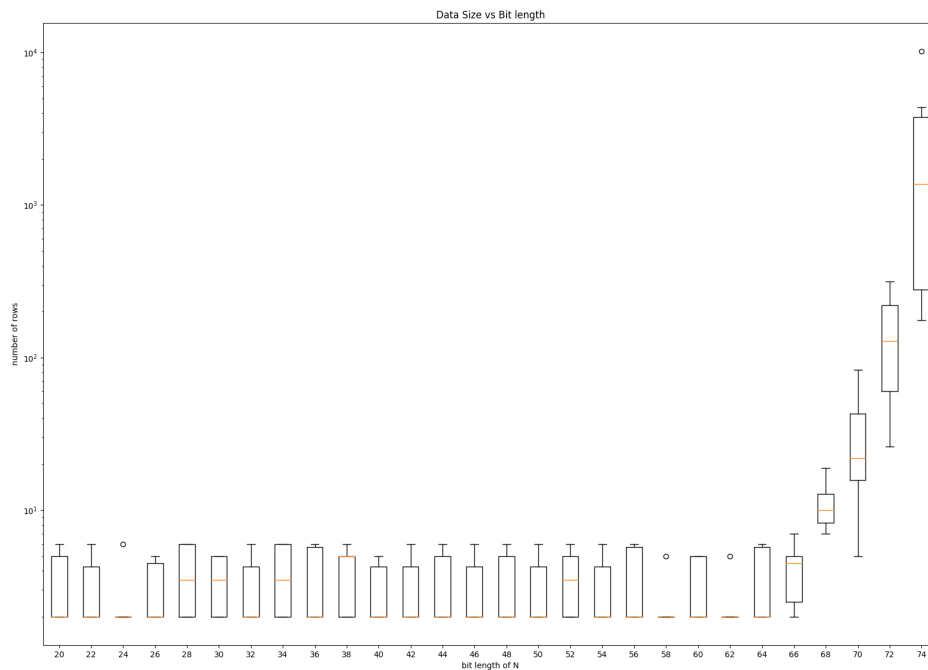


Figure 5.7: Box plot of data size at the end of factoring N

5.2.4. Comparing Different Filter methods

We also ran the algorithm with only one or none of the filtering methods. This was to see which of the filtration methods were the most effective; to see if all of them work; and to see what the result would be if we were not to apply any filtration methods at all. Because of time constraints, we decided to only run the algorithm on numbers that are between 20 and 26 in bit length.

As can be seen in figures 5.8, 5.9 and 5.10, the dynamic filtration is the fastest and has the least amount of iterations and the least amount of rows. This is probably because the budget was so large in comparison to the size of the number that the algorithm was factoring, which meant that it could rather quickly solve the problem.

Furthermore, we can see that in figures 5.8 and 5.10 that after the dynamic filtration method, the min-max filtration method is second. Then we have the prime filtration method in third, this makes sense as the prime filtration method is meant to only start filtering towards the end of the process.

Finally, in figure 5.9 there is less of a clear cut difference between the prime and min-max filtration methods. It is even the case for the bit length of 24 that the min-max method took more iterations on average than having no filter, this is most likely a fluke because of one of the factors being found when merging.

In conclusion, we can see that the dynamic filtration method is able to quickly solve the smaller numbers, and is most likely the reason why in figures 5.5 and 5.6 there is a plateau till the 70 bit mark. This is probably because the dynamic filtration method is able to find the answer quickly and therefore avoid greater increase in data. In graph 5.2 we can see that that the increase in merge time starts earlier, around the 48 bit length mark. This suggests that earlier on the dynamic filtration method takes a long time to run.

Therefore, dynamic filtration even though it is constant in time theoretically, does take a long time when the distance that it can maximally scan has been reached. As figures 5.5 and 5.6 show, it is effective to a certain point, after which it does not scale well any more.

Furthermore, from our observations we would suggest to either adapt dynamic filtration so that it is more efficient, one possible avenue would be to look at sieving methods [52]. Otherwise, we would suggest looking for new filtration methods or improving the other two filtration methods.

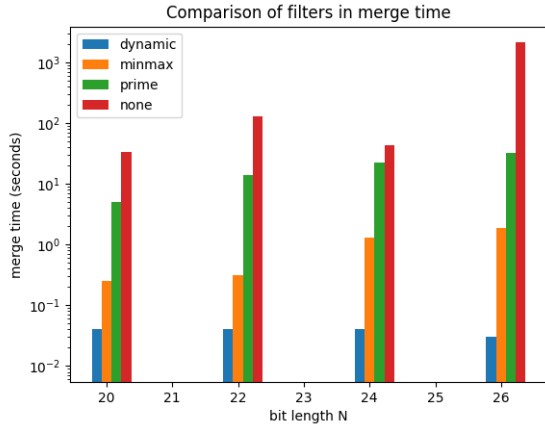


Figure 5.8: Time in seconds till solution is found, when running filtering methods on their own

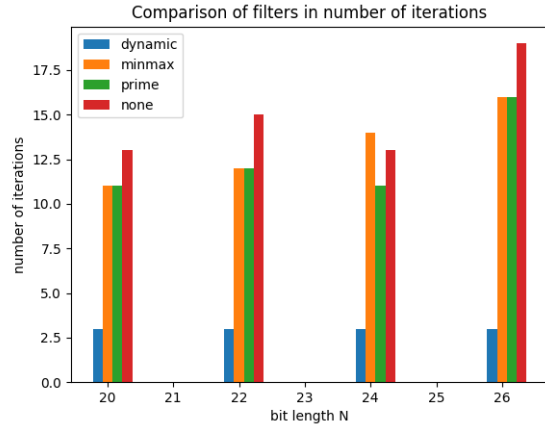


Figure 5.9: Number of iterations till solution is found, when running filtering methods on their own

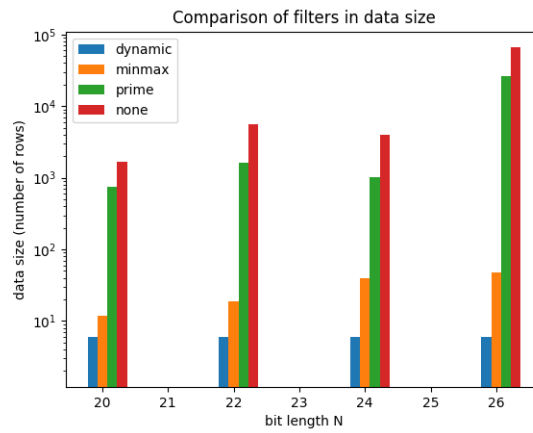


Figure 5.10: Data size at then end of the algorithm, when running filtering methods on their own

5.2.5. Reduction method

Firstly, we have looked at how much the reduction methods reduce the entire problem by. We ran two versions of the reduction method, normal and ruthless, on all numbers between 20 - 40 bits in length. Ruthless being the version where we substitute the values if two values are equal or opposite, normal being the case where that does not happen.

In figure 5.11 we can see the mean number of rows before and after the running of the reduction methods for every number ranging from bit length 20 to 40. We can clearly see that there is some reduction for all cases. However, we can also see that the reduction methods are not able to reduce the size of the problem to a size that is radically smaller than the original problem size.

In figure 5.12 we can see a box plot for both reduction methods. We can see that the normal reduction method is able to reduce the size of the problem at around 4% on average and the ruthless reduction method is able to reduce the size on average by about 7%. There are, however, cases where the methods have been able to reduce the problem in a far greater amount, with the ruthless reduction method being able to reduce the problem size by 12%. Unfortunately, there have been cases where the normal reduction was less effective, where we have a reduction of just around 3%.

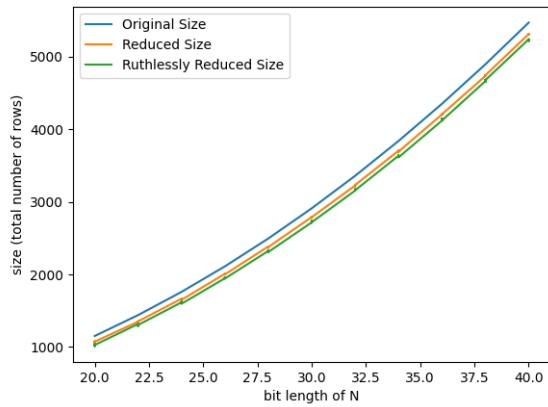


Figure 5.11: Line plot of mean number of rows for every number between 20 - 40 bits in length.

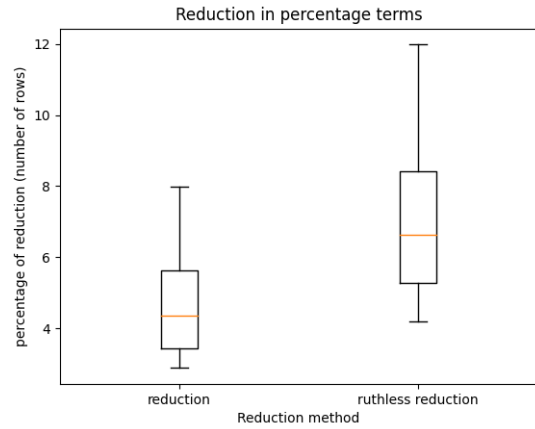


Figure 5.12: Box plot of both reduction methods, looking at the percentage of the reduction it has managed to make.

In figures 5.13, 5.14 and 5.15 we can see the actual difference in performance using the different reduction methods and not using the reduction method. Firstly, in figure 5.13 we can see the mean time (in seconds) it takes to factor numbers. In all cases there does not seem to be any constant trend where any of the reduction methods outperforms in time. Furthermore, it does seem that for many cases that the non-reduction method outperforms the reduction methods. Moreover, in the case for number of iterations in figure 5.14 it is usually the case that the no reduction method takes less or just as many iterations as the reduction methods. Finally, this trend continues in the last graph 5.15 for the size of data.

Therefore, we can conclude that on average the reduction methods in their current form have no advantage in being able to factor numbers faster than not using a reduction method.

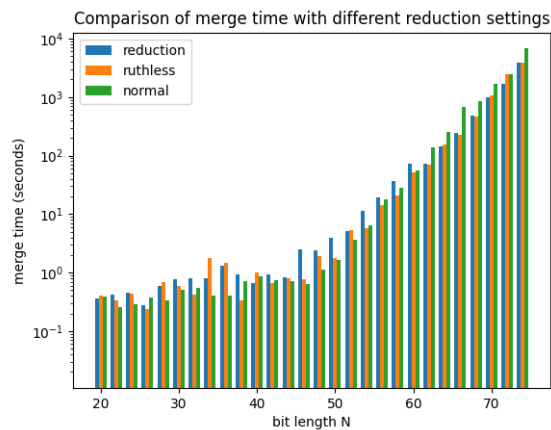


Figure 5.13: Time in seconds till solution is found, when running different reduction methods

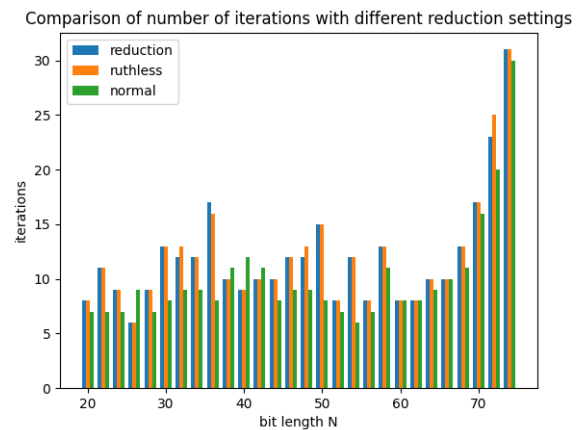


Figure 5.14: Number of iterations till solution is found, when running different reduction methods

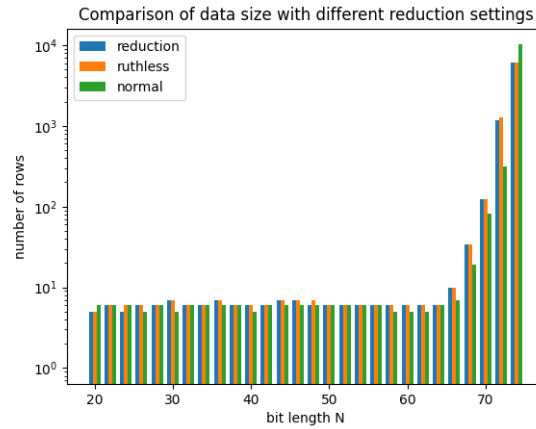


Figure 5.15: Data size at then end of the algorithm, when running different reduction methods

Moving on to look at the box plots of the amount of time it takes to factor numbers in figures 5.16, 5.17 and 5.18, we see a slightly different picture. Again the average seems to be the same for all methods. However, there is a significantly greater amount of variance. Most notably for the numbers with the bit length 62 and 58. This does suggest that it does have a greater advantage for certain numbers, but on average it does not.

Therefore, we can conclude that the reduction methods do not show a special advantage for some numbers.

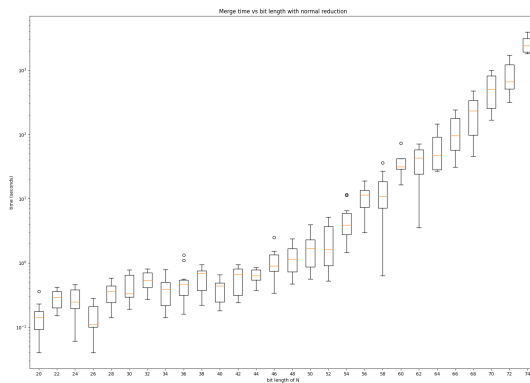


Figure 5.16: Box plot of time it takes to factor numbers using normal reduction method

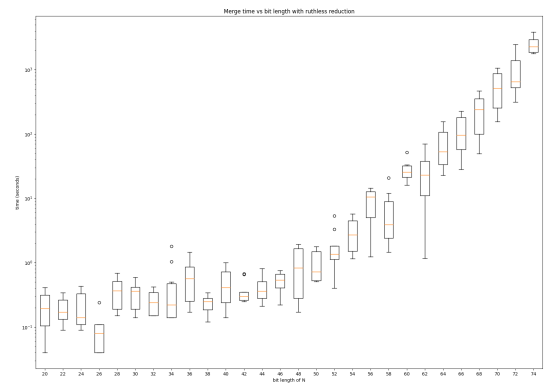


Figure 5.17: Box plot of time it takes to factor numbers using ruthless reduction method

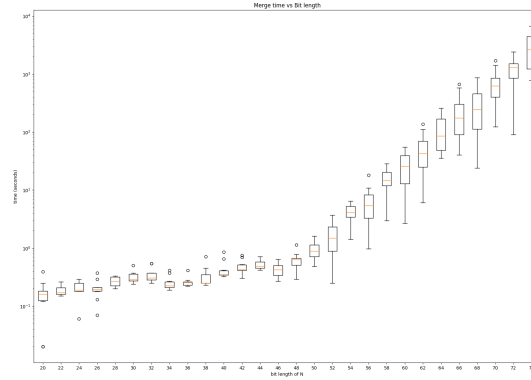


Figure 5.18: Box plot of time it takes to factor numbers

5.2.6. Comparison with other factoring methods

To give a quick comparison on how long the current state of the art factoring methods take to factor numbers we can look at figure 5.19 where we ran Sage math's implementation of Paul Zimmermann's GMP-ECM algorithm [56]. The time it takes to factor the RSA numbers is about a factor of ten faster than what is displayed, this is because running sage through the command line is a lot slower than opening sage and running the program.

Still the point is clear, in that we can easily see that there is little to no difference in time it takes to run GMP-ECM and takes significantly less time than our factoring method.

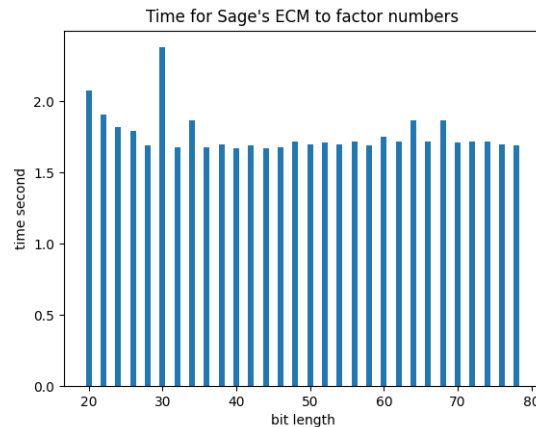


Figure 5.19: GMP-ECM factoring all the same numbers

5.2.7. Summary

In summary we can see that our factoring method is not fast, nor seems to be able to factor any number faster than the current best case factoring methods.

However, in comparison to factoring methods that use quantum computers, our method greatly outperforms all of them. Moreover, our method can with ease factor numbers that are around the 40 bit mark, the largest number that has been factored with the use of a quantum computer [31].

As seen in the time comparison section 5.2.1 and as said in the theoretical evaluation, most of the time it takes to factor the large numbers is based on the fact that the merging process takes a long time due to the method taking $O(K \log K)$ and the size of the final results doubling every iteration. We can however say that the filtration methods are able to reduce the size of the data by a great deal looking at the graphs 5.8 to 5.10.

However the main reason why the increase in time to solve the problem is relatively flat from 20 till 50 bits as seen in figure 5.2 is probably because of the dynamic filtration method, where the budget

is so large that through brute force it manages to find at least one of the factors relatively quickly. We assume this by looking at the graphs 5.8 to 5.10 where in all cases the dynamic filtration method is the fastest.

6

Conclusion

RSA is a very important public encryption standard that is used frequently. It is therefore of great importance to constantly study and check the security of the encryption standard. Research either in finding new general purpose factoring methods or through looking for new special purpose factoring methods like Fermat's factoring method.

In this final chapter we shall answer the research question that we have set ourselves in this thesis and go over the limitations of this research and future possible research.

6.1. Discussion

To reiterate the initial research question:

How can we build a scalable distributive factoring algorithm that is based on a discrete optimisation formulation?

Our method of distributive factoring that we showed in chapter 4 is indeed able to factor numbers that have a bit length of 20 to 74 as shown in chapter 5. Therefore, we can argue that given an unlimited amount of memory and an unlimited amount of threads RSA can be broken in a short amount of time. Unfortunately in real life, we do not have an unlimited amount of memory or threads, and as our memory approximately doubles every iteration during the merge part of the algorithm this is by far not a good general purpose factoring method.

Therefore, if we were to seriously look at being able to break RSA numbers that get used for encryption that are of 1024, 2048 or 4096 bits in length then the general answer would be that at its current state it would not be possible. This is due to the large amount of data that gets created as the factoring process goes on. Keeping many options in data and not having been able to reduce the size of the data, has been the major drawback of our implementation.

Furthermore, if we were to compare our method to current state of the art factoring methods we can see that our method underperforms by quite a significant margin.

Even though theoretically this method should be able to factor certain numbers faster than others, we have not been able to find any empirical evidence that this method of factorisation has any special purpose advantage to the current state of the art factoring methods. Therefore, we can currently say that its special purpose advantage is inconclusive.

However, what can be shown is that this method of factoring is at the current state of quantum computing a far better method than Shor's factoring method and any discrete optimisation method where the entire problem is placed onto the quantum computer. In the future as quantum computers get better at solving larger and more complex problems, so too can we increase the size of the cells and factor even larger RSA numbers. Thereby reducing the spatial complexity of the problem, and being able to break larger RSA keys.

Furthermore, we are to the best of our knowledge the first people to have described the semiprime factoring problem as a search problem. We believe that this will give a new insight into how to possibly break RSA in general or in special cases. Either by devising search algorithms for this problem, or by

inventing new filtration methods. So that larger RSA keys will become vulnerable to special factoring methods.

Moreover, we have no doubt that there are more methods of filtration that can be applied to this problem. Therefore, possibly one day be able to reduce the spatial complexity from being exponential to subexponential or possibly even polynomial.

Finally, we hope this method will be the first in a new generation of factoring methods and through future methods we shall either be able to factor certain semiprime numbers faster or will be able to factor semiprime numbers faster in general.

6.2. Limitations

The first limitation of this research is that we have not tested more numbers, and have only found numbers where this method is rather bad at in comparison with general purpose factoring methods. As mentioned before we do know that this algorithm may be able to factor certain RSA numbers better than the current best factoring methods around, and that this method may be able to factor certain numbers in seconds, minutes or hours when the current best factoring methods may only factor number in days, weeks or years.

Furthermore, we have not gone into trying to factor RSA keys that are of encryption standard length i.e. 1024, 2048 or 4096 bits in length. Giving the factorisation method a time limit of x minutes and then seeing if it was able to factor the RSA number.

Thirdly, another limitation was that the code was written in Python. Even though Python is a helpful language in terms of ease of use and number of libraries, it is not the best in terms speed. Therefore, our algorithm could be significantly faster and may even factor larger numbers if it were programmed in C++, however it does not change the general pitfalls of the algorithm.

Finally, the budget for the dynamic case filtration was critical for determining what size numbers the algorithm could factor. If it had been larger or smaller, then our results would have been slightly different.

6.3. Future Work

As has been alluded to, there are many new avenues for future research either in improving the current factoring algorithm or creating a new factoring algorithm by itself. Here we shall quickly go over sections that we believe to be of interest and/or could improve this algorithm even more.

Different Data Structure Looking for new data structures for the data so that the data ordering can be done in a more efficient way, merging the data can be done more efficiently, and/or filtering can be done more efficiently.

Merge Process Furthermore, we believe that the current method of merging with the same data structure can be done in $O(K)$ as opposed to $K \log(K)$ by simply making sure that the final results table is constantly ordered by initially ordering the table that will be merged into the final result table. Moreover, we believe that the process of merging can have a smaller data complexity due to a better "Merge strategy", where we possibly select the tables that get merged better by being more selective on either size or intersection of variables.

Filtration Methods Other than the three filtration methods already shown, we believe that there must be more ways of reducing the data complexity of this problem. One such method may be done by using the other two table methods. Where after having all candidates for the variables p_2 and q_2 till p_x and q_x , the solutions get fed into a column or a block to see if it were possible to be a valid solution.

Reduction process As mentioned before it has been possible to solve certain formulations of the problem by hand. It therefore may be of use to go deeper into looking at how one could optimise the reduction process in order to reduce the problem to a far smaller size, or even solve the entire problem outright.

Testing more numbers The current tests have been done on a select amount of numbers. It may be the case that we were unlucky and that there are numbers where this method is better at factoring RSA numbers than current state of the art factoring methods. Where this factoring method has the same property as Fermat's, Euler's and Pollard ρ 's factoring methods in that it is a "special purpose" factoring method.

Larger cells In this factoring method we have chosen the smallest possible chunks in the cell method. However, it is easy to merge all cells with one, two or more cells together and then go about

solving the amalgamated cells and then start the process of reducing, merging and filtering. This may mean that we get a smaller data complexity and therefore can factor larger numbers.

6.4. Concluding Remarks

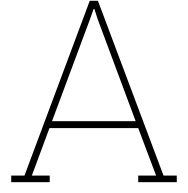
In conclusion, this thesis has shown that for semiprime numbers between 20-74 bits in length, it is possible to factor them using our distributive method. This has mainly been thanks to our filtration methods which have been able to reduce the data complexity by quite a margin. Therefore, far outperforming current state of the art quantum factoring methods. However, in comparison to the current state of the art classical factoring methods, it is slow. During our practical evaluation, we have not come across any discernible advantage in factoring certain special numbers over general purpose factoring methods. However, we have reason to believe that there are numbers that can be factored faster using our factoring method. Finally, we have shown that one can view the factoring problem as a search problem, this may mean that in future research new special purpose factoring methods will be invented that exploit weaknesses in RSA.

References

- [1] URL: <https://ccadb-public.secure.force.com/mozilla/IncludedCACertificateReport>.
- [2] Hedayat Alghassi, Raouf Dridi, and Sridhar Tayur. "Graver bases via quantum annealing with application to non-linear integer programs". In: *arXiv preprint arXiv:1902.04215* (2019).
- [3] Evgeny Andriyash et al. "Boosting integer factoring performance via quantum annealing offsets". In: *D-Wave Technical Report Series* 14 (2016).
- [4] Eric R. Anschuetz et al. *Variational Quantum Factoring*. 2018. arXiv: 1808.08927 [quant-ph].
- [5] Maliheh Aramon et al. "Physics-inspired optimization for quadratic unconstrained problems using a digital annealer". In: *Frontiers in Physics* 7 (2019), p. 48.
- [6] Nicolas Auger et al. "On the Worst-Case Complexity of TimSort". In: *26th Annual European Symposium on Algorithms (ESA 2018)*. Ed. by Yossi Azar, Hannah Bast, and Grzegorz Herman. Vol. 112. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 4:1–4:13. ISBN: 978-3-95977-081-1. DOI: 10.4230/LIPIcs.ESA.2018.4. URL: <http://drops.dagstuhl.de/opus/volltexte/2018/9467>.
- [7] Connelly Barnes. "Integer factorization algorithms". In: *Department of Physics Oregon State University* (2004).
- [8] David E Bernal, Sridhar Tayur, and Davide Venturelli. "Quantum Integer Programming (QIP) 47-779: Lecture Notes". In: *arXiv preprint arXiv:2012.11382* (2020).
- [9] Dimitris Bertsimas, Georgia Perakis, and Sridhar Tayur. "A new algebraic geometry algorithm for integer programming". In: *Management Science* 46.7 (2000), pp. 999–1008.
- [10] Endre Boros and Peter L Hammer. "Pseudo-boolean optimization". In: *Discrete applied mathematics* 123.1-3 (2002), pp. 155–225.
- [11] Matthew Edward Briggs. "An introduction to the general number field sieve". PhD thesis. Virginia Tech, 1998.
- [12] Bruno Buchberger. "A theoretical basis for the reduction of polynomials to canonical forms". In: *ACM SIGSAM Bulletin* 10.3 (1976), pp. 19–29.
- [13] Christopher J. C. Burges. "Factoring as Optimization". In: 2002.
- [14] Cristian S Calude, Michael J Dinneen, and Richard Hua. "QUBO formulations for the graph isomorphism problem and related problems". In: *Theoretical Computer Science* 701 (2017), pp. 54–69.
- [15] KEITH CONRAD. "THE INFINITUDE OF THE PRIMES". In: ().
- [16] Pasqualina Conti and Carlo Traverso. "Buchberger algorithm and integer programming". In: *International Symposium on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*. Springer. 1991, pp. 130–139.
- [17] Harald Cramer. "On the order of magnitude of the difference between consecutive prime numbers". In: (1936).
- [18] Nike Dattani. "Quadratization in discrete optimization and quantum mechanics". In: *arXiv preprint arXiv:1901.04405* (2019).
- [19] Nikesh S. Dattani and Nathaniel Bryans. *Quantum factorization of 56153 with only 4 qubits*. 2014. arXiv: 1411.6758 [quant-ph].
- [20] Poussin C De La Vallée. "Recherches analytiques de la théorie des nombres premiers/C. De La Vallée Poussin". In: *Annales de la Societe Scientifique de Bruxelles*. Vol. 21. 1896, p. 351.
- [21] Raouf Dridi and Hedayat Alghassi. "Prime factorization using quantum annealing and computational algebraic geometry". In: *Scientific Reports* 7.1 (Feb. 2017). ISSN: 2045-2322. DOI: 10.1038/srep43048. URL: <http://dx.doi.org/10.1038/srep43048>.

- [22] Jean-Charles Faugere. “A new efficient algorithm for computing Gr obner bases without reduction to zero (F5)(15/6/2004)”. In: ().
- [23] Jean-Charles Faugere. “A new efficient algorithm for computing Grobner basis”. In: *(F4)* (2002).
- [24] Craig Gidney and Martin Ekerå. “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”. In: *Quantum* 5 (2021), p. 433.
- [25] Fred Glover, Gary Kochenberger, and Yu Du. “A tutorial on formulating and using QUBO models”. In: *arXiv preprint arXiv:1811.11538* (2018).
- [26] Jack E. Graver. “On the foundations of linear and integer linear programming I”. In: *Mathematical Programming* 9 (1975), pp. 207–226.
- [27] Gian Giacomo Guerreschi. “Solving Quadratic Unconstrained Binary Optimization with divide-and-conquer and quantum algorithms”. In: *arXiv preprint arXiv:2101.07813* (2021).
- [28] *HPC software developments help to break cryptography records*. Oct. 2020. URL: <https://prace-ri.eu/hpc-software-developments-help-to-break-cryptography-records/>.
- [29] Hiroshi Ishikawa. “Higher-order clique reduction without auxiliary variables”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 1362–1369.
- [30] Shuxian Jiang et al. *Quantum Annealing for Prime Factorization*. 2018. arXiv: 1804.02733 [quant-ph].
- [31] Amir H Karamlou et al. “Analyzing the performance of variational quantum factoring on a superconducting quantum processor”. In: *npj Quantum Information* 7.1 (2021), pp. 1–6.
- [32] Andrew Lucas. “Ising formulations of many NP problems”. In: *Frontiers in physics* (2014), p. 5.
- [33] Enrique Martin-Lopez et al. “Experimental realization of Shor’s quantum factoring algorithm using qubit recycling”. In: *Nature photonics* 6.11 (2012), pp. 773–776.
- [34] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: <https://doi.org/10.7717/peerj-cs.103>.
- [35] Naeimeh Mohseni, Peter L McMahon, and Tim Byrnes. “Ising machines as hardware solvers of combinatorial optimization problems”. In: *Nature Reviews Physics* 4.6 (2022), pp. 363–379.
- [36] Emile Okada, Richard Tanburn, and Nikesh S Dattani. “Reducing multi-qubit interactions in adiabatic quantum computation without adding auxiliary qubits. part 2: The” split-reduc” method and its application to quantum determination of ramsey numbers”. In: *arXiv preprint arXiv:1508.07190* (2015).
- [37] Shmuel Onn. “Nonlinear discrete optimization”. In: *Zurich Lectures in Advanced Mathematics, European Mathematical Society* (2010), p. 75.
- [38] Soham Pal et al. “Hybrid scheme for factorization: Factoring 551 using a 3-qubit NMR quantum adiabatic processor”. In: *Pramana* 92 (Nov. 2016). DOI: 10.1007/s12043-018-1684-0.
- [39] WangChun Peng et al. “Factoring larger integers with fewer qubits via quantum annealing with optimized parameters”. In: *SCIENCE CHINA Physics, Mechanics & Astronomy* 62.6 (2019), p. 60311.
- [40] Loic Pettier and Projet SAFIR. “The Euclidean Algorithm in Dimension n”. In: (1996).
- [41] John Preskill. “Quantum computing in the NISQ era and beyond”. In: *Quantum* 2 (2018), p. 79.
- [42] Lauren Pusey-Nazzaro et al. “Adiabatic quantum optimization fails to solve the knapsack problem”. In: *arXiv preprint arXiv:2008.07456* (2020).
- [43] Ronald L Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.
- [44] Gernot Schaller and Ralf Schützhold. *The role of symmetries in adiabatic quantum algorithms*. 2009. arXiv: 0708.1882 [quant-ph].
- [45] Peter W Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.
- [46] Richard Tanburn and Nike Dattani. *Quantum-Factorization-By-Minimization*. <https://github.com/HPQC-LABS/Quantum-Factorization-By-Minimization>. 2018.

- [47] Richard Tanburn, Oliver Lunt, and Nikesh S. Dattani. *Crushing runtimes in adiabatic quantum computation with Energy Landscape Manipulation (ELM): Application to Quantum Factoring*. 2015. arXiv: 1510.07420 [quant-ph].
- [48] Richard Tanburn, Emile Okada, and Nike Dattani. *Reducing multi-qubit interactions in adiabatic quantum computation without adding auxiliary qubits. Part 1: The "deduc-reduc" method and its application to quantum factorization of numbers*. 2015. arXiv: 1508.04816 [quant-ph].
- [49] BaoNan WANG et al. "Quantum annealing distributed integer decomposition study of local field coefficient h and coupling coefficient J with stability Ising model". In: *SCIENTIA SINICA Physica, Mechanica & Astronomica* 50.3 (2020), p. 030301.
- [50] Baonan Wang et al. "Prime factorization algorithm based on parameter optimization of Ising model". In: *Scientific reports* 10.1 (2020), pp. 1–10.
- [51] Richard H. Warren. "Factoring on a Quantum Annealing Computer". In: *Quantum Info. Comput.* 19.3–4 (Mar. 2019), pp. 252–261. ISSN: 1533-7146.
- [52] Eric W Weisstein. "Sieve of eratosthenes". In: <https://mathworld.wolfram.com/> (2004).
- [53] Eric W Weisstein. "Twin Prime Conjecture". In: <https://mathworld.wolfram.com/> (2003).
- [54] Marek Wolf. "Nearest-neighbor-spacing distribution of prime numbers and quantum chaos". In: *Physical Review E* 89.2 (2014), p. 022922.
- [55] Nanyang Xu et al. "Quantum factorization of 143 on a dipolar-coupling nuclear magnetic resonance system." In: *Physical review letters* 108 13 (2012), p. 130501.
- [56] Paul Zimmermann and Inria Lorraine. "GMP-ECM: yet another implementation of the Elliptic Curve Method (or how to find a 40-digit prime factor within $2 \cdot 10^{11}$ modular multiplications)". In: *Workshop Computational Number Theory of FoCM*. Vol. 99. 1999.



Cell Method example

Equations of cell method

$$\begin{aligned}
& -2 * S_{10} + S_{11} + z_{11} + 1 \\
& -2 * S_{11} + S_{12} - S_{21} + q2 + z_{12} \\
& -2 * S_{12} - S_{22} + q3 + z_{13} \\
& -2 * S_{13} - S_{23} + q4 \\
& S_{21} + p2 - 2 * z_{11} + z_{21} \\
& S_{22} - S_{31} + p2 * q2 - 2 * z_{12} + z_{22} \\
& S_{23} - S_{32} + p2 * q3 - 2 * z_{13} + z_{23} \\
& - S_{33} + p2 * q4 \\
& S_{31} + p3 - 2 * z_{21} + z_{31} \\
& S_{32} - S_{41} + p3 * q2 - 2 * z_{22} + z_{32} \\
& S_{33} - S_{42} + p3 * q3 - 2 * z_{23} + z_{33} \\
& - S_{43} + p3 * q4 \\
& S_{41} + p4 - 2 * z_{31} - 1 \\
& S_{42} + p4 * q2 - 2 * z_{32} - 1 \\
& S_{43} + p4 * q3 - 2 * z_{33} - 1 \\
& p4 * q4 - 1
\end{aligned}$$

(A.1)

	S1_0	S1_1	z1_1		
0	1	0	1		
1	1	1	0		
	S1_1	S1_2	S2_1	q2	z1_2
0	0	0	0	0	0
1	0	0	1	0	1
2	0	0	1	1	0
3	0	1	1	0	0
4	1	0	0	1	1
5	1	1	0	0	1
6	1	1	0	1	0
7	1	1	1	1	1
	S1_2	S2_2	q3	z1_3	
0	0	0	0	0	
1	0	1	0	1	

```

2    0    1  1    0
3    1    0  1    1
    S1_3 S2_3 q4
0    0    0  0
1    0    1  1
    S2_1 p2 z1_1 z2_1
0    0  0    0    0
1    0  1    1    1
2    1  0    1    1
3    1  1    1    0
    S2_2 S3_1 p2 q2 z1_2 z2_2
0    0    0  0  0    0    0
1    0    0  0  1    0    0
2    0    0  1  0    0    0
3    0    0  1  1    1    1
4    0    1  0  0    0    1
5    0    1  0  1    0    1
6    0    1  1  0    0    1
7    0    1  1  1    0    0
8    1    0  0  0    1    1
9    1    0  0  1    1    1
10   1    0  1  0    1    1
11   1    0  1  1    1    0
12   1    1  0  0    0    0
13   1    1  0  1    0    0
14   1    1  1  0    0    0
15   1    1  1  1    1    1
    S2_3 S3_2 p2 q3 z1_3 z2_3
0    0    0  0  0    0    0
1    0    0  0  1    0    0
2    0    0  1  0    0    0
3    0    0  1  1    1    1
4    0    1  0  0    0    1
5    0    1  0  1    0    1
6    0    1  1  0    0    1
7    0    1  1  1    0    0
8    1    0  0  0    1    1
9    1    0  0  1    1    1
10   1    0  1  0    1    1
11   1    0  1  1    1    0
12   1    1  0  0    0    0
13   1    1  0  1    0    0
14   1    1  1  0    0    0
15   1    1  1  1    1    1
    S3_3 p2 q4
0    0  0  0
1    0  0  1
2    0  1  0
3    1  1  1
    S3_1 p3 z2_1 z3_1
0    0  0    0    0
1    0  1    1    1
2    1  0    1    1
3    1  1    1    0
    S3_2 S4_1 p3 q2 z2_2 z3_2
0    0    0  0  0    0    0

```

1	0	0	0	1	0	0
2	0	0	1	0	0	0
3	0	0	1	1	1	1
4	0	1	0	0	0	1
5	0	1	0	1	0	1
6	0	1	1	0	0	1
7	0	1	1	1	0	0
8	1	0	0	0	1	1
9	1	0	0	1	1	1
10	1	0	1	0	1	1
11	1	0	1	1	1	0
12	1	1	0	0	0	0
13	1	1	0	1	0	0
14	1	1	1	0	0	0
15	1	1	1	1	1	1
S3_3 S4_2 p3 q3 z2_3 z3_3						
0	0	0	0	0	0	0
1	0	0	0	1	0	0
2	0	0	1	0	0	0
3	0	0	1	1	1	1
4	0	1	0	0	0	1
5	0	1	0	1	0	1
6	0	1	1	0	0	1
7	0	1	1	1	0	0
8	1	0	0	0	1	1
9	1	0	0	1	1	1
10	1	0	1	0	1	1
11	1	0	1	1	1	0
12	1	1	0	0	0	0
13	1	1	0	1	0	0
14	1	1	1	0	0	0
15	1	1	1	1	1	1
S4_3 p3 q4						
0	0	0	0			
1	0	0	1			
2	0	1	0			
3	1	1	1			
S4_1 p4 z3_1						
0	0	1	0			
1	1	0	0			
S4_2 p4 q2 z3_2						
0	0	1	1	0		
1	1	0	0	0		
2	1	0	1	0		
3	1	1	0	0		
S4_3 p4 q3 z3_3						
0	0	1	1	0		
1	1	0	0	0		
2	1	0	1	0		
3	1	1	0	0		
p4 q4						
0	1	1				

A.1. RSA Numbers

bit_length,numbers
20,557983

20,664007
20,581717
20,462169
20,476941
20,697343
20,581543
20,848059
20,401963
20,711743
22,2448997
22,2378251
22,1696223
22,3258881
22,1999649
22,3062797
22,1822031
22,2106319
22,1811363
22,1872931
24,7624259
24,10119817
24,13496789
24,10309699
24,6445081
24,7703119
24,11976053
24,9024497
24,10288573
24,5735351
26,19786243
26,29167067
26,56296057
26,42197003
26,45366583
26,30208097
26,39545369
26,56274857
26,40713859
26,42527131
28,183845707
28,197932171
28,209120059
28,161384609
28,127811399
28,142582079
28,133060841
28,94993919
28,147069287
28,112076893
30,472737289
30,714633329
30,557986531
30,645569279
30,346537141
30,676016269
30,646426969

30,632062349
30,886484219
30,404154983
32,2046012737
32,3624499621
32,2761703267
32,2820455081
32,2936154413
32,2256606907
32,3893280947
32,3090736723
32,2409656549
32,1758806171
34,8103838487
34,7566818569
34,8114075797
34,10669323073
34,8334557719
34,8252377381
34,12098580251
34,15027476591
34,15812084437
34,9774346931
36,22935986897
36,33138126169
36,43596357289
36,38818588889
36,31400652883
36,33267928999
36,30804444701
36,37612883447
36,37667931761
36,29776613161
38,186948333437
38,237495888499
38,229141977451
38,151822648543
38,186640014377
38,202691243303
38,175114789769
38,76430528617
38,137986631899
38,179077394861
40,551508189013
40,298860185297
40,771629809109
40,715468313881
40,315334398499
40,431019749357
40,414951735091
40,583234853563
40,427008420931
40,522133240393
42,1226580636979
42,2367542034643
42,1939839808117

42,2316229587187
42,2848809364591
42,1992898495073
42,1891143689779
42,3888618617827
42,2278712606761
42,2165764242497
44,8792448322831
44,15219790380013
44,7768394269963
44,7126400629583
44,13658795239757
44,6792642517853
44,12907799423879
44,10593407186801
44,11050312374931
44,9006576830627
46,42244912567811
46,41285196651737
46,30277882397593
46,21647072785279
46,50840412795677
46,38156909161181
46,25164789431413
46,24572504317967
46,58150374927233
46,31793894186269
48,174673050905537
48,146265634948883
48,165040432864487
48,148101049493203
48,122774979277387
48,207785681727167
48,144908062562149
48,156372067084609
48,254506424694059
48,219181227407819
50,872493669463387
50,633922813882093
50,908941571134687
50,555628279894759
50,642301706246609
50,640460318983567
50,358229698426723
50,473934601272149
50,500880535968317
50,772536090874063
52,2718240285350959
52,3312307150041941
52,2688798276517819
52,2749728016615253
52,3300782118971881
52,2894175630522877
52,2086420843364089
52,3096063660217403
52,1589011240266371

52,2128462890212257
54,8882666652028931
54,11895416836537337
54,8239077264200033
54,12908383741026941
54,13301046661238879
54,10618836390838567
54,8468400874234501
54,11925802235472557
54,7739161933321561
54,8755223251805759
56,29395116158465989
56,50676092065612369
56,33315925358675299
56,54361101420776221
56,35817363758798939
56,37995224909047661
56,47239227354034481
56,53351322147726481
56,29631127495198921
56,37370753145683383
58,128239684950614927
58,100378011985019077
58,128945482392452311
58,194532391087886549
58,274389974354667181
58,147486360644456461
58,160723965489364391
58,183823201626555287
58,102002639838743663
58,202827251709564569
60,570378200408178797
60,675183015311043157
60,948002903702860217
60,869864916837857477
60,431495932511632693
60,830479821042818669
60,618770053549125293
60,378655441515821009
60,673468261036911493
60,796945072691982979
62,3583593961228849031
62,1905880712921579539
62,3024154658306697287
62,2058471775775019757
62,2653569704892700093
62,2203434866479432973
62,3302236964237838637
62,2028417098861853041
62,3939797000953741633
62,2506030462262773067
64,6425397346950162271
64,12314567633253381589
64,12549149351745564823
64,8697179519513662127
64,13858417306686544997

64,6753032504969945863
64,15605639495765202469
64,10482221135631438379
64,12876867855879868459
64,8556511821565620239
66,51436584698296859471
66,38427035456331156073
66,34013846111423370197
66,48644049464009359961
66,65219442926926643143
66,42431377523350214051
66,36912777185920640351
66,26096524129949665097
66,27703383391433070617
66,31156885293239972117
68,148937268486128559341
68,157473369860717924879
68,182641765842044726339
68,192693572852573383099
68,159926168784716697821
68,201541623254686575827
68,168091419965165689703
68,203485916420599442057
68,138762493463495273017
68,188854839286317016447
70,701643544509778315439
70,856903729322358000089
70,1033140804665629446851
70,877620497917156635001
70,651647762289959803993
70,717268152688953209797
70,731279262250147466963
70,378869868040391615597
70,688141646676714600793
70,431309271445154407159
72,3553931160054096287777
72,1512078965234330003089
72,2437623989525992263593
72,2370648160107943389799
72,2905623807380054430289
72,2218715140711173709961
72,2760068852748213927121
72,3430647283866057045829
72,1891365003006981347771
72,2673202871494076719789
74,12068078432590670895443
74,14571464221308550730767
74,8506650017733305442071
74,13610338837587772491749
74,8186966926224521798287
74,13515060684708668446693
74,11440425733401180230809
74,11251561255141568527487
74,16268299380072617681927
74,5330523977057877717131