

Shor's Algorithm in Superposition and the Comparison of Resources used by Two Different Quantum Algorithms for Factoring Large Numbers

Matthew Thorne

MSc in Computer Science

The University of Bath

September 2020

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:

A handwritten signature in black ink, appearing to read "M. Thorne". The signature is written in a cursive style with a long horizontal stroke at the top.

Shor's Algorithm in Superposition and the Comparison of Resources used by Two Different Quantum Algorithms for Factoring Large Numbers

submitted by

Matthew Thorne

for the degree of MSc in Computer Science of the

University of Bath

September 2020

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the author unless otherwise specified below, in accordance with the University of Bath's policy on intellectual property (see <http://www.bath.ac.uk/ordinances/22.pdf>). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of MSc in Computer Science in the Department of Computer Science. No portion of the work in this thesis has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signature of Author



Matthew Thorne

Abstract

Shor's algorithm is capable of breaking most modern encryption systems given a sufficiently large quantum computer. However, it is proving extremely difficult to build large, scalable quantum computers. This puts the size of quantum computer required for Shor's algorithm to break any common encryptions far out of reach of current technology. Shor's algorithm is a factoring algorithm and the best current implementation requires $2n + 3$ qubits to break an RSA encryption, where n is the bitsize of the encryption key. Resulting in a linear rate of growth compared to n .

In 2017 Bernstein, Biassie and Mosca presented a new algorithm that had a sublinear, $n^{2/3}$, rate of growth of qubits. This means asymptotically fewer qubits are required to factor a number than Shor's algorithm. However the size of n at which the BBM algorithm starts to use less qubits is unknown. Should the BBM algorithm use fewer qubits to factor common RSA key sizes it could potentially be implemented sooner on a physical quantum computer. A major component of the BBM algorithm is Shor's algorithm run in a superposition to factor a superposition of numbers.

This dissertation implemented Shor's algorithm in a superposition on Microsoft's quantum simulator allowing accurate resource measurements. Using these resources measurements a minimum bound on the number of qubits required by the BBM algorithm to factor a number of size n was calculated. It was found that the BBM algorithm required at least $13n^{2/3} + \log(1.44(n^{2/3} + 1))(11n^{2/3} + 22) + 11$ qubits to factor a number of size n . This gave a cross over point of $n = 0.42 \cdot 10^6$, between Shor's algorithm and the BBM algorithm, meaning for numbers greater than $0.42 \cdot 10^6$ the BBM algorithm would indeed use less qubits than Shor's algorithm to factor a number. However, for common RSA key sizes (for example, 2048-bit) the BBM algorithm would use more than double the number of qubits that Shor's algorithm would. Shor's algorithm is therefore more likely to be implemented sooner than the BBM algorithm to factor common RSA keys.

The implementation for Shor's algorithm in superposition can be accessed at:
<https://github.com/MattThorne/Quantum>

Acknowledgements

Thank you to Professor James H. Davenport and to Dr Benjamin Pring for guiding me through this project.

Contents

1	Problem Description	2
2	Literature Review	4
2.1	Computational Complexity	4
2.2	Cryptography	5
2.3	Classical Factoring Algorithms	9
2.3.1	Fermat and Kraitchik	9
2.3.2	Dixon’s Random Squares Algorithm	10
2.3.3	The Quadratic Sieve	10
2.3.4	The Number Field Sieve	12
2.4	Quantum Computation	14
2.4.1	Fundamentals	14
2.4.2	Quantum Circuits	16
2.4.3	Reversible Computation	18
2.4.4	Physical Realisation of Quantum Computers	23
2.5	Quantum Algorithms	25
2.5.1	Useful Maths	25
2.5.2	Grover’s Quantum Search Algorithm	28
2.5.3	Shor’s Quantum Fourier Transform	30
2.6	Quantum factoring Algorithms	33
2.6.1	Shor’s Algorithm	33
2.6.2	Bernstein, Biasse, and Mosca Low-Resource Quantum Fac- toring Algorithm	35
2.6.3	Shor’s Algorithm in Superposition	37
2.6.4	Comparing Quantum Algorithms to Classical Algorithms	40
2.7	Quantum Computer Simulators	41
2.8	Summary	43
3	Implementation	45
3.1	Implementing Fundamental Operations	45
3.1.1	Modular Square	45
3.1.2	Modular Multiply	47

3.1.3	Signed Multiply	47
3.1.4	Signed Subtract	48
3.2	Implementing Components In Superposition	50
3.2.1	Selecting x in Superposition	50
3.2.2	Modular Exponentiation in Superposition	51
3.2.3	Continued Fraction in Superposition	54
3.2.4	GCD in Superposition	56
3.3	Implementing Shor's algorithm In Superposition	56
4	Testing	58
4.1	Modular Square	60
4.2	Modular Multiply	60
4.3	Signed Multiply	61
4.4	Signed Subtract	61
4.5	Selection of x	62
4.6	Square and Multiply Exponentiation	62
4.7	Continued Fractions	62
4.8	Greatest Common Divisor	63
4.9	Shor's algorithm in superposition	63
5	Analysis	65
5.1	Modular Square	65
5.2	Modular Multiply	66
5.3	Signed Multiply	66
5.4	Signed Subtract	67
5.5	Selection of x	67
5.6	Square and Multiply Exponentiation	68
5.7	Continued Fractions	69
5.8	Greatest Common Divisor	70
5.9	Shor's algorithm in Superposition	71
6	Discussion	72
7	Conclusion	75
	Bibliography	76
	Appendices	81
I	12 Points of Ethics	81

Section 1

Problem Description

If physical realisation of scalable quantum computers is achieved, it is known that the most significant modern encryption systems can be broken using Shor's quantum factoring algorithm [Sho99]. In this dissertation, we concentrate on the RSA cryptosystem, which Shor's algorithm breaks by factoring a large number $N = pq$, the product of two distinct large primes. However, implementation of Shor's algorithm is currently far out of reach from current quantum computers, since engineering large quantum computers has proven an extremely challenging task. Quantum computers use a fundamentally different method of conducting calculations, based on the quantum bit or qubit. The limiting factor for Shor's algorithm is that the number of qubits required to break an encryption grows linearly with number of bits in the input N . Since common encryption systems can use 2048 bit numbers, the number of qubits required to implement Shor's algorithm is beyond what will be possible in the near future. However, new quantum factoring algorithms such as the one presented by Bernstein, Biassie and Mosca (BBM) [BBM17] have been developed which have a slower, sub-linear, rate of growth of qubits, potentially allowing them to be physically implemented sooner.

The fact that the BBM algorithm uses fewer qubits than Shor's algorithm for sufficiently large inputs does not give away if fewer qubits are used for common encryption key sizes. Although the asymptotic complexities are known for both Shor's algorithm and the BBM algorithm, this notation can hide substantial constant factors which are only revealed once the algorithms have been implemented. These constants determine the exact resources required by the algorithms. To establish if fewer qubits are used to break common encryption keys, the algorithms thus need to be implemented.

Method	Qubits	Time
Classical	0	$O(L(N)^{1.9+o(1)})$
Shor's	$O(\log_2(N)^{1+o(1)})$	$O(\text{poly}(\log_2(N)))$
BBM	$O(\log_2(N)^{2/3+o(1)})$	$O(L(N)^{1.38+o(1)})$

Table 1.1: Comparison of qubit and time complexities for different factoring algorithms. $L = \exp(((\log_2(N))^{1/3}(\log_2 \log_2 N)^{2/3}))$

A crucial part of the BBM algorithm uses Shor's algorithm in a superposition as a sub-routine. This project implements Shor's algorithm in superposition, on Microsoft's quantum simulator. A lower bound for the resources used by the BBM algorithm was then calculated. This lower bound was then used to compare the resources used by the BBM algorithm with an implementation of Shor's conventional algorithm. Comparison of the asymptotic qubit and time complexities for the different factoring methods are shown in table 1.1.

Section 2

Literature Review

This chapter covers the necessary background and current literature to understand the justification and motivation for this project. To start, cryptography is crossed to understand the importance of factoring algorithms. The best current classical factoring algorithms are then inspected before moving onto the quantum world and eventually quantum factoring algorithms. The prospects of physical realisation of quantum computers is also discussed.

2.1 Computational Complexity

In computer science, the efficiency of an algorithm is very important. The efficiency is given by the amount of resources required to run the algorithm; known as the complexity of the algorithm. Particularly important are the time and memory requirements. Since the time in seconds an algorithm takes to run is highly dependent on the computer, the time complexity is instead generally measured in the number of elementary operations a computer must complete; these are known as steps. The number of steps required varies with the input, and typically the larger the input the greater the number of steps and thus the greater the time complexity. However, it is impossible to know the number of steps required for all possible inputs, therefore complexity is typically expressed as a function of the size of the input n in bits. The number of steps for different inputs of the same size can also vary, meaning different functions can be used to show different complexities. The worst-case complexity is the maximum complexity for all inputs of size n .

Time and space complexity are often represented using big O notation. This is a convenient method for describing how the time and space it takes to run an algorithm grows as the input size grows. For example an algorithm of linear complexity is notated $O(n)$. Big O notation can be useful for comparing the efficiency of algorithms, allowing you to clearly see which algorithm will be more efficient with a sufficiently large input. However, it is important to remember that

crucial constants and factors are hidden behind the big O notation, for example algorithm a that requires $10 \cdot n^3$ operations and algorithm b that requires $100000 \cdot n^2$ operations, will have a big O of, $O(n^3)$ and $O(n^2)$ respectively. Algorithm a will thus appear to lose out to b according to the big O notation, but, when actually implemented at small values of n algorithm a will do less operations due to the smaller constant. Occasionally little o notation is also used, these are terms which tend to 0 as n gets large.

The space complexity of an algorithm is split into the space required to store the input and the auxiliary space required to carry out the computation. An important point to note is that although the space and time complexities are known for the BBM algorithm and Shor's algorithm, the exact number of qubits required is not known. Instead just the rate at which the number of qubits required grows is known. To discover the actual number of qubits required in practice, it is necessary to implement the algorithms. This is the major motivation for the project, to get a lower bound for the resources used by the BBM algorithm to enable a comparison between BBM and Shor's algorithm's when they are each implemented.

2.2 Cryptography

For many millennia cryptography has been used to ensure communications between two parties are kept confidential. The simplest form utilises a secret key agreed upon by two people, traditionally called Alice and Bob. Should Alice want to send a secure message to Bob, she must first transform her original message, called plain-text, using the secret key. The key scrambles the message to a message that cannot be understood, called the cipher-text. This is the process of encryption. When Bob then receives the cipher-text, he can transform it back to the original plain-text using the key. This is the process of decryption. Continuing the naming traditions, if a third person Oscar were to intercept the message between Alice and Bob, he would only receive the cipher text. He would thus find this message unintelligible; unless of course he also knew the secret key. The main limitation of secret-key cryptography is before any messages can be sent, both Alice and Bob must agree on a key, without Oscar also knowing. This proves challenging when using modern communications. One solution is to instead use a public-key cryptosystem.

The revolutionary public-key cryptosystem was first proposed by Diffie and Hellman [DH76]. They proposed that two keys could be used. A public key, which

was known to everyone, including Oscar, and a private key known only to one person. The public key would be used to encrypt a message, which could then only be decrypted using the private key. So if Alice wanted to send a message to Bob, she could encrypt a message using Bob's public key. This message could only be decrypted by Bob, using his private key. This has the massive advantage that no prior communication between Alice and Bob needs to have taken place. This is often compared to Alice placing a message in a box and locking it with a combination lock left by Bob. Since only Bob knows the combination, only he can read the message.

The key idea behind public-key cryptosystems is that it is infeasible to determine the decryption (private) key, from the encryption (public) key. Diffie's and Hellman's procedure has the following four properties:

- (a) Deciphering an enciphered message yields the original message. Formally,

$$D(E(M)) = M \tag{2.1}$$

- (b) Enciphering a deciphered message yields the original message. Formally,

$$E(D(M)) = M \tag{2.2}$$

- (c) There is no easy method to determine D if E is known

- (d) Both E and D are easy to compute.

Where D , E , and M are the decryption function, encryption function and the message respectively. This allows messages to be sent between two parties without prior communication. Mathematically, different public-key cryptosystems are based on different computational problems, which are all infeasible one way, but relatively easy the other way. These are often called trapdoor one-way functions. They are not truly one way functions since computed inverses exist [Dav99]. But given an algorithm it is computationally infeasible to calculate the reverse unless a certain trapdoor is known. Currently there are no problems which can be proved to be one way, although there are many that are believed to be [SP19]. The most widely used implementation of a public-key cryptosystem is the RSA cryptosystem, created by Rivest, Shamir and Adleman (also creators of the Alice and Bob naming tradition)[RSA78]. The security of RSA is based upon the difficulty of factoring large integers. The below description of RSA system is based on the original paper [RSA78]. Their method is as follows:

1. First represent the message to be sent as an integer M between 0 and $n - 1$. This is the necessary numerical form ready for encryption.
2. Next encrypt message M by raising it to the e th power modulo n . That is the cipher text C is the remainder when M^e is divided by n , or $C = M^e \bmod n$.
3. Finally, the cipher-text C can then be decrypted by raising it to the d th power modulo n . That is $M = C^d \bmod n$.

The public key is therefore the pair of integers (e, n) and the private key is similarly (d, n) . For clarity this means the encryptions and decryption algorithms are as follows:

$$E(M) = M^e \bmod n$$

$$D(C) = C^d \bmod n$$

For an adversary to be able to recover M , they must first determine d . The next step is to know how to pick these keys to ensure a secure system. To start, n is determined as the product of two large random prime numbers, p and q .

$$n = p \cdot q$$

Since the problem of factorising n is enormously hard, the values of p and q are effectively also private. This is where the security of the system lies as will be shown below. d is then picked to be a large integer which is relatively prime to $\phi(n)$. $\phi(n)$ is the euler-phi function, which represents the total number of positive integers less than n which are relatively prime to n . For prime numbers $\phi(n) = p - 1$. For the case of n ,

$$\phi(n) = \phi(p) \cdot \phi(q)$$

$$\phi(n) = (p - 1) \cdot (q - 1)$$

The importance of using $\phi(n)$ is also shown a little later. Finally the integer e can be computed as the multiplicative inverse of d . That is,

$$e \cdot d \equiv 1 \bmod \phi(n)$$

where, as above $\phi(n) = (p - 1)(q - 1)$

An important identity to proving the RSA method is correct is Euler's generalisation of Fermat's little theorem.

$$M^{\phi(n)} \equiv 1 \pmod{n} \quad (2.3)$$

This identity can be utilised to show that equations 2.1 and 2.2 hold true. Formally that,

$$(M^e)^d \equiv M \pmod{n}$$

Where M is the message represented as a numerical form, and e , d , and n are the keys. Since we have by definition that:

$$e \cdot d = k \cdot \phi(n) + 1$$

For some integer k , it follows that:

$$(M^e)^d \equiv M^{k \cdot \phi(n) + 1} \pmod{n}$$

Using equation 2.3:

$$(M^e)^d \equiv 1^k M \pmod{n}$$

Therefore:

$$(M^e)^d \equiv M \pmod{n}$$

Operating on M with the encryption function followed by the decryption function returns M , and equivalently if the functions are applied the other way around M is also returned. The RSA method therefore satisfies equations 2.1 and 2.2.

Both the decryption and encryption algorithms are examples of modular exponentiation. Although the use of these functions may initially seem complex, efficient algorithms are known, for example the Chinese remainder theorem. Similarly efficient methods are also known for generating large primes.

The final property to ensure that RSA satisfies is that knowing E it is infeasible to determine D . More strictly, with knowledge of (e, n) it is infeasible to resolve (d, n) . One method to determine (d, n) would be to factor n into p and q , to then determine $\phi(n)$ and thus find d . In fact, it can be shown that all other approaches to breaking this system are at least as hard as factoring n [RSA78]. Fortunately factoring large numbers into large primes is currently an enormously hard task with current algorithms. The current fastest algorithm was presented by Coppersmith [Cop93] who made a slight improvement to the algorithm developed by Buhler, Lenstra and Pomerance [BLP93] which utilises a number field sieve

method discussed below.

The RSA cryptosystems security is therefore based on the fact that factoring large integers is currently computationally infeasible using a classical computer, with the current best algorithms. However, the highly interesting part is that an efficient algorithm is known for quantum computers. It is also worth noting that other popular public-key cryptosystems such as the ElGamal cryptosystem are based on a different computational problem, the discrete logarithm problem. However, Shor [Sho99] also showed an algorithm for solving this problem on a quantum computer, so there is no escape. Since most modern encryptions utilise a public-key cryptosystem based on either the discrete logarithm problem or the factorisation problem, should a sufficiently large quantum computer be realised, most modern communications would no longer be confidential.

2.3 Classical Factoring Algorithms

2.3.1 Fermat and Kraitchik

As mentioned above the most evident method of breaking RSA cryptography is to factor n into its prime components. A trivial method of trial division of all primes up to \sqrt{n} would guarantee finding the solution. However, should $n > 10^{12}$ this is an infeasible method [SP19].

An early factorisation method was suggested by Fermat. The trick is to find a pair of squares such that $x^2 - y^2 = n$. If this is found then both $(x - y)$ and $(x + y)$ are non-trivial factors of n . The basis of most modern factoring algorithms is Kraitchik's enhancement of Fermat's difference of squares technique [PE96]. He reasoned that it might suffice to find x and y such that $x^2 - y^2$ equalled a multiple of n . Formally $x^2 - y^2 \equiv 0 \pmod{n}$. Such congruences can have uninteresting solutions such as $x \equiv \pm y \pmod{n}$ and interesting solutions such $x \not\equiv \pm y \pmod{n}$. Should such a pair be found it means that $n \mid (x + y)(x - y)$, but neither $(x + y)$ or $(x - y)$ is divisible by n . It thus follows that $\gcd(x + y, n)$ and $\gcd(x - y, n)$ are non-trivial factors of n . Kraitchik found that if n is odd and divisible by two primes, at least half of the solutions are of the interesting variety. Using these interesting solutions, it is found that the $\gcd(x - y, n)$ is a non-trivial factor of n . Since this is known the problem of factoring is now changed to finding pairs of x and y .

2.3.2 Dixon's Random Squares Algorithm

The simplest method is Dixon's random squares algorithm. As the name suggests this effectively randomly checks numbers x to see if $(x^2 \bmod n)$ is also a square. This on its own is very rare and would result in very slow factoring. However due to Kraitchik's enhancement Dixon has another option. The product $Q(x)$ of several numbers $(x^2 \bmod n)$ can also be checked to see if $Q(x)$ is a square. This on the face of it may seem more complicated, but there is a handy trick of utilising a factor base. A factor base is a set B of small primes. For example, $B = \{1, 2, 3, 5, 7, 11\}$. The fundamental theorem of arithmetic states that every positive integer can be represented in exactly one way as the product of one or more primes. This fact can be used in conjunction with a factor base to check if $Q(x)$ is also a square. The algorithm is as follows, first obtain several integers x such that the prime factors of $(x^2 \bmod n)$ are all in the factor base. Next take the product of a subset of x 's such that each prime factor in the factor base is used an even number of times. There are a number of algorithms that help quickly find this linear dependency [SP19]. If each factor in B is used an even number of times the product must be a square. This will result in the desired congruence of $x^2 \equiv y^2 \bmod n$, which may lead to the factorisation of n if $x \not\equiv \pm y \bmod n$. Dixon selects the x 's at random to see if $(x^2 \bmod n)$ factors into the factor base. However he finds it useful to start with numbers around \sqrt{kn} where $k = 1, 2, 3, \dots$, since when these numbers are squared modulo n they are small. This means they are more likely to have prime factors in the chosen factor base. If a number factors into a factor base B it is known as B -smooth. More efficient methods of selecting numbers that are more likely to be B -smooth than just testing random numbers are known and are discussed below. Dixon's method can be thought of as two steps, firstly finding a set S of B -smooth numbers, secondly finding a subset of S such that the product is a square.

2.3.3 The Quadratic Sieve

A widely used algorithm to speed up the process of finding smooth numbers is the Quadratic sieve due to Pomerance [Pom94]. The idea is to search for congruences of the form $\prod (x^2) \equiv \prod (x^2 - n) \bmod n$ where x runs across consecutive integers. Since the left hand side is already a square, the problem is to find a product of values $(x^2 - n)$ which are also a square. The form of the polynomial $f(x) = x^2 - n$ is very important, since if:

$$f(x) = x^2 - n$$

then

$$\begin{aligned} f(x + kp) &= (x + kp)^2 - n \\ f(x + kp) &= x^2 + 2kpx + (kp)^2 - n \\ f(x + kp) &= f(x) + 2kpx + (kp)^2 \equiv f(x) \pmod{p} \end{aligned}$$

therefore

$$f(x + kp) \equiv f(x) \pmod{p}$$

Therefore solving $f(x) = x^2 - n \equiv 0 \pmod{p}$ for one x generates a whole sequence of numbers which are also divisible by p . If you set p as a prime number in the factor base B then this can be used to rapidly find other numbers that also divide by p . This is extremely similar to the sieve of Eratosthenes which finds prime numbers up to a certain limit. The quadratic sieve algorithm thus goes as follows:

1. First choose a factor base B of small primes.
2. Next collect a set of values to be sieved, say $X(x) = \{\sqrt{n}, \sqrt{n} + 1, \sqrt{n} + 2, \dots, \sqrt{n} + 99\}$. Resulting in $Y(x) = \{(\sqrt{n} - n), ((\sqrt{n} + 1) - n), ((\sqrt{n} + 2) - n), \dots, ((\sqrt{n} + 99) - n)\}$.
3. For each prime p in the factor base do the following:
 - (a) Solve the equation $Y(x) = (\sqrt{n} + x)^2 - n \equiv 0 \pmod{p}$.
 - (b) Use the sieve described above to find other values in $Y(x)$ which are divisible by p .
 - (c) Any values in $Y(x)$ that are divisible by p should be divided by p until no longer divisible by p .
4. After the set $Y(x)$ has been completely sieved by the factor base B any value in $Y(x)$ equal to 1 is a smooth number.
5. These smooth numbers can then be used as in Dixon's random squares algorithm to factor n .

The quadratic sieve vastly speeds up the process of finding smooth numbers, and results in a much faster factoring algorithm than a random search. However, still faster algorithms are known that share a lot in common with the quadratic sieve. The break-through comes by realising the quadratic polynomial used for the quadratic sieve does not necessarily have to be quadratic. Perhaps certain

higher degree polynomials could produce more smooth numbers than quadratics. Another more subtle improvement comes from utilising rings of numbers other than \mathbb{Z} and $\mathbb{Z}/n\mathbb{Z}$. The idea is that these other rings of numbers could have a notion of smoothness that produces more smooth numbers than in \mathbb{Z} [Bri98].

2.3.4 The Number Field Sieve

The special number field sieve was invented by John Pollard and a more general version was presented by Buhler, Lenstra and Pomerance which allowed factorisation of an arbitrary integer into prime factors [BLP93]. The current fastest classical method to factor arbitrary integers is Coppersmith's improvement to the general number field sieve [Cop93]. This has a time complexity of $\approx O(L(N)^{1.9+o(1)})$. Nevertheless, this is still an exponential time complexity, meaning it is infeasible for a classical computer to carry it out fast. However, (as will be seen below) the General number field sieve can be improved by using quantum computers. The general number field sieve is very similar to the special version with more subtleties to allow arbitrary integers to be factored. For simplicity, just the special number field sieve is discussed.

The Special Number Field Sieve

The special number field sieve (NFS) is an integer factorisation algorithm that is efficient for integers of the form $r^e \pm s$ where r and s are small. It is very similar in method to the quadratic sieve, however, a more efficient method for finding smooth numbers is used by utilising number fields.

The quadratic sieve searches for smooth numbers of size $\approx n^{1/2}$. The number field sieve searches for two numbers of size $\approx n^{1/d}$ where d is often ≥ 4 . It turns out the chance of finding two numbers of size $n^{1/4}$ being smooth is greater than the chance of finding one number of size $n^{1/2}$ being smooth. This is where the advantage of the number field sieve over the quadratic sieve comes from [Ste08].

The special number field sieve goes as follows. An irreducible polynomial f with integer coefficients along with an integer m such that $f(m) \equiv 0 \pmod{n}$ is chosen. If α is a root of f then we can form the ring $\mathbb{Z}[\alpha]$. This is a ring consisting of all sums of the form $\sum_{k=0}^{\infty} a_k \alpha^k$. Since $f(\alpha) = 0$ and $f(m) \equiv 0 \pmod{n}$ we have a natural map ϕ from $\mathbb{Z}[\alpha]$ to $\mathbb{Z}/(n\mathbb{Z})$ [PE96]. The idea of the number field sieve is to then search for a set S of pairs (a, b) where both the algebraic integer $(a + \alpha b)$ and the integer $(a + mb)$ are smooth. Because $\phi(a + \alpha b) = (a + mb \pmod{n})$ each pair (a, b) produces a congruence \pmod{n} . Sufficiently many of these congruences

allows a difference of squares method such that $x^2 \equiv y^2 \pmod n$ to be set up. This can then be used to help factor n as in the previous methods [LLMP93].

The question then is how to find a set S of pairs (a, b) where both the algebraic integers $(a + \alpha b)$ and the integers $(a + mb)$ are smooth. Similar to the quadratic sieve we start by deciding how smooth we want the integers to be by picking a factor base. We set up two parallel factor bases, one in $\mathbb{Z}[\alpha]$ to be used for finding smooth algebraic integers $(a + \alpha b)$, and one in \mathbb{Z} for finding smooth integers $(a + mb)$. The base in $\mathbb{Z}[\alpha]$ is a set of all prime ideals in $\mathbb{Z}[\alpha]$ whose norm is bound by a chosen value. The base in \mathbb{Z} consists of prime integers up to some bound, just as in the quadratic sieve. The pairs are then found using a sieving process just as in the quadratic sieve.

As in the previous factoring algorithms a subset of S can be taken such that the prime factors in the factor bases are used an even number of times, therefore resulting in a difference of squares, hopefully allowing n to be factored.

The NFS Trade Off

The number field sieve can be thought of in two main steps:

1. Search for a set S of pairs (a, b) , where both the algebraic integer $(a + \alpha b)$ and the integer $(a + mb)$ are smooth.
2. Use linear algebra to find a subset of S that results in a congruence of the form $x^2 \equiv y^2 \pmod n$.

There is a trade off between these two steps, since the larger the set S , the more pairs (a, b) you have, therefore the longer it will take to find a linear dependency where all the exponents are even. The size of S thus wants to be minimised to speed up step 2. However, to reduce the size of S whilst still ensuring a linear dependence exists, the factor bases used in step 1 must be made smaller. This reduces the chance of finding pairs (a, b) which produce smooth integers in those factor bases, thus increasing the time taken for step 1. The trade off between these two steps is something that must be optimised. As we will see below the BBM quantum factoring algorithm rapidly speeds up the process of finding pairs (a, b) , this trade off is likely something that will need to be considered when implementing the BBM quantum factoring algorithm.

Large-Prime Variants

There have been many attempts to improve the number field sieve. One particularly important variant is the Large-Prime variant. Traditionally only smooth integers are considered, that is integers whose prime factors are less than some bound B . The large-prime variant also allows integers with one prime factor bigger than the bound to be considered, that is integers which have one factor $R > B$ and all other factors are $< B$. If two such integers are found with the same R , multiplying them together yields an integer with a factor R^2 . Since the exponent of R is even this integer can be useful in the linear algebra step to step up a difference of squares. Allowing large-primes can therefore increase the number of smooth numbers found [BR96]. There are also larger variants allowing 2,3, and even 4 primes above the bound B [LM94] [DL95]. Experiments have shown that a large proportion of the smooth numbers found are from the large prime variants. When comparing the implementations of the quantum algorithms back to the NFS it will be important to consider variants like this.

2.4 Quantum Computation

2.4.1 Fundamentals

Quantum computing utilises the properties of quantum mechanics to perform calculations. Exploiting these counter intuitive rules could provide an essential efficiency advantage over classical computers. Classical computers use the fundamental concept of a bit, with two states, 0 and 1. Similarly quantum computers use a fundamental concept known as a qubit, which can exist in a state $|0\rangle$ or $|1\rangle$. However, qubits can also exist in a state other than $|0\rangle$ or $|1\rangle$. Linear combinations of states can be formed, these are known as superpositions.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where α and β are complex numbers.

To determine the state of a classical bit you can examine it, getting either 1 or 0. Remarkably, despite the superposition, qubits are also resolved to either 0 or 1, but in a probabilistic way. A 0 or 1 is measured with probability $|\alpha|^2$ and $|\beta|^2$ respectively, thus a measurement only gives restricted information about the quantum state. Inherently, the probabilities must sum to 1 meaning $|\alpha|^2 + |\beta|^2 = 1$. Therefore only indirect correspondence with the quantum state

is possible. Unfortunately upon measuring a qubit the quantum state collapses down to the measurement meaning many repeated measurements cannot be used to obtain the quantum state. For example if the qubit above was measured to be 0, any subsequent measurements on the same qubit would measure 0. Despite this indirect relationship with the qubit state using these states in superpositions are essential to unlocking the potential power of quantum computation.

Combining multiple bits increases the number of possible states. This is the same for multiple qubits. The possible states with 2 bits are:

$$00 \ 01 \ 10 \ 11$$

The quantum state of two qubits is:

$$|\psi\rangle = \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle$$

Similarly to single qubits the probability of measuring each state is $|\alpha_x|^2$ where $x = 00, 01, 10, 11$. Computations can be conducted with qubits by manipulating the probabilities of measuring each state. This is done using quantum gates, mirroring classical logic gates.

Quantum superpositions allow functions to be evaluated using many different values simultaneously. This is known as quantum parallelism, and can be thought of as an experiment behaving as if it proceeds down all paths simultaneously. Each of these paths has a probability amplitude determined by the physics of the experiment. The number of values that can be evaluated is equivalent to the number of possible states that can be measured. This is equal to 2^n where n is the number of qubits. This therefore grows exponentially with respect to the number of qubits. The exponential growth in the number of values that can be evaluated simultaneously demonstrates the potential power of quantum computers. However, due to the probabilistic nature of measuring qubits simply evaluating the function for many values is not enough as information about only one value would be extracted at random. Instead the circuit must be slightly modified to extract a global property that can be used to determine the evaluations. The key behind fast and reliable quantum computations is to set a quantum superposition that calculates all possible answers at once whilst being arranged in such a way for all the wrong answers to destructively interfere. By properly adjusting the phases of various operations, successful computations reinforce each other while others interfere randomly [Gro96].

Before continuing, it is worth mentioning the prospect that our current knowl-

edge on quantum mechanics is incorrect. This could fundamentally preclude the large-scale realisation of functioning quantum computers [K.18]. However, it could also result in quantum computers being more powerful than we currently think [Aar05]. Many new surprises and discoveries are possible in this previously unexplored area, and they could have considerable consequences in all areas of science. Whether this will enable or prohibit the development of quantum computers we will have to wait and see.

Determining whether an algorithm can be efficiently computed is an integral problem in the field of computer science. It seems it was Feynman [Fey82] who first suggested that utilising quantum mechanics may allow more powerful computations than classical computers. Deutsch was the first to explicitly ask whether a quantum computer can efficiently solve problems where no efficient solution is known for a classical computer. This was addressed by Deutsch and Jozsa [DJ92]. They showed that there were problems which could be solved efficiently on a quantum computer where an efficient solution for a classical computer was not known.

The question that remains is, are quantum computers truly more powerful than classical computers? Or are the algorithms just yet to be found on a classical computer? So much is still unknown in the answer to these questions that it makes the study of quantum computation a great and exciting challenge.

2.4.2 Quantum Circuits

Manipulation of the quantum state of a qubit allows computations to be carried out. In order to describe these manipulations it is useful to look at quantum circuits and the fundamental operations which allow these states to be changed. A quantum state can be simply represented as a matrix, for example the quantum state $\alpha|0\rangle + \beta|1\rangle$ of a single qubit can be written as:

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Where the top value corresponds to the amplitude for $|0\rangle$ and the bottom value the amplitude for $|1\rangle$. Operations are carried out using quantum gates, which can be also be described using matrices. The simplest operation is a single qubit gate. Two important examples are shown below.

$$\begin{array}{c}
 \text{---} \boxed{X} \text{---} \\
 \text{---} \boxed{H} \text{---}
 \end{array}
 \begin{array}{c}
 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\
 \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}
 \end{array}$$

These are the NOT gate and the Hadamard gate. Similarly to the classical NOT gate the quantum NOT gate similarly swaps the probability of measuring a $|0\rangle$ or a $|1\rangle$. The Hadamard gate can be described as a square root of NOT gate. It turns a state $|0\rangle$ into $\frac{(|0\rangle+|1\rangle)}{\sqrt{2}}$ and a state $|1\rangle$ into $\frac{(|0\rangle-|1\rangle)}{\sqrt{2}}$. Equally a state $\frac{(|0\rangle-|1\rangle)}{\sqrt{2}}$ goes to $|1\rangle$ and a state $\frac{(|0\rangle+|1\rangle)}{\sqrt{2}}$ goes to $|0\rangle$.

To demonstrate a single qubit gate the output from a quantum NOT gate is:

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

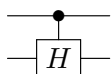
Just as in classical computers quantum gates can act across multiple qubits. The prototypical multi-qubit gate is the CNOT gate shown below. This gate takes two qubits, one as a control qubit and the second as a target qubit. If the control is set to 0 the target is left alone, but if the control qubit is set to 1 the target qubit is flipped.

$$\begin{array}{c}
 \text{---} \bullet \text{---} \\
 \text{---} \oplus \text{---}
 \end{array}
 \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

To demonstrate the potential effects of interference that can occur within quantum computation we can look at the controlled Hadamard gate shown below.

If we suppose our two qubit system is initially in a superposition of states

$$\frac{1}{\sqrt{2}} |10\rangle - \frac{1}{\sqrt{2}} |11\rangle,$$



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

and we apply the controlled Hadamard gate to the system, the resulting output would be:

$$\frac{1}{2}(|10\rangle - |11\rangle) - \frac{1}{2}(|10\rangle - |11\rangle) = |11\rangle.$$

The resulting amplitudes for the state $|10\rangle$ cancel out due to the superposition. Had the initial state not been a superposition there would have been a chance of observing the state $|10\rangle$. Quantum gates are therefore easily described using matrices. The only constraint is that the matrices give rise to unitary transformations, allowing physically feasible quantum gates.

Quantum computers can not be universal in the same way that classical computers can since there are an infinite number of transformations that can act on a single qubit. In contrast there are only four functions which can map one bit to another bit on a classical computer. This makes it impossible for a quantum computer to implement every program since it cannot implement every transformation. Instead a universal quantum computer is one that can approximate every transformation within a finite error using a finite sequence of gates. This is therefore weaker than a set of universal gates for a classical computer. For a quantum computer a common universal gate set is the Clifford + T gate set. The Clifford group is a set of gates which can be simulated efficiently on a classical computer. Whereas the T -gates make it possible to reach all different points.

2.4.3 Reversible Computation

The laws of physics appear to be completely reversible. This places a large constraint on any system built using these laws, mainly that the system must itself be reversible. That is knowing the state of a system, allows unique determination of the system in the past. In the quantum computing context that means knowing the quantum state after a quantum gate tells you exactly what the quantum state must have been before the quantum gate. Classical computers get around this challenge by dissipating some of the information of the system as heat energy, allowing irreversible computations. Since the superposition of quantum states needs

to be maintained to make use of quantum parallelism, quantum computers must use reversible computation. This is what puts the constraint on quantum gates to be unitary transformations. Reversible computation often imposes extra costs in comparison to irreversible computation. Bennett showed that any irreversible computation can be made reversible [Ben73]. The idea is to use extra memory to store the extra information needed to reverse the computation. For example the irreversible NAND gate takes two inputs a and b and has one output $\neg ab$, therefore if you obtain an output of 1, inputs a and b could have been any combination of 00, 01 and 10. However, you can make a reversible NAND gate by giving three inputs a, b and 1 and taking three outputs a, b and $1 \oplus ab = \neg ab$, therefore given any combination of outputs it is possible to precisely determine the input. The downside of course is that both a and b must now be kept in memory to keep the computation reversible. These reversible gates are known as Toffoli gates [Tof80]. Bennett uses three stages to turn any irreversible computation into a reversible computation.

1. First create a copy of the input and compute the calculation storing any intermediary steps, and any necessary extra information to allow each step to be reversed.
2. Secondly create a copy of the output.
3. Finally using the stored intermediary steps and extra information reverse the original computation.

This method is very useful to enable reversible computations, however for large multi-step calculations the memory requirements of storing every intermediary value and extra bits can soon become infeasible. For example if you were to try to compute $3 \cdot 122$ by adding 3 to itself 122 times it would be necessary to store all 120 intermediary values in order to make it reversible. Instead Bennett further suggested an improvement to limit the storage requirements, to the detriment of increased computational time. Bennett showed that any irreversible computation that uses space S and takes time T can be made reversible with time $T' = O(T)$ and space $S' = O(S \log(T))$ [Ben89]. In order to reduce the space required for the reversible computation Bennett suggested that a multi-step calculation be done in segments of length k steps. The process goes as follows, to start k steps are computed, and all necessary intermediary steps are stored. Next the k 'th step is stored and all previous $k - 1$ steps are reversed, leaving just the input and the k 'th step in memory. The process is then repeated starting from the k 'th

step. This can be iterated until the desired result is obtained, before reversing the whole process to clear the all memory. An example of the steps required to compute $64 \cdot a$ by simply doubling a for both a standard reversible process and reversible process requiring less memory using segments of length $k = 2$ are shown in table 2.1. As can be seen the low storage reversible computation takes more steps, but requires only three intermediary steps to be stored at any one time. In contrast the standard reversible process is quicker to finish but needs up to seven intermediary steps stored. Since using extra qubits can be very costly in current quantum computers, methods like these to reduce the number of extra qubits required are very useful.

Steps	Standard Reversible Process	Low Storage Reversible Process
0	a	a
1	$a \ 2a$	$a \ 2a$
2	$a \ 2a \ 4a$	$a \ 2a \ 4a$
3	$a \ 2a \ 4a \ 8a$	$a \ 4a$
4	$a \ 2a \ 4a \ 8a \ 16a$	$a \ 8a \ 4a$
5	$a \ 2a \ 4a \ 8a \ 16a \ 32a$	$a \ 8a \ 4a \ 16a$
6	$a \ 2a \ 4a \ 8a \ 16a \ 32a \ 64a$	$a \ 4a \ 16a$
7	$a \ 2a \ 4a \ 8a \ 16a \ 32a \ 64a \ 128a$	$a \ 2a \ 4a \ 16a$
8	$a \ 2a \ 4a \ 8a \ 16a \ 32a \ 64a \ 128a \ 256a$	$a \ 2a \ 16a$
9	$a \ 2a \ 4a \ 8a \ 16a \ 32a \ 64a \ 256a$	$a \ 16a$
10	$a \ 2a \ 4a \ 8a \ 16a \ 32a \ 256a$	$a \ 32a \ 16a$
11	$a \ 2a \ 4a \ 8a \ 16a \ 256a$	$a \ 32a \ 64a \ 16a$
12	$a \ 2a \ 4a \ 8a \ 256a$	$a \ 64a \ 16a$
13	$a \ 2a \ 4a \ 256a$	$a \ 128a \ 64a \ 16a$
14	$a \ 2a \ 256a$	$a \ 128a \ 64a \ 16a \ 256a$
15	$a \ 256a$	$a \ 64a \ 16a \ 256a$
16		$a \ 32a \ 64a \ 16a \ 256a$
17		$a \ 32a \ 16a \ 256a$
18		$a \ 16a \ 256a$
19		$a \ 2a \ 16a \ 256a$
20		$a \ 2a \ 4a \ 16a \ 256a$
21		$a \ 4a \ 16a \ 256a$
22		$a \ 8a \ 4a \ 16a \ 256a$
23		$a \ 8a \ 4a \ 256a$
24		$a \ 4a \ 256a$
25		$a \ 2a \ 4a \ 256a$
26		$a \ 2a \ 256a$
27		$a \ 256a$

Table 2.1: The steps of both a standard reversible process and a reversible process requiring less storage of calculating $64 \cdot a$.

The process is usefully thought of as a pebbling game where there is just one rule: *If step i is occupied by a pebble then you may place a pebble on step $i + 1$ if it is not occupied, or you may remove a pebble from step $i + 1$ if it is occupied.* Algorithm 1 shows two mutually recursive functions which implement Bennett's scheme of a low storage reversible process for the pebble game [LTV98]. Starting

from step s with n free pebbles, it is possible to reach step $s + 2^{n-1}$. The *Step()* function simply carries out a single step, or reverses a single step. To implement this game it is necessary to keep track of which pebbles are at which step and which are free, this can be achieved simply using an array.

Algorithm 1: Pebbling algorithm

```

1 Function Pebble( $s, n$ ):
2   if  $n == 0$  then
3     Return;
4   else
5     let  $t = s + 2^{n-1}$  ;
6     Pebble( $s, n-1$ );
7     Step(+1)/* Place free pebble on node  $t$                 */
8     Unpebble( $s, n-1$ );
9     Pebble( $t, n-1$ );
10  end
11 end
12 Function Unpebble( $s, n$ ):
13   if  $n == 0$  then
14     Return;
15   else
16     let  $t = s + 2^{n-1}$  ;
17     Unpebble( $t, n-1$ );
18     Pebble( $s, n-1$ );
19     Step(-1)/* Remove pebble from node  $t$                 */
20     Unpebble( $s, n-1$ );
21   end
22 end

```

An important point to note is that Bennett's scheme is not proven optimal, and is simply a method to convert non-reversible algorithms into reversible algorithms. In fact there is actually a slight flaw in Bennett's paper due to a hidden constant on the reversible space bound [LS90]. In fact the space bound is really given by $S' \approx \epsilon 2^{1/\epsilon} S \ln(T/S)$. Where $\epsilon = \frac{\ln(2-(1/k))}{\ln k}$. If ϵ is chosen to be tiny then the space required by Bennett's conversion will grow exponentially, Figure 2-1 shows the exponential explosion of space requirements as ϵ tends to zero for different ratio of S/T .

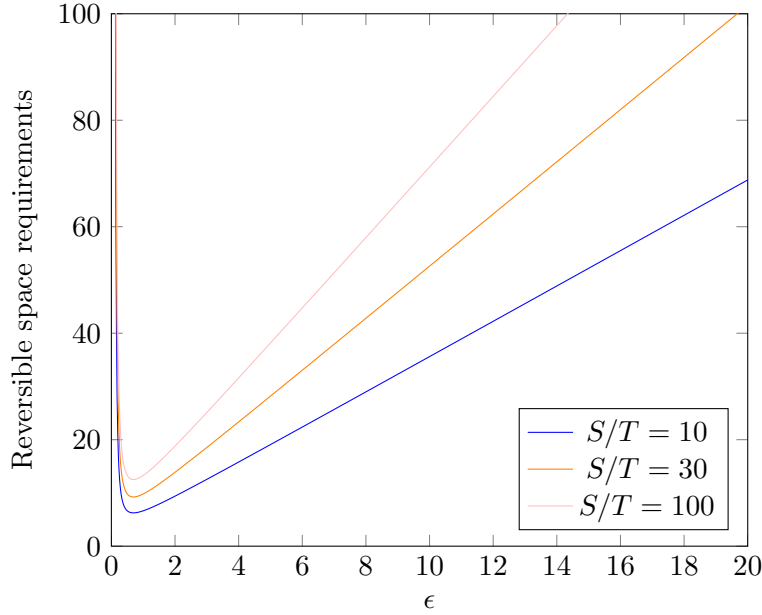


Figure 2-1: Plots show the explosion of space requirements when ϵ tends to zero.

2.4.4 Physical Realisation of Quantum Computers

Physical realisation of quantum computers has proven extremely hard [NC19]. Research into the feasibility of building large scale quantum computers began following Shor's discovery [CJL⁺16]. For a long time it was thought the fragile nature of quantum states would introduce too many errors to ever make a scalable technology. This fragile nature makes quantum states highly susceptible to noise. Noise is the result of unwanted interactions with an environment outside the system. It can cause the state of a qubit to be affected. For example, if an electron is used as the qubit it could unwantedly interact with charged particles, altering its quantum state.

A large body of research has gone into developing quantum error correcting codes to tackle this problem. In classical computers, noise can affect bits by causing them to flip with some probability p . Classical error correcting codes traditionally work by encoding redundancy into the message. For example instead of sending a single bit of say 0, three bits, 000, are sent. The probability of noise affecting all three bits, causing them all to flip is much lower than the probability of the single bit being flipped. In some ways this is analogous to spoken language. If initially you do not understand someone, you can ask them to repeat the message. Quantum error correcting codes are more challenging due

to the no-cloning theorem, the continuous nature of a quantum state, and the fact that measurement collapses the quantum state. However, sufficient progress has been made to show that noise likely poses no fundamental barrier to large-scale quantum computers [NC19]. One method used similar to classical error correcting codes is to use multiple physical qubits to encode one logical qubit. The number of qubits required by a quantum algorithm is therefore not only determined by the complexity of the algorithm, but also critically the quality of these qubits.

These challenges make the depths and widths of algorithms particularly important. For example, an algorithm with a larger depth requires that the qubits be kept longer in a particular state which could introduce errors as the quantum states breakdown. Equally an algorithm with a large width requires many qubits which is also hard to achieve. Often algorithms come with different depth/width trade offs which must be carefully balanced. The importance of reducing the width versus reducing the depth of a quantum algorithm is not yet clear as quantum computers of useful size are not yet available. We have to keep in mind that error correction will create an overhead in the number of qubits used. It makes sense to minimise the number of qubits before applying error correction if qubits are hard to come by [Bea03].

A central result is the threshold theorem which shows that a noisy quantum computer can accurately and efficiently simulate computations so long as the strength of the noise is below a certain critical ‘threshold’ [Ali07]. These developments in quantum error correcting codes have yielded high fault tolerance.

In 2019 Google announced they had built and used a 53 qubit quantum computer. Conducting ‘the first calculation that can be performed only on a quantum processor [ea19], they thus announced quantum supremacy. Despite these advancements, a large number of technical leaps are still required to build large, scalable, fault tolerant systems that can realise the full promise of quantum computing. A recent progress forecast in the domain of quantum computing predicted there was a $< 5\%$ probability that a generalised logical qubit will be achieved before 2026. This would roughly mark the beginning of scalable quantum computation. Additionally, there was a $< 5\%$ chance that RSA-2048 Shor attacks would be feasible with a quantum computer before 2039 [SR20]. There is therefore still a long way to go and we are very much in the early days of quantum computing.

Another point to note about physical quantum computers is their different physical architectures. A quantum circuit is not necessarily able to allow any two arbitrary qubits to interact, instead it may be the case that only qubits immediately next to each other can interact, if this is the case a series of swap

operations must be performed to get two arbitrary qubits next to each other. Different current quantum architectures have varying connectivity's [FDH04].

In order to compare different quantum computers IBM introduced the Quantum Volume metric [CBS⁺19]. In classical computers it is simple to compare the transistor count, however comparing the qubit count of quantum computers is not the same since qubits decohere, losing their quantum information. A quantum computer with a few low-error and highly connected qubits is likely capable of performing more advanced quantum algorithms than a computer with many error-prone unconnected qubits. The quantum volume is a single number meant to encapsulate the performance of a quantum computer. It quantifies the largest random circuit of equal width and depth that the computer successfully implements. In 2020 IBM achieved a Quantum Volume of 64, meaning an arbitrary circuit with up to a depth and width of 6 can be successfully implemented [Jea20].

2.5 Quantum Algorithms

2.5.1 Useful Maths

Before tackling quantum algorithms it is useful to cover some important mathematical algorithms that are very useful to the understanding of both Shor's algorithm and the BBM algorithm.

Square and Multiply Algorithm

Square and multiply algorithms allow for fast computation of large positive integer powers of a number. It is based on the observation that for an integer n ,

$$x^n = \begin{cases} x(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$$

This observation is very easy and convenient to convert to binary. For example to calculate y^{13} , first convert 13 to binary, getting 1101. Starting with the value y you then simply traverse the exponent starting with the most significant bit and if a 0 is encountered the value is just squared. If a 1 is encountered the value is squared and then multiplied by y . This allows large powers to be calculated in far fewer steps than simple repeated multiplying. The number of steps required is equal to the number of bits in n . For example, the calculation above can be done in 4 steps as shown in table 2.2. This method can just as easily be used for

modular exponentiation, as it is used in the BBM algorithm by using modular squaring and modular multiplication.

Step	Bit	Work	Description
1	1	y	Start process with y
2	1	$(y^2) \cdot y$	Since the bit is 1 the value is squared and multiplied.
3	0	$((y^2) \cdot y)^2$	Since the bit is 0 the value is just squared
4	1	$((y^2) \cdot y)^2 \cdot y$	Since the bit is 1 the value is squared and multiplied.

Table 2.2: Example Calculation of y^{13} using Square and multiply.

Continued Fractions

The continued fractions algorithm allows an irrational number θ to be approximated to a rational fraction, this is done through a series of split and invert steps. It is best demonstrated by example, lets try to represent $\theta = 31/13$ using a continued fraction. To start we split $31/13$ into its integer and fraction,

$$\frac{31}{13} = 2 + \frac{5}{13}.$$

Next the fraction is inverted

$$\frac{31}{13} = 2 + \frac{1}{\frac{13}{5}}.$$

The next split and invert iteration is then applied to $13/5$, resulting in

$$\frac{31}{13} = 2 + \frac{1}{2 + \frac{3}{5}}.$$

These iterations continue until the decomposition terminates with a 1, the full decomposition of $31/13$ is

$$\frac{31}{13} = 2 + \frac{1}{2 + \frac{1}{1 + \frac{1}{1 + \frac{1}{2}}}}.$$

The worst case for continued fractions is that the quotient is always 1, this results from the Fibonacci numbers, which can therefore be used to determine the maximum number of steps the continued fraction algorithm will takes. It turns out if θ is an L bit number it will take $1.44L$ steps for the algorithm to terminate. This is the rate of growth for the Fibonacci numbers [Dav99].

Of importance to approximating an irrational number as a rational fraction are

the convergences to a continued fraction. A convergence is obtained by stopping a continued fraction before it terminates, for example stopping at $\frac{31}{13} = 2 + \frac{1}{2}$ in the example above. This gives an approximation

$$\frac{31}{13} \approx 2 + \frac{1}{2} = \frac{4}{2} + \frac{1}{2} = \frac{5}{2}$$

It is interesting to note that each successive convergent gives a better approximation to the real number, and that the convergences give the best possible approximation by a fraction with a given denominator [Dav99]. Thus any fraction which gives a better approximation must have a larger denominator. If you are interested in determining an approximation of an irrational number to a certain accuracy a bound can be given for the denominator, allowing the best approximation within that bound to be found.

The implementation of continued fractions is split into recursive iterations of split and invert. Each iteration takes the numerator n and the denominator d . During the calculation we can keep track of the approximate fraction s_m/r_m where $s_m = q_m s_{m-1} + s_{m-2}$ and $r_m = q_m r_{m-1} + r_{m-2}$, and q_m is the quotient from the m 'th split and invert step. To calculate the approximate fraction on the m 'th iteration, $s_{m-1}, s_{m-2}, r_{m-1}$ and r_{m-2} are all needed. Since we are given a bound N we can stop the split and invert steps there. Each split and invert iteration therefore takes the inputs $(n, d, s1, s2, r1, r2, N)$. An outline of the algorithm is shown in algorithm 2

Algorithm 2: Continued Fractions

```

input:  $(n, d, s1, s2, r1, r2, N)$ 
1 if  $d = 0$  Or  $r2 > N$  then
2   if  $d = 0$  and  $r2 \leq N$  then Return  $s2/r2$ ;
3   else Return  $s1/r1$ ;
4 else
5   let  $quotient = n/d$  ;
6   let  $(n, d) = (d, n - quotient \cdot d)$  ;
7   let  $(s1, s2) = (s2, s1 - quotient \cdot s2)$ ;
8   let  $(r1, r2) = (r2, r1 - quotient \cdot r2)$ ;
9   Return Continued Fractions( $n, d, s1, s2, r1, r2, N$ )
10 end
```

Greatest Common Divisor

A very similar algorithm to the continued fractions algorithm is the extended Euclidean algorithm which is an efficient method for computing the greatest common divisor. Conveniently the extended Euclid's algorithm uses the same process as in the continued fractions algorithm and it also has the same maximum number of steps. The greatest common divisor is given by $s_m \cdot n + r_m \cdot d$. An outline of the algorithm is shown in algorithm 3.

Algorithm 3: Greatest Common Divisor

```

input:  $(n, d, s1, s2, r1, r2, N)$ 
1 if  $d = 0$  then
2   | Return  $s1/r1$ 
3 else
4   | let  $quotient = n/d$  ;
5   | let  $(n, d) = (d, n - quotient \cdot d)$  ;
6   | let  $(s1, s2) = (s2, s1 - quotient \cdot s2)$ ;
7   | let  $(r1, r2) = (r2, r1 - quotient \cdot r2)$ ;
8   | Return Greatest Common Divisor( $n, d, s1, s2, r1, r2, N$ )
9 end

```

2.5.2 Grover's Quantum Search Algorithm

There are two fundamental quantum algorithms which can be used in a wide range of applications, and make up the foundation of both Shor's and the BBM algorithm. The two fundamental algorithms are discussed below.

The first algorithm is Grover's quantum search algorithm. Grover's algorithm attacks the following problem: given a space to search of size N , find an element that satisfies some property, with no prior knowledge. One application of an algorithm which attacks this search problem is to find the shortest route passing through all cities on a map. If there were N routes, a classical computer would take $O(N)$ operations to determine the shortest route, effectively checking each route one-by-one. Grover's algorithm makes a quadratic speed up to this, allowing a quantum computer to find the shortest route in just $O(\sqrt{N})$ operations [Gro96].

To identify solutions to the search problem Grover's algorithm makes use of a component called a quantum oracle. A quantum oracle is a black box which upon recognising solutions will mark solutions to the problem by shifting their phase. The oracle can be thought of as utilising a function $f(x)$, where if x is a solution $f(x) = 1$ and if x is not a solution $f(x) = 0$. Formally, the action of the quantum

oracle is defined by:

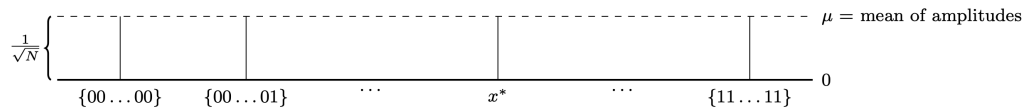
$$|x\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle$$

if x is a solution $f(x) = 1$ and thus the phase of $|x\rangle$ is flipped. If x is not a solution $f(x) = 0$ and $|x\rangle$ is left alone. A crucial note to make about the oracle is the distinction between the oracle knowing a solution and the oracle being able to recognise a solution. For example, if you had many keys you may not know which one unlocks a particular door, but it would be easy to recognise if a key did unlock a door by simply trying the key in the lock and the door opening. In the same way it is possible to construct a quantum oracle which recognises a solution to a search problem when it sees one. The challenge is designing a quantum circuit to recognise solutions to the particular search problem you are interested in.

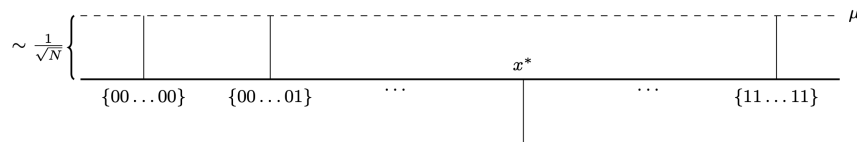
Simply flipping the phase of solutions is not enough to identify them, since a measurement would only yield only a single computational basis state and at a probability of its amplitude squared. Therefore flipping the phase does not change the probability of measurement at all! What we really want is to increase the probability of measuring a solution so that we are overwhelmingly likely measure the solution. The clever trick is the use of Grover's diffusion operator, which is sometimes referred to as inversion about the mean. Grover's diffusion operator flips the phase of every computational basis around the mean of the amplitudes of every computational basis. Grover realised that placing the oracle and the diffusion operator in sequence would amplify the solutions the oracle had identified, increasing the probability of measuring them. This quantum subroutine known as the Grover iteration goes as follows:

1. Apply the oracle O
2. Apply Grover's diffusion operator

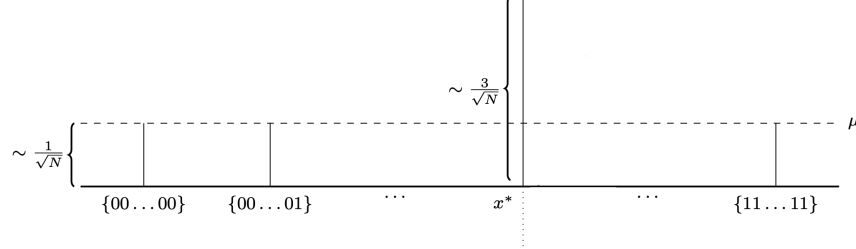
To see how this amplification occurs, the Grover iteration can be helpfully visualised graphically [Hui]. To start the amplitudes of each basis state is equal:



Next, the oracle is used to flip the state of a solution.



Finally, the amplitudes of all the basis are inverted about the mean, resulting in the solution state being amplified.



Applying the Grover iteration multiple times massively increases the amplitude of a computational basis which is a solution. Grover found that to have a near certain probability of measuring a solution the Grover iteration must be performed $O(\sqrt{N/M})$ times, where N is the search space and M is the number of solutions in the search space. That means if there is only one solution it can be found in just $O(\sqrt{N})$ operations on a quantum computer. This is a quadratic speed up to the $O(N)$ operations it would take on a classical computer.

2.5.3 Shor's Quantum Fourier Transform

The second fundamental algorithm for quantum computing is Shor's quantum Fourier transform (QFT). The QFT is a critical component in Shor's factoring algorithm as well as numerous other quantum applications. Classically a Fourier Transform is used to transform a signal from the time domain to the frequency domain. For example a sinusoidal wave with a 1Hz frequency is transformed to a delta functions with a value of 1. Correspondingly, a delta function of with a value of 1 would be transformed to a sinusoidal wave of frequency 1Hz. One very useful property of the classical Fourier transform is if a function in the time domain has a period r , its transformation in the frequency domain has a period $1/r$. Formally the classical Fourier transform is defined as:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i j k / N}$$

A QFT performs the same transformation, but due to the quantum nature is defined as a linear operator acting on a quantum register as follows:

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i j k / N} |k\rangle.$$

As in the classical case a delta function is transformed to a sinusoidal function. For example a state $|10\rangle$ is transformed to $|00\rangle - |01\rangle + |10\rangle - |11\rangle$ and vice versa. Since this transformation is unitary it can be implemented on a quantum computer. Unfortunately there are some difficult problems with the QFT as a standalone algorithm. Firstly, since a measurement just returns the computational basis and not its amplitude, the amplitudes cannot be accessed. Secondly, there is no general way to prepare an arbitrary state to be Fourier transformed, so the process may seem redundant. However, when combined within other algorithms the QFT can become an incredibly useful subroutine.

Quantum Phase Estimation

The Quantum Fourier transform is a key element in a quantum procedure known as phase estimation. Phase estimation solves the typical linear algebra problem - given a matrix and its eigenvector, find the eigenvalue. Solving this problem on a quantum computer makes phase estimation key for a number of quantum algorithms. For quantum computers the matrix represents a unitary operator and the eigenvector is represented by a register of qubits, as shown in equation 2.4. The corresponding eigenvalue is an exponential complex function with a real part θ . Phase estimation is a procedure to estimate the value of θ .

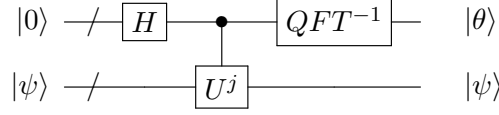
$$U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle \quad (2.4)$$

The estimation is completed using two registers, the first initially in state $|0\rangle$ and the second in state $|\psi\rangle$. To start the first register is put into a superposition of all possible values using Hadamard gates. Controlled- U operations are then used to create a state that can be processed by a QFT to estimate θ . U is raised to successive powers of 2 for each qubit in the first register. Finally an inverse QFT is used to put θ into a register. Overall the phase estimation produces the following transformations:

$$|0\rangle|\psi\rangle \xrightarrow{H} |j\rangle|\psi\rangle \xrightarrow{U^j} \frac{1}{\sqrt{2^t}} e^{2\pi i j \theta} |j\rangle|\psi\rangle \xrightarrow{QFT^{-1}} |\theta\rangle|\psi\rangle$$

where t is the size of the first register. The circuit for quantum phase estimation is shown below. The accuracy of the approximation of θ is dependant on the length of the first register t . The larger t is the more digits of accuracy θ can be approximated to.

Similarly to the QFT the phase estimation algorithm on its own is not enough



to be useful, and there are two hurdles to overcome. Firstly, it must be possible to prepare a state $|\psi\rangle$ and secondly, it is necessary to have operators able to perform the controlled- U operations. If these hurdles are overcome phase estimation is a very useful sub-routine that enables interesting problems to be solved. One important problem is order finding, which is now discussed.

Quantum Order Finding

The order of x modulo N is the smallest integer r , such that $x^r \equiv 1 \pmod{N}$. The order finding problem is to determine r given x and N . It is useful to note that $x^r \pmod{N}$ repeats in a cyclic pattern as r is incremented, with a period of at most $N - 1$. Therefore the maximum value of the order is $N - 1$. Quantum order finding applies the phase estimation algorithm to the unitary operator U and the corresponding eigenvector $|u_s\rangle$ and eigenvalue shown below.

$$\begin{aligned}
 U|y\rangle &\equiv |xy \pmod{N}\rangle \\
 |u_s\rangle &\equiv \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \exp\left\{\frac{-2\pi i s k}{r}\right\} |x^k \pmod{N}\rangle \\
 U|u_s\rangle &= \exp\left\{\frac{2\pi i s}{r}\right\} |u_s\rangle
 \end{aligned}$$

Where r is the order we are aiming to calculate. In order to carry out phase estimation using this operator, eigenvector and eigenvalue the two hurdles for phase estimation must be overcome. The first hurdle of being able to prepare the state $|u_s\rangle$ is overcome by a clever observation, which is,

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |u_s\rangle = |1\rangle.$$

This means carrying out phase estimation using $|1\rangle$ in the second register, is equivalent to carrying out phase estimation with a superposition of $|u_s\rangle$ in the second register. Therefore for each s in the range 0 to $r - 1$ we obtain an estimate of the phase $\theta \approx s/r$, all in superposition. Thus all we need to prepare to get an estimate of θ is the state $|1\rangle$ in the second register, which is trivial.

The second hurdle of performing the controlled- U operations is overcome using a classical procedure known as modular exponentiation. Modular exponentiation allows the computation of $x^{2^i} \bmod N$ for all $x < t$ to be computed efficiently on a classical computer. These classical values can then be used by multiplying the quantum register where they are required. Since modular multiplication of a quantum register by a classical number can be implemented.

Since both hurdles have been overcome the phase estimation procedure can be used to get an estimate of $\theta = s/r$. This is done through the following transformations:

$$|0\rangle |1\rangle \xrightarrow{H} |j\rangle |1\rangle \xrightarrow{U^j} |j\rangle |x^j \bmod N\rangle \approx \frac{1}{\sqrt{2^t}} e^{2\pi i j \frac{s}{r}} |j\rangle |u_s\rangle \xrightarrow{QFT^{-1}} |s/r\rangle |u_s\rangle$$

Next this first register now containing s/r can be measured obtaining a classical integer. The final step is to now calculate r from this classical integer, this can be done efficiently using the classical continued fractions method.

Phase estimation utilising a QFT can therefore calculate the order of x modulo N . This problem is in fact equivalent to the factoring problem and is the basis for Shor's algorithm.

2.6 Quantum factoring Algorithms

2.6.1 Shor's Algorithm

Building on Deutsch and Jozsa's work Shor showed that a quantum computer could efficiently solve two critical problems — the discrete logarithm problem and the problem of finding prime factors [Sho99]. In order to rapidly speed up the process of factoring large numbers Shor's algorithm exploits the counter intuitive rules of quantum mechanics.

Similarly to the classical factoring algorithms Shor's aim is to find an x such that $x^2 - 1 = (x + 1)(x - 1) = p \cdot q = N$ or in other terms that $x^2 \equiv 1 \bmod N$. Since this gives us non-trivial divisors $\gcd(N, x + 1)$ and $\gcd(N, x - 1)$ of N . The challenge is therefore finding squared values which are congruent 1 $\bmod N$. Shor does this by solving the order finding problem for a random integer x where $0 < x < N$. That is finding an integer r such that $x^r \equiv 1 \bmod N$. If r is even then x^r is a square and thus $x^r - 1 = (x^{r/2} + 1)(x^{r/2} - 1) = p \cdot q = N$, and $\gcd(N, x^{r/2} + 1)$ and $\gcd(N, x^{r/2} - 1)$ are non-trivial divisors. In order to find the

order of x modulo N Shor uses the quantum order finding algorithm discussed above. Shor's algorithm is summarised in algorithm 4.

Algorithm 4: Shor's Algorithm

```

input :  $N$ 
output: A non-trivial factor of  $N$ 
1 if  $N$  is even then Return 2.;
2 if  $N$  is prime power then Return prime.;
3 Choose random integer  $x$  such that  $0 < x < N$ ;
4 if  $\gcd(N, x) > 1$  then Return  $\gcd(N, x)$ ;
5 Use quantum order finding sub-routine to find  $r$  such that  $x^r \equiv 1$ 
   mod  $N$ .;
6 if  $r$  is even and  $x^{r/2} \not\equiv -1 \pmod{N}$  then
7   | if either  $\gcd(x^{r/2} \pm 1, N)$  are non-trivial factors then Return
   | non-trivial factor;
8   | else go to step 3;
9 end

```

Shor's algorithm has a time complexity of $O(\text{poly}(\log(N)))$ compared to, the number field sieve which has a time complexity of $O(L(N)^{1.9+o(1)})$. It thus has an exponential speed up for factoring a large integer compared to the best classical methods. Enabling the ability to efficiently break an RSA encryption. However, the number of qubits required grows linearly with the number of bits in N . This means to factor the key sizes used in RSA vast numbers of qubits are required.

Shor's algorithm can be implemented in many different ways, all with their own space/ time trade-offs. A lot of work has been done to minimise the time complexity of Shor's algorithm, this is because the run-time of an algorithm is highly related to the depth of the circuit required to run it. For quantum algorithms the bigger the depth of the circuit the longer the amount of time qubits must be in existence for to carry out an algorithm. Since qubits are currently not perfect the longer they are in existence the longer they must be stabilised by quantum error correcting codes and the more error that can creep in, making results less accurate. Therefore fast versions of Shor's algorithm have been developed [Zal98]. However, optimising the depth often has a pay off of increasing the width required for the computation, meaning potentially more qubits are required. For this report the focus is on the maximum number of qubits required at any one time. There has been some effort to reduce the number of qubits required, for example [HRS17], [Sei01]. However the rate of growth remains linear. The best current

implementation attempts have reduce the number of qubits required at any one time to just $2L + 3$ where L is the number of bits required to store the number being factored N [Bea03]. This means factoring 15 with a general algorithm would require just 11 qubits. It is interesting to note that for the special case of factoring 15 further improvements have been made since many of the multiplications are simply the identity, resulting in unused qubits. One impressive physical implementation factored 15 using NMR with just seven qubits [VSB⁺01].

2.6.2 Bernstein, Biasse, and Mosca Low-Resource Quantum Factoring Algorithm

The BBM low-resource quantum factoring algorithm is an attempt to use less qubits to factor an integer N than Shor’s algorithm. It makes use of a quantum computer to rapidly speed up the general number field sieve [BBM17]. As mentioned above the NFS is split into two important steps. Firstly searching for a set S of pairs of numbers (a, b) , such that $(a + mb)$ and $(a + \alpha b)$ are both smooth, and secondly using linear algebra to find a subset of S to form a difference of squares $x^2 \equiv y^2 \pmod{N}$. The BBM algorithm utilises Grover’s quantum search algorithm to speed up the searching step, allowing the set S of pairs (a, b) to be found quadratically quicker.

The BBM algorithm has a time complexity of $O(L(N)^{1.38+o(1)})$, making the time it takes asymptotically worse than Shor’s algorithm, however the qubit complexity is just $O((\log_2(N))^{2/3+o(1)})$, meaning the number of qubits required is asymptotically better than Shor’s algorithm, since BBM is sublinear. Additionally the time complexity has a considerable improvement to the classical general NFS. Since the BBM algorithm uses asymptotically less qubits than Shor’s algorithm, the BBM algorithm may be able to be implemented sooner to factor common RSA key sizes, as a smaller quantum computer may be required. However, as discussed above big O notation hides constants, and we only actually know which algorithm wins out asymptotically. An important open question is whether the BBM algorithm would use fewer qubits than Shor’s algorithm to break common RSA key sizes. This report explores this by implementing the major quantum component of the BBM algorithm on a quantum simulator. Allowing the a minimum bound for the resources required to factor numbers of RSA key sizes to be compared to the resources required by an implementation of Shor’s algorithm.

The BBM Quantum Search

The NFS is dependent of finding pairs of numbers (a, b) such that $(a + mb)$ and $(a + \alpha b)$ are both smooth numbers. This is equivalent to finding numbers such that the product $(a + mb)(a + \alpha b)$ is smooth. The number of bits in $(a + mb)(a + \alpha b)$ is at most $(\log N)^{2/3+o(1)}$. The BBM algorithm uses a careful reversible algorithm design to require just $(\log N)^{2/3+o(1)}$ qubits. This is asymptotically sublinear in the length of N .

In order to find smooth integers, the BBM algorithm utilises Grover's quantum search with an oracle such that:

- $U_y |s\rangle = -|s\rangle$ if s is y -smooth
- $U_y |s\rangle = |s\rangle$ if s is not y -smooth

That is given a superposition of numbers the oracle will flip the amplitude of the numbers which are smooth. Thus when using this oracle in Grover's quantum search the amplitudes of smooth numbers will be increased, making them highly likely to be measured. One run of Grover's quantum search will only identify one smooth number. Since multiple smooth numbers are needed BBM partition the search space, and each part is systematically searched with a high probability of finding a smooth number. Once a sufficient number of smooth numbers have been found linear algebra can be used to factor N , as in the classical NFS.

In order to identify smooth numbers the oracle used in the BBM algorithm carries out Shor's algorithm repeatedly. This divides N into its divisors, which are themselves then divided into their divisors, and so on. Once Shor's algorithm is run sufficient number of times it is easy to identify smooth numbers by collecting any numbers where all of the divisors are below the smooth bound. There are two important distinctions between how Shor's algorithm is used in the BBM algorithm and how it is used as a standalone algorithm. Firstly, the BBM algorithm uses Shor's algorithm to factor numbers of bitsize $(\log_2(N))^{2/3+o(1)}$ as opposed to size $\log_2(N)$, this is where the major potential advantage for the BBM algorithm over Shor's conventional algorithm comes from. Secondly, when used in the BBM algorithm Shor's algorithm must be carried out all in superposition. Changing Shor's algorithm to all be in superposition is discussed below.

BBM present both a parallel and a serial construction for their quantum factoring algorithm, which both use Shor's algorithm in superposition as a sub-routine. They also both have the same space complexity.

2.6.3 Shor's Algorithm in Superposition

Conventionally Shor's algorithm is applied to a single integer N resulting in a divisor N_1 . The BBM algorithm applies Shor's algorithm to a superposition of inputs $|N\rangle$ obtaining a superposition of divisors $|N_1\rangle$. When carrying out Shor's algorithm in the conventional way many steps can be completed using a classical computer, mainly the selection of x , the two crucial classical computations in the quantum order finding sub-routine of modular exponentiation and, continued fractions, the calculation of $x^{r/2} \pm 1$ and finally the greatest common divisor computation. However, when carrying out Shor's algorithm in superposition all these steps must also be completed in superposition using the quantum computer. This changes the cost of how many qubits are required for the full algorithm. The main challenge of this report is therefore to implement Shor's algorithm in superposition, allowing the creation of an oracle which identifies smooth numbers.

The method given by BBM fits Shor's algorithm in superposition into just $O((\log x)^{1+o(1)})$ qubits, this is the same space complexity as Shor's original algorithm however, the BBM algorithm applies to numbers of bits at most $(\log N)^{2/3+o(1)}$ long, resulting in the entire algorithm fitting into just $O((\log N)^{2/3+o(1)})$ qubits. A number of alterations are required to conduct Shor's algorithm in superposition without using many qubits. The components that must be altered are discussed below.

Selecting x

To start the trivial step of selecting an integer x randomly between 1 and $N - 1$ must be conducted carefully with the superposition $|N\rangle$. Since every N in $|N\rangle$ needs a corresponding x we also need a superposition $|x\rangle$. Therefore whereas conventionally x is a classical number, it must now be a quantum number stored in a quantum register. the selection of x therefore performs the following transformation:

$$|N\rangle |0\rangle \rightarrow |N\rangle |x\rangle,$$

taking two input quantum registers, one containing the superposition of N 's and the other empty, and outputting two registers, containing the superpositions of N 's and corresponding x 's. To select an appropriate x for every N BBM choose to select one random integer R and then defines $x = 1 + (R \bmod (N - 1))$. This produces a superposition $|x\rangle$ such that $1 < x < (N - 1)$ for every value in the superposition $|N\rangle$. Following the transformation the two registers are entangled, meaning if you measured the first register as y , you would then be guaranteed to

measure $x = 1 + (R \bmod (y - 1))$ in the second register.

Square and Multiply Modular Exponentiation

The next component which must be changed to work in superposition is the modular exponentiation used during the quantum order finding sub-routine. Conventionally this component calculates the superposition $|x^j \bmod N\rangle$, where x and N are classical integers and j is a quantum number in a superposition $|j\rangle$. Shor does this by precomputing (using a classical computer) the sequence $x^2 \bmod N, x^4 \bmod N, \dots$ allowing the calculation of the superposition $|x^j \bmod N\rangle$. This requires $O((\log N)^{2+o(1)})$ bits on the classical computer, since each item in the sequence is $O(\log N)$ bits long and we require a sequence $O(\log N)$ long.

The difficulty arises since we start instead with superpositions $|x\rangle$ and $|N\rangle$, as well as $|j\rangle$. The precomputed sequence would thus also have to be in a superposition stored in a quantum register. To do this would thus require $O((\log N)^{2+o(1)})$ qubits. Instead of using Shor's conventional strategy, to reduce the qubits required BBM use a square and multiply modular exponentiation algorithm. Overall the modular exponentiation must perform the transformation:

$$|N\rangle |x\rangle |j\rangle |0\rangle \rightarrow |N\rangle |x\rangle |j\rangle |x^j \bmod N\rangle.$$

As with the selection of x following this transformation these four registers are now entangled. This transformation can be achieved using Bennett's generic conversion on the square and multiply algorithm. Usefully the square and multiply algorithm is split into steps which can be used as the steps in Bennett's conversion. This allows the modular exponentiation algorithm to fit into just $O((\log x)^{1+o(1)})$ space. Usefully this same square and multiply component can be reused later on in Shor's algorithm in superposition to calculate $x^{r/2} \bmod N$.

Continued Fractions

Next the continued fractions component must be altered to be conducted in superposition. Conventionally this would take a classical irrational number θ obtained from the measuring step in the quantum order finding sub-routine, as well as a bound for the limit of the continued fractions. It would then output the best approximation to that irrational number as a rational fraction, with the denominator less than the bound. For the case of Shor's algorithm the bound is N and the denominator is the order r . Since the whole process needs to be conducted in superposition there would be no measurement step as this would collapse the su-

perposition, instead continued fractions must be conducted on the superposition of all θ 's. For the case of Shor's algorithm in superposition we are only interested in the approximated fractions denominator r , therefore the transformation needed is:

$$|N\rangle |\theta\rangle |0\rangle \rightarrow |N\rangle |\theta\rangle |r\rangle ,$$

This transformation can again be carried out using the Bennett's generic conversion on the conventional continued fractions method. Each 'split and invert' step can be used as a step in Bennett's conversion.

Greatest Common Divisor

The final component of the conventional Shor's algorithm that must be altered is finding the greatest common divisor. Conventionally this is just finding the gcd of two classical integers. However, we now need a method for finding the greatest common divisor of two quantum numbers. Thankfully this can also be done using Bennett's generic conversion of the extended Euclidean algorithm, performing the transformation:

$$|a\rangle |b\rangle |0\rangle \rightarrow |a\rangle |b\rangle |\gcd(a, b)\rangle ,$$

Since the continued fractions method is nearly identical to calculating the gcd via the extended Euclidean algorithm the 'split and invert' steps can again be used as the steps in Bennett's conversion. A helpful point to note is that the $\gcd(a, b) = \gcd(a \bmod b, b)$. Therefore we can use the square and multiply modular exponentiation in superposition to calculate $(x^{r/2} \pm 1) \bmod N$ which is then input to the greatest common divisor component.

Overview

Now we have all the necessary components to calculate Shor's algorithm in superposition, we now put all the transformations together, as shown in algorithm 5. Since all the intermediate values must be removed at the end the all the components are reversed to just leave $|N\rangle |N_1\rangle$.

Algorithm 5: Shor's Algorithm in Superposition

input : $|N\rangle |0\rangle$
output: $|N\rangle |N_1\rangle$

- 1 $\rightarrow |N\rangle |0\rangle;$
- 2 $\rightarrow |N\rangle |0\rangle |x\rangle //$ Selecting x
- 3 $\rightarrow |N\rangle |0\rangle |x\rangle |j\rangle //$ Creating j
- 4 $\rightarrow |N\rangle |0\rangle |x\rangle |j\rangle |a^j \bmod M\rangle //$ Modular Exponentiation
- 5 $\rightarrow |N\rangle |0\rangle |x\rangle |\theta\rangle |a^j \bmod N\rangle //$ Inverse QFT
- 6 $\rightarrow |N\rangle |0\rangle |x\rangle |\theta\rangle |a^j \bmod N\rangle |r\rangle //$ Continued Fractions
- 7 $\rightarrow |N\rangle |0\rangle |x\rangle |\theta\rangle |a^j \bmod N\rangle |r\rangle |(x^{r/2} - 1) \bmod N\rangle //$ Calc. $(x^{r/2} - 1)$
 $\bmod N$
- 8 $\rightarrow |N\rangle |N_1\rangle |x\rangle |\theta\rangle |a^j \bmod N\rangle |r\rangle |(x^{r/2} - 1) \bmod N\rangle //$ Calc. N_1
- 9 $\rightarrow |N\rangle |N_1\rangle |x\rangle |\theta\rangle |a^j \bmod N\rangle |r\rangle //$ Reversing Calc. $(x^{r/2} - 1)$
 $\bmod N$
- 10 $\rightarrow |N\rangle |N_1\rangle |x\rangle |\theta\rangle |a^j \bmod N\rangle //$ Reversed Continued Fractions
- 11 $\rightarrow |N\rangle |N_1\rangle |x\rangle |j\rangle |a^j \bmod N\rangle //$ QFT
- 12 $\rightarrow |N\rangle |N_1\rangle |x\rangle |j\rangle //$ Reversed Modular Exponentiation
- 13 $\rightarrow |N\rangle |N_1\rangle |x\rangle //$ Removing j
- 14 $\rightarrow |N\rangle |N_1\rangle //$ Removing x

2.6.4 Comparing Quantum Algorithms to Classical Algorithms

To compare the resources used by quantum algorithms to classical algorithms you need to be able to evaluate the cost of the computation. Assuming that a step in a quantum computer is equivalent to a classical computer; you could simply sum the number of steps required for the algorithms. However, realistic design for large scale quantum computers call this assumption into question.

The difficulties arise due to noise causing the coherence of the qubits to be finite. As discussed above the decoherence of qubits means quantum error correcting codes must be used to prevent the quantum state being corrupted. Preserving a quantum state is therefore an active process even if no quantum gates are performed on the qubit. Classical computers are required for the quantum error correcting codes, resulting in an overall increased amount of computation needed for the quantum algorithm. Additionally as the input increases it would be expected the number of logical qubits needed increases, therefore more classical computation will be required for the quantum error correcting codes.

In order to make a direct comparison between quantum algorithms and classical algorithms it is clear that the classical computation for quantum error correcting codes should be taken into account. This will make the comparison highly dependent on the architecture of the quantum computer used. There have been some attempts to directly compare the resources of quantum algorithms with classical algorithms. The idea in [AMG⁺17] is to take some metrics concerning your quantum circuit and compute the classical resources required to implement the quantum error-correction protocols, which would give a minimal cost to the quantum computation and therefore allow for a direct comparison of resources.

This project ignores the overhead of quantum error-correction. This of course will make it more difficult to directly compare the resources used by NFS, Shor and BBM, however, it will allow a simpler implementation and comparison between Shor and BBM.

2.7 Quantum Computer Simulators

Since quantum computers are currently not scalable and widely available this project will use a simulator to investigate and compare the algorithms. A quantum simulator works by using a random number generator to achieve the probabilistic nature of qubits [Ker], the unitary matrices which describe the quantum gates can then be multiplied to simulate the quantum circuit.

To simulate a quantum computer on a classical computer you must store all the information about the current quantum state. Due to the exponential growth in the number of possible states with the number of qubits this soon gets too large for a classical computer to handle. For example trying to simulate 500 qubits would require at least 2^{500} amplitudes, this is larger than the number of atoms in the observable universe [NC19]. The simulators are therefore limited on how many qubits can be fully simulated.

Microsoft's Quantum Development Kit

There are a number of different freely available quantum simulators, all at different stages of development, for example, IBM's Qiskit, Googles Cirq, and Microsoft's Quantum development kit. All these simulators offer a range of tools and functions to allow you to program and control a new quantum circuit. For this report Microsoft's quantum development kit (MQDK) was used since it offers a key advantage of having different types of simulators. The MQDK is based on the domain-specific language Q# [SGT⁺18] which uses the universal Clifford

+ T quantum gate set. Although still limited by how many qubits can be fully simulated, MQDK allows the design and costing of quantum circuits, irrespective of the actual number of qubits required. It accomplishes this by executing the quantum operation without actually simulating the state of a quantum computer, it can therefore estimate the resources for operations that would require thousands of qubits. Since the main goal of the project is to make a comparison of the resources used by quantum algorithms, the resource estimator in MQDK will be very beneficial.

MQDK also comes with a full quantum simulator that allows execution and debugging of quantum algorithms. The scale of algorithms which can be run is limited due to the exponential growth of states as described above. In the MQDK the number of qubits that can be fully simulated is roughly limited to about 30 qubits. However, it will be useful to run small instances for debugging and investigation purposes. A noteworthy feature is the ability to view the quantum state at any point during a computation without the collapse of the state, this of course is not possible in a real quantum computer, but is very useful for debugging purposes when using a simulator.

The final simulator that MQDK offers is the Toffoli simulator. This simulator is a special purpose simulator with a limited scope that only supports X, CNOT and multi-controlled X gates. This limits the simulator to just running binary operations, meaning superpositions can not be used with this simulator. While the Toffoli simulator is far more restricted, it can be used with millions of qubits, far more than the full quantum simulator. All of the new circuits and operations created in this report were within the scope of the Toffoli simulator. The Toffoli simulator thus became a vital tool for testing and debugging, allowing a quantum state not in superposition to be put in to a new circuit and receiving a quantum state not in superposition out very quickly, just like a classical computer. Many different inputs can then be tested for the correct output, giving confidence in the accuracy of the new circuit. This testing process is identical to that of a new classical function.

Another major benefit of the MQDK is that it has implementations of both Shor's quantum algorithm and Grover's quantum search algorithm. These proved extremely helpful for both learning the MQDK and the algorithm's themselves.

MQDK has a huge range of readily available operations, for example, there are arithmetic operations to add, increment, multiply, square, divide and compare two quantum registers. These operations will be incredibly useful when implementing Shor's algorithm in superposition, since many of the steps conventionally done on

a classical computer now must be done in quantum registers, for example modular exponentiation, requires both squaring and multiplying. There are however still operations which have not yet been implemented as part of the standard MQDK that are required for this report, such as modular and signed arithmetic. These will therefore have to be implemented to allow Shor's algorithm to be run in superposition. Below in table 2.3 of some already implemented arithmetic circuits.

Operation	Quantum Circuit Diagram
Add	
Multiply	
Square	
Divide	
Greater Than	

Table 2.3: Table of MQDK implemented arithmetic operations

2.8 Summary

This chapter has covered the necessary background and the current research to justify and motivate this project. To start it was found that some of the most significant modern encryption systems are based on the problem of factoring large integers; a problem that is infeasible on a classical computer. The current best method to factor an arbitrary integer on a classical computer, the general number field sieve, was explored, although it should be noted that it is not a very fast process.

The review then considered quantum computers. By taking advantage of the strange properties of quantum mechanics Shor showed that the problem of

factoring arbitrary integers was feasible on a quantum computer. However the rate of growth of the required number of qubits puts a physical implementation far out of reach of current quantum computers. A low resource quantum factoring algorithm presented by Bernstein, Biasse and Mosca required a sub-linear number of qubits but at a worse time complexity. Therefore at sufficiently large key sizes BBM would need less qubits than Shor's algorithm, however at what key size the BBM algorithm would use less qubits is unknown. The crucial sub-routine of the BBM algorithm was in fact Shor's algorithm, but it is used to factor smaller numbers and vitally is all conducted in superposition. The various components that would need modifying to implement Shor's algorithm in superposition were then discussed.

This project aims to implement Shor's algorithm in superposition with the hope of discovering the resources it requires. This will help determine if the whether the BBM algorithm would require less qubits than Shor's algorithm for common key sizes such as 2048-bit RSA keys. The implementation and investigation will be carried out using Microsoft's quantum development kit.

Section 3

Implementation

This chapter describes the implementation of Shor's algorithm in superposition, the crucial sub-routine in the BBM algorithm. To start four new fundamental operations that are required are implemented, before implementing the three main altered components putting them into superposition.

3.1 Implementing Fundamental Operations

In order to allow Shor's algorithm to be conducted in superposition four fundamental operations were required which in superposition which had not previously been implemented in the MQDK. Below is a description of the implementations of the four operations required in superposition.

3.1.1 Modular Square

In order to perform square and multiply modular exponentiation both a modular square and a modular multiply operation are required. To start the implementation of the modular square operation is tackled. The modular square operation must perform the transformation:

$$|a\rangle |b\rangle |0\rangle \rightarrow |a\rangle |b\rangle |a^2 \mod b\rangle.$$

To implement this transformation three operations already implemented in MQDK are used, the square operation, the divide operation, and the add operation, as shown in Table 2.3. Figure 3-1 shows the quantum circuit diagram for the implementation of the modular square operation. To ensure the computation is reversible two auxiliary registers are used and reset to zero by the end of the computation so they can be released. The implementation is split into five steps, each described in table 3.1.

A key point to note is that different registers are required to be different sizes. For example, the target register *R1* for the squaring operation in step 1, must

be double the size of register $R2$ storing a , since squaring a number can at most double the number of bits required to stored it. For the edge case of $b = 0$ a control qubit can be used to control the whole operation, giving an output of 0.

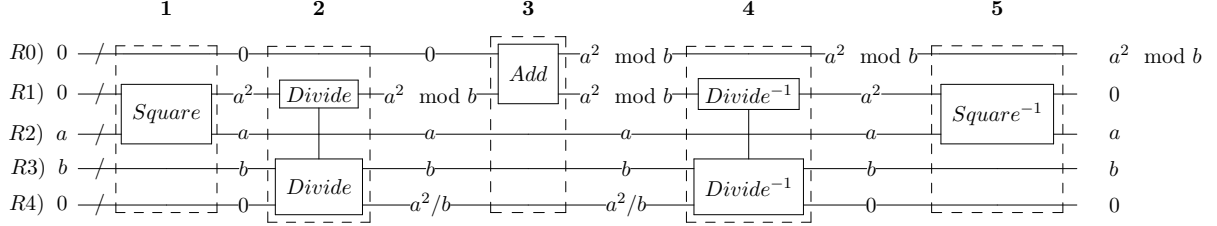


Figure 3-1: Modular Square Quantum Circuit

Step	Description
Input	An input of 3 registers all of equal size L is taken, storing the a , the modular b and a target register to take the result.
1	The first step is simply to square a into the auxiliary register $R1$. Register $R1$ must be of size $2L$ since a^2 can at most be double the number of bits in a
2	The next step is to divide a^2 by b , this outputs both the floor of the divide and the remainder. Since the divide operation takes three registers all of equal size, register $R4$ must be of size $2L$ since $R1$ is that size. Also an additional padding register of length L is used to pad the register $R3$ storing b since $R3$ is only of size L .
3	Next the remainder from the previous step is added to the target register $R0$, since this is in state $ 0\rangle$ this simply results in the target register now just containing the remainder. The remainder stored in $R1$ is at most $b - 1$, therefore $R0$ only needs to be of size L .
4	Next the auxiliary registers must be reset, this is done by reversing the previous steps. Firstly the divide operation is reversed.
5	The final step is to reverse the initial squaring operation, leaving all auxiliary registers reset to $ 0\rangle$ allowing them to be reused for a later calculation.
Output	Three registers all of size L are output containing a , b and $a \bmod b$

Table 3.1: Modular Square Operation

3.1.2 Modular Multiply

Modular multiplication is very similar to modular squaring, and a near identical implementation is used. The modular multiplication operator must perform the transformation:

$$|a\rangle |b\rangle |c\rangle |0\rangle \rightarrow |a\rangle |b\rangle |c\rangle |a \cdot b \bmod c\rangle.$$

The full quantum circuit is shown in figure 3-2. The extra input register containing b is taken and the multiply operation is used in step 1 and 5 instead of the squaring operation. The full quantum circuit is shown below.

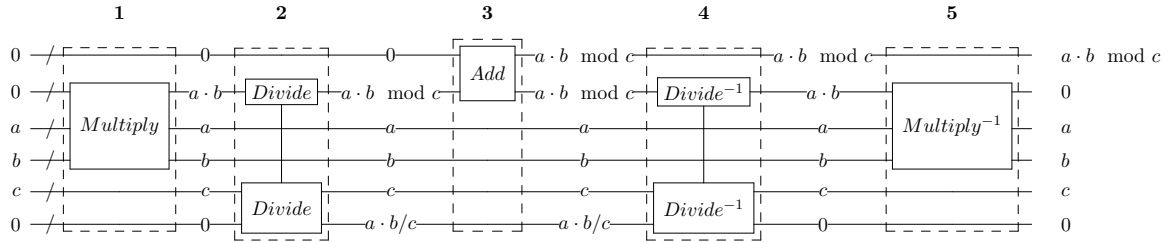


Figure 3-2: Modular Multiply Quantum Circuit

3.1.3 Signed Multiply

To complete the continued fractions step of Shor's algorithm in superposition a signed multiplication operation is required, allowing two signed numbers a and b to be multiplied. The convention of using an additional bit to represent the sign of a number in classical computation was adopted by using an additional qubit to represent the sign of a quantum number. A value of zero and one is given to the sign bit if the number is positive or negative respectively. The operation must perform the transformation:

$$|a\rangle |b\rangle |0\rangle \rightarrow |a\rangle |b\rangle |a \cdot b\rangle$$

The implementation of the signed multiply operation is shown in figure 3-3. The operation is split into 2 steps described in table 3.2.

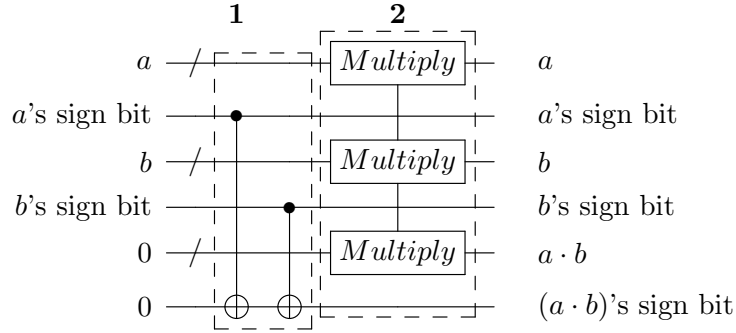


Figure 3-3: Signed Multiply Quantum Circuit

Step	Description
Input	An input of 3 registers is taken. Two are of equal size $L + 1$, storing the signed integers a and b . The third register is the target register and is of size $2L + 1$.
1	The first step is to perform an XOR operation on the two integers signed bits, determining the signed of the output bit. This is done through two consecutive CNOT gates
2	The next step is to simply perform a multiplication on the two integers to determine the absolute value of $a \cdot b$. Combined with the signed bit from step this produces the signed output.
Output	Three registers all of size L are output containing a , b and $a \bmod b$

Table 3.2: Signed Multiplication Operation

3.1.4 Signed Subtract

The final operation required is a signed subtraction, allowing two signed numbers a and b to be subtracted. It performs the transformation:

$$|a\rangle |b\rangle \rightarrow |a - b\rangle |garbage\rangle$$

The quantum circuit diagram is shown in figure 3-4. The operation is split into five key steps, described in table 3.3. The greater than operation part of the MQDK is used to flip a control bit if b is bigger than a . One useful observation is that if a and b have different signs, the output $a - b$ will have the same sign as a .

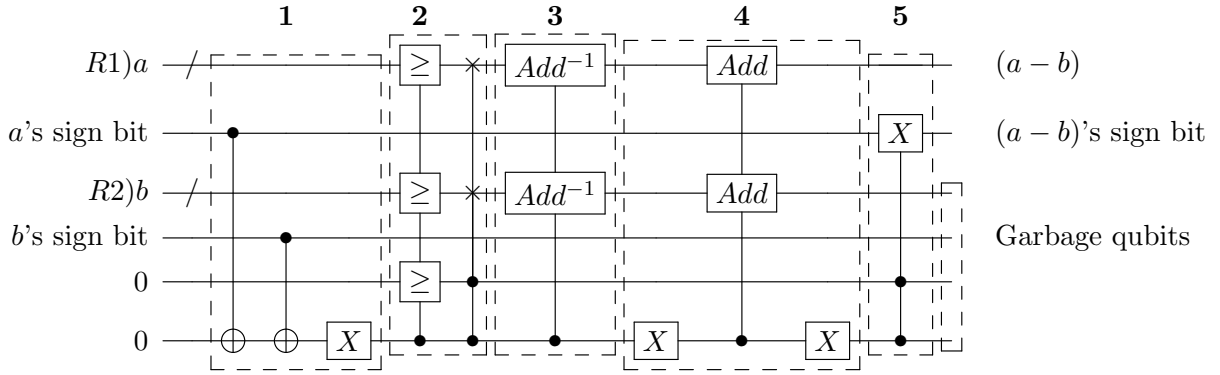


Figure 3-4: Signed Subtract Quantum Circuit

Step	Description
Input	An input of 2 registers and two additional qubits is taken. The two registers are of equal size $L + 1$, storing the signed integers a and b . The two qubits are used as control qubits.
1	The first step is to perform an XOR operation on the two integers signed bits, helping to determine the sign of the output bit. This is done through two consecutive CNOT gates
2	The next step determines whether $b > a$ storing the result in a control qubit. If $b > a$ then the two registers a and b are swapped.
3	The third step is to perform a controlled inverse Add, this is equivalent to a subtract of $R1 - R2$. This is only performed if a and b have the same sign
4	The forth step is to do a controlled add between $R1$ and $R2$. This is only done if a and b have different sign's.
5	The final step is to set the sign of the output using the two control qubits.
Output	An output 1 register containing $a \cdot b$ along with $L + 2$ garbage qubits.

Table 3.3: Signed Subtraction Operation

Two additional qubits are used to control the logic of the signed subtraction, and the output contains a set of garbage qubits that must be stored in order to make the calculation reversible. Having additional garbage qubits that are of no use is not optimal and there likely is a more optimal implementation of a signed subtraction with fewer or no garbage qubits. However, this implementation met the minimum requirements for implementing Shor's algorithm in superposition and met the required space bound so no further optimisation was tried.

3.2 Implementing Components In Superposition

3.2.1 Selecting x in Superposition

The first component of Shor's algorithm that requires altering is the selection of x , this must now be done in superposition, performing the transformation:

$$|N\rangle |0\rangle \rightarrow |N\rangle |x\rangle,$$

where $x = 1 + (R \bmod (N - 1))$ and R is a random integer. The quantum circuit diagram for the implementation is shown in figure 3-5. The process is split into five steps described in table 3.4.

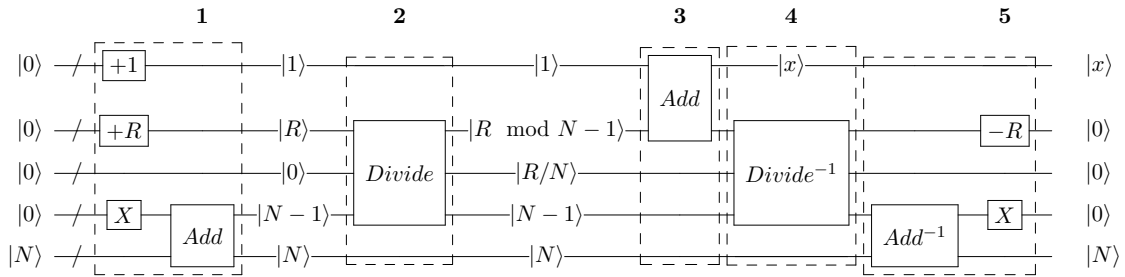


Figure 3-5: Circuit for the selection of x

Step	Description
Input	An input of 2 registers, the first containing N and the second is the target register to store x . A random classical number R is also taken.
1	The first step performs a few minor actions to set up the calculation. $N - 1$ is calculated into an auxiliary register, the classical number R is encoded into a quantum auxiliary register and another auxiliary register is put in state $ 1\rangle$
2	The next step performs a divide operation calculating the remainder $R \bmod N - 1$
3	The third step simply adds the remainder to the auxiliary register in state $ 1\rangle$ producing the state $ 1 + (R \bmod N - 1)\rangle = x\rangle$
4	The forth step is to reverse the divide operation.
5	The final step undoes the initial actions to reset the auxiliary registers.
Output	An output of 2 registers containing N and x .

Table 3.4: Selecting x

3.2.2 Modular Exponentiation in Superposition

The next component that is needed is the square and multiply modular exponentiation, performing the transformation:

$$|a\rangle |b\rangle |c\rangle |0\rangle \rightarrow |a\rangle |b\rangle |c\rangle |a^b \bmod c\rangle.$$

As mentioned above the square and multiply algorithm is handily split into repetitive steps. The implementation of a single square and multiply modular step is first discussed. Then how this step is then used in Bennett's generic conversion to complete a full square and multiply algorithm is discussed.

Square and Multiply Step

Each step takes a single qubit from the exponent b and uses that to determine if the current value is just squared or both squared and multiplied. This produces the transformation:

$$\text{Square and multiply step: } |a\rangle |b_i\rangle |c\rangle |v_i\rangle |0\rangle \rightarrow |a\rangle |b_i\rangle |c\rangle |v_i\rangle |v_{i+1}\rangle,$$

where i is the current step, v_i is the current working value and b_i is the i 'th qubit of b . For example say the current working value was a^2 and $b_i = 1$, this would lead to the transformation:

$$|a\rangle |1\rangle |c\rangle |a^2\rangle |0\rangle \rightarrow |a\rangle |1\rangle |c\rangle |a^2\rangle |(a^2)^2 \cdot a\rangle,$$

where a^2 has been both squared and multiplied since the exponent qubit b_i was in state 1. The quantum circuit used to implement a single square and multiply step is shown in figure 3-6. Each step is split into four parts described in table 3.5. Both the modular square and the modular multiply operations implemented above are used. Since this operation is reversible it can also be used to reverse a step. For example taking the values v_i and v_{i+1} and running the circuit backwards would output the value v_i and an empty register.

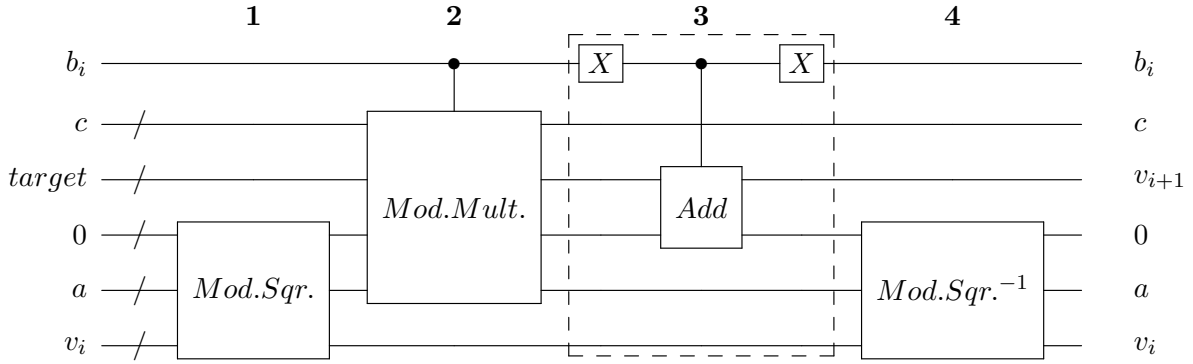


Figure 3-6: Square and Multiply Step

Part	Description
Input	An input of 4 registers and one qubit is taken storing a, c, v_i and the i 'th qubit of b as well as one being an empty target register.
1	Firstly the current working value is squared into an auxiliary register, since squaring happens regardless of whether b_i is 0 or 1
2	Second if b_i is 1 the now squared value must be multiplied by a into the final target register
3	Thirdly if b_i is 0 the value from the auxiliary register is added into the target register. Since part 2 would have not have effected the target register this is analogous to copying the squared value into the target.
4	Finally the part 1 is undone, resetting the auxiliary register back to 0.
Output	An output of 4 registers containing a, c, v_i, v_{i+1} and the qubit containing b_i .

Table 3.5: Square and multiply step

Reversible square and multiply

This Square and Multiply step can now be used to produce a fully reversible Square and multiply implementation using Bennett's generic conversion. Before plugging the square and multiply step into the recursive Pebble functions for Bennett's conversion it is useful to know how many intermediary steps must be stored at one time, or in other words how many free pebbles we require. This allows calculation of how many qubits must be used for the conversion. The total number of square and multiply steps required to complete the computation is simply equal to the number of qubits in $|b\rangle - 1$ that is $\log_2(b) - 1$. The -1 comes from the first step in the general square and multiply algorithm not being

required since it is simply setting the starting value which in our case is given as our input a . Using Bennett's conversion the number of intermediary steps that must be stored at any one time is therefore $\log_2(\log_2(b) - 1)$, meaning we need this many free pebbles. Each stored step will store the current working value at that step v_i in a quantum register. This register needs at most the same number of qubits as in $|c\rangle$. Thus $\log_2(c) \cdot \log_2(\log_2(b) - 1)$ qubits are needed in total to store all the intermediate steps.

Now the number of intermediary steps required is known we can initialise the correct number of quantum registers required, these can be thought of as the free pebbles. A classical array is used to keep track of which registers are storing which step of the calculation and which registers are free. This is like keeping track of which pebble is at which step in the pebble game.

In order to calculate step $i + 1$ the square and multiply step requires the register containing the step i and a free register as input. Likewise in order to reverse step $i + 1$ it must be given the register containing i and $i + 1$.

Thus using the classical tracking array, at every *step()* in the pebble algorithms the correct two registers can be passed. For example, if we are at step i and are looking to calculate $i + 1$, the register containing v_i and an empty register will be used. This is just like having a pebble which is at step i and then placing a pebble at step $i + 1$. Similarly, if we are looking to remove a pebble from the occupied step $i + 1$ the registers containing v_i and v_{i+1} can be selected. The implementation is described in table 3.6.

Part	Description
1	Firstly, the number of auxiliary registers required is calculated as $\log_2(\log_2(b) - 1)$.
2	Next, the required qubit registers and the classical array to track them are initialised.
3	Next, the recursive Pebble function is called, every time the Step function is called inside the Pebble function the following happens: <ol style="list-style-type: none"> 1. The register storing the current step v_i is found, that is the equivalent to finding the pebble on step i. 2. Next if we are conducting a step we find the first empty register to store the next working value, equivalent to finding a free pebble. We then carry out a step using the register storing v_i and the free register as input to the square and multiply step. 3. If instead we are reversing a step we find the register storing v_{i+1}, equivalent to finding the pebble on step $i + 1$. We then reverse a step by using the registers storing v_i and v_{i+1} as input to the reverse square and multiply step. 4. If the target number of steps has been reached the value v_{i+1} is copied into the target register.
4	Finally, the Unpebbling function is called to reset all the auxiliary registers back to empty.

Table 3.6: Square and Multiply Implementation

3.2.3 Continued Fraction in Superposition

The next component that must be implemented is the Continued Fractions algorithm in superposition. This performs the transformation:

$$|N\rangle |\theta\rangle |0\rangle \rightarrow |N\rangle |\theta\rangle |r\rangle,$$

where r is the denominator of the best rational approximation to θ , and N is the bound for the accuracy of the approximation. The implementation uses the recursive algorithm shown in algorithm 2. Each recursive iteration is taken as a step in Bennett's generic conversion. Since we are only interested in the denominator

of the approximation we must only keep track of $r1$ and $r2$ and not $s1$ and $s2$ as in the recursive algorithm. Each step produces the transformation:

$$|n\rangle |d\rangle |r1\rangle |r2\rangle |0\rangle \rightarrow |d\rangle |n \bmod d\rangle |r2\rangle |r1 - (n/d) \cdot r2\rangle |garbage\rangle. \quad (3.1)$$

The implementation of a step is split into two main parts, firstly we must check to see if the best approximation has been found, that is if $d == 0$ or $r2 \geq N$, line 1 of algorithm 2. A control qubit was used to set if the best approximation had been found. The second part was to carry out the appropriate calculations for the split and invert. The circuit diagram producing the transformation in equation 3.1 is given in figure 3-7.

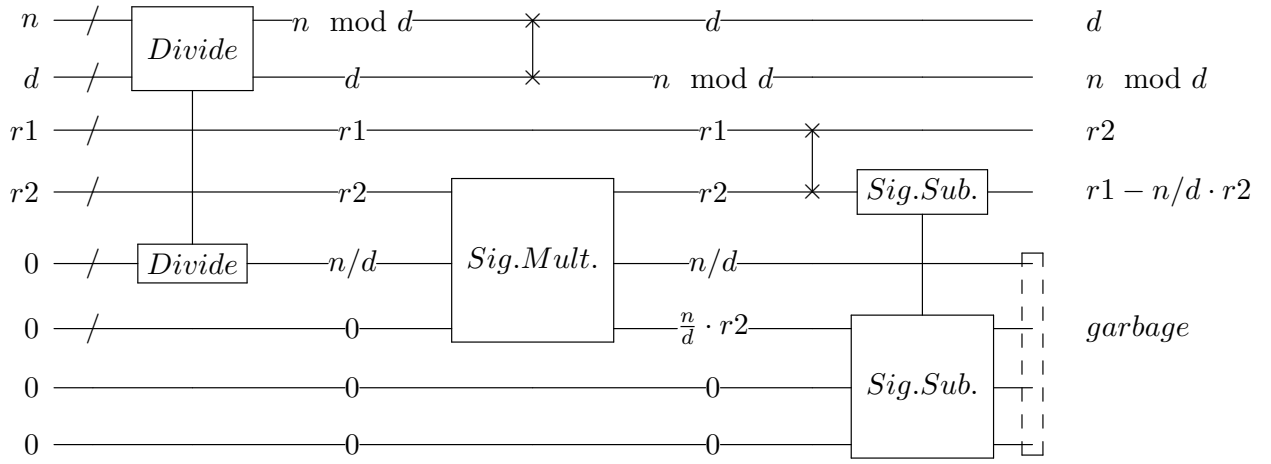


Figure 3-7: Continued Fractions Step

As in the implementation of the square and multiply exponentiation above the Continued fractions step can be used to create a fully reversible Continued Fractions implementation using Bennett's conversion. The required number of steps to complete the continued fraction is at most $1.44L$ where L is the number of bits in θ . Thus the required number of steps that must be stored at one time is $\log(1.44L)$ using Bennett. Since each stored step must contain values for registers $|n\rangle |d\rangle |r1\rangle |r2\rangle |garbage\rangle$ each of the same size as $|M\rangle$ the number of qubits that must be used is equal to $5 \cdot \log_2(N) \cdot \log(1.44L)$.

Again a classical array is used to keep track of which register is storing which step and which is currently free.

The same implementation as was used for the square and multiply reversible implementation was utilised for the continued fractions implementation. The

square and multiply step was simply replaced by first checking if the best approximation had been found followed by the continued fraction step. Once the best approximation had been found it would be added into the target register.

3.2.4 GCD in Superposition

The final step that must be implemented into superposition is finding the greatest common divisor. This was done by utilising the extended Euclidean algorithm. As discussed above this is very convenient since it is nearly identical to the continued fractions algorithm. Finding the greatest common divisor in superposition performs the following transformation:

$$|a\rangle |b\rangle |0\rangle \rightarrow |a\rangle |b\rangle |\text{gcd}(a, b)\rangle,$$

The same continued fractions step can be used in the greatest common divisor algorithm. However, the key difference is that both $s1$ and $r1$ are required to calculate the gcd. We must therefore keep track of all of $s1, s2, r1, r2$ meaning $7 \cdot \log_2(M) \cdot \log(1.44L)$ qubits are required, just to store the reversible values. An additional calculation is also required to calculate the gcd and copy it into the target register.

3.3 Implementing Shor's algorithm In Superposition

Now we have implemented all the necessary components it is possible to perform Shor's algorithm in superposition. This overall circuit is shown in figure 3-8.

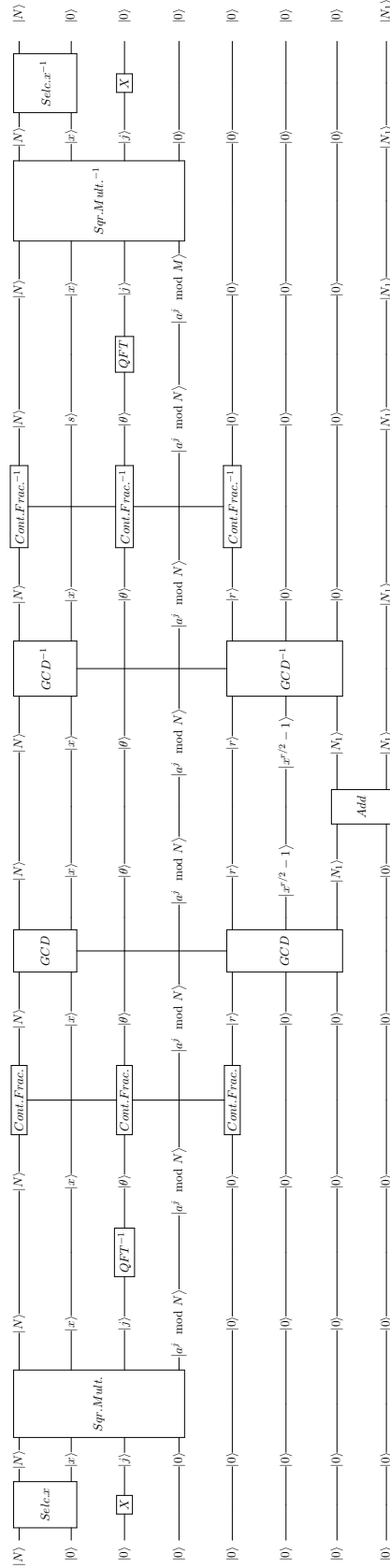


Figure 3-8: Shor's algorithm in superposition

Section 4

Testing

Each new operation was run through a series of tests to ensure their accuracy. This chapter describes the tests carried out on each of the new implementations. All tests conducted use the black-box principle of putting in a certain input and observing that the output is as expected.

Where possible three main series of tests were conducted on each of the operations. Firstly the full state simulator was used. A superposition was input and the quantum state afterwards was viewed to ensure the correct transformation had occurred. If the correct transformation had occurred the inverse operation would be used to reset the qubits. This thus simultaneously checked the reversibility of each operator. Equal superposition of all values possible with the available qubits was tested. For example with the modular square operation with a and b both as 2 bit numbers the output is shown in figure 4-1. The full state simulator could be used to test operations requiring up to 30 qubits.

\overline{a} \overline{m} \overline{t}	\overline{a} \overline{m} $\overline{(a^2 \bmod m)}$
00 00 00	00 00 00
00 01 00	00 01 00
00 10 00	00 10 00
00 11 00	00 11 00
01 00 00	01 00 00
01 01 00	01 01 00
01 10 00	$\xrightarrow{Mod.Sqr.}$ 01 10 01
01 11 00	01 11 01
10 00 00	10 00 00
10 01 00	10 01 00
10 10 00	10 10 00
10 11 00	10 11 01
11 00 00	11 00 00
11 01 00	11 01 00
11 10 00	11 10 01
11 11 00	11 11 00

Figure 4-1: Full state simulation of modular squaring with a and m as 2 bit numbers

The second test employed the Toffoli simulator to exhaustively run larger cases where more than 30 qubits were required. It was necessary to run many iterations to test many different inputs. An automated test was created to put each combination of the inputs through the quantum operation and measure the output. The results were then compared to a classical computation using the same input to check for accuracy. Like the first test the output was put back through the operator to check the operator reversed the operation correctly.

The third test was to investigate very large inputs. Since an exhaustive test soon became unfeasible when inputs beyond a certain point, instead patches of the test space were investigated. The bitsize of the inputs were incremented and then tested, up to a maximum of 64 bits for each input. Testing with numbers larger than 64 bit could not be conducted due to current limitations in the Microsoft's *Q#* language. Particular focus was given to edge case areas, in particular the transition from one bitsize to the next on particular inputs. Again the outputs were used as inputs to the reverse operators to check for reversibility. For each

operation the tests conducted are summarised in a table.

4.1 Modular Square

Test	Simulator	Bitsize of a and b	Number of Tests	Description
1	Full state	1-4	4	Both a and b were put into an equal superposition of all values possible with the available bits.
2	Toffoli	1-9	349524	Exhaustive test of each combination of a and m .
3	Toffoli	10-64	27000	For each bitsize the first and the last 100 combinations were tested, and then 100 evenly space numbers in between were tested.

Table 4.1: Testing of the Modular Square Operator

4.2 Modular Multiply

Test	Simulator	Bitsize of a, b and c	Number of Tests	Description
1	Full state	1-3	3	a , b and m were put into an equal superposition of all values possible with the available bits.
2	Toffoli	1-5	37448	Exhaustive test of each combination of a, b and c .
3	Toffoli	6-64	18328	For each bitsize the first and the last 108 combinations were tested, and then 100 evenly space numbers in between were tested.

Table 4.2: Testing of the Modular Multiply Operator

4.3 Signed Multiply

Test	Simulator	Bitsize of a and b	Number of Tests	Description
1	Full state	1-6	6	a and b were put into an equal superposition of all values possible with the available bits.
2	Toffoli	1-10	699048	Exhaustive test of each combination of a and b .
3	Toffoli	11-64	26500	For each bitsize the first and the last 100 combinations were tested, and then 100 evenly space numbers in between were tested.

Table 4.3: Testing of the Signed Multiply Operator

4.4 Signed Subtract

Test	Simulator	Bitsize of a and b	Number of Tests	Description
1	Full state	1-11	11	a and b were put into an equal superposition of all values possible with the available bits.
2	Toffoli	11-64	26500	For each bitsize the first and the last 100 combinations were tested, and then 100 evenly space numbers in between were tested.

Table 4.4: Testing of the Signed Subtract Operator

4.5 Selection of x

Test	Simulator	Bitsize of N	Number of Tests	Description
1	Full state	1-5	62	N was put into an equal superposition of all values possible with the available bits, and a test was then run for each possible the random integer R could take.
2	Toffoli	1-64	32000	For each bitsize m was tested at 50 equally space values. For each m , R was tested at 10 equally space values.

Table 4.5: Testing of the Generation of x Operator

4.6 Square and Multiply Exponentiation

Test	Simulator	Bitsize of a	Number of Tests	Description
1	Toffoli	2-64	1701	For each bit size, a , the exponent b and the modular value were tested each at 3 different values resulting in $3^3 = 27$ tests for each bit size

Table 4.6: Testing of Square and Multiply Operator

4.7 Continued Fractions

Test	Simulator	Bitsize of θ	Number of Tests	Description
1	Toffoli	1-64	3200	The register bitsize, the irrational number and the continued fraction limit were all varied.

Table 4.7: Testing of Continued Fractions Operator

4.8 Greatest Common Divisor

Test	Simulator	Bitsize of a and b	Number of Tests	Description
1	Toffoli	1-64	3200	The register bitsize of a and b was varied and even space values of a and b where tested for each bitsize.

Table 4.8: Testing of GCD Operator

4.9 Shor's algorithm in superposition

Testing Shor's algorithm in superposition as a whole using the full state simulator is not possible in the MQDK since it uses far more than 30 qubits even for a small input. Unfortunately, it is also not possible to test the whole algorithm using the Toffoli simulator because the quantum Fourier transform puts it outside the scope for the Toffoli simulator. Instead Shor's algorithm in superposition was tested in sections. Looking to figure 3-8 the circuit is split into three sections by the inverse QFT and the standard QFT. The first section contains the selection of x , and the square and multiply modular exponentiation. The second section carries out the continued fractions, the greatest common divisor, the addition into the target register and then the inverse of the GCD and continued fractions. The third section is simply the inverse of the first section. The tests conducted are shown in table 4.9. Although testing of the QFT inline with the algorithm couldn't be conducted it is a very common component of quantum algorithms that has been widely implemented. Meaning we can place confidence it the operation it will conduct. Additionally it also uses no extra qubits, resulting in it not effecting the resource measurements we are interested in.

Section	Bitsize of N	Number of Tests	Description
1	2-64	630	For each bit size 10 evenly space tests were conducted where N was varied.
2	2-64	630	For each bit size 10 evenly space tests were conducted where both N , x and θ were varied.
3	2-64	630	For each bit size 10 evenly space tests were conducted where N was varied.










Table 4.9: Testing of Shor's algorithm in superposition

Section 5









Analysis

The Resource estimator included in the MQDK was used to analyse the number of qubits used for each operation. The resource estimator is able to give a breakdown of where during the operation the qubits are created and removed. Since the resource estimator does not simulate the state of the system it is possible to test Shor's algorithm in superposition as a whole with a breakdown of where each qubit is used. To help better illustrate the ratios of the qubits used for each step inside each operation a bar plot is shown for a given register size. The length of the bar is equal to the number of qubits used.






5.1 Modular Square

Description	Number of qubits	Plot of qubits in use when $L = 10$.
An input of 3 registers all of equal size L is taken, storing the a , the modular b and a target register to take the result.	$3L$	
An auxiliary control qubit is used to control if the modular b is zero	1	
An auxiliary register of size $2L$ is used to store a^2 (step 2 in table 3.1)	$2L$	
Two auxiliary registers are used for the divide step (step 3 in table 3.1), one of size L and the other of size $2L$.	$3L$	
An auxiliary control qubit is used inside the MQDK Divide and Square operations.	1	 
Overall		
Input:	$3L$	
Auxiliary:	$5L + 2$	
Total:	$8L + 2$	






5.2 Modular Multiply

Description	Number of qubits	Plot of qubits in use when $L = 10$
An input of 4 registers all of equal size L is taken, storing the a and b , the modular c and a target register to take the result.	$4L$	
An auxiliary control qubit is used to control if the modular c is zero	1	
An auxiliary register of size $2L$ is used to store $a \cdot b$	$2L$	
Two registers are used for the divide step, one of size L and the other of size $2L$.	$3L$	
An auxiliary control qubit is used inside the MQDK Divide and Multiply operations	1	
Overall		
Input:	$4L$	
Auxiliary:	$5L + 2$	
Total:	$9L + 2$	







5.3 Signed Multiply

Description	Number of qubits	Plot of qubits in use when $L = 10$
An input of 3 registers is taken, 2 of size L storing the a and b , and the other a target register of size $2L - 1$ to take the result.	$4L - 1$	
An auxiliary control qubit is used inside the MQDK Multiply operation	1	
Overall		
Input:	$4L - 1$	
Auxiliary:	1	
Total:	$4L$	

5.4 Signed Subtract










Description	Number of qubits	Plot of qubits in use when $L = 10$
An input of 2 registers and 2 control qubits is taken. The first register stores a and is of size $L + 1$, the second register stores b and is of size L	$2L + 3$	
An auxiliary control qubit is used inside the MQDK Greater Than and Add operations	1	
Overall		
Input:	$2L + 3$	
Auxiliary:	1	
Total:	$4L$	

5.5 Selection of x

Description	Number of qubits	Plot of qubits in use when $L = 10$
An input of 2 registers both of size L is taken. The first contains the supposition of N and the second is the target register for x .	$2L$	
2 auxiliary registers of size L are created to store R and $N - 1$	$2L$	
1 auxiliary register of size L is created to store the result from the divide operation	$1L$	
Overall		
Input:	$2L$	
Auxiliary:	$3L$	
Total:	$5L$	

5.6 Square and Multiply Exponentiation








The number of qubits used in square and multiply modular exponentiation is dependant on the bitsize of a, b and c . An increase in the size of the exponent b leads to more stages in the reversible process and thus more auxiliary registers required, whereas an increase in a or c leads to bigger auxiliary registers being required.

Description	Number of qubits	Plot of qubits in use when $L = K = 10$
An input of 4 registers is taken. Three of these registers are of the same size L storing a , the modular c and the target register to hold the result. The forth register stores the exponent b , which we will say is of size K .	$3L + K$	
Bennett's reversible scheme uses a number of auxiliary registers required for the pebbling process. The number required is equal to $(\log(K - 1) + 1)$ resulting in $L(\log(K - 1) + 1)$ qubits.	$L \log(K - 1) + L$	
4 auxiliary control qubits used to control each step of the square and multiply.	4	
An auxiliary register of size L is temporarily used to store the output from each square and multiply step.	L	
$5L + 2$ auxiliary qubits are temporarily used for both the modular squaring and the modular multiplication operations.	$5L + 2$	
2 auxiliary control qubits are required to control the square and multiply step in reverse	2	
Overall		
Input:	$3L + K$	
Auxiliary:	$7L + L \log(K - 1) + 8$	
Total:	$10L + K + L \log(K - 1) + 8$	











5.7 Continued Fractions

Description	Number of qubits	Plot of qubits in use when $L = 5$
An input of 3 registers is taken all of size L , storing the limit N, θ , and a target register to hold the result.	$3L$	■
An auxiliary control bit is used to control the calculation of $\theta = 0$.	1	■
Bennett's reversible scheme uses a number of auxiliary registers required for the pebbling process. The number of steps stored at one time is equal to $\log(1.44(L + 1))$. For each step $n, d, r1$ and $r2$ are stored along with the garbage registers. n and d are each of length $L + 1$. $r1$ and $r2$ are each of length $L + 2$ and the quotient times the current approximation is of length $2L + 3$. The garbage uses $3L + 7$ qubits. This results in a total of $\log(1.44(L + 1))(7L + 13)$ qubits.	$\log(1.44(L + 1))(7L + 13)$	■ ██████████
3 auxiliary control qubits are temporarily used and then removed to check if the best approximation has been found before every 'split and invert' step,	3	■ ██████████
An auxiliary register of size $L + 3$ is temporarily used for the split and invert step. After the step the register is removed.	$L + 3$	■ ██████████
4 auxiliary control qubits are created to select whether the previous or the current approximation is the desired result, once the best approximation has been found. They are then removed.	4	■ ██████████
Overall		
Input:	$3L$	■
Maximum Auxiliary:	$L + \log(1.44(L + 1))(7L + 13) + 4$	██████████
Total:	$4L + \log(1.44(L + 1))(7L + 13) + 4$	■ ██████████

5.8 Greatest Common Divisor

Description	Number of qubits	Plot of qubits in use when $L = 5$
An input of 3 registers is taken all of size L , storing a and b , and a target register to hold the result.	$3L$	
Bennett's reversible scheme uses a number of auxiliary registers required for the pebbling process. The number of steps stored at one time is equal to $\log(1.44(L+1))$. For each step $n, d, r1, r2, s1$ and $s2$ are stored along with the garbage qubits. n and d are each of length $L+1$. $r1, r2, s1$ and $s2$ are all of length $L+2$. The garbage qubits use $5L+12$ qubits. This results in a total of $\log(1.44(L+1))(11L+22)$ qubits.	$\log(1.44(L+1))(11L+22)$	
An auxiliary register of size $L+3$ is used for the split and invert step	$L+3$	
2 auxiliary registers both of size $2L+3$ and 4 control qubits are used to do the final calculation of the greatest common divisor.	$2(2L+3)+4$	
Overall		
Input:	$3L$	
Maximum Auxiliary:	$4L + \log(1.44(L+1))(11L+22) + 10$	
Total:	$7L + \log(1.44(L+1))(7L+12) + 10$	

5.9 Shor's algorithm in Superposition

Description	Number of qubits	Plot of qubits in use when $L = 4$
An input of 2 registers is taken both of size L , the first containing the superposition N to be factored and the second is an empty target register to store the result	$2L$	
Next the 6 auxiliary registers required to store the output from each stage are created. 4 of the registers are of size L , and the final one is of size $2L+1$, this will store θ .	$7L + 1$	
Next x is generated into one of the auxiliary registers, temporarily using an additional $3L$ auxiliary qubits.	$3L$	
Next square and multiply modular exponentiation is conducted taking the auxiliary register of size $2L + 1$ as the exponent and one of the auxiliary registers of size L to store the result. This operation thus temporarily uses $7L + L \log((2L + 1) - 1) + 8$ auxiliary qubits	$7L + L \log(2L) + 8$	
Following the inverse Fourier transform, continued fractions is conducted. This uses $L + \log(1.44(L + 1))(7L + 12) + 4$ temporary auxiliary qubits	$L + \log(1.44(L + 1))(7L + 12) + 4$	
Next $(a^{r/2} - 1) \bmod m$ is calculated using square and multiply modular exponentiation. This uses $7L + L \log((L - 1) + 8)$ auxiliary qubits	$7L + L \log((L - 1) + 8)$	
Finally the greatest common divisor is found using $4L + \log(1.44(L + 1))(11L + 22) + 10$ auxiliary qubits.	$4L + \log(1.44(L + 1))(11L + 22) + 10$	
Overall		
Input:	$2L$	
Maximum Auxiliary:	$11L + \log(1.44(L + 1))(11L + 22) + 11$	
Total:	$13L + \log(1.44(L + 1))(11L + 22) + 11$	

Section 6

Discussion

Following this analysis the resources required to run Shor's algorithm in superposition are known. This allows a lower bound of the resources required by the BBM algorithm to be determined. Both the parallel and the serial construction of the BBM algorithm repeatedly use Shor's algorithm in superposition to factor N into its divisors, in order to identify if N is a smooth number. Therefore the BBM algorithm needs at least the number of qubits to run Shor's algorithm in superposition, since it must also store the output from each run of Shor's algorithm in superposition. For this discussion we thus take that the minimum number of qubits required to run the BBM algorithm as equal to a single run of Shor's algorithm in superposition.

Using this lower bound for the quantum resources required for the BBM algorithm it is possible to compare to Shor's conventional algorithm. The BBM algorithm only needs to factor numbers of size $\log_2(N)^{2/3}$, whereas Shor's conventional algorithm must factor numbers of size $\log_2(N)$. A plot of the qubits required by each algorithm to factor N is shown in figure 6-1. The lowest resource implementation currently known for Shor's conventional algorithm is used, using just $2\log_2(N) + 3$ qubits.

As can be seen, Shor's algorithm uses less qubits up to a point, beyond which the BBM algorithm uses less qubits. The two algorithms intersect at $\log_2(N) = 0.422055 \cdot 10^6$, meaning for an N with a bit size of $> 0.422055 \cdot 10^6$ the BBM uses less qubits than Shor's algorithm to factor N . However, for an N with a bitsize $< 0.422055 \cdot 10^6$ Shor's algorithm will use less qubits to factor N . Therefore for common 2048-bit RSA keys Shor's algorithm will use less qubits. As can be seen in the more focussed graph in figure 6-1 for a 2048-bit key size the BBM algorithm uses more than double the number of qubits that Shor's algorithm does. This means a larger quantum computer would be required to run the BBM algorithm. Therefore, Shor's algorithm can likely be physically implemented sooner than the BBM algorithm to factor common RSA keys.

Upon implementing Shor's algorithm in superposition it is perhaps not that

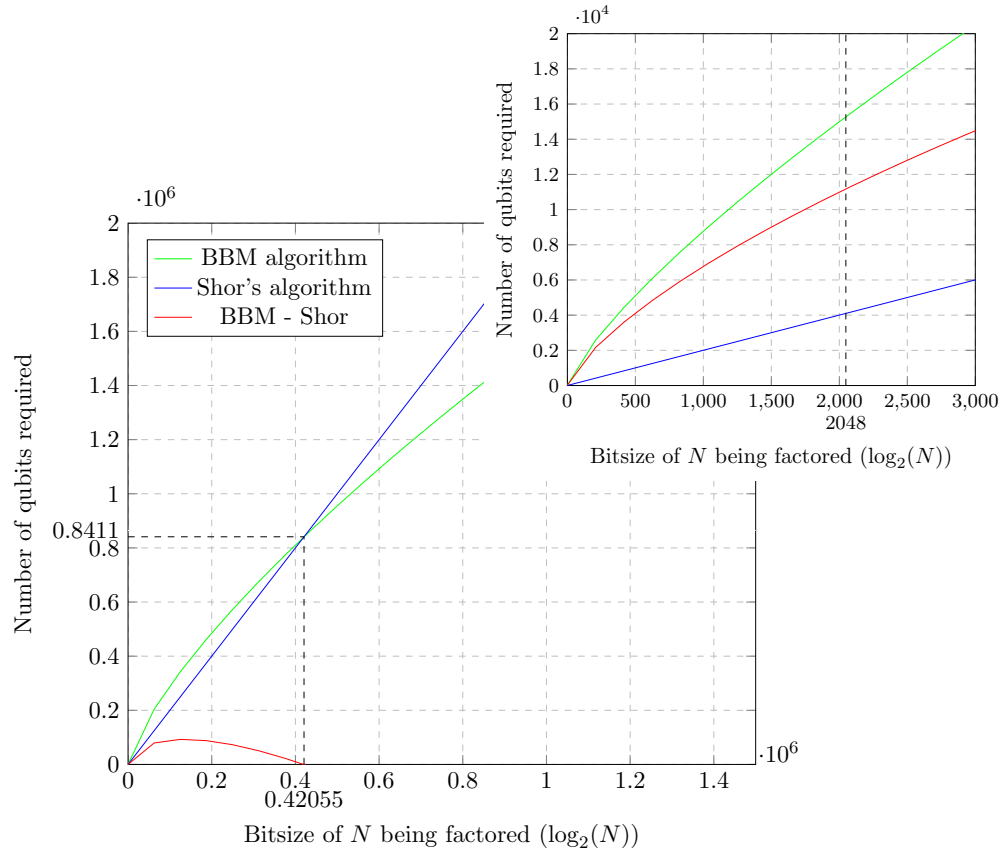


Figure 6-1: Comparison of the quantum resources used by the BBM algorithm and Shor's algorithm.

surprising that the cross over point at which the BBM algorithm uses less qubits than Shor's algorithm is at such large bitsizes. The major bottlenecks are the computations (such as the greatest common divisor) that Shor's algorithm does on a classical computer that must now be done on a quantum computer for the BBM algorithm, this results in two main points. Firstly every value during each of the computations must be stored in a quantum register since they are all in a superposition. Secondly these computations must now be completed reversibly, meaning intermediary values must now be also stored. The combination of these points means a large number of quantum registers is required. For commonly used bit sizes the number of registers required far outweighs the advantage of only having to factor numbers of size $\log_2(N)^{2/3}$ as opposed to $\log_2(N)$. A second effect of having to now conduct these computations reversibly is that this very likely increases the depth of the circuits, further making them more difficult to implement since the qubits must be kept stable for longer.

One important aspect to discuss is that this implementation for Shor's algorithm in superposition and the operations which make it up are very likely not optimal. Care was taken to ensure the space and time bounds that BBM give for each of the steps were met, but beyond that the operators were not optimised. One such potential optimisation that BBM note in their paper is during the selection of x , that the value R need not be stored in a quantum register since it is the same for all values. Another likely improvement is during the calculation of continued fractions and the greatest common divisor. Each intermediary step stored contains values for both the current and the previous approximation, meaning the same value is stored in two consecutive steps. It may be possible to reduce this so the value is only stored once. It is also very likely that the general scheme used for implementing Bennett's conversion is not optimal in time.

Section 7

Conclusion

To conclude, the aim of this report was to implement Shor's algorithm in superposition using Microsoft's Quantum Development Kit and then to compare the quantum resources used by Bernstein, Biassie and Mosca's quantum factoring algorithm with that of the conventional Shor's algorithm. To implement Shor's algorithm in superposition required implementing three new quantum operators to conduct calculations that during the conventional Shor's algorithm are done on a classical computer. These are square and multiply modular exponentiation, continued fractions, and the greatest common divisor. Additionally four new fundamental operators were implemented, these were modular squaring, modular multiplication, signed subtraction and signed multiplication. Combining these operations allowed Shor's algorithm in superposition to be implemented, and the resources it takes analysed. Finally taking the minimum resources required for the BBM quantum factoring algorithm as equal to the resources required by one run of Shor's algorithm in superposition, the BBM algorithm was compared to Shor's algorithm. It was found that Shor's conventional algorithm used less qubits than the BBM algorithm for factoring numbers N with bitsize $< 0.42 \cdot 10^6$. For numbers N with bitsize $> 0.42 \cdot 10^6$ the BBM algorithm required less qubits than Shor's algorithm. It was therefore concluded that for the common RSA key size of 2048 Shor's algorithm required a smaller quantum computer, and could thus be physically implemented sooner.

Bibliography

- [Aar05] S. Aaronson. Quantum computing and hidden variables. *Physical Review A*, 71(3), Mar 2005.
- [Ali07] P. Aliferis. *Level Reduction and the Quantum Threshold Theorem*. PhD thesis, California Institute Of Technology, 2007.
- [AMG⁺17] M. Amy, O. Matteo, V. Gheorghiu, M. Mosca and A. Parent, and J. Schanck. Estimating the cost of generic quantum pre-image attacks on sha-2 and sha-3. In *Selected Areas in Cryptography – SAC 2016*, pages 317–337, Cham, 2017. Springer International Publishing.
- [BBM17] D. J. Bernstein, J. Biasse, and M. Mosca. A low-resource quantum factoring algorithm. *Post-Quantum Cryptography Lecture Notes in Computer Science*, page 330–346, 2017.
- [Bea03] S. Beauregard. Circuit for shor’s algorithm using $2n+3$ qubits. *Quantum Info. Comput.*, 3(2):175–185, March 2003.
- [Ben73] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.
- [Ben89] C. H. Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.
- [BLP93] J. P. Buhler, H. W. Lenstra, and Carl Pomerance. Factoring integers with the number field sieve. *Lecture Notes in Mathematics The development of the number field sieve*, page 50–94, 1993.
- [BR96] H. Boender and H. J. J. te Riele. Factoring Integers with Large-Prime Variations of the Quadratic Sieve. *Experimental Mathematics*, 5(4):257–273, 1996.
- [Bri98] M. Briggs. An Introduction to the General Number Field Sieve. *VTechWorks Home*, Apr 1998.
- [CBS⁺19] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3), Sep 2019.

- [CJL⁺16] L. Chen, S. Jordan, Y. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography, 2016.
- [Cop93] D Coppersmith. Modifications to the number field sieve. *Journal of Cryptology*, 6(3), 1993.
- [Dav99] H. Davenport. *The higher arithmetic*. Cambridge University Press, 1999.
- [DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
- [DJ92] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proc Royal Society London*, A(439):553–558, 1992.
- [DL95] B. Dodson and A. K. Lenstra. NFS with Four Large Primes: An Explosive Experiment. *Advances in Cryptology — CRYPTO’ 95 Lecture Notes in Computer Science*, page 372–385, 1995.
- [ea19] F. Arute et al. Quantum supremacy using a programmable superconducting processor. *Nature* 574, pages 505–510, 2019.
- [FDH04] A. G. Fowler, S. J. Devitt, and L. C. L. Hollenberg. Implementation of shor’s algorithm on a linear nearest neighbour qubit array. *Quantum Info. Comput.*, 4(4):237–251, July 2004.
- [Fey82] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7), 1982.
- [Gro96] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC ’96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.
- [HRS17] T. Häner, M. Roetteler, and K. M. Svore. Factoring Using $2n + 2$ Qubits with Toffoli Based Modular Multiplication. *Quantum Info. Comput.*, 17(7–8):673–684, June 2017.
- [Hui] J. Hui. Quantum computing series. https://medium.com/@jonathan_hui/qc-quantum-computing-series-10ddd7977abd. Accessed: 2020-06-30.

- [Jea20] P. Jurcevic and A. Javadi-Abhari et al. Demonstration of quantum volume 64 on a superconducting quantum computing system. <https://arxiv.org/abs/2008.08571>, 2020.
- [K.18] Gil K. Three puzzles on mathematics, computation, and games. <https://arxiv.org/abs/1801.02602>, 2018. Accessed: 2020-09-09.
- [Ker] N. Kershaw. Get started with the quantum development kit (qdk) - microsoft quantum. <https://docs.microsoft.com/en-us/quantum/quickstarts/get-started>. Accessed: 2020-02-30.
- [LLMP93] A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and J. M. Pollard. The number field sieve. *Lecture Notes in Mathematics The development of the number field sieve*, page 11–42, 1993.
- [LM94] A. K. Lenstra and M. S. Manasse. Factoring with two large primes. *Mathematics of Computation*, 63(208):785–785, 1994.
- [LS90] R. Y. Levine and A. T. Sherman. A note on bennett’s time-space tradeoff for reversible computation. *SIAM Journal on Computing*, 19(4):673–677, 1990.
- [LTV98] M. Li, J. Tromp, and P. Vitányi. Reversible simulation of irreversible computation. *Physica D: Nonlinear Phenomena*, 120(1-2):168–176, Sep 1998.
- [NC19] M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2019.
- [PE96] C. Pomerance and P. Erdős. A tale of two sieves. *NOTICES AMER. MATH. SOC*, 43:1473–1485, 1996.
- [Pom94] C. Pomerance. The number field sieve. *Mathematics of Computation 1943–1993: A Half-Century of Computational Mathematics Proceedings of Symposia in Applied Mathematics*, page 465–480, 1994.
- [RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems, Jan 1978.
- [Sei01] J. Seifert. Using Fewer Qubits in Shor’s Factorization Algorithm via Simultaneous Diophantine Approximation. *Topics in Cryptology — CT-RSA 2001 Lecture Notes in Computer Science*, page 319–327, 2001.

- [SGT⁺18] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q# enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, 2018.
- [Sho99] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.
- [SP19] D. R. Stinson and M. B. Paterson. *Cryptography: theory and practice*. CRC press, 2019.
- [SR20] J. Sevilla and C. J. Riedel. Forecasting timelines of quantum computing. <https://arxiv.org/abs/2009.05045>, 2020.
- [Ste08] P. Stevenhagen. The number field sieve. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography*, 44:83–100, 2008.
- [Tof80] T. Toffoli. Reversible computing. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 632–644, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.
- [VSB⁺01] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang. Experimental realization of shor’s quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414(6866):883–887, Dec 2001.
- [Zal98] C. Zalka. Fast versions of shor’s quantum factoring algorithm. *arXiv: Quantum Physics*, 1998.

Appendices

I 12 Points of Ethics



Department of Computer Science
12-Point Ethics Checklist for UG and MSc Projects

Student Matthew Thorne

**Academic Year
or Project Title** 2019/2020

Supervisor Professor Davenport

Does your project involve people for the collection of data other than you and your supervisor(s)?

YES / **NO**

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Have you prepared a briefing script for volunteers?* YES / NO

Briefing means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?* YES / NO

All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. *Is there any intentional deception of the participants?* YES / NO

Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. *Will participants be de-briefed?* YES / NO

The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. This phase might wait until after the study is completed where this is necessary to protect the integrity of the study.

5. ***Will participants voluntarily give informed consent?*** YES / NO
Participants MUST consent before taking part in the study, informed by the briefing sheet. Participants should give their consent explicitly and in a form that is persistent –e.g. signing a form or sending an email. Signed consent forms should be kept by the supervisor after the study is complete.
6. ***Will the participants be exposed to any risks greater than those encountered in their normal work life (e.g., through the use of non-standard equipment)?*** YES / NO
Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life.
7. ***Are you offering any incentive to the participants?*** YES / NO
The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.
8. ***Are you in a position of authority or influence over any of your participants?*** YES / NO
A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.
9. ***Are any of your participants under the age of 16?*** YES / NO
Parental consent is required for participants under the age of 16.
10. ***Do any of your participants have an impairment that will limit Their understanding or communication?*** YES / NO
Additional consent is required for participants with impairments.
11. ***Will the participants be informed of your contact details?*** YES / NO
All participants must be able to contact the investigator after the investigation. They should be given the details of the Supervisor as part of the debriefing.
12. ***Do you have a data management plan for all recorded data?*** YES / NO
All participant data (hard copy and soft copy) should be stored securely, and in anonymous form, on university servers (not the cloud). If the study is part of a larger study, there should be a data management plan.