Quantum Computing and Shor's Algorithm

Matthew Hayward

Original Spring 1999, Revised 2015

Contents

1 Preface

This paper is intended to be a beginners introduction to the field of quantum computing. As I began to study this topic at the University of Illinois under the supervision of DrṘoy Campbell in 1999, there was a lack of introductory material to the topic. This paper will hopefully serve as an introduction to the rudiments of quantum computing and the specifics of Shor's algorithm for factoring large numbers.

In order to get the most out of this paper, readers should be familiar with the following topics:

- Binary representation of numbers
- Complex numbers
- Vector mathematics

That being said there are appendices covering the bare essentials of those topics that are needed to fully comprehend this paper.

The field of quantum computing has its own vocabulary, most of the novel terms used in this paper are described in the glossary at the end.

2 Introduction

Astounding advances have been made in the manufacture of computers between 1950 and 1999. The number of atoms needed to represent a bit in memory has been decreasing exponentially since 1950. Likewise the number of transistors per chip, clock speed, and energy dissipated per logical operation have all followed their own improving exponential trends. Despite these fantastic advances, the manner in which all computers function has been essentially identical: combinations of a few simple logical operations that process some number of operations per unit time. This rate of improvement in transistor size cannot be sustained much longer, at the current rate as of 199 in the year 2020 one bit of information will requite only one atom to represent it. The problem is that at that level of miniaturization the behavior of the components of a computer will become dominated by the principles of quantum physics. (Williams, Clearwater)

With the size of components in classical computers shrinking to where the behavior of the components may soon be dominated more by quantum physics than classical physics researchers have begun investigating the potential of these quantum behaviors for computation. Surprisingly it seems that a computer whose components are able to function in a quantum are more efficient than any classical computer can be.

It is the physical limitations of the classical computer, and the possibilities for the quantum computer to perform certain useful tasks more rapidly than any classical computer which drive the study of quantum computing.

3 The Classical Computer

3.1 Turing Machines

One of the people most directly responsible for the current concept of computing machines is Alan Turing. Other early giants in the field of Computer Science include Kurt Godel, Emil Post, and Alonzo Church.

Turing and others proposed mathematical models for computing which allowed for the study of algorithms and in absence of any particular computer hardware. These abstraction have proved invaluable in the field of computer science, allowing for general results about algorithms and their run time complexity to be obtained without having to consider a variety of potential computation engines on which those algorithms would run. Turing's model is called a Turing machine. The design of the Turing machine is the following: The machine consists of an infinite tape divided into cells, each cell can contain a 1, a 0, or a blank. There is also a head which can move about the tape, read a cell, write to a cell, change its internal state, or end computation. It is the internal state of this head that is the program which decides what action the head will take next. (Steane)

As simple as this model appears, it encompasses all the functionality of today's most modern supercomputer. That is to say, given the time and memory, there is not a single computation that can be performed on a modern supercomputer that can not be performed on a Turing machine. The immense value of the Turing machine is that when examining what sort of functions can be computed by a computer, one need only to examine a Turing machine, and not one of the millions of potential computing devices to determine if a computation is possible. If a Turing machine can perform the computation, then it is computable. If a Turing machine can not, then the function can not be computed by any classical computer. (Shor)

3.2 Church-Turing Thesis

This bold claim, that any computer is essentially equivalent to a Turing machine grew out of contemporaneous work by Alonzo Church and Alan Turing, and is variously referred to as Church's Thesis, the Church-Turing Thesis, the Turing-Church thesis, the Church-Turing conjecture, and Turing's thesis. There are several equivalent formulations of the thesis stated at different times and to different degrees of rigor, but the central idea is that:

> Every effectively calculable function can be produced by a Turing machine.

The claim here is that anything which one would like to compute, which one reasonably could compute (for instance given pen and paper, and enough time), are precisely those things which a Turing machine can compute.

The Church Turing thesis is perhaps best understood as a **definition** of the types of functions that are calculable in the real world - not as a theorem to be proven. As evidence for the suitability of this as a definition, multiple (indeed every one considered to far) distinct models of computation have been shown to be equivalent to the Turing model with regard to what functions they can compute.

Beyond the mere ability of a computing device to calculate a function is the consideration of how long it would take to do so. This notion is generalized by the time complexity of a given algorithm, which considers the number of steps or operations an algorithm would have to perform relative to the length of the input in order to compute the function.

There are various extensions of the Church-Turing thesis that address the relative efficiency of different models of computation, such as the **Complexity-Theoretic Church-Turing Thesis** states that:

> A Turing machine can efficiently simulate any realistic model of computation.

Here *efficiently* means a Turing machine can simulate any realistic model of computation with at most polynomial time expansion of the computation time or other resources (such as memory). This hypothesis may be viewed either as a definition of what a realistic model of computation is, or as a statement of fact (which may turn out to be false) about realistic models of computation. There are no known models of computation which are thought to be realizable that violate this thesis. The discovery of a model which did violate this thesis would likely be an astonishing breakthrough in the fields of computational complexity, computer science, and/or physics.

These theses gives us insight into the power of computing machines. If a computer theoretically can compute all the functions a Turing machine can compute (given enough memory and time) then it is as powerful as a Turing machine.

3.3 Running Time and Complexity Classes

Turing produced the Turing machine in an effort to answer the question "Is there a mechanical procedure by which the truth or falsity of any mathematical conjecture can be decided?" Turing reasoned that if there was such a method, and if it were truly mechanical in nature and requiring no creative effort, then it could be performed by a Turing machine. The proof offered by Turing that there is no such procedure is the same as the proof of Turing's Halting problem, which states that no Turing machine H can in general be given any arbitrary Turing machine X and its input (I_x) as its own (H's) input, and determine whether X will run to completion on I_x. The demonstration by Turing of a problem which can not be solved by a Turing machine suggests the division of functions into two groups: those that can be computed by a Turing machine and those that can not.

By the Church-Turing thesis we can generally say that functions that are not Turing computable are simply "not computable." It should be stressed at this point that this classification of a function that is not Turing computable as not computable in a general sense is due to the **definitional** aspect of the Church-Turing thesis. It is a logical possibility that some future computing machine will indeed be able to compute some function that a Turing machine can not - however no such machine is known, and the development of such a machine would be a large breakthrough. Arguments can be made that no such machine would be physically possible - however this is not a settled question.

Determining if a function is computable at all is interesting, but this distinction is not fine enough to determine if a function can realistically be solved by a physical computer in a reasonable amount of time. If a computation will take billions of years to compute, it may as well be uncomputable from the perspective of the pragmatist. A algorithm can be characterized by the number of operations and amount of memory it requires to compute an answer given an input of size n. These characterizations of the algorithm determine what is called the algorithm's *running time* or *time complexity*. Specifically, the time complexity of an algorithm to compute a function is determined by looking at how the number of operations of the algorithm scale with the size of the input of the function it computes. We'll take the size of the input to be the number of bits needed to represent that number in binary.

An algorithm which computes its output for an input of size n using resources (computational steps, memory, etc.) that is bounded above by a polynomial function of n are said to be of *polynomial time*. For example if an algorithm with an input size of 10 bits took $10^4 + 7 * 10^2 + 1001$ operations to compute its answer it would be considered a polynomial time algorithm.

Problems which can be solved in polynomial time or less are generally deemed to be *tractable*. An algorithm which solves a problem in polynomial time may be referred to as *efficient*. Problems which require more than polynomial time are usually considered to be *intractable*, for example an algorithm which would require 2^n operations for an input size of n would be considered intractable, as the number of operations grows exponentially with the input size, not polynomially. Generally tractable problems are considered to be those which may practically be solved in the real world. (Shor)

In complexity theory, problems are often restated in terms of a *decision problem* - this means that the function of interest takes in its input and produces a yes or no answer.

Computer Scientists have grouped problems into a variety of complexity classes, below are some of the more well known.

P : The class of decision problems which can be answered in polynomial time.

NP : Nondeterministic polynomial time, the class of decision problems where a candidate for an answer can be verified as a correct answer or not in polynomial time.

NP-complete : The "hardest" problems in NP, this set has the interesting property that if any NP-complete problem can be solved in polynomial time, then P and NP are in fact the same. Whether P equals NP is an outstanding problem in computer science and complexity theory.

(Cormen, Leiserson, Rivest)

Given these distinctions, to determine if an function may be *efficiently* computed by a classical computer there are two questions that must be answered. First, can a Turing machine compute the function, if so then the function is called computable. Second the time complexity of the algorithm to be used must by considered. In general if an algorithm requires polynomial time or less to compute it is considered to be tractable, and if not it is considered to be intractable.

Accordingly, interest in quantum computing is twofold. First it is of interest if a quantum computer can compute functions which are uncomputable on a classical computer, and second it is of interest whether an algorithm which is intractable on a Turing machine may be tractable on a quantum computer.

4 The Quantum Computer

4.1 Quantum Physics

When considering the possible efficiency of a computer which makes use of the results of quantum physics, it is helpful to know a little of quantum physics itself. Quantum physics arose from the failure of classical physics to offer correct predictions on the behavior of photons and other elementary particles. Quantum physics has since been under intense scrutiny, as some of its predictions seem very strange indeed. Nevertheless experiments verify the same strange behavior which leads skeptics to challenge the veracity of Quantum physics.

4.2 The Classical Bit

To understand the ways in which a quantum computer is different from a classical computer you must first understand the rudiments of the classical computer. The most fundamental building block of a classical computer is the bit. A bit is capable of storing one piece of information, it can have a value of either 0 or 1. Any amount of information can be encoded into a list of bits. In a classical computer a bit is typically stored in a silicone chip, or on a metal hard drive platter, or on a magnetic tape. About 10^{10} atoms were used to represent one bit of information in 1999. The smallest conceivable storage for a bit involves a single elementary particle of some sort. For example, for any particle with a spin-1/2 characteristic (such as a proton, neutron, or electron), it's spin-1/2 characteristic on measurement will be either $+1/2$ or $-1/2$. We can thus encode a bit using a single particle by mapping 1 to be spin $+1/2$ and 0 to be $-1/2$, assuming we can measure and manipulate the spin of such a particle. If we were to try to use this spin-1/2 particle as a classical bit, one that is always in the 0 or 1 state, we would fail. We would be trying to apply classical physics on a scale where it simply is not applicable. This single spin-1/2 particle will instead act in a quantum manner. (Williams, Clearwater)

This spin-1/2 particle which behaves in a quantum manner could be the fundamental building block of a Quantum computer. We could call it a qubit, to denote that it is analogous in some ways to a bit in a classical computer. Just as a memory register in a classical computer is an array of bits, a quantum memory register is composed of several spin-1/2 particles, or qubits. There is no particular need for the spin-1/2 particle, equally well we could use a Hydrogen atom, and designate its electron being measured in the ground state to be the 0 state, and it being in the first excited state to be the 1 state. There are a multitude of possible physical qubit representations that could work. For simplicity I will discus only the spin-1/2 particle from here on, but analogous arguments could be made for many things.

4.3 State Vectors and Dirac Notation

We wish to know exactly how the behavior of the spin-1/2 particle, our qubit, differs from a that of a classical bit. Recall that a classical bit can store either a 1 or a 0, and when measured the value observed will always be the value stored. Quantum physics states that when we measure the spin-1/2 particles state we will determine that it is in the $+1/2$ state, or the spin $-1/2$ state. In this manner our qubit is not different from a classical bit, for it can be measured to be in the $+1/2$, or 1 state, or the $-1/2$, or 0 state. The differences between the qubit and the bit come from what sort of information a qubit can store when it is not being measured.

According to quantum physics we may describe the state of this spin-1/2 particle by a state vector in a Hilbert Space.

A Hilbert Space is a linear, complex vector space.

In a linear vector space you may add and multiply vectors that lie within the space together and the resulting vector will still lie within that space. For example you are probably familiar with the x, y, z coordinate system where the x, y, and z axes are mutually perpendicular real number lines, which coincide at the point $x = 0$, $y = 0$, $z = 0$ - this is an example of a linear vector space.

In a complex vector space the lengths of the vectors within the space are described with complex numbers. Complex numbers are numbers which take the form $a + i * b$, where a and b are real numbers, and i is defined to be the square root of negative one. (Williams, Clearwater)

In the Hilbert Space for our state vector, which describes the state of our spin-1/2 particle, these perpendicular axes will correspond to each possible state that the system can be measured in. So our Hilbert Space for a single qubit will have two perpendicular axes, one corresponding to the spin-1/2 particle being in the $+1/2$ state, and the other to the particle being in the $-1/2$ state. These states which the vector can be measured to be are referred to as *eigenstates.* The vector which exists somewhere in this space which represents the state of our spin-1/2 particle is called the *state vector.* The projection of the state vector onto one of the axes shows the contribution of that axes' eigenstate to the whole state. This means that in general, the state of the spin-1/2 particle can be any combination of the base states. In this manner a qubit is totally unlike a bit, for a bit can exist in only the 0 or 1 state, but the qubit can exist, in principle, in any combination of the 0 and 1 state, and is only constrained to be in the 0 or 1 state when we measure the state.

Now I will introduce the standard notation for state vectors in Quantum physics. The state vector is written with the following way and called a *ket vector* $|\psi\rangle$. Where ψ is a list of numbers which contain information about the projection of the state vector onto its base states. The term ket and this notation come from the physicist Paul Dirac who wanted a concise shorthand for writing formulas that occur in Quantum physics. These formulas frequently took the form of the product of a row vector with a column vector. Thus he referred to row vectors as *bra vectors* represented as $\langle y|$. The product of a bra and a ket vector would be written $\langle y|x\rangle$, and would be referred to as a *bracket.* (Williams, Clearwater)

There is nothing special about this vector notation, you may think of any state vector being written as a letter with a line over it, or as a bold letter as vectors are normally denoted. If you do further reading into this area you will almost assuredly come across the bra and ket notation, which is why it is presented.

4.4 Superposition and Eigenstates

Earlier I said that the projection of the state vector onto one of the perpendicular axes of its Hilbert Space shows the contribution of that axes' eigenstate to the whole state. You may wonder what is meant by the *whole state.* You would think (and rightly so, according to classical physics) that our spin-1/2 particle could only exist entirely in one of the possible $+1/2$ and $-1/2$ states, and accordingly that its state vector could only exist lying completely along one of its coordinate axes. If the particle's axes are called x and y, and the state vector's x coordinate which denotes the contribution to the $-1/2$, or 0 state, and y coordinate which denotes the contribution to the $+1/2$, or 1 state, should only be (1,0), or (0,1).

That seems perfectly reasonable, but it simply is not correct. According to quantum physics a quantum system can exist in a mix of all of its allowed states simultaneously. This is the Principle of Superposition, and it is key to the power of the quantum computer. While the physics of superposition is not simple at all, mathematically it is not difficult to characterize this kind of behavior.

4.5 The Quantum qubit

Back to our qubit, our spin-1/2 particle. Now that we know that while it can only be measured to have a spin of $+1/2$ or $-1/2$, it may in general be in a superposition of these states when we are not measuring it. We could refer to its state in the following manner.

Let x_1 be the eigenstate corresponding to the spin $+1/2$ state, and let x_0 be the eigenstate corresponding to the spin $-1/2$ state. Let X be the total state of our state vector, and let w_1 and w_0 be the complex numbers that weight the contribution of the base states to our total state, then in general:

$$|X\rangle = w_0 * |x_0\rangle + w_1 * |x_1\rangle \equiv (w_0, w_1)$$

At this point it should be remembered that w_0 and w_1, the weighting factors of the base states are complex numbers, and that when the state of X is measured, we are guaranteed to find it to be in either the state:

$$0 * |x_0\rangle + w_1 * |x_1\rangle \equiv (0, w_1)$$

or the state

$$w_0 * |x_0\rangle + 0 * |x_1\rangle \equiv (w_0, 0)$$

This is analogous to a system you may be more familiar with, a vector with real weighting factors in the a two dimensional plane. Let the base states for this two dimensional plane be the unit vectors x, and y. In this case we know that the state of any vector V can be described in the following manner:

$$V = x_0 * x + y_0 * y \equiv (x_0, y_0)$$

Our state vector is a unit vector in a Hilbert space, which is similar to vector spaces you may be more familiar with, but it differs in that the lengths of the vectors are complex numbers. It is not necessary from a physics perspective for the state vector to be a unit vector (by which I mean it has a length of 1), but it makes for easier calculations further on, so I will assume from here on out that the state vector has length 1. This assumption does not invalidate any claims about the behavior of the state vector. To see how to convert a state vector of any length to length 1 see appendix A.

With all this in mind we have fully defined the basic building block of our quantum computer, the qubit. It is fundamentally different from our classical bit in that it can exist in any superposition of the 0 and 1 states when it is not being measured. (Barenco, Ekert, Sanpera, Machiavello)

4.6 The Quantum Memory Register

Up till now we have considered a two state quantum system, specifically our spin-1/2 particle. However a quantum system is by no means constrained to be a two state system. Much of the above discussion for a 2 state quantum system is applicable to a general n state quantum system.

In an n state system our Hilbert Space has n perpendicular axes, or eigenstates, which represent the possible states that the system can be measured in. As with the two state system, when we measure our n state quantum system, we will always find it to be in exactly one of the n states, and not a superposition of the n states. The system is still allowed to exist in any superposition of the n states while it is not being measured.

Mathematically if two state quantum system with coordinate axes x_0, x_1 can be fully described by:

$$|X\rangle = w_0 * |x_0\rangle + w_1 * |x_1\rangle \equiv (w_0, w_1)$$

Then an n state quantum system with coordinate axes $x_0, x_1, \ldots, x_{n-1}$ can be fully described by:

$$|X\rangle = \sum_{k=0}^{n-1} w_k * |x_k\rangle$$

In general a quantum system with n base states can be represented by the n complex numbers w_0 to w_{n-1}. When this is done the state may be written as:

$$|X\rangle = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{n-1} \end{pmatrix}$$

Where it is understood that w_k refers to the complex weighting factor for the k'th eigenstate.

Using this information we can construct a quantum memory register out of the qubits described in the previous section. We may store any number n in our quantum memory register as long as we have enough qubits, just as we may store any number n in a classical register as long as we have enough classical bits to represent that number. The state of the quantum register with n states is give by the formula above. Note that in general a quantum register composed of n qubits requires 2^n complex numbers to completely describe its state. A n qubit register can be measured to be in one of 2^n states, and each state requires one complex number to represent the projection of that total state onto that state. In contrast a classical register composed of n bits requires only n integers to fully describe its state.

This means that one can store an exponential amount of information in a quantum register relative to the size of that register. Here we see some of the first hints that a quantum computer could be exponentially more efficient than a classical computer in some respects. Recall that from our discussion of complexity that problems which can be solved in polynomial time are generally thought of as being tractable, and that problems which can be solved in exponential time are thought of as intractable. If a quantum computer can be is exponentially more efficient than a classical computer for some algorithms, then some problems currently intractable may become tractable! This is a large part of the motivation for the study of quantum computing.

4.7 Probability Interpretation

Now that we know how to represent our state vector as a superposition of states, and we know that we can only measure the state vector to be in one of the base states, it would seem that there would be some sort of discrepancy. We must determine what happens when we measure the state vector. We know from quantum physics that given an initial condition the state vector will evolve in time in accordance with Schrödenger's equation:

$$ih\frac{\partial|X(t)\rangle}{\partial t} = H(t)|X(t)\rangle$$

Where i is the square root of negative one, h is $1.0545 * 10^{-34}$ Js, and H is the Hamiltonian operator, which is determined by the physical characteristics of the system being evolved.

In our notation this expression is:

$$ih\frac{\partial w_i(t)}{\partial t} = \sum_j H(t)_{ij} * w_j(t)$$

This evolution would appear to be continuous, but these equations only apply to a quantum mechanical system evolving in isolation from the environment. The only way to observe the state of the state vector is to in some way cause the quantum mechanical system to interact with the environment. When the state vector is observed it makes a sudden discontinuous jump to one of the eigenstates, but this is not a violation of Schrödenger's equation. When the quantum mechanical system interacts with the outside environment and measured the state vector is said to have collapsed. (Williams, Clearwater)

Now understanding that a state vector will collapse when it interacts with the external environment, we still need to know in what manner this collapse happens. To perform any sort of useful calculation we must be able so say something about which base state a quantum mechanical system will collapse into. The probability that the state vector will collapse into the j'th eigenstate, is given by $|w_j|^2$ which is defined to be $a_j^2 + b_j^2$ if $w_j = a_j + i * b_j$, where w_j is the complex projection of the state vector onto the j'th eigenstate. In general the chance of choosing any given state is

$$Prob(j) = \frac{|w_j|^2}{\sum_{k=0}^{n-1} |w_k|^2}$$

but as mentioned earlier we will insist on having the state vector of length one, and in this case the probability expression simplifies to $Prob(j) = |w_j|^2$.

So now we know how to construct an n state quantum system, which can be placed in an arbitrary superposition of states. We also know how to measure the resultant superposition and get a base state with a certain probability. This is all that we need to understand about our quantum memory register to be able to simulate its behavior.

4.8 Quantum Parallelism

Given that our quantum memory register differs from a classical one in that it can store a superposition of the base states of the register, one might wonder what this implies as to the efficiency of quantum computing. The study of quantum computing is relatively new, most give credit to Richard Feynman for being the first to suggest that there were tasks that a quantum computer could perform exponentially better than a classical computer. Feynman observed that a classical computer could not simulate a quantum mechanical system without suffering from exponential slowdown. At the same time he hinted that perhaps by using a device whose behavior was inherently quantum in nature one could simulate such a system without this exponential slowdown. (Feynman)

Several quantum algorithms rely on something called quantum parallelism. Quantum parallelism arises from the ability of a quantum memory register to exist in a superposition of base states. A quantum memory register can exist in a superposition of states, each component of this superposition may be thought of as a single argument to a function. A function performed on the register in a superposition of states is thus performed on each of the components of the superposition, but this function is only applied one time. Since the number of possible states is 2^n where n is the number of qubits in the quantum register, you can perform in one operation on a quantum computer what would take an exponential number of operations on a classical computer. This is fantastic, but the more superposed states that exist in you register, the smaller the probability that you will measure any particular one will become.

As an example suppose that you are using a quantum computer to calculate the function $\mathcal{F}(x) = 2 * x \bmod 7$, for x integers between 0 and 7 inclusive. You could prepare a quantum register that was in a equally weighted superposition of the states 0-7. Then you could perform the $2 * x \bmod 7$ operation once, and the register would contain the equally weighted superposition of 1,2,4,6,1,3,5,0 states, these being the outputs of the function $2*x \bmod 7$ for inputs 0 - 7. When measuring the quantum register you would have a 2/8 chance of measuring 1, and a 1/8 chance of measuring any of the other outputs. It would seem that this sort of parallelism is not useful, as the more we benefit from parallelism the less likely we are to measure a value of a function for a particular input. Some clever algorithms have been devised, most notably by Peter Shor and L. K. Grover which succeed in using quantum parallelism on a function where they are interested in some property of all the inputs, not just a particular one.

4.9 The Efficiency of the Quantum Computer

Given the possible efficiency of quantum parallelism much work has been done to show formally with mathematical proofs how quantum computers differ from classical ones in the complexity classes and running times of various algorithms. Here is a short list of some of the landmarks in the study of the efficiency of quantum computers.

In 1980 Paul Benioff offered a classical Turing machine which used quantum mechanics in its workings, thus showing that theoretically a quantum computer was at least as powerful as a classical computer. (Benioff)

In 1982 Richard Feynman showed that a classical Turing Machine (and hence any classical computer if the Complexity-Theoretic Church-Turing Thesis holds) could not simulate a quantum mechanical system without suffering exponential slowdown. (Feynman)

David Deutsch and Richard Jozsa showed in a paper in 1992 that there was an algorithm that could be run in poly-log time on a quantum computer, but required linear time on a deterministic Turing machine. This may have been the first example of a quantum computer being shown to be exponentially faster than a deterministic Turing machine. However, the problem could also be solved in poly-log time in a probabilistic Turing machine, a Turing machine which is capable of making a random choice. (Deutsch, Jozsa)

Andre Berthiaume and Giles Brassard proved in a 1992 paper that $P \subset QP$, where P is a complexity class as mentioned earlier and QP (also denoted as EQP) corresponds to problems which can be solved in worst case polynomial time by a quantum computer, so there are indeed problems which can be solved in polynomial time on a quantum computer that can not be solved in a polynomial time with a deterministic Turing machine. (Berthiaume, Brassard)

5 Quantum Algorithms

5.1 Introduction to Shor's Algorithm

By the early nineties it was known that a quantum computer could be more efficient than any classical computer for certain tasks of the Complexity-Theoretic Church-Turing thesis holds . Nonetheless investigations into these areas were largely driven by academic curiosity. There was not much economic motive for people to spend lots of money or time trying to build a quantum computer.

That changed in 1994 when Peter Shor, a scientist working for Bell Labs, devised a polynomial time algorithm for factoring large numbers on a quantum computer. This discovery drew great attention to the field of quantum computing.

5.2 Motivation for Shor's Algorithm

The algorithm was viewed as important because the difficulty of factoring large numbers is relied upon for many cryptography systems. If an efficient method of factoring large numbers is implemented most of the many encryption schemes would be next to worthless to protect their data from prying eyes. While it has not been proven that factoring large numbers can not be archived on a classical computer in polynomial time, as of 2015 the fastest algorithm publicly available for factoring large number runs in $O(exp(\frac{64}{9}n^{1/3}(log\ n)^{2/3})$, operations where n is the number of bits used to represent the number: this runtime exceeds polynomial time. In contrast Shor's algorithm runs in $O((\log n)^2 * \log\log n)$ on a quantum computer, and then must perform $O(\log n)$ steps of post processing on a classical computer. Overall then this time is polynomial. This discovery propelled the study of quantum computing forward, as such an algorithm is much sought after. (Shor)

5.3 Overview of Shor's Algorithm

Shor's algorithm hinges on a result from number theory. This result is:

> The function $\mathcal{F}(a) = x^a \bmod n$ is a periodic function, where x is an integer coprime to n. In the context of Shor's algorithm n will be the number we wish to factor. When two numbers are coprime it means that their greatest common divisor is 1.

Calculating this function for an exponential number of a's would take exponential time on a classical computer. Shor's algorithm utilizes quantum parallelism to perform the exponential number of operations in one step.

The reason why this function is of utility in factoring large numbers is this:

> Since $\mathcal{F}(a)$ is a periodic function, it has some period r. We know that $x^0 \bmod n = 1$, so $x^r \bmod n = 1$, and $x^{2r} \bmod n = 1$ and so on since the function is periodic.

Given this information and through the following algebraic manipulation:

$$x^r \equiv 1 \bmod n$$

$$(x^{r/2})^2 = x^r \equiv 1 \bmod n$$

$$(x^{r/2})^2 - 1 \equiv 0 \bmod n$$

and if r is an even number

$$(x^{r/2} - 1)(x^{r/2} + 1) \equiv 0 \bmod n$$

We can see that the product $(x^{r/2} - 1)(x^{r/2} + 1)$ is an integer multiple of n, the number to be factored. So long as $x^{r/2}$ is not equal to ± 1, then at least one of $(x^{r/2} - 1)$ or $(x^{r/2} + 1)$ must have a nontrivial factor in common with n. So by computing $\gcd(x^{r/2} - 1, n)$, and $\gcd(x^{r/2} + 1, n)$, we will obtain a factor of n, where gcd is the greatest common denominator function.

Here is a brief overview of Shor's algorithm, which is explained in detail in the next section. Shor's algorithm tries to find r, the period of $x^a \bmod n$, where n is the number to be factored and x is an integer coprime to n. To do this Shor's algorithm creates a quantum memory register with two parts. In the first part the algorithm places a superposition of the integers which are to be a's in the $x^a \bmod n$ function. We will choose our a's to be the integers 0 through $q - 1$, where q is the power of two such that $n^2 \leq q < 2n^2$. Then the algorithm calculates $x^a \bmod n$, where a is the superposition of the states, and places the result in the second part of the quantum memory register.

Next the algorithm measures the state of the second register, the one that contains the superposition of all possible outcomes for $x^a \bmod n$. Measuring this register has the effect of collapsing the state into some observed value, say k. It also has the side effect of projecting the first part of the quantum register into a state consistent with the value measured in the second part.

Since we have partitioned our quantum register into two parts measurement of the second part collapses that part into exactly one value, while the other partition collapses into a state consistent with the observed value in the other portion of the register. It is still possible for the non measured part of the register to exist in a superposition of base states, as long as each of those base states are consistent with the measured value in the other part of the register. What this means in this instance is that after this measurement the second part of the register contains the value k, and the first part of the register contains a superposition of the base states which when plugged into $x^a \bmod n$ produce k. Since we know $x^a \bmod n$ is a periodic function, we know that the first part of the register will contain the values c, $c + r$, $c + 2r \ldots$ and so on, where c is the lowest integer such that $x^c \bmod n = k$.

The next step is to perform a discrete Fourier transform on the contents of first part of the register (the one containing all integers such that $x^a \bmod n = k$) and to put the result back into register one. The application of the discrete Fourier transformation has the effect of peaking the probability amplitudes of the first part of the register at integer multiples of the quantity q/r.

Now measuring the first part of the quantum register will yield an integer multiple of the inverse period. Once this number is retrieved from the quantum memory register, a classical computer can do some analysis of this number, make a guess as to the actual value of r, and from that compute the possible factors of n. This post processing will be covered in more detail later. (Shor)

5.4 Steps to Shor's Algorithm

Shor's algorithm for factoring a given integer n can be broken into some simple steps.

Step 1 Determine if the number n is a prime, a even number, or an integer power of a prime number. If it is we will not use Shor's algorithm. There are efficient classical methods for determining if a integer n belongs to one of the above groups, and providing factors for it if it is. This step would be performed on a classical computer.

Step 2 Pick a integer q that is the power of 2 such that $n^2 \leq q < 2n^2$. This step would be done on a classical computer.

Step 3 Pick a random integer x that is coprime to n. When two numbers are coprime it means that their greatest common divisor is 1. There are efficient classical methods for picking such an x. This step would be done on a classical computer.

Step 4 Create a quantum register and partition it into two parts, register 1 and register 2. Thus the state of our quantum computer can be given by: $|\text{reg1}, \text{reg2}\rangle$. Register 1 must have enough qubits to represent integers as large as $q - 1$. Register 2 must have enough qubits to represent integers as large as $n - 1$. The calculations for how many qubits are needed would be done on a classical computer.

Step 5 Load register 1 with an equally weighted superposition of all integers from 0 to $q-1$. Load register 2 with all zeros. This operation would be performed by our quantum computer. The total state of the quantum memory register at this point is:

$$\frac{1}{\sqrt{q}}\sum_{a=0}^{q-1}|a,0\rangle$$

Step 6 Now apply the transformation x^a mod n to for each number stored in register 1 and store the result in register 2. Due to quantum parallelism this will take only one step, as the quantum computer will only calculate $x^{|a\rangle}$ mod n, where $|a\rangle$ is the superposition of states created in step 5. This step is performed on the quantum computer. The state of the quantum memory register at this point is:

$$\frac{1}{\sqrt{q}}\sum_{a=0}^{q-1}|a, x^a \text{ mod } n\rangle$$

Step 7 Measure the second register, and observe some value k. This has the side effect of collapsing register one into a equal superposition of each value a between 0 and $q-1$ such that

$$x^a \text{ mod } n = k$$

This operation is performed by the quantum computer. The state of the quantum memory register after this step is:

$$\frac{1}{\sqrt{||A||}}\sum_{a'=a'\in A}|a', k\rangle$$

Where A is the set of a's such that x^a mod $n = k$, and $||A||$ is the number of elements in that set.

Step 8 Now compute the discrete Fourier transform on register one. The discrete Fourier transform when applied to a state $|a\rangle$ changes it in the following manner:

$$|a\rangle = \frac{1}{\sqrt{q}}\sum_{c=0}^{q-1}|c\rangle * e^{2\pi iac/q}$$

This step is performed by the quantum computer in one step through quantum parallelism. After the discrete Fourier transform our register is in the state:

$$\frac{1}{\sqrt{||A||}}\sum_{a'\in A}\frac{1}{\sqrt{q}}\sum_{c=0}^{q-1}|c,k\rangle * e^{2\pi ia'c/q}$$

Step 9 Measure the state of register one, call this value m, this integer m has a very high probability of being a multiple of q/r, where r is the desired period. This step is performed by the quantum computer.

Step 10 Take the value m, and on a classical computer do some post processing which calculates r based on knowledge of m and q. In particular:

- m has a high probability of being $= \lambda * (q/r)$ where λ is an integer
- If we perform floating point division of m/q, and then calculate the best rational approximation to m/q whose denominator is less than or equal to q
- We take this denominator to be a candidate for r.
- If our candidate r is odd we either double it if doing so leads to a value less than q

There are efficient ways to do this post processing on a classical computer.

Step 11 Once you have attained r, a factor of n can be determined by taking $\gcd(x^{r/2}+1, n)$ and $\gcd(x^{r/2}-1, n)$. If you have found a factor of n, then stop, if not go to step 4. This final step is done on a classical computer.

Step 11 contains a provision for what to do if Shor's algorithm failed to produce factors of n. There are a few reasons why Shor's algorithm can fail, for example the discrete Fourier transform could be measured to be 0 in step 9, making the post processing in step 10 impossible. The algorithm will sometimes find factors of 1 and n, which is not useful either. For these reasons step 11 must be able to jump back to step 4 to start over. (Williams, Clearwater)

5.5 Other Quantum Algorithms

Shor's algorithm is not the only algorithm that seems to be better on a quantum computer than any classical computer for a problem which is considered to be useful. In 1994 L. K. Grover, also of Bell Labs, devised an algorithm to find an item in an unsorted list of N items in $.758\sqrt{N}$ operations. No classical algorithm can guarantee finding the item in less than N operations, and in the average case it would take $N/2$ operations. (Grover)

6 A Simulation of Shor's Algorithm on a Classical Computer

In order to make the steps of Shor's algorithm more concrete to me I made a program which simulates the operation of a quantum computer performing Shor's algorithm. As noted by Feynman and others it is impossible to simulate a quantum mechanical system on a classical computing device without incurring exponential slowdown, thus the simulation is only tolerable to observe for relatively small numbers. The simulation implements all of the steps in Shor's algorithm, and successfully factors numbers. The code is written in C++ and can be found in Appendix D.

6.1 Introduction to the Code for the Simulation

The code base for the simulation of Shor's algorithm consists of four files.

complex.C : This file contains the simple complex number class that I wrote for storing state information about my quantum memory register in the simulation.

qureg.C : This file contains the quantum register class that simulates the behavior of the quantum memory register in Shor's algorithm. It is generic enough that it may be made to simulate any quantum memory register, although what happens in the event of a collapse not due to the direct measurement of the register is left to the programmer to perform. An example of such a collapse that is the collapse of the first partition of Shor's quantum memory register in step 7 of his algorithm.

shor.C : This is the main body of the simulation, which contains the Steps of Shor's algorithm as described above.

util.C : This is a library of useful functions used by shor.C

Makefile : A simple makefile to produce the shor-simulator executable.

6.2 The Complex Number Class

The complex class is needed because the probability amplitudes, or the lengths, of a vector in a Hilbert Space are in general complex. The state of our quantum memory register may be thought of as a vector in a Hilbert Space. Each complex number can be set to a complex value, which is stored internally as a real part, and an imaginary part, both of which are double precision floating point numbers in this simulation.

6.3 The Quantum Memory Register Class

The qureg class, my quantum memory register, is defined in qureg.C. When a quantum memory register is constructed it takes as input the number of qubits which it will contain. In my implementation quantum memory register of n bits requires 2^n complex numbers to represent. This is because a register of n bits can exist in any one of 2^n base states, and a quantum register may exist in any superposition of those base states. I use one complex number per base state to describe the probability of that state being measured.

For example, if a quantum register has 3 bits it can exist in any one of the following states:

$$|0,0,0\rangle$$
$$|0,0,1\rangle$$
$$|0,1,0\rangle$$
$$|0,1,1\rangle$$
$$|1,0,0\rangle$$
$$|1,0,1\rangle$$
$$|1,1,0\rangle$$
$$|1,1,1\rangle$$

If I take these values to be binary numbers where the most significant bit is the left most bit then these states correspond to the numbers 0,1,2,3,4,5,6,7. There are $2^3 = 8$possible base states.

I follow the convention that the probability of measuring the j'th state whose complex amplitude is w_j is $|w_j|^2/\sum_j |w_j|^2$. I also ensure throughout that algorithm that the $\sum_j |w_j|^2$ are equal to 1, so the probability of measuring the j'th state is $|w_j|^2$. Since my qureg class is only being used in Shor's algorithm, any given base state that is measured will correspond to an integer value, for example, in steps 7 and 9 we will measure only integer values. These the base states of our n bit quantum register can be thought of as being the integers 0 through 2^{n-1}. In my code a quantum register of size n will be represented by an array of complex numbers of size 2^n. The value stored at the j'th array position is the complex probability amplitude associated with measuring the number j. For example, in a quantum memory register of size 2 that contains an equal superposition of the numbers 0-3 would be represented by:

$$State[0] = 1/2 + i * 0$$
$$State[1] = 1/2 + i * 0$$
$$State[2] = 1/2 + i * 0$$
$$State[3] = 1/2 + i * 0$$

Observe that the probability of measuring any given state is $|w_j|^2 = \frac{1}{2}^2 + 0^2 = 1/4$.

If we attempt to measure this register, we will with equal probability get the number 0,1,2,3. Let us assume we measure the state and get the value 2, this has the effect of collapsing all probabilities not measured to 0, so the new state of our quantum register is:

$$State[0] = 0 + i * 0$$
$$State[1] = 0 + i * 0$$

$$State[2] = 1 + i * 0$$

$$State[3] = 0 + i * 0$$

If we were to measure this register again we would measure the value 2 without fail.

Sometimes during the course of operation we cause a state to exist in the quantum memory register where the sum over the squares of the probability amplitudes is not 1, for example a quantum register could be in the state:

$$State[0] = 1 + i * 0$$

$$State[1] = 1 + i * 0$$

We would expect that when measured 0 and 1 would be equally likely outcomes, however we would also like for simplicity to have $|w_j|^2$ be the probability of measuring the j'th state. To accommodate this desire the qureg class contains a subroutine which will normalize the probability amplitudes, such that sum of the $|w_j|^2$ over all j's is one.

The qureg class also allows the state of the register to be set to any arbitrary state. This functionality is used in steps 5, 6, 7, and 8 of Shor's algorithm as described above. The quantum memory register class has two member functions which do things that a real quantum memory register could not do.

The first such function is the Dump() function which dumps the entire state information of the register without collapsing it, and is included for debugging. The reason why a real quantum memory register could not do such a thing is that to detect the state information of a real quantum memory register without applying some measurement, which would collapse the state vector of the quantum memory register into one of the base states.

The second such function is the GetProb(state) which gets the probability amplitude of any given state. This function suffers from the same problems as the Dump function, it is impossible to attain information about the state vector without causing it to collapse into one of the base states. The GetProb function is used in the main program to calculate the discrete Fourier transform in step 8. While a real quantum computer could not use this information in step 8, a real quantum computer would not need to. In a real quantum computer the Fourier transformation would take place in one step, and the transform function would take as its argument a state which could in general be any superposition of base states. This function is used to simulate this use of a function which takes a superposition of states as its argument.

6.4 The Simulation of Shor's Algorithm

The implementation of Shor's algorithm found in shor.C follows the steps outlined in the description of Shor's algorithm found above. There are some significant differences in the behavior of the simulator and the behavior of a actual quantum computer.

The simulator uses 2^{n+1} double precision floating point numbers to represent the state of the quantum register. It uses one Complex object to represent the probability amplitude of each of the 2^n eigenstates of a n bit quantum memory register, and each Complex object uses two double precision floating point numbers. On a machine in which a double precision floating point number is represented with 64 bits the simulator uses approximately 2^{n+7} classical bits used to represent the state of the quantum memory register, where n is the number of bits required to represent the number the simulator is trying to factor. A real quantum computer would use exactly n qubits as its memory register. This exponential space usage is apparently unavoidable, as a classical computer cannot simply cannot simulate a quantum computer without suffering from exponential time or space overhead if the Complexity-Theoretic Church-Turing Thesis holds.

A second large difference is that during the modular exponentiation step of Shor's algorithm (Step 6 above) a quantum computer would perform one operation of $x^a \bmod n$, where a is the superposition of states in register 1. The simulation must calculate the superposition of values caused by calculating $x^a \bmod n$ for $a = 0$ through $q-1$ iteratively. The simulation also stores the result of each modular exponentiation, and uses that information to collapse register 1 in step 7 in Shor's algorithm. A quantum computer would not be capable of performing this book keeping, as examining any particular result would collapse the existing superposition to be placed in register 2 at the end of step 6 of Shor's algorithm. A quantum computer would have no need whatsoever to do such bookkeeping, as when when register 2 is measured in step 7, the collapse of register 1 is an automatic and unavoidable consequence of the measurement. A analogous argument follows for the use of the get probability function in step 8 of Shor's algorithm for calculating the discrete Fourier transform.

Aside from these differences, which are necessitated by the inability of a classical computer to accurately depict the behavior of a quantum mechanical system, the operations are performed by the shor.C program are identical to those called for in the description of Shor's algorithm.

As the algorithm runs the state of the quantum memory register changes in the manner laid out in the description of Shor's algorithm. After the final measurement of register 1 in step 9 we obtain some integer m, which has a high probability of being an integer multiple λ of q/r, where λ is some integer, and r is the period of $x^a \bmod n$ that we are trying to find, so that we may calculate $x^{r/2} - 1 \bmod n$ and $x^{r/2} + 1 \bmod n$ in an effort to find numbers which share factors with n.

In the post processing step shor.C takes the integer m and divides it by q, thus yielding some number c which is approximately equal to λ/r, where λ is an integer, and r is the desired period. Then using a helper function it calculates the best rational approximation to c which has a denominator that is less than or equal to q (recall that q is the power of two such that $n^2 \leq q < 2n^2$, where n is the number to be factored by Shor's algorithm).

We take this denominator to be our period. Shor's algorithm can only use even periods in determining factors of n, and so we check to see if r is even, if not we check to see if doubling the period would still yield a period less than q, if so we double our guessed period.

Taking the period, which is now guaranteed to be even, we proceed to calculate $x^{r/2}-1 \bmod n$ and $x^{r/2}+1 \bmod n$, calling these values a and b, we compute the greatest common denominator of a and n, and b and n, to see if we have attained a nontrivial factor of n.

6.5 Utility Functions for the Simulation

The util.C file contains functions which are called from shor.C. Many of the functions are written to avoid overflow when calculating functions in which there are intermediate steps which may have large values. For example while factoring 15, $x^{255} \bmod 15$ is calculated, but the modular exponentiation function in util.C calculates it in such a manner that x^{255} need not be explicitly calculated. I have commented these helper functions in the code itself.

7 Conclusion

From the dawn of computer science the field has benefited from abstractions which simulate actual computing devices. Between the Turing machine and the Church-Turing Thesis a strong foundation was made for study of the computable and uncomputable. Complexity analysis provides a way to distinguish classes of problems based on their runtime characteristics, and the rough grouping of problems of polynomial runtime as tractable and others as intractable combined with the presumed correctness of the Complexity-Theoretic Church-Turing hypothesis suggest whether or not a problem is tractable is not depended on the model of computation used.

Through the principle of superposition in quantum systems we can create useful memory components that are on the scale of an atom or smaller. These quantum memory registers may be able to facilitate exponential computational speed increases in algorithms that can take advantage of quantum parallelism.

Peter Shor has shown an algorithm which makes factoring large numbers tractable for a quantum computer, where no such algorithm is published for a classical computer. In doing so has drawn great attention to the field of quantum computing. Due to Shor's algorithm, we may someday have to turn to other means of encrypting data than are today typically employed. L. K. Grover's database search algorithm shows another noteworthy task that a quantum computer can perform faster than any classical computer.(Brassard)

The efforts to build a real quantum memory register that functions are in the most preliminary stages. As of the original time of writing this paper, 3-qubit registers had been built. In 2001 Shor's Algorithm was applied to the number 15 at IBM's Almaden Research Center and Stanford University. In 2005 the first qubyte (g. 8-bit quantum register) was created at The Institute of Quantum Optics and Quantum Information at the University of Innsbruck in Austria. In 2009 NIST reads and writes individual qubits, and demonstrates some computing operations on qubits. (Timeline of quantum computing)

Operational quantum computers are by no means an inevitable consequence of this research. It may be that the problems surrounding keeping a quantum memory register isolated from any disturbance long enough for a calculation to take place will be insurmountable. In any case, quantum computing will remain an exciting topic for experimentalists and theorists alike for years to come. Hopefully this paper, and the simulation of Shor's algorithm have been as enlightening and fun for the reader as they were for the author.

8 Bibliography

Barenco, Ekert, Sanpera and Machiavello. "Un saut d'echelle pour les calculateurs," *La Recherche*, November 1996.

Benioff, p. "The Computer as a Physical System: A Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines," *Journal of Statistical Physics*, Vol. 22 (1980), pp. 563-591.

Berthiaume, Andre and Brassard, Gilles. "The quantum Challenge to Complexity Theory," *Proceedings of the 7th IEEE Conference on Structure in Complexity Theory* (1992), pp. 132-137.

Brassard, Gilles. "Searching a Quantum Phone Book," *Science*, 31 January 1997.

Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. "Introduction to Algorithms," St. Louis: McGraw-Hill, 1994.

Deutsch, David and Jozsa, Richard. "Rapid Solution of Problems by Quantum Computation," *Proceedings Royal Society London*, Vol. 439A (1992), pp. 553-558.

Feynman, Richard. "Simulating Physics with Computers," *Optics News* Vol. 11 (1982), pp. 467-488.

Grover, L. K. "A Fast Quantum Mechanical Algorithm for Database Search," *Proceedings of the 28'th Annual ACM Symposium on the Theory of Computing* (1996), pp. 212-219.

Shor, Peter. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124-134.

Steane, Andrew. "Quantum Computing," *Reports on Progress in Physics*, vol 61 (1998), pp 117-173.

Timeline of quantum computing (2015). *Timeline of quantum computing.* Available at: http://en.wikipedia.org/wiki/Timeline_of_quantum_computing [Accessed 14 Jan. 2015].

Williams, Colin P. and Clearwater, Scott H. "Explorations in Quantum Computing," New York: Springer-Verlag, 1998.

9 Glossary

This is a glossary of terms and variables used throughout this paper.

λ : In the context of Shor's algorithm in integer such that $m = \frac{\lambda q}{r}$.

a : A variable that when used in the context of Shor's algorithm is an argument to the function $\mathcal{F}(a) = x^a \bmod n$. It may be a single integer, or it may denote a superposition of states.

Algorithm : A method for producing some answer given some inputs.

Binary : The base two number system. For more information see Appendix A.

Bit : A thing which can store one item of information, either a 1 or a 0.

Church-Turing Thesis : The Church-Turing Thesis can be thought of as defining the class of functions which are effectively calculable as those which can be computed by a Turing machine.

Classical computer : A computer whose internal workings behave in manner consistent with classical physics. Data registers in a classical computer can not exist in a superposition of states. By definition (e.g. the Church-Turing Thesis) a classical computer can only compute functions that a Turing machine can compute.

Classical physics : The model that was used to describe physical phenomenon before the advent of quantum physics. The predictions of classical physics with regard to the behavior of fundamental particles are incorrect.

Collapse : A collapse in the context of this paper is what happens to the state vector of a quantum mechanical system when that system is observed or measured. Since the system can only be measured to be in one of its base states, the state vector will collapse from some superposition of base states into the measured state only.

Complex number : A number of the form $a + i * b$, where a and b are real numbers and i is defined to be the square root of negative one.

Complex vector space : A vector space in which the coordinates of a vector are complex numbers.

Complexity class : A grouping of algorithms based on how their memory usage and number of operations scale with the size of the input.

Complexity-Theoretic Church-Turing Thesis : The hypothesis that a Turing machine can efficiently simulate any realistic model of computation. Here efficiently means with polynomial time overhead. It should be noted that this is a hypothesis only, however the advent of a physically realizable computing device which violates this thesis would be a major breakthrough, and there are arguments as to why such a breakthrough may never occur.

Coprime : Integers a and b are coprime if their greatest common denominator is one.

Discrete Fourier Transform : A transformation converts a finite list of equally spaced samples of a function into a list of coefficients of finite combinations of circles, ordered by their frequencies, that have the same values. In Shor's algorithm it is used to calculate and multiple of the inverse period, where the period is the quantity which enables Shor's algorithm to find factors of a number n.

Exponential : A function which goes as: $\mathcal{F}(x) = a^x$, where a is some constant greater than 1.

Exponential time : This is an attribute of an algorithm which means the number of operations required to compute the answer grows exponentially with the size of the input.

gcd : This is an abbreviation for the mathematical function which calculates the greatest common denominator of two integers. The greatest common denominator of two integers a and b is the largest integer c such that a/c and b/c are integers.

Grover's Search Algorithm : An algorithm designed by L. K. Grover of Bell Labs which finds a element in an unsorted database of size n in $O(\sqrt{n})$ operations on a quantum computer. The lowest known running time for this problem on a classical computer is $O(n)$.

Hilbert Space : A complex linear vector space. The complete state of a n state quantum mechanical system can be represented by a vector in an n dimensional Hilbert Space.

i : The square root of -1.

Linear vector space : A vector space such that vectors within the space which are added or multiplied together result in vectors that also lie within the same space.

Memory register : A array of bits on a classical computer. A memory register of size n may store one of 2^n values.

Mutually perpendicular : In the context of vector spaces mutually perpendicular items are items such that no one can be decomposed into components of the others.

n : In the context of Shor's algorithm, a number to be factored.

Periodic function : A function with a period r such that $\mathcal{F}(x) = \mathcal{F}(x+r) = \mathcal{F}(x+2r)$ and so on. Sine and Cosine are typical examples of periodic functions.

Polynomial time : This is an attribute of an algorithm which means the number of operations required to compute the answer grows polynomially with the size of the input n (E.g. like n^c for some constant c).

q : In the context of Shor's algorithm the power of 2 such that $n^2 \leq q < 2n^2$.

Quantum memory register : A array of n qubits which can exist in any superposition of its 2^n base states.

Quantum parallelism : The ability of a quantum computer to perform an operation on a quantum memory register which results in the simultaneous calculation of a function function of 2^n different values where n is the size of the quantum memory register.

Quantum physics : Currently the most complete model for describing the behavior of small physical systems.

Qubit : A two state quantum mechanical system, which can exist in any superposition of the 0 and 1 state. In this paper I have considered a spin-1/2 particle as a possible candidate for a qubit in a physical implementation of a qubit.

r : In the context of Shor's algorithm the period of the periodic function x^a mod n.

Shor's Algorithm : A algorithm designed by Peter Shor of Bell Labs which finds factors of a number n in polynomial time relative to the number of bits in n's binary representation on a quantum computer. The fastest published algorithm on a classical computer is slower than polynomial time, and the presumed intractability of this problem is the basis for many cryptographic systems.

Spin-1/2 particle : A particle which can be characterized as having a spin of +1/2 or -1/2. Examples include the proton, neutron, and electron.

State vector : In the context of this paper, the state vector in a Hilbert Space which completely describes a quantum mechanical state vector - such as the state of quantum memory register.

Superposition of states : A mixture of base states. The state vector for a quantum mechanical systems which can be measured in one of n base states can exist as any combination of components of the base states.

Turing machine : A theoretical computing device consisting of an infinite tape divided into cells which can hold a 1, a 0, or a blank and a head which can move around the tape, read and write bits, and change its own internal state. The Church-Turing Thesis defines effectively computable functions as those which can be computed by a Turing machine.

Unit vector : A vector whose length is 1.

x : In the context of Shor's algorithm a integer which is coprime to n and used in the function $\mathcal{F}(a) = x^a \bmod n$.

A Mathematics Used in this Paper

This appendix will review some of the mathematics used in the paper. The sections are not intended to be comprehensive for their topic, they only cover what is needed to understand this paper.

A.1 Binary Representation of Numbers

We commonly represent number is base 10, there are 10 elements in our base 10 numbering system, 0,1,2,3,4,5,6,7,8, and 9. In a base n counting system there are n distinct elements, 0 through $n-1$.

When a number which is greater than $n-1$ needs to be displayed in base n it is represented by a string of the $n-1$ elements. The value of any given symbol in the string is found by multiplying that symbol by n^x, where x is the number of symbols in the string that are to the right of the symbol in question.

For example, in base 10 the number 982 is equal to $9*10^2+8*10^1+2*10^0$.

Likewise, in base two the number 10101001 is equal to $1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0 = 169$ in base 10.

A.2 Complex Numbers

A complex number is a number of the form $a+i*b$, where a and b are real numbers, and i is defined to be the square root of negative one. Addition of two complex numbers c_1 and c_2 is defined to be:

$$c_1 = a_1 + i * b_1$$

$$c_2 = a_2 + i * b_2$$

$$c_1 + c_2 = a_1 + a_2 + i * (b_1 + b_2)$$

The complex conjugate of a complex number c, denoted c^* is defined to be:

$$c = a + i * b$$

$$c^* = a - i * b$$

Multiplication of two complex numbers c_1 and c_2 is defined to be:

$$c_1 = a_1 + i * b_1$$

$$c_2 = a_2 + i * b_2$$

$$c_1 * c_2 = a_1 * a_2 - b_1 * b_2 + i * (a_1 * b_2 + a_2 * b_1)$$

Euler's Formula for complex numbers states that $e^{ix} = \cos x + i * \sin x$, this relationship is used in the discrete Fourier transform of Shor's algorithm.

A.3 Vector Mathematics

The only operations that are used in Shor's algorithm on vectors are addition, length determination, and scaling. The vector in question is that state vector of a quantum mechanical system, it is a vector in a Hilbert Space. Being a vector in a Hilbert space means that the vector can be represented by projections of the vector onto each of the perpendicular base vectors which define the Hilbert Space.

For example, a n state quantum system requires a n dimensional Hilbert Space to represent its state vector. The quantum system can be measured in any of the n states, and to represent this we imagine each of the n states as mutually perpendicular axes within our Hilbert space. Thus the state vector for a system in the j'th state is equal to:

$$\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$

For the n states, where the number at the top of the column is the length of the state vector projected onto the 1st state, and the 1 appears in the j'th row.

To add two vectors we simply add their components.

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix}$$

Since this vector lies in a Hilbert Space the projections of the state vector onto the coordinate axes are allowed to be complex numbers, thus the definition of length is slightly different from what is expected.

The length of a vector in a Hilbert space with n components is defined to be: $\sqrt{\sum_{j=1}^{n} |w_j|^2}$ where w_j is the value of the j'th component of the vector, and $|w_j|^2$ is defined to be w_j times its complex conjugate, or when $w_j = a + i * b$, $|w_j|^2 = a^2 + b^2$.

To scale a vector by any length l you simply multiply each component of the vector by the value l. In particular to scale a vector to length 1 you multiply each component by the inverse length of the vector.

B Actual Quantum Computers and Further Research

There is great interest in building an actual quantum computer. Many strategies for building a quantum memory register are currently being evaluated. The study of quantum computing is a relatively new field, and the experimental implementation is still in its infancy. Consequently the world wide web is the best place to search for information regarding current research.

Note: This section has not been updated since 1999.

One possible way to implement a quantum memory register is by using nuclear magnetic resonance techniques to manipulate classical fluids to perform quantum calculations. To read more on this see:

http://www.sciam.com/1998/0698issue/0698gershenfeld.html

http://squint.stanford.edu/qc/overview.html

A second approach is to use ion traps and lasers to hold atomic ions in place. To read more about this method check:

http://physics.colorado.edu/faculty/monroe.c.html

http://www.bldrdoc.gov/timefreq/ion/index.htm

A good place to look for original research papers is the Los Alamos National Laboratories E-print archive at:

http://quickreviews.org/

This is by no means a comprehensive list of approaches to and research regarding quantum computing. Given the changing topology of the web and the youth of the subject you will probably be able to search for and find a wealth of information on quantum computing from your favorite web search engine or directory service.

C Other Quantum Computer Simulators

On a more theoretical note there is some people are writing other, more robust and generic simulators of a quantum computer. The following web sites have information about quantum computing, and have code for simulating the action of a quantum computer. I found these sites to be great resources during my study of quantum computing.

Note: This section has not been updated since 1999.

http://tph.tuwien.ac.at/ oemer/qc/qcl/qcl.html

http://tph.tuwien.ac.at/ oemer/prakt/prakt.html

http://www.openqubit.org/

D Code for my Simulation of Shor's Algorithm

Here is included the code for my simulation of Shor's algorithm. I have tried to document the code as well as possible, buy if you find something difficult please write me at mjhayward@gmail.com and I will attempt to explain it as well as I can. The code was originally compiled in 1999 with g++ version 2.8.1 on a Pentium Pro 200 running Linux 2.0.33, and had also been compiled it on a UltraSparc running SunOS 5.6 also using g++ 2.8.1, with one modification.

Since then, the code has been updated to compile with gcc/g++ 4.8.2 and GNU Make 3.81. I hope that you will find it easy to compile the code with g++. If you have trouble compiling please send me information about your compiler, OS, and error messages, and I will see what I can do.

D.1 complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex {
public:
  Complex();                   //Default constructor
  Complex( double, double );  // Argument constructor
  virtual ~Complex();          //Default destructor.
  void Set(double new_real, double new_imaginary); //Set data members.
  double Real() const;              //Return the real part.
  double Imaginary() const;         //Return the imaginary part.
  Complex & operator+(const Complex&); //Overloaded + operator
  Complex & operator*(const Complex&); //Overloaded * operator
  Complex & operator=(const Complex&); //Overloaded = operator
  bool operator==(const Complex&) const;    //Overloaded == operator
private:
  double real;
  double imaginary;
};

#endif
```

D.2 complex.C

```
#include <math.h>
#include "complex.h"

//Complex constructor, initializes to 0 + i0.
Complex::Complex(): real( 0 ), imaginary( 0 ) {}

//Argument constructor.
Complex::Complex( double r, double i ): real( r ), imaginary( i ) {}

//Complex destructor.
Complex::~Complex() {}
```

```
//Overloaded = operator.
Complex & Complex::operator=(const Complex &c) {
  if (&c != this) {
    real = c.Real();
    imaginary = c.Imaginary();
    return *this;
  }
}

//Overloaded + operator.
Complex & Complex::operator+(const Complex &c) {
  real += c.Real();
  imaginary += c.Imaginary();
  return *this;
}

//Overloaded * operator.
Complex & Complex::operator*(const Complex &c) {
  real = real * c.Real() - imaginary * c.Imaginary();
  imaginary = real * c.Imaginary() + imaginary * c.Real();
  return *this;
}

//Overloaded == operator.  Small error tolerances.
bool Complex::operator==(const Complex &c) const {
  //This is to take care of round off errors.
  if (fabs(c.Real() - real) > pow(10,-14)) {
    return false;
  }
  if (fabs(c.Imaginary()- imaginary) > pow(10,-14)) {
    return false;
  }
  return true;
}

//Sets private data members.
void Complex::Set(double new_real, double new_imaginary) {
  real = new_real;
  imaginary = new_imaginary;
  return;
}

//Returns the real part of the complex number.
double Complex::Real() const {
  return real;
}
```

```
//Returns the imaginary part of the complex number.
double Complex::Imaginary() const {
  return imaginary;
}
```

D.3 qureg.h

```
#include "complex.h"

#ifndef QUREG_H
#define QUREG_H

using namespace std;

class QuReg {
public:
  //Default constructor.  Size is the size in bits of our register.
  //In our implementation of Shor's algorithm we will need size bits
  //to represent our value for "q" which is a number we have chosen
  //with small prime factors which is between 2n^2 and 3n^2 inclusive
  //where n is the number we are trying to factor.  We envision our
  //the description of our register of size "S" as 2^S complex
  //numbers, representing the probability of finding the register on
  //one of or 2^S base states.  Thus we use an array of size 2^S, of
  //Complex numbers.  Thus if the size of our register is 3 bits
  //array[7] is the probability amplitude of the state |1,1,1>, and
  //array[7] * Complex Conjugate(array[7]) = probability of choosing
  //that state.  We use normalized state vectors thought the
  //simulation, thus the sum of all possible states times their
  //complex conjugates is = 1.
  QuReg(unsigned long long int size);

  QuReg(); //Default Constructor

  QuReg(const QuReg &); //Copy constructor

  virtual ~QuReg(); //Default destructor.

  //Measures our quantum register, and returns the decimal
  //interpretation of the bit string measured.
  unsigned long long int DecMeasure();

  //Dumps all the information about the quantum register.  This
  //function would not be possible for an actual quantum register, it
  //is only there for debugging.  When verbose != 0 we return every
  //value, when verbose = 0 we return only probability amplitudes
  //which differ from 0.
  void Dump(int verbose) const;

  //Sets state of the qubits using the arrays of complex amplitudes.
  void SetState(Complex *new_state);

  //Sets the state to an equal superposition of all possible states
  //between 0 and number inclusive.
  void SetAverage(unsigned long long int number);

  //Normalize the state amplitudes.
  void Norm();

  //Get the probability of a given state.  This is used in the
  //discrete Fourier transformation.  In a real quantum computer such
  //an operation would not be possible, on the flip side, it would
  //also not be necessary as you could simply build a DFT gate, and
  //run your superposition through it to get the right answer.
  Complex GetProb(unsigned long long int state) const;

  //Return the size of the register.
  int Size() const;

private:
  unsigned long long int reg_size;
  Complex *State;
};

#endif
```

D.4 qureg.C

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "complex.h"
#include "qureg.h"

using namespace std;

QuReg::QuReg(): reg_size( 0 ), State( NULL ) {}

//Constructor.
QuReg::QuReg(unsigned long long int size) {
  reg_size = size;
  State = new Complex[(unsigned long long int)pow(2, reg_size)];
  srand(time(NULL));
}

//Copy Constructor
QuReg::QuReg(const QuReg & old) {
  reg_size = old.reg_size;
  unsigned long long int reg_length = (unsigned long long int) pow(2, reg_size);
  State = new Complex[reg_length];
  for (unsigned int i = 0 ; i < reg_length ; i++) {
    State[i] = old.State[i];
  }
}

//Destructor.
QuReg::~QuReg() {
  if ( State ) {
    delete [] State;
  }
}

//Return the probability amplitude of the state'th state.
Complex QuReg::GetProb(unsigned long long int state) const {
  if (state >= pow(2, reg_size)) {
    cout << "Invalid state index " << state << " requested!"
 << endl << flush;
    throw -1;
  } else {
    return(State[state]);
  }
}

//Normalize the probability amplitude, this ensures that the sum of
//the sum of the squares of all the real and imaginary components is
//equal to one.
void QuReg::Norm() {
  double b = 0;
  double f, g;
  for (unsigned long long int i = 0; i < pow(2, reg_size) ; i++) {
    b += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
  }
  b = pow(b, -.5);
  for (unsigned long long int i = 0; i < pow(2, reg_size) ; i++) {
    f = State[i].Real() * b;
    g = State[i].Imaginary() * b;
    State[i].Set(f, g);
  }
}

//Returns the size of the register.
int QuReg::Size() const {
  return reg_size;
}

//Measure a state, and return the decimal value measured.  Collapse
//the state so that the probability of measuring the measured value in
//the future is 1, and the probability of measuring any other state is
//0.
unsigned long long int QuReg::DecMeasure() {
  int done = 0;
  int DecVal = -1; //-1 is an error, we did not measure anything.
  double rand1, a, b;
  rand1 = rand()/(double)RAND_MAX;
  a = b = 0;
  for (unsigned long long int i = 0 ; i < pow(2, reg_size)  ;i++) {
    if (!done ){
      b += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
      if (b > rand1 && rand1 > a) {
//We have just measured the i state.
for (unsigned long long int j = 0; j < pow(2, reg_size) ; j++) {
  State[j].Set(0,0);
}
State[i].Set(1,0);
DecVal = i;
done = 1;
      }
      a += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
    }
  }
  return DecVal;
}

//For debugging, output information about the register.
void QuReg::Dump(int verbose) const {
  for (unsigned long long int i = 0 ; i < pow(2, reg_size) ; i++) {
    if (verbose || fabs(State[i].Real()) > pow(10,-14)
|| fabs(State[i].Imaginary()) > pow(10,-14)) {
      cout << "State " << i << " has probability amplitude "
   << State[i].Real() << " + i" << State[i].Imaginary()
   << endl << flush;
    }
  }
}

//Set the states to those given in the new_state array.
void QuReg::SetState(Complex *new_state) {
  //Beware, this function will cause segfaults if new_state is too
  //small.
  for (unsigned long long int i = 0 ; i < pow(2, reg_size) ; i++) {
    State[i].Set(new_state[i].Real(), new_state[i].Imaginary());
  }
}

//Set the State to an equal superposition of the integers 0 -> number
//- 1
void QuReg::SetAverage(unsigned long long int number) {
  if (number >= pow(2, reg_size)) {
    cout << "Error, initializing past end of array in qureg::SetAverage.\n";
  } else {
    double prob = pow(number, -.5);
    for (unsigned long long int i = 0 ; i <= number ; i++) {
      State[i].Set(prob, 0);
    }
  }
}
```

D.5 shor.C

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#include "complex.h"
#include "qureg.h"
#include "util.h"

using namespace std;

int main() {
  //Establish a random seed.
  srand(time(NULL));

  //Output standard greeting.
  cout << "Welcome to the simulation of Shor's algorithm." << endl
       << "There are four restrictions for Shor's algorithm:" << endl
       << "1) The number to be factored (n) must be >= 15." << endl
```

```
     << "2) The number to be factored must be odd." << endl
     << "3) The number must not be prime." << endl
     << "4) The number must not be a prime power." << endl
     << endl << "There are efficient classical methods of factoring "
     << "any of the above numbers, or determining that they are prime."
     << endl << endl << "Input the number you wish to factor." << endl
     << flush;

//n is the number we are going to factor, get n.
unsigned long long int n;
cin >> n;

cout << "Step 1 starting." << endl << flush;
//Test to see if n is factorable by Shor's algorithm.
//Exit if the number is even.
if (n%2 == 0) {
  cout << "\tError, the number must be odd!" << endl << flush;
  exit(0);
}
//Exit if the number is prime.
if (TestPrime(n)) {
  cout << "\tError, the number must not be prime!" << endl << flush;
  exit(0);
}
//Prime powers are prime numbers raised to integral powers.
//Exit if the number is a prime power.
if (TestPrimePower(n)) {
  cout << "\tError, the number must not be a prime power!" << endl << flush;
  exit(0);
}
cout << "Step 1 complete." << endl << flush;

cout << "Step 2 starting." << endl << flush;
//Now we must figure out how big a quantum register we need for our
//input, n.  We must establish a quantum register big enough to hold
//an equal superposition of all integers 0 through q - 1 where q is
//the power of two such that n^2 <= q < 2n^2.
unsigned long long int q;
cout << "\tSearching for q, the smallest power of 2 greater than or equal to n^2." <<
q = GetQ(n);
cout << "\tFound q to be " << q << "." << endl << flush;
cout << "Step 2 complete." << endl << flush;

cout << "Step 3 starting." << endl << flush;
//Now we must pick a random integer x, coprime to n.  Numbers are
//coprime when their greatest common denominator is one.  One is not
```

```
//a useful number for the algorithm.
unsigned long long int x = 0;
cout << "\tSearching for x, a random integer coprime to n." << endl << flush;
x = 1+ (unsigned long long int)((n-1)*(double)rand()/(double)RAND_MAX);
while (GCD(n,x) != 1 || x == 1) {
    x = 1 + (unsigned long long int)((n-1)*(double)rand()/(double)RAND_MAX);
}
cout << "\tFound x to be " << x << "." << endl << flush;
cout << "Step 3 complete." << endl << flush;

//Create the register.
cout << "Step 4 starting." << endl << flush;
QuReg * reg1 = new QuReg(RegSize(q) - 1);
cout << "\tMade register 1 with register size = " << RegSize(q) << endl
     << flush;

//This array will remember what values of q produced for x^q mod n.
//It is necessary to retain these values for use when we collapse
//register one after measuring register two.  In a real quantum
//computer these registers would be entangled, and thus this extra
//bookkeeping would not be needed at all.  The laws of quantum
//mechanics dictate that register one would collapse as well, and
//into a state consistent with the measured value in resister two.
unsigned long long int * modex = new unsigned long long int[q];

//This array holds the probability amplitudes of the collapsed state
//of register one, after register two has been measured it is used
//to put register one in a state consistent with that measured in
//register two.
Complex * collapse = new Complex[q];

//This is a temporary value.
Complex tmp;

//This is a new array of probability amplitudes for our second
//quantum register, that populated by the results of x^a mod n.
Complex * mdx = new Complex[(unsigned long long int)pow(2,RegSize(n))];

// This is the second register.  It needs to be big enough to hold
// the superposition of numbers ranging from 0 -> n - 1.
QuReg *reg2 = new QuReg(RegSize(n));
cout << "\tCreated register 2 of size " << RegSize(n) << endl << flush;
cout << "Step 4 complete." << endl << flush;

//This is a temporary value.
unsigned long long int tmpval;
```

```
//This is a temporary value.
unsigned long long int value;

//c is some multiple lambda of q/r, where q is q in this program,
//and r is the period we are trying to find to factor n.  m is the
//value we measure from register one after the Fourier
//transformation.
double c, m;

//This is used to store the denominator of the fraction p / den where
//p / den is the best approximation to c with den <= q.
unsigned long long int den;

//This is used to store the numerator of the fraction p / den where
//p / den is the best approximation to c with den <= q.
unsigned long long int p;

//The integers e, a, and b are used in the end of the program when
//we attempts to calculate the factors of n given the period it
//measured.
//Factor is the factor that we find.
unsigned long long int e, a, b, factor;

//Shor's algorithm can sometimes fail, in which case you do it
//again.  The done variable is set to 0 when the algorithm has
//failed.  Only try a maximum number of tries.
unsigned int done = 0;
unsigned int tries = 0;
while (!done) {
  if (tries >= 5) {
    cout << "\tThere have been five failures, giving up." << endl << flush;
    exit(0);
  }

  cout << "Step 5 starting attempt: " << tries+1 << endl << flush;
  //Now populate register one in an even superposition of the
  //integers 0 -> q - 1.
  reg1->SetAverage(q - 1);
  cout << "Step 5 complete." << endl << flush;

  cout << "Step 6 starting attempt: " << tries+1 << endl << flush;
  //Now we preform a modular exponentiation on the superposed
  //elements of reg 1.  That is, perform x^a mod n, but exploiting
  //quantum parallelism a quantum computer could do this in one
  //step, whereas we must calculate it once for each possible
```

```
    //measurable value in register one.  We store the result in a new
    //register, reg2, which is entangled with the first register.
    //This means that when one is measured, and collapses into a base
    //state, the other register must collapse into a superposition of
    //states consistent with the measured value in the other..  The
    //size of the result modular exponentiation will be at most n, so
    //the number of bits we will need is therefore less than or equal
    //to log2 of n.  At this point we also maintain a array of what
    //each state produced when modularly exponised, this is because
    //these registers would actually be entangled in a real quantum
    //computer, this information is needed when collapsing the first
    //register later.

    //This counter variable is used to increase our probability amplitude.
    tmp.Set(1,0);

    //This for loop ranges over q, and puts the value of x^a mod n in
    //modex[a].  It also increases the probability amplitude of the value
    //of mdx[x^a mod n] in our array of complex probabilities.
    for (unsigned long long int i = 0 ; i < q ; i++) {
      //We must use this version of modexp instead of c++ builtins as
      //they overflow when x^i is large.
      tmpval = modexp(x,i,n);
      modex[i] = tmpval;
      mdx[tmpval] = mdx[tmpval] + tmp;
    }

    //Set the state of register two to what we calculated it should be.
    reg2->SetState(mdx);

    //Normalize register two, so that the probability of measuring a
    //state is given by summing the squares of its probability
    //amplitude.
    reg2->Norm();
    cout << "Step 6 complete." << endl << flush;

    cout << "Step 7 starting attempt: " << tries+1 << endl << flush;
    //Now we measure reg2.
    value = reg2->DecMeasure();

    //Now we must using the information in the array modex collapse
    //the state of register one into a state consistent with the value
    //we measured in register two.
    for (unsigned long long int i = 0 ; i < q ; i++) {
      if (modex[i] == value) {
collapse[i].Set(1,0);
```

```
      } else {
collapse[i].Set(0,0);
      }
    }

    //Now we set the state of register one to be consistent with what
    //we measured in state two, and normalize the probability
    //amplitudes.
    reg1->SetState(collapse);
    reg1->Norm();
    cout << "Step 7 complete." << endl << flush;

    //Here we do our Fourier transformation.
    cout << "Step 8 starting attempt: " << tries+1 << endl << flush;
    DFT(reg1, q);
    cout << "Step 8 complete." << endl << flush;

    cout << "Step 9 starting attempt: " << tries+1 << endl << flush;
    //Next we measure register one, due to the Fourier transform the
    //number we measure, m will be some multiple of lambda/r, where
    //lambda is an integer and r is the desired period.
    m = reg1->DecMeasure();
    cout << "\tValue of m measured as: " << m << endl << flush;
    cout << "Step 9 complete." << endl << flush;

    //If nothing goes wrong from here on out we are done.
    done = 1;

    //If we measured zero, we have gained no new information about the
    //period, we must try again.
    if (m == 0) {
      cout << "\tMeasured, 0 this trial a failure!" << endl << flush;
      done = 0;
    }

    //The DecMeasure subroutine will return -1 as an error code, due
    //to rounding errors it will occasionally fail to measure a state.
    if (m == -1) {
      cout << "\tWe failed to measure anything, this trial a failure!" << endl << flush;
      done = 0;
    }

    //If nothing has gone wrong, try to determine the period of our
    //function, and get factors of n.
    if (done) {
      //Now c =~ lambda / r for some integer lambda.  Borrowed with
      //modifications from Berhnard Ohpner.
      c = (double)m  / (double)q;

      cout << "Steps 10 and 11 starting attempt: " << tries+1 << endl << flush;
      //Calculate the denominator of the best rational approximation
      //to c with den < q.  Since c is lambda / r for some integer
      //lambda, this will provide us with our guess for r, our period.
      den = denominator(c, q);

      //Calculate the numerator from the denominator.
      p = (unsigned long long int)floor(den * c + 0.5);

      //Give user information.
      cout << "\tMeasured m: " << m << ", rational approximation for m/q=" << c << " is:
   << p << " / " << den << endl << flush;

      //The denominator is our period, and an odd period is not
      //useful as a result of Shor's algorithm.  If the denominator
      //times two is still less than q we can use that.
      if (den % 2 == 1 && 2 * den < q ){
cout << "\tOdd candidate for r found, expanding by 2\n";
p = 2 * p;
den = 2 * den;
      }

      //Initialize helper variables.
      e = a = b = factor = 0;

      // Failed if odd denominator.
      if (den % 2 == 1)  {
cout <<  "\tOdd period found.  This trial failed."
     << " \t Trying again." << endl << flush;
done = 0;
      } else {
//Calculate candidates for possible common factors with n.
cout <<  "\tCandidate period is " << den << endl << flush;
e = modexp(x, den / 2, n);
a = (e + 1) % n;
b = (e + n - 1) % n;
cout << "\t" << x << "^" << den / 2 << " + 1 mod " << n << " = " << a
     << "," << endl
     << "\t" << x << "^" << den / 2 << " - 1 mod " << n << " = " << b
     << endl << flush;
factor = max(GCD(n,a),GCD(n,b));
      }
    }
    //GCD will return a -1 if it tried to calculate the GCD of two
    //numbers where at some point it tries to take the modulus of a
    //number and 0.
    if (factor == -1) {
      cout << "\tError, tried to calculate n mod 0 for some n.  Trying again."
   << endl << flush;
      done = 0;
    }

    if ((factor == n || factor == 1) && done == 1) {
      cout << "\tFound trivial factors 1 and " << n
   << ".  Trying again." << endl << flush;
      done = 0;
    }

    //If nothing else has gone wrong, and we got a factor we are
    //finished.  Otherwise start over.
    if (factor != 0 && done == 1) {
      cout << "\t" << n << " = " << factor << " * " << n / factor << endl << flush;
    } else if (done == 1) {
      cout << "\tFound factor to be 0, error.  Trying again." << endl
   << flush;
      done = 0;
    }
    cout << "Steps 10 and 11 complete." << endl << flush;
    tries++;
  }
  delete reg1;
  delete reg2;
  delete [] modex;
  delete [] collapse;
  delete [] mdx;
  return 1;
}
```

D.6 util.h

```
#include "qureg.h"

#ifndef UTIL_H
#define UTIL_H

//This function takes an integer input and returns 1 if it is a prime
//number, and 0 otherwise.
unsigned long long int TestPrime(unsigned long long int n);
```

```
//This function takes an integer input and returns 1 if it is equal to
//a prime number raised to an integer power, and 0 otherwise.
unsigned long long int TestPrimePower(unsigned long long int n);

//This function computes the greatest common denominator of two integers.
//Since the modulus of a number mod 0 is not defined, we return a -1 as
//an error code if we ever would try to take the modulus of something and
//zero.
unsigned long long int GCD(unsigned long long int a, unsigned long long int b);

//This function takes and integer argument, and returns the size in bits
//needed to represent that integer.
unsigned long long int RegSize(unsigned long long int a);

//q is the power of two such that n^2 <= q < 2n^2.
unsigned long long int GetQ(unsigned long long int n);

//This function takes three integers, x, a, and n, and returns x^a mod
//n.  This algorithm is known as the "Russian peasant method," I
//believe, and avoids overflow by never calculating x^a directly.
unsigned long long int modexp(unsigned long long int x, unsigned long long int a, unsign

// This function finds the denominator q of the best rational
// denominator q for approximating p / q for c with q < qmax.
unsigned long long int denominator(double c, unsigned long long int qmax);

//This function takes two integer arguments and returns the greater of
//the two.
unsigned long long int max(unsigned long long int a, unsigned long long int b);

//This function computes the discrete Fourier transformation on a register's
//0 -> q - 1 entries.
void DFT(QuReg * reg, unsigned long long int q);

#endif
```

D.7 util.C

```
#include <iostream>
#include <math.h>
#include "qureg.h"

using namespace std;

//This function takes an integer input and returns 1 if it is a prime
```

```
//number, and 0 otherwise.
//
// Not optimized at all.
unsigned long long int TestPrime(unsigned long long int n) {
  unsigned long long int i;
  for (i = 2 ; i <= floor(sqrt(n)) ; i++) {
    if (n % i == 0) {
      return(0);
    }
  }
  return(1);
}

//This function takes an integer input and returns 1 if it is equal to
//a prime number raised to an integer power, and 0 otherwise.
unsigned long long int TestPrimePower(unsigned long long int n) {
  unsigned long long int i,j;
  j = 0;
  i = 2;
  while ((i <= floor(pow(n, .5))) && (j == 0)) {
    if((n % i) == 0) {
      j = i;
    }
    i++;
  }
  for (unsigned long long int i = 2 ; i <= (floor(log(n) / log(j)) + 1) ; i++) {
    if(pow(j , i) == n) {
      return(1);
    }
  }
  return(0);
}

//This function computes the greatest common denominator of two integers.
//Since the modulus of a number mod 0 is not defined, we return a -1 as
//an error code if we ever would try to take the modulus of something and
//zero.
unsigned long long int GCD(unsigned long long int a, unsigned long long int b) {
  unsigned long long int d;
  if (b != 0) {
    while (a % b != 0) {
      d = a % b;
      a = b;
      b = d;
    }
  } else {
    return -1;
  }
  return(b);
}

//This function takes and integer argument, and returns the size in bits
//needed to represent that integer.
unsigned long long int RegSize(unsigned long long int a) {
  unsigned long long int size = 0;
  while(a != 0) {
    a = a>>1;
    size++;
  }
  return(size);
}


//q is the power of two such that n^2 <= q < 2n^2.
unsigned long long int GetQ(unsigned long long int n) {
  unsigned long long int power = 8; //256 is the smallest q ever is.
  while (pow(2,power) < pow(n,2)) {
    power = power + 1;
  }
  return 1 << power;
}

//This function takes three integers, x, a, and n, and returns x^a mod
//n.  This algorithm is known as the "Russian peasant method," I
//believe, and avoids overflow by never calculating x^a directly.
unsigned long long int modexp(unsigned long long int x, unsigned long long int a, unsign
  unsigned long long int value = 1;
  unsigned long long int tmp;
  tmp = x % n;
  while (a > 0) {
    if (a & 1) {
      value = (value * tmp) % n;
    }
    tmp = tmp * tmp % n;
    a = a>>1;
  }
  return value;
}

// This function finds the denominator q of the best rational
// denominator q for approximating p / q for c with q < qmax.
unsigned long long int denominator(double c, unsigned long long int qmax) {
  double y = c;
  double z;
  unsigned long long int q0 = 0;
  unsigned long long int q1 = 1;
  unsigned long long int q2 = 0;
  while (1) {
    z = y - floor(y);
    if (z < 0.5 / pow(qmax,2)) {
      return(q1);
    }
    if (z != 0) {
      //Can't divide by 0.
      y = 1 / z;
    } else {
      //Warning this is broken if q1 == 0, but that should never happen.
      return(q1);
    }
    q2 = (unsigned long long int)floor(y) * q1 + q0;
    if (q2 >= qmax) {
      return(q1);
    }
    q0 = q1;
    q1 = q2;
  }
}

//This function takes two integer arguments and returns the greater of
//the two.
unsigned long long int max(unsigned long long int a, unsigned long long int b) {
  if (a > b) {
    return(a);
  }
  return(b);
}

//This function computes the discrete Fourier transformation on a register's
//0 -> q - 1 entries.
void DFT(QuReg * reg, unsigned long long int q) {
  //The Fourier transform maps functions in the time domain to
  //functions in the frequency domain.  Frequency is 1/period, thus
  //this Fourier transform will take our periodic register, and peak it
  //at multiples of the inverse period.  Our Fourier transformation on
  //the state a takes it to the state: q^(-.5) * Sum[c = 0 -> c = q - 1,
  //c * e^(2*Pi*i*a*c / q)].  Remember, e^ix = cos x + i*sin x.

  Complex * init = new Complex[q];
  Complex tmpcomp( 0, 0 );

  //Here we do things that a real quantum computer couldn't do, such
  //as look as individual values without collapsing state.  The good
  //news is that in a real quantum computer you could build a gate
  //which would what this out all in one step.

  unsigned long long int count = 0;
  double tmpreal = 0;
  double tmpimag = 0;
  double tmpprob = 0;
  for (unsigned long long int a = 0 ; a < q ; a++) {
    //This if statement helps prevent previous round off errors from
    //propagating further.
    if ((pow(reg->GetProb(a).Real(),2) +
 pow(reg->GetProb(a).Imaginary(),2)) > pow(10,-14)) {
      for (unsigned long long int c = 0 ; c < q ; c++) {
tmpcomp.Set(pow(q,-.5) * cos(2*M_PI*a*c/q),
    pow(q,-.5) * sin(2*M_PI*a*c/q));
init[c] = init[c] + (reg->GetProb(a) * tmpcomp);
      }
    }
    count++;
    if (count == 100) {
      cout << "Making progress in Fourier transform, "
   << 100*((double)a / (double)(q - 1)) << "% done!"
   << endl << flush;
      count = 0;
    }
  }
  reg->SetState(init);
  reg->Norm();
  delete [] init;
}
```

D.8 qubit.C

Below is the much simpler code for a single qubit. You may find it entertaining and educational to write a simple driver program for this class to toy with a single qubit.

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "complex.h"
```

```
using namespace std;

class Qubit {
public:
  Qubit();            //Default constructor.
  virtual ~Qubit();           //Default destructor.
  int Measure();      //Returns zero_state = 0 or one_state = 1 in accordance
                      //with the probabilities of zero_state and one_state.
  void Dump() const;        //Prints our zero_state, and one_state, without
                      //disturbing anything, this operation has no physical
                      //realization, it is only for information and debugging.
                      //It should never be used in an algorithm for
                      //information.
  void SetState(const Complex& zero_prob, const Complex& one_prob);
                      // Takes two complex numbers and sets the states to
                      //those values.
  void SetAverage();  //Sets the state to 2^(1/2) zero_state, 2^(1/2)
                      //one_state.  No imaginary/phase component.
  double MCC(int state) const;  //Multiply the zero or one state by its complex
                        //conjugate, and return the value.  This value
                        //will always be a real number, with no imaginary
                        //component.

private:
  Complex zero_state;
  //The probability of finding the Qubit in the zero or all imaginary
  //state.  I currently use only the real portion.

  Complex one_state;
  //The probability of finding the Qubit in the one or all real state.
  //I currently use only the real portion.

  //|zero_state|^2 + |one_state|^2 should always be 1.
  //This notation means z_s * ComplexConjugate(z_s) + o_s *
  //ComplexConjugate(o_s) = 1.
};

//Qubit constructor, initially sets things in the zero state.
Qubit::Qubit() {
  zero_state = Complex(1,0);
  one_state = Complex(0,0);
  srand(time(NULL));
}

//Returns <state>_state * ComplexConjugate(<state>_state).
double Qubit::MCC(int state) const {
  if (state == 0) {
    return (pow(zero_state.Real(), 2) + pow(zero_state.Imaginary(), 2));
  } else {
    return (pow(one_state.Real(), 2) + pow(one_state.Imaginary(), 2));
  }
}

//Measurement operator.  Destructively collapses superpositions.
int Qubit::Measure() {
  double rand1 = rand()/(double)RAND_MAX;

  //This assumes that the sum of the squares of the amplitudes are = 1
  if (MCC(0) > rand1) {
    zero_state.Set(1,0);
    one_state.Set(0,0);
    return 0;
  } else {
    zero_state.Set(0,0);
    one_state.Set(1,0);
    return 1;
  }
}

//Outputs state info for our qubit.  For debugging purposes.
void Qubit::Dump() const{
  cout << "Amplitude of the zero state is "
       << zero_state.Real() << " +i" << zero_state.Imaginary() << endl
       << flush;
  cout << "Amplitude of the one state is "
       << one_state.Real() << " +i" << one_state.Imaginary() << endl
       << flush;
}

//Sets the zero and one states to arbitrary amplitudes.  Outputs
//an error message if the two values MCC'ed != 1 + 0i.
void Qubit::SetState(const Complex& zero_prob, const Complex& one_prob) {
  zero_state = zero_prob;
  one_state = one_prob;
  //Determine if |zero_state|^2 + |one_state|^2 == 1, if not we
  //are not in a valid state.
  double probab;
  probab = MCC(0) + MCC(1);
  if (fabs(probab - 1) > pow(10, -10)) {
    //This funny expression
    //allows us some rounding errors.
    cout << "Warning, total probability for in SetState is different from 1." << endl <<
  }
}

//Sets the qubit 1/2 way between the 0 state and the 1 state.  No phase.
void Qubit::SetAverage() {
  zero_state.Set(pow(2, -.5), 0);
  one_state.Set(pow(2, -.5), 0);
}
```

E Sample Output

E.1 Sample Output for $n = 17$

```
Welcome to the simulation of Shor's algorithm.
There are four restrictions for Shor's algorithm:
1) The number to be factored must be >= 15.
2) The number to be factored must be odd.
3) The number must not be prime.
4) The number must not be a prime power.

There are efficient classical methods of factoring any of the above
numbers, or determining that they are prime.

Input the number you wish to factor.
17
Step 1 starting.
Error, the number must not be prime!
```

E.2 Sample Output for $n = 15$

```
Welcome to the simulation of Shor's algorithm.
There are four restrictions for Shor's algorithm:
1) The number to be factored (n) must be >= 15.
2) The number to be factored must be odd.
3) The number must not be prime.
4) The number must not be a prime power.

There are efficient classical methods of factoring any of the above numbers, or determin

Input the number you wish to factor.
15
Step 1 starting.
Step 1 complete.
Step 2 starting.
Searching for q, the smallest power of 2 greater than or equal to n^2.
Found q to be 256.
Step 2 complete.
Step 3 starting.
Searching for x, a random integer coprime to n.
Found x to be 13.
Step 3 complete.
Step 4 starting.
Made register 1 with register size = 9
Created register 2 of size 4
Step 4 complete.
Step 5 starting attempt: 1
Step 5 complete.
Step 6 starting attempt: 1
Step 6 complete.
Step 7 starting attempt: 1
Step 7 complete.
Step 8 starting attempt: 1
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Step 8 complete.
Step 9 starting attempt: 1
Value of m measured as: 0
Step 9 complete.
Measured, 0 this trial a failure!
Steps 10 and 11 complete.
Step 5 starting attempt: 2
Step 5 complete.
Step 6 starting attempt: 2
Step 6 complete.
Step 7 starting attempt: 2
Step 7 complete.
Step 8 starting attempt: 2
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Step 8 complete.
Step 9 starting attempt: 2
Value of m measured as: 0
Step 9 complete.
Measured, 0 this trial a failure!
Steps 10 and 11 complete.
Step 5 starting attempt: 3
Step 5 complete.
Step 6 starting attempt: 3
Step 6 complete.
Step 7 starting attempt: 3
Step 7 complete.
Step 8 starting attempt: 3
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Step 8 complete.
Step 9 starting attempt: 3
Value of m measured as: 128
Step 9 complete.
Steps 10 and 11 starting attempt: 3
Measured m: 128, rational approximation for m/q=0.5 is: 1 / 2
Candidate period is 2
13^1 + 1 mod 15 = 14,
13^1 - 1 mod 15 = 12
15 = 3 * 5
Steps 10 and 11 complete.
```

E.3 Sample Output for $n = 33$

```
Welcome to the simulation of Shor's algorithm.
There are four restrictions for Shor's algorithm:
1) The number to be factored (n) must be >= 15.
2) The number to be factored must be odd.
3) The number must not be prime.
4) The number must not be a prime power.

There are efficient classical methods of factoring any of the above numbers, or determin

Input the number you wish to factor.
33
Step 1 starting.
Step 1 complete.
Step 2 starting.
Searching for q, the smallest power of 2 greater than or equal to n^2.
Found q to be 2048.
Step 2 complete.
Step 3 starting.
Searching for x, a random integer coprime to n.
Found x to be 8.
Step 3 complete.
Step 4 starting.
Made register 1 with register size = 12
Created register 2 of size 6
Step 4 complete.
Step 5 starting attempt: 1
Step 5 complete.
Step 6 starting attempt: 1
Step 6 complete.
Step 7 starting attempt: 1
Step 7 complete.
Step 8 starting attempt: 1
Making progress in Fourier transform, 4.83635% done!
Making progress in Fourier transform, 9.72154% done!
Making progress in Fourier transform, 14.6067% done!
Making progress in Fourier transform, 19.4919% done!
Making progress in Fourier transform, 24.3771% done!
Making progress in Fourier transform, 29.2623% done!
Making progress in Fourier transform, 34.1475% done!
Making progress in Fourier transform, 39.0327% done!
Making progress in Fourier transform, 43.9179% done!
Making progress in Fourier transform, 48.8031% done!
Making progress in Fourier transform, 53.6883% done!
Making progress in Fourier transform, 58.5735% done!
Making progress in Fourier transform, 63.4587% done!
Making progress in Fourier transform, 68.3439% done!
Making progress in Fourier transform, 73.2291% done!
Making progress in Fourier transform, 78.1143% done!
Making progress in Fourier transform, 82.9995% done!
Making progress in Fourier transform, 87.8847% done!
Making progress in Fourier transform, 92.7699% done!
Making progress in Fourier transform, 97.6551% done!
Step 8 complete.
Step 9 starting attempt: 1
Value of m measured as: 409
Step 9 complete.
Steps 10 and 11 starting attempt: 1
Measured m: 409, rational approximation for m/q=0.199707 is: 273 / 1367
Odd period found.  This trial failed.   Trying again.
Steps 10 and 11 complete.
Step 5 starting attempt: 2
Step 5 complete.
Step 6 starting attempt: 2
Step 6 complete.
Step 7 starting attempt: 2
Step 7 complete.
Step 8 starting attempt: 2
Making progress in Fourier transform, 4.83635% done!
Making progress in Fourier transform, 9.72154% done!
Making progress in Fourier transform, 14.6067% done!
Making progress in Fourier transform, 19.4919% done!
Making progress in Fourier transform, 24.3771% done!
Making progress in Fourier transform, 29.2623% done!
Making progress in Fourier transform, 34.1475% done!
Making progress in Fourier transform, 39.0327% done!
Making progress in Fourier transform, 43.9179% done!
Making progress in Fourier transform, 48.8031% done!
Making progress in Fourier transform, 53.6883% done!
Making progress in Fourier transform, 58.5735% done!
Making progress in Fourier transform, 63.4587% done!
Making progress in Fourier transform, 68.3439% done!
Making progress in Fourier transform, 73.2291% done!
Making progress in Fourier transform, 78.1143% done!
Making progress in Fourier transform, 82.9995% done!
Making progress in Fourier transform, 87.8847% done!
Making progress in Fourier transform, 92.7699% done!
Making progress in Fourier transform, 97.6551% done!
Step 8 complete.
Step 9 starting attempt: 2
Value of m measured as: 1438
```

```
Step 9 complete.
Steps 10 and 11 starting attempt: 2
Measured m: 1438, rational approximation for m/q=0.702148 is: 719 / 1024
Candidate period is 1024
8^512 + 1 mod 33 = 32,
8^512 - 1 mod 33 = 30
33 = 3 * 11
Steps 10 and 11 complete.
```