COM3401, Quantum Simulation and the Further Examination of Shor's Algorithm

Ross James Bevington
Department of Computer Science
University of Exeter

April 2005

Abstract

Within the next twenty years the computing industry may hit a barrier on today's computing technology. This barrier is not one that can be overcome with human ingenuity but is a physical one that may stop the continuing miniaturisation of electronic components.

Quantum Computing may be the answer to this, already algorithms have been developed to take advantage of strange quantum effects. These algorithms has been shown to be many hundreds of times faster than their classical counterparts being able to sort databases and crack encryption quickly.

However there is no actual quantum computer to experiment with, research in this field must be run on simulators running on classical hardware.

In this document I hope to develop a simulator and learn more from quantum algorithms and to explain some of the problems inherent to them.

Contents

1 Introduction 4
2 Background 4
2.1 The need for a quantum computer 4
2.2 The Benefits of Quantum Computing 5
2.3 Building a Quantum Computing 6
2.4 Summary 8
3 Preliminary Research 9
3.1 Past Papers 9
3.1.1 Peter Shor 9
3.1.2 Richard Josza 10
3.1.3 Mark Moore 10
3.1.4 David Crick 10
3.2 Shor's Quantum Factoring Algorithm 11
3.3 Existing Simulators 12
3.3.1 QCL 12
3.3.2 M. Hayward Factoring Simulator 13
3.3.3 jaQuzzi 14
3.4 Conclusions 14
4 Problem Description 17
4.1 Simulator 17
4.2 Research 18
5 Analysis and Design 19
5.1 Development Model 19
5.2 Programing Language 19
5.2.1 GCJ 20
5.2.2 Programming Techniques 21
5.3 Requirements 21
5.3.1 Primary 21
5.3.2 Secondary 21
5.4 Interface Design 22
5.5 Classes 22
6 Testing 25
6.1 Functionality Testing 25
6.2 Performance Testing 26
6.3 User Testing 26
6.4 Black and white box testing 27
6.5 Testing Evaluation 27
7 Findings 29
7.1 Efficiency 29

7.2 The Introductory Period 30
8 Critical Assessment 31

Appendices **34**

A qAbs Usage **35**
1 Example output 35
2 Compiling qAbs 35
3 List of factors composed of two primes 36

B QCL Source Code **37**

C Test Data **39**
1 qAbs 39
1.1 JVM 39
1.2 Native 42
2 M. Hayward 45
3 QCL 56

List of Figures

1 Transistors per square inch, [Intel Corporation, 2005] 5
2 Comparing Computations on classical and quantum hardware 6
3 Contents of register after $x^a \ mod \ n$ is performed 12
4 Contents of register one after register two has been measured 13
5 Contents of register one after Fourier Transform 14
6 Current simulators, based upon [Wallace, 2004] 15
7 Example QCL source code 15
8 Sample of source code from Hayward's simulator 16
9 Functionality of varies data structures 17
10 Spiral model used in the development of this project 20
11 Flow diagram detailing the structure of the user interface 23
12 UML Diagram showing components used in the simulator of Shor's algorithm 24
13 Results of testing 25
14 Simulator Performance Comparison 26
15 Tests performed by users on the interactive interface 27
16 collapse(container) method of the qContainer class 28
17 Results of white box testing 28
18 Register one after Fourier transform 29
19 Growth of q as n increases 30
20 Sequence generated in register two before measurement, n=33 x=9 . . 30

.

1 Introduction

Over the last decade there has been a lot of interest in new types of computable machines, due partly over the fears on conventional, classical, computing. Research by leading physicists, mathematicians and computer scientists has shown quantum computers could be the future, as they may be able to solve some computationally difficult problems much faster than is currently possible on a classical computer. Quantum computers could be able to break modern encryption or sort databases very quickly compared to classical computers.

Unfortunately the quantum effects that these computers must use make it very difficult to create a quantum computer, although there has been much work in the the last five years. We are still a very long way away before a usable quantum computer can be built, thats not to say one cannot be simulated. David Deutsch has already created a working model for a quantum computer, the quantum Turing machine or QTM Deutsch [1985]. This has been proved as a model for computation on a quantum and has allowed algorithms to be developed

Even though the mathematical foundations for quantum computing have been set there is still a slow adoption of quantum computing in general, with the number of quantum algorithms is relatively small, this is probably largely due to the multidisciplinary skills and different formalisms that are needed. Quantum computing is also without a structured quantum programming language that is complete enough to describe all quantum programs, this lack of a methodology is stifling growth in algorithm design.

Even though there are relatively few algorithms that could be run on quantum hardware, being able to simulate these will hopefully allow a methodology to be created. For a long time quantum simulation *will* be the closest many will come to a real quantum computer, but this does come with its own problems. Because of the overhead even the fastest quantum algorithms will run at exponential time and memory which can mean research over large inputs is very time consuming.

It is the aim of this project to construct a quantum simulator capable of simulating Peter Shor's quantum factoring algorithm and abstract enough to allow future developers to construct and test other algorithms inspired by Shor's approach. From using my simulator I hope to gain new insight into Shor's algorithm and explain some of the reasons behind its efficiency and failings.

2 Background

2.1 The need for a quantum computer

In 1965 Gordon Moore, one of the original founders of Intel Corporation, noted that every year the number of transistors on an integrated circuit board doubled, yet the space required remained the same [Moore, 1965]. Moore's law as it was later known has continued to the present day and is the main reason for the strong growth in the computing industry.

However even Moore admits this growth cannot continue:

> "In terms of size [of transistor] you can see that we're approaching the size of atoms which is a fundamental barrier, but it'll be two or three generations before we get that far - but that's as far out as we've ever been able to

Year	*Transistors*
1974	5,000
1978	29,000
1982	120,000
1985	275,000
1989	1,180,000
1993	3,100,000
1997	7,500,000
1999	24,000,000
2000	42,000,000

Figure 1: Transistors per square inch, [Intel Corporation, 2005]

> see. We have another 10 to 20 years before we reach a fundamental limit"
> Gordon Moore [Manek Dubash, 2005]

The fundamental limit Moore mentions is the level of the nanometre, once electronic components are at the point where they are constructed from only a few atoms their behaviour cannot be guaranteed. This is because at this level physics is not longer governed by classical mechanics but quantum mechanics, at this level logic gates that previously would only allow true or false now are able to give both simultaneously

When the behaviour of these logical gates cannot be guaranteed the algorithms, the operating system and programs built on it will not function or will produce undesirable side affects.

Because of this manufactures will no longer be able to build faster and bigger computer and memory chips, this will have the affect of causing the specifications on computer systems to plateau. But the computing industry has preempted this and has poured vast amounts of money into research and development of future technologies that could allow us to push this barrier further into the future or remove this all together. The most promising of these seems to be quantum computing, exploiting the very affects that limit the computing industry seems a fitting way to push the computing industry into a new era of computing.

2.2 The Benefits of Quantum Computing

A quantum computer is a theoretical computing device put forward initially by Richard Feyman during at time when scientists were studying the limits of computation [Feynman, 1985]. As discussed previously packing more and more information into smaller and smaller spaces cannot continue indefinitely once this reaches the level of the nanometre, classical physics no longer applies. Feyman and others wondered if these quantum effects could be used to create a new computing device, a quantum computer. They hoped that by using quantum mechanics they could not only make computers on a much smaller scale than available today but also use quantum properties to make computation more efficient.

The implication of the creation of a fully functional quantum computer is great, both Shor and Deutsch have claimed that NP-Complete problems may be solvable in polynomial time on a quantum computer [Deutsch, 1985] [Shor, 1994]. This is a significant claim as NP-Complete problems can be mapped onto each other so if a solution is found to one of these then a solution to the whole group can be found. If

this prediction is true quantum computing could revolutionise the computing industry allowing previously intractable problems such as the travelling salesman problem to be solves in a relatively short time.

A number of people have said that it may also be possible to crack todays public key encryption in seconds using a quantum computer, this has important consequences for governments, banks and almost all industries that want to safe guard their data, a quantum computer would make this impossible.

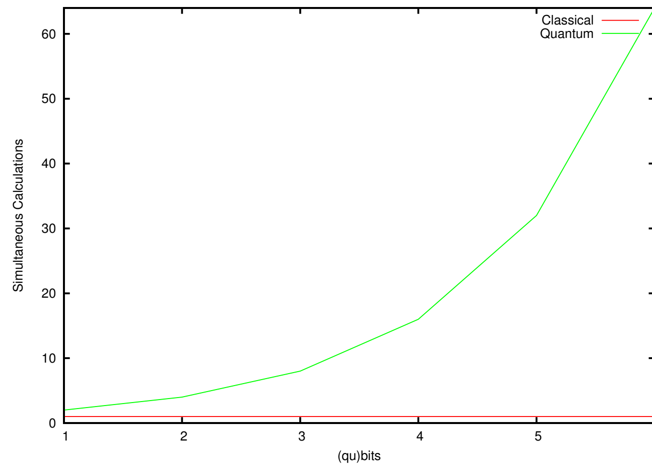


Figure 2: Comparing Computations on classical and quantum hardware

The ability for a quantum computer to compute much more efficiently comes from the principle of superposition, in essence by exploiting proposed parallel universes quantum computing enables variables to take on multiple values at the same time. This enables a quantum computer to use massive parallelism, this is the key to quantum computing, if algorithms can exploit this then they can become much more efficient than their classical counterparts.

Figure 2 shows the processing power of one classical computer to one future quantum computer. As the quantum computer can process multiple values at the same time it immediately out paces a classical computer. This massive parallelism should allow for fast more efficient algorithms to be run which normally would never complete on a classical computer.

2.3 Building a Quantum Computing

Already a mathematical model of a quantum computer has been created, in 1985 David Deutsch proposed the first true QTM, quantum Turing machine [Deutsch, 1985]. This has been used as mathematical base for research and proposed algorithms however we are still very far away from a physical quantum computer.

A large amount of research is going on in the physics community, already NMR (nuclear magnetic resonance) has been able to successful run two quantum algorithms, Shor's factoring algorithm and Lov Grover's database sorting algorithm [Steffen et al., 2001]. Although these approaches are good for showing the quantum computing is in fact possible and the algorithms do work the technology that this method uses is not extendable, it is doubtful that any future quantum computer will use this approach.

Recent approaches such as quantum dots which use *caged* electrons inside a ring of atoms show promise, but this technology is not mature enough to create a usable or reliable quantum computer. With a possible quantum computer many years away simulation of a quantum computer is the only possible way to experiment with a quantum computer.

The main limitation in quantum computing is that it is not possible to view the result of a calculation before the algorithm has been completed. The contents of each variable can be though of as being in a separate parallel universes, these cannot be viewed or measured without effecting the others, in fact measuring a variable halfway through a computation may cause the algorithm to fail. It is not just the programmer that can cause a variable to be measured, any outside force such as a stray electron, photon, atom etc can disrupt the delicate system inside the quantum computer. Because of this the computer itself must be kept completely isolated from any outside interference, this is perhaps the most difficult hurdle, one that physicists have yet to overcome.

In order to gain a more comprehensive understanding of quantum computing in general it is useful to have some knowledge of the various quantum effects that a real quantum would use and that I intend to simulate, these are outlined below. For a further explanation of these please see [Hey and Walters, 2003].

Superposition

If the state of a particle is not known then the principle of superposition states that it is in all states simultaneously; a state is a description of the properties of a particle at any given time. If any kind of measurement is made upon the particle that is in superposition then the state collapses to one of its possible states, in this way quantum physics allows for particles to have multiple states concurrently.

Imagine a box that is subject to the laws of quantum physics, if the contents of the box is unknown then quantum physics dictates it must be in all possible states simultaneously.

Entanglement

Particles that have interacted with each other retain a link between them and can be entangled. These particles can then be separated and measured, measuring one particle causes the state of the other particle to instantly change. Entanglement allows for action at a distance, no matter how great the distance between two entangled particles knowledge of the measured state will propagate to the other entangled particle causing its state to change. By using entanglement knowledge can effectively be passed faster than light and it is possible for particle to teleported.

Entanglement is a difficult concept to understand, it tries to explain how objects with no connection can pass information on what they have been measured to be instantly between each other.

Randomness

At the heart of quantum mechanics is pure randomness, it is *impossible* to say with complete accuracy the state to which a particle in superposition will collapse to, all that can be given is a probability that describes the states it could fall into this is called a wave function. Because classical computers are unable to produce a random number

a source exhibiting quantum effects of a quantum computer can be used to generate true random numbers.
The nature of quantum physics it would seem that any type of computation is not suited to the random world in which a quantum computer would inhabit. Although this randomness means that the correct answer or even an answer at all may not always be obtained superposition and entanglement do allow for vast parallelism that could not even be recreated today.

By using the properties of quantum mechanics stated earlier it is possible to create two fundamental quantum data types, the qubit and the quantum register. These are used in almost all the quantum algorithms I have investigated.

The Qubit

The qubit can be thought of as the counterpart to the bit in classical computing, however in classical computing there is only two states, zero and one, where as a qubit can have a mixture of the two described by a probability. A qubit can have both states simultaneously because of superposition, this allows both values to exist at the same time and be worked on simultaneously.

This is an incredibly powerful concept, and is how a quantum computer can use immense parallelism as shown in Figure 2.

The Quantum Register

A quantum register is a collection of qubits, and is a data structure is commonly used to store a superposition of integers in byte form.

A register has a number of important properties, firstly it can be put into a superposition of values, secondly it can be entangled with other registers and finally operations can be performed upon the register. If the contents are in a state of superposition and the register is processed, for example used in a function, but not measured the superposition is maintained.

2.4 Summary

Although the ability to compute with a quantum computer has been shown there is still no realised quantum computer. Even with current advances in quantum dots progress has been slow with the ability to create a fully featured quantum computer moving further and further into the future.

Without the availability of a simulator the ability to develop algorithms that could take advantage of quantum effects such as superposition and entanglement to solve NP-Complete problems is severely limited. The importance of a quantum computing simulator is therefore quite high.

3 Preliminary Research

The ability to compute on a quantum computer has already been proven by Benioff, Feynman and Deutsch, with the creation of a QTM. The QTM has provided a new platform on which quantum algorithms can be run, both Peter Shor and Lov Grover have proposed algorithms that are able to be run on this and have shown to be more efficient than their best classical counterparts.

Although these algorithms have been put forward there has been relatively little exploration into the working of these algorithms. There has been successful work into testing both Shor's and Grover's sorting algorithm by physicists working with NMR technology, but no work into why these algorithms work has been done.

Shor and Grovers algorithms were proposed almost ten years ago, since then there has few new quantum algorithms, the reason for is most likely due to a lack of methodology for their creation. By subjecting the existing quantum algorithms to heavy examination the computer science community may be able to create a methodology to program new quantum algorithms.

I will be examining Shor's factoring algorithm along with simulators capable of running this algorithm.

3.1 Past Papers

The following sections summarise the findings of the few papers that focus on Shor's algorithm.

3.1.1 Peter Shor

In Shor's paper he theorises that NP-Hard or NP-Complete problems could be solved in polynomial time on a quantum computer [Shor, 1994] . NP-Complete problems are mappable in that if a solution to one can be found in polynomial time then a solution to all in the class can be found in polynomial time. If such an algorithm were found it would open new possibilities in computer science, Shor has likened it to finding the holy grail for computer science.
His factorisation algorithm is undoubtedly the most famous algorithm capable of running on a quantum computer, his paper explains in great detail the states a quantum computer must be manipulated into to run his quantum algorithm, details of which are described in the following section. The paper also proposes a similar way to compute the discrete logarithm of a number, to date no simulator has been written that can simulate this algorithm nor has much further research been conducted.
The discrete logarithm algorithm is interesting in that it is the only other algorithm that uses the method seen in the factoring algorithm, the reasons for lack of interest in this algorithm is probably due to the complex mathematics involved. Shor uses complex terminology and methods from group theory which many of the physicists and computer scientists have much less background in. This may be the reason why his prime factorisation algorithm was much more celebrated than the discrete logarithm algorithm.

This has meant there are many more sources of information on prime factorisation, in fact it seems there are only three sources of information on the discrete logarithm algorithm. The first being Shor's original paper, his extended paper[Shor, 1997] and Richard Josza's paper which analyses Shor's paper in greater depth[Jozsa, 2000].

Surprisingly discrete logarithms are used in the RSA encryption scheme just like prime factorisation, researchers who are hoping to be able to break much of the encryption in use today should seriously look at the use of discrete logarithms.

3.1.2 Richard Josza

Richard Josza's paper on the algorithms proposed in Shor's paper is perhaps the best exploration on Shor's algorithm available [Jozsa, 2000]. The paper works through each stage of both the factoring algorithm and the discrete logarithm algorithm.

From my research it seems that this paper is the only not written by Shor that examines the discrete logarithm algorithm in enough depth to allow the reader to implement the algorithm.

The paper goes on to generalise both algorithms and prove that Shor's technique for solving problems using periodic functions can be applied to any finite group, this is called the abelian hidden subgroup problem.

3.1.3 Mark Moore

In Mark Moore's research paper entitled 'Quantum-inspired Algorithm and a Methodology for their construction' he puts forward an algorithm which is more efficient than Shor's algorithm for finding prime factors and even factors [Moore, 1995], his algorithm seems to be the first to actively use Shor's quantum factoring algorithm as a base for future development.

Moore's algorithm works by measuring what he calls the introductory period, a short set of numbers that sometimes appear in the first part of Shor's algorithm. His work states that by measuring this and forming another sequence based on this algorithm can be used to determine odd and even factors. The problem of the introductory period is examined in much more detail in section 7.2.

Although his algorithm seems sensible enough the contents of the register is in a state of superposition, Moore takes many measurements of this register without explaining how it continues to be in superposition. Because of this his algorithm would not be able to run on a quantum computer and therefore can only be said to be quantum inspired, as the title of paper suggests.

3.1.4 David Crick

In 1999 David Crick wrote a simulator with the goal of proving that Shor's factoring algorithm did use a exponential amount of memory. While this is true on classical hardware it is still unsure whether this is true on quantum hardware. The simulator written for the project was fast, being able to factor a nine digit number in only a few days, although this seems a long time the hardware he used makes this a relatively fast simulation. Crick's work claimed to have results to prove that the memory requirements did increase exponentially however I believe the results from his project may be unsound.

Firstly the simulator he used to obtain these results used the visualisation of Shor's algorithm [Narayanan et al., 1999], there are two main differences between the visualisation and the actual algorithm. The visualisation does not use probabilities or a Fourier transform, essential ingredients. Crick's simulator was also mentioned in 'Quantum Computer Software' in which Wallace says it does not keep any quantum state, thus

the simulator did not seem to run to use any of the fundamentals of a quantum computer such as superposition and entanglement.

Crick's paper fails to take into account that most of the *space* used would in fact be held within superposition inside a register. Unfortunately because of this the results found by Crick cannot be relied upon as the simulator he was using could be deemed classical rather than quantum.

In this project I hope to use my simulator to examine parts of Crick's paper in further detail as his conclusion may still be valid and should be explored.

3.2 Shor's Quantum Factoring Algorithm

A prime factor is a number made from the product of two prime numbers. For example 15 is a prime factor and is made from the product of 3 and 5, 15 = 3 * 5. Shor's algorithm is given a number made of two primes will produce the factors if run enough times. Although this seems simple enough it is computationally expensive with the best running times on a classical computer being exponential. Because factoring in to primes is very difficult it is used a lot in encryption the most famous scheme being RSA.

Shor's algorithm is based upon Euler's theorem, part of number theory [Wolfram Research Inc, 2005]. Shor used this theorem and the massive parallelism inherent in a quantum computer to find a metaphorical needle in a haystack.

Firstly a number of variables dependant of the number to be factored must be found, this can be done classically using know fast algorithms. A number, q, must be found which is the smallest power of two possible to hold n then a random number, x, is calculate. This random factor must be co prime to the factor.

Once there two variable are found two quantum register are set, the first is put into a superposition of integers between zero and q, the second is set to zero. Now the function $x^a \ mod \ n$ is performed on register one and the result placed in superposition in register two, as register one is in a state of superposition this calculation takes place on all values held inside the register simultaneously. Figure 3 shows the result of this[1].

By measuring the second register a repeating pattern is projected into the first, this now holds only values of $x^a \ mod \ n$ that are equal to the measurement in the second register. This is done through entanglement, mentioned earlier in the document.

The period in the first register is what is required by the algorithm to work out the factors but this cannot be measured in one measurement, it requires at least two, possibly more. Because the values held in the first register are held in superposition only one measurement can be made.

To get around this a quantum Fourier transform can be used to peak register one multiples of the period. By measuring register one this may reveal either the period or a multiple, however there is a chance that nothing or that an incorrect value could be measured because of the probabilistic nature of superposition.

Figure 5 shows that although the multiples of the frequency are peaked with a very high probability values, values very close to it also have a small but possible chance of being measured. This can be seen by the fall off centred around the peaks.

Providing that a correct value for the period is measured then it is a simple to form a continued fraction to find the base period. Once the period has been found

[1]These diagrams were inspired those found in [Williams and Clearwater, 1998] but taken from the debug output of qAbs

Figure 3: Contents of register after $x^a \ mod \ n$ is performed

$gcd(x^{r/2}, \ n)$ will give one factor.

3.3 Existing Simulators

Figure 6 has a list of the simulators available today, this is based upon the results in Wallace's quantum simulator review [Wallace, 2001] [Wallace, 2004]. I have updated the list so that it reflects more accurately the simulators available today, simulators that are highly incomplete or no longer available have been removed.

As Figure 6 shows there has been little work on the Java platform to create a quantum computer simulator, it is also interesting to note that only two simulators are currently in active development with most of the others projects related to academic research and discontinued after a short time of development.

I will be investigating three of these simulators, QCL, M. Hayward's simulator and jaQuzzi. I chose these as the source code is available and all are implemented very differently from each other.

3.3.1 QCL

QCL[2] is an open source quantum simulator and perhaps the most mature available. The simulator is written by Bernhard Ömer and provides simulation of programs written in a C like language called QCL, this is an interpreted language and is the only attempt at creating a high level language for the construction of quantum algorithms so far.

Figure 7 is an example of the kind of source that QCL can parse, it is a functional language which closely resembles C in which all operation on quantum registers must be previously defined. Dealing with registers is made relatively simple as the language takes care of any in between states to make sure a computation collapses successfully.

QCL is a very good simulator but it relies a lot on a background of functional programing. The language itself does have a number of flaws the major one being that it is not comprehensive enough for all the currently known quantum algorithms to be

[2] http://tph.tuwien.ac.at/~oemer/qcl.html

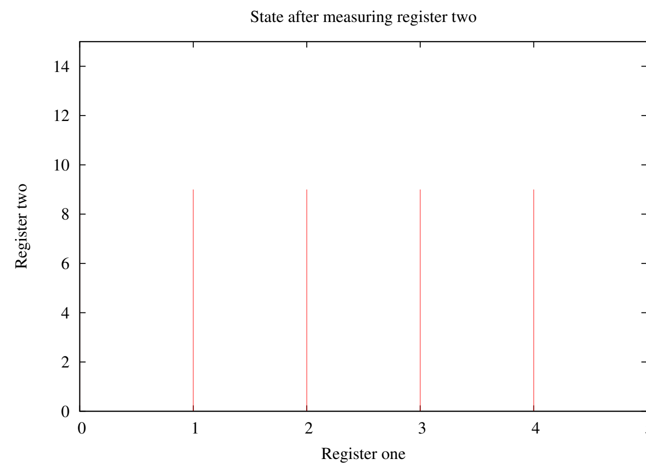


Figure 4: Contents of register one after register two has been measured

written in QCL. Further development of this project could allow a methodology for the creation of quantum algorithms to develop.

The simulator is much slower than any simulator built solely for an algorithm, this is to be expected as input must be parsed and interpreted, because of this QCL is more of a conventional simulator than any other program available to simulate a quantum algorithm.

3.3.2 M. Hayward Factoring Simulator

The structure of most of the simulators available are similar in to the design implemented in Matthew Hayward's simulator[3].

Almost all of the simulators available are written in C or C++ like Hayward's simulator, this allows for much faster execution. Although this is not always necessary as Shor's algorithm suffers from exponential growth on classical architecture.

Although the speed is a useful feature of these kind of algorithms the language chosen can make it difficult to initially learn how Shor's algorithm works. For example Shor's algorithm uses only two registers however Hayward's simulator must make used of many temporary arrays which support his register classes. These store the data relating to entanglement and superposition. This results in the program is more difficult to understand than QCL as the contents of each register can be spread over three arrays at some points, Figure 8 shows and example of this.

On further inspection of the source code of the simulator there are some points in which the state of the system has been reproduced incorrectly, for example the qubit class uses two variables to store the states but then only goes onto use one, the state of the entire system is not assured to be a reflection of any quantum system because of this.

[3]`http://alumni.imsa.edu/\~matth/quant/index.html`

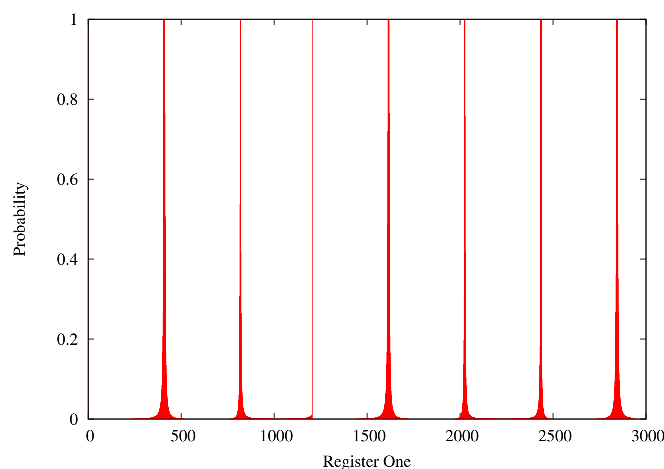


Figure 5: Contents of register one after Fourier Transform

3.3.3 jaQuzzi

As noted in the simulator table, Figure 6, jaQuzzi is the only available simulator to be written in Java. The available documentation for the simulator explains that it enables the user to construct algorithms using quantum gates. Because of this it should be able to simulator Shor's algorithm with the current quantum circuit diagrams [Beauregard, 2003].

JaQuzzi was written by Felix Schuermann in 2000 as part of a Masters thesis, the website for jaQuzzi[4] allows the simulator to be downloaded in a variety of formats. The author allows for the source code, precompiled binaries and an applet to be downloaded from the website. Unfortunately I have not been able to run any of these formats, both the applets and jar file immediately crash before the application starts and the source file does not compile cleanly. As the Java language was designed to be backward compatible with previous versions and because of this and the errors in the source file it is difficult to tell what state the program was in, or whether it was even runnable.

Because of this I have been unable to find no other Java simulator capable of factoring Shor's algorithm. I have not taken Java applets into account because all of the ones I have seen attempt to visualise Shor's algorithm.

3.4 Conclusions

Shor's pioneering paper on factoring and logarithms has been heavily cited but almost no work has been done to explain in detail why the algorithm works. All the papers that I have been able to read on the topic of Shor's algorithm are only able to explain the steps required to run the algorithm. It is possible that the discrete logarithm algorithm in Shor's paper has never even been simulated outside of Shor's development process, considering that discrete logarithms are as useful to encryption as factoring it is strange that no paper has been written with the goal of any further analysis of this algorithm. It is also interesting to note that the amount of mature simulators has fallen from those

[4] http://www.eng.buffalo.edu/~phygons/jaQuzzi/index.html

Name	*Language*	*Status*	*Source*	*Algorithms*	*Comments*
UH Quantum Computer	Java Applet	Final	No	Grover's	Graphical Java applet
Quantum-Entanglement	Perl Library	Final	Yes	Can be used to implement Shor's algorithm	
QuaSi	Java Applet	Unknown	No	Shor's, Grover's, Deutsch-Josza	Graphical Java applet
OpenQUACS	Maple	Final	Yes	Unknown	
CS 596	Matlab	Final	Yes	Shor's	
QuCalc	Mathmatica	Final	Yes	Quantum Circuits	
Universal Quantum Computation Simulator	Unknown	Final	No	Shor's, Grover's	Proprietary
Quantum Computer Simulator	C++	Final	Yes	Unknown	Library Functions
QCE	Unknown	In Development	No	Shor's, Grover's, Deutsch-Josza	
OpenQubit	C	Final	Yes	Shor's	
M. Hayward's Quantum Simulator	C++	Final	Yes	Shor's	
QCL	C++	In Development	Yes	Shor's, Grover's	Most Active Simulator found
Finite State Machine (FSM) Simulation	C	Final	Yes	Grover's	Paper with sample code
Qubiter	C++	Final	Yes	Quantum Circuits	covered by a US Patent
QCompute	Pascal	Final	Yes	Quantum Circuits	
Quantum	Unknown	Final	Yes	Quantum Circuits	
Q-gol	CaML., TcL	Final	Yes	Shor's algorithm	
QTM simulator	C++	Final	Yes	Quantum Turning Machine	
Quantum Qudit Simulator	Visual Basic	Final	Yes	Quantum Circuits	Not found it [Wallace, 2004]
jaQuzzi	Java	Final	Yes	Unknown	Not found it [Wallace, 2004]

Figure 6: Current simulators, based upon [Wallace, 2004]

```
procedure shor(int number) {
  int width=ceil(log(number,2));// size of number in bits
  qureg reg1[2*width];              // first register
  qureg reg2[width];                  // second register

...

  print "chosen random x =",x;
  Mix(reg1);                              // Hadamard transform
  expn(x,number,reg1,reg2);  // modular exponentiation
  measure reg2;                     // measure 2nd register
  dft(reg1);                              // Fourier transform
  measure reg1,m;                 // measure 2st register
  reset;                                      // clear local registers
```

Figure 7: Example QCL source code

```
    for (int i = 0 ; i < q ; i++) {
      //We must use this version of modexp instead of c++ builtins as
      //they overflow when x^i > 2^31.
      tmpval = modexp(x,i,n);
      modex[i] = tmpval;
      mdx[tmpval] = mdx[tmpval] + tmp;
    }
   reg2->SetState(mdx);
```

Figure 8: Sample of source code from Hayward's simulator

available during Wallace's simulator review. This could be a sign that QCL is emerging as the most popular simulator, this problem with this is QCL is not mature enough to describe all of the few quantum algorithms.

4 Problem Description

Shor's algorithm is one of the most famous algorithms capable of running on a quantum computer and one of a few to be actually tested using NMR [Steffen et al., 2001]. However much of Shor's proposed work is not well understood, his discrete logarithm algorithm for example may not have even been simulated.

In this project I will develop a simulator that is able to run Shor's factoring algorithm yet abstract enough to allow future algorithms to be run on it simulator with little to no modification of the main source files. I will also be using my simulator to examine Shor's factoring algorithm in much greater depth, examining the causes for it failures and its efficiency.

4.1 Simulator

I have chosen to develop a simulator in the Java programing language, this will allow my simulator to work on a wide range of platforms

Most if not all of the simulators today with the exception of QCL are not expendable to other quantum algorithms. One of the main reasons behind this is that the programming used does not resemble the quantum data structures they are meant to represent, for example Hayward's simulator does not have a complete class for the representation of a quantum register. I will try to solve this by creating an abstract class than can be used to model a quantum register and is expendable to be used with other possible quantum data structures.

I propose the idea of a quantum container, acting like a programming black box. The quantum container will allowing input, output, superposition and entanglement. As a quantum register does it could hold a range of integers in superposition and be linked to other registers by entanglement. Further than this, other data types could easily be stored; further more the programmer would need no knowledge of setting the superposition or maintaining it after collapse of a depending container.

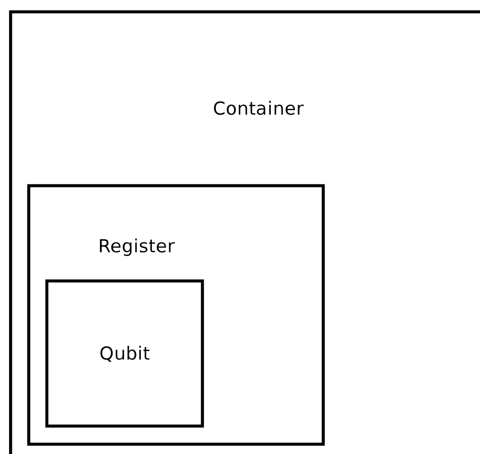


Figure 9: Functionality of varies data structures

Figure 9 show the functionality of the various data structure available in my simulator. A register now can be though of a subset of the functionality of a container, where a register can store only boolean values a container could plausibly hold any

data structure that memory and quantum error would allow.

By implementing a quantum container this should make my program much easier to understand. Functions that are needed only on registers can be removed, such as the Hadamard transform, this will make Shor's algorithm more abstract and allow very different algorithms to reuse the same data structure. Like QCL my simulator will also hide intermediate states from the programmer, this should make the structure of the algorithm much easier to understand unlike the source code for Hayward's simulator.

With the removal of the functions only needed for quantum registers and the simplification of the processes required to maintain the data structures used Shor's factoring algorithm the source code for my program should be much easier to understand than Shor's algorithm and form perhaps the only working Java quantum simulator.

4.2 Research

There is very little research on the inner workings of Shor's factoring algorithm, why the algorithm fails or why is completes quicker for some values of the random x factor. I will be using my simulator to attempt to find answers to these problems, a simulator is able to view the intermediate states that a quantum computer would store in superposition.

I will divide my research into a number of sections, firstly does Shor's algorithm sometimes fail to factorise correctly. I will investigate the contents of the registers during the computation and the introductory period mentioned by Moore. Secondly, I will be analysing as to why Shor's algorithm can complete quickly for one value of the random factor x but relatively slowly for another value.

5 Analysis and Design

The following section describes the software design methods used in the creation of my simulator including reasons as to why these methods were chosen.

5.1 Development Model

The waterfall model is a popular way to structure the development of a software project. It consists of a number of stages with distinct goals which once completed are never returned to, the main disadvantage of the waterfall model is that is does not allow many changes or revisions. My project is not suited to the waterfall model, because of its research led nature the design is liable to change drastically depending on what is developed and found through research.
Because of this I plan to adopt the spiral model for the development of my simulator this model is very good for large and complicated projects. It combines the waterfall model with features of the prototype model which allows for the design to be changed in light of new feedback. The spiral model has a number of clear steps,

1. The requirements are defined in as much detail as possible
2. A preliminary design of the system is created
3. A prototype of from this design is create which usually lacks many complex features.
4. By evaluating the prototype and comparing it to the requirements a second prototype is designed
5. The existing prototype is evaluated in the same manner as was the previous prototype, and, if necessary, another prototype is developed from it.
6. Once the system has satisfied the requirements it is refined and the final system is designed
7. The system is evaluated and tested with maintenance carried out on a continuing basis

Because of the time constraints and short development time on this project it may not be possible to stick rigorously to the steps in a conventional spiral model. Because of this I have designed my own spiral model based upon the above.

Figure 10 shows the waterfall model that I will be using. I have chosen to develop a linear simulator then possibly two simulator prototypes, The linear simulator should allow me to implement Shor's algorithm quickly using the visualisation given in [Narayanan et al., 1999], by evaluating this I may be able to preempt problems that may occur in the full quantum simulation which is much more complex.
The simulator prototypes will be based on the requirements in the next section.

5.2 Programing Language

The first stage of the development process was to choose a suitable programming language. Having programed large systems before with Java made this language one of the most obvious choices for the implementation of my simulator. Programming the

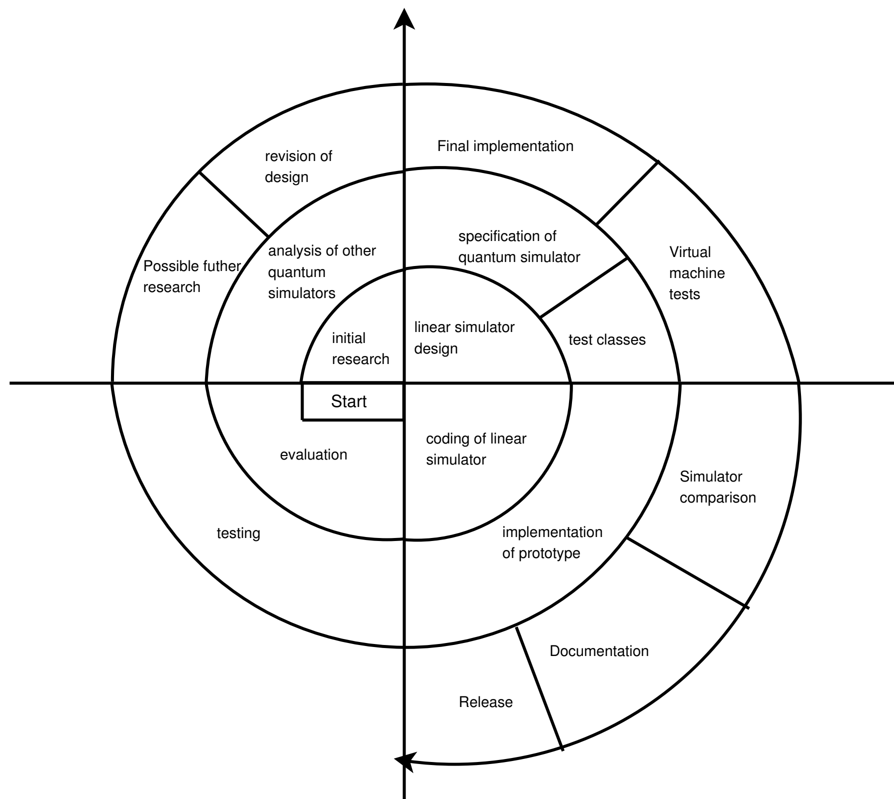


Figure 10: Spiral model used in the development of this project

simulator in Java allows for it to be run on many different platforms, this is because all code is run on a virtual machine.

However I was concerned that there was no other working quantum simulator written in the Java language, most of the other simulators were written in C or C++; two very efficient languages. An object orientated language was needed for the implementation of my simulator so C was removed from consideration as this in not object based.

After developing my linear simulator in Java I believe it is suitable for quantum simulator construction, even though the fast execution speed would be a bonus if the simulator was written in C++ the learning curve and memory management would slow down developments.

5.2.1 GCJ

After choosing Java as the programming language to be used in my project I found that the Java language can be compiled to byte code using a tool called GCJ.

GCJ[5] is part of the open source compiler GCC this allows Java source and class files to be compiled directly to machine code. By compiling the source code for my simulator using GCJ it may be possible run the simulator with execution times similar to those simulators written in C and C++.

[5] http://gcc.gnu.org/java/

5.2.2 Programming Techniques

For this project I will be using a set style in which I will be programming in, outlined below. The reason for this is to aid future development by allowing others to quickly understand the source code presented in my simulator. Some of this techniques will also allow my code to be reused by people in the future, this is important as there is currently no working simulator written in Java that can be used for any kind of base for future projects.
Debug Responses, Any debug method or methods that would not be acceptable in a quantum computer must display a warning to the user and explain that if this was used on a real quantum computer the result would corrupt the algorithm.
Object Orientation, To aid in development an object orientated structure will be adopted, and quantum classes should not be mixed with classical and a strict class naming convention will be used. All classes that deal with quantum behaviour will start with the letter 'q' this will mean the differences between classes are easily seen.

5.3 Requirements

I have set two groups for my requirements, the first set are primary requirements for my program to be considered successful and the second set secondary. I have done this to allow rapid development of a working simulator that can then be used to research Shor's factoring algorithm, another of my project's aims.

This is because the development time of my project may increase rapidly leaving me no time to focus on the research goals of my project, by discarding the secondary goals I should still have time to throughly research Shor's factoring algorithm.

Many of these requirements are based upon the results of the quantum simulator review

5.3.1 Primary

- The simulator should be written in the Java language.
- The simulator should be capable of running Shor's factoring algorithm and hold a quantum state.
- The simulator should have an interface to allow Shor's algorithm to be run interactively.
- The simulator should be designed so that future variations of Shor's algorithm can run with minimal changes.
- The simulator should allow easy debugging of quantum data structures.

5.3.2 Secondary

- The simulator should be able to take command line arguments.
- The simulator should be able to factor the number fifteen within one second.
- The simulator should allow the factorisation of a three digit number within a reasonable amount of time.
- The simulator should allow graph of registers to be generated.

- The simulator should be faster than QCL over a range of inputs.
- The simulator should be compiled into native byte code and tested, using GCJ.

5.4 Interface Design

Figure 11 shows the conceptual design for the interface of my simulator, this interface will allow the user to either set the input interactively or provide arguments on the command line.

The front end is planned to be console based but as the system is separating into distinct classes there is room for future development of a graphical user interface.

5.5 Classes

As said previously the structure of my simulator will be separated through a number of distinct classes. Figure 12 shows a UML diagram showing some of these classes as used in simulating Shor's factoring algorithm.

An outline of each class is also given here, this includes classes that are not used in context of Shor's factoring algorithm

complexNum: This class implements a complex number, a complex number is in the form of a+bi where a and b are real numbers and i is the square root of negative one.

qubit: Although the concept of qubits is not used directly in my simulator it is useful to implement this class for use in future algorithms. The qubit is made up of a complex number, which holds its state, and a series of methods that enforce the integrity of this state. The probability of each of the two states is given by the square of both the parts of the complex number state, this must be equal to one.

utility: This class contains frequently used classical functions such as calculating the greatest common divisor for two integers, and finding the variables q and x.

function, orFunction, shorFunction: The abstract function class and its sub classes orFunction and shorFunction allow for an extendable system. By creating a new sub class different functions can be designed and used without changing any code in the qContainer class. Although the orFunction is not used in my simulator it is provided as a proof of concept of this idea for future development.

qContainer: This is my abstract quantum data structure, it should be able to mimic most commonly used data structures. For the contents of this simulator it is being used as a quantum register holding integers. This class also implements a discrete Fourier transform based on the one found in Hayward's simulator.

shor: This class implement Shor's factoring algorithm, it uses qContainer's are registers and links them using the shorFunction class.

main: This class contains the user interface for the simulator and runs the shor class.

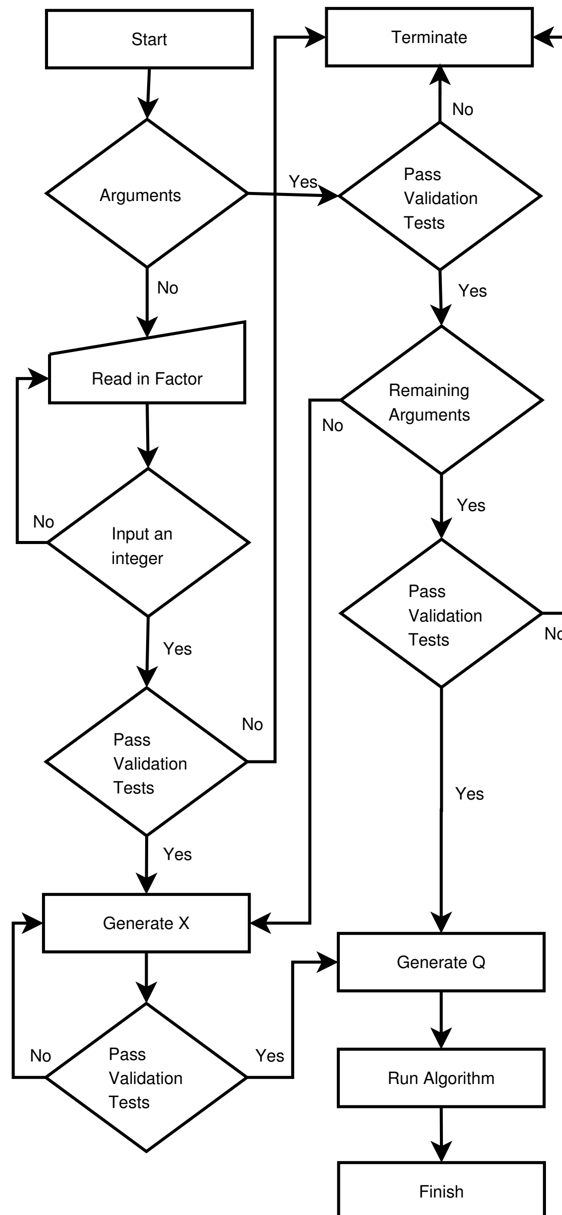


Figure 11: Flow diagram detailing the structure of the user interface

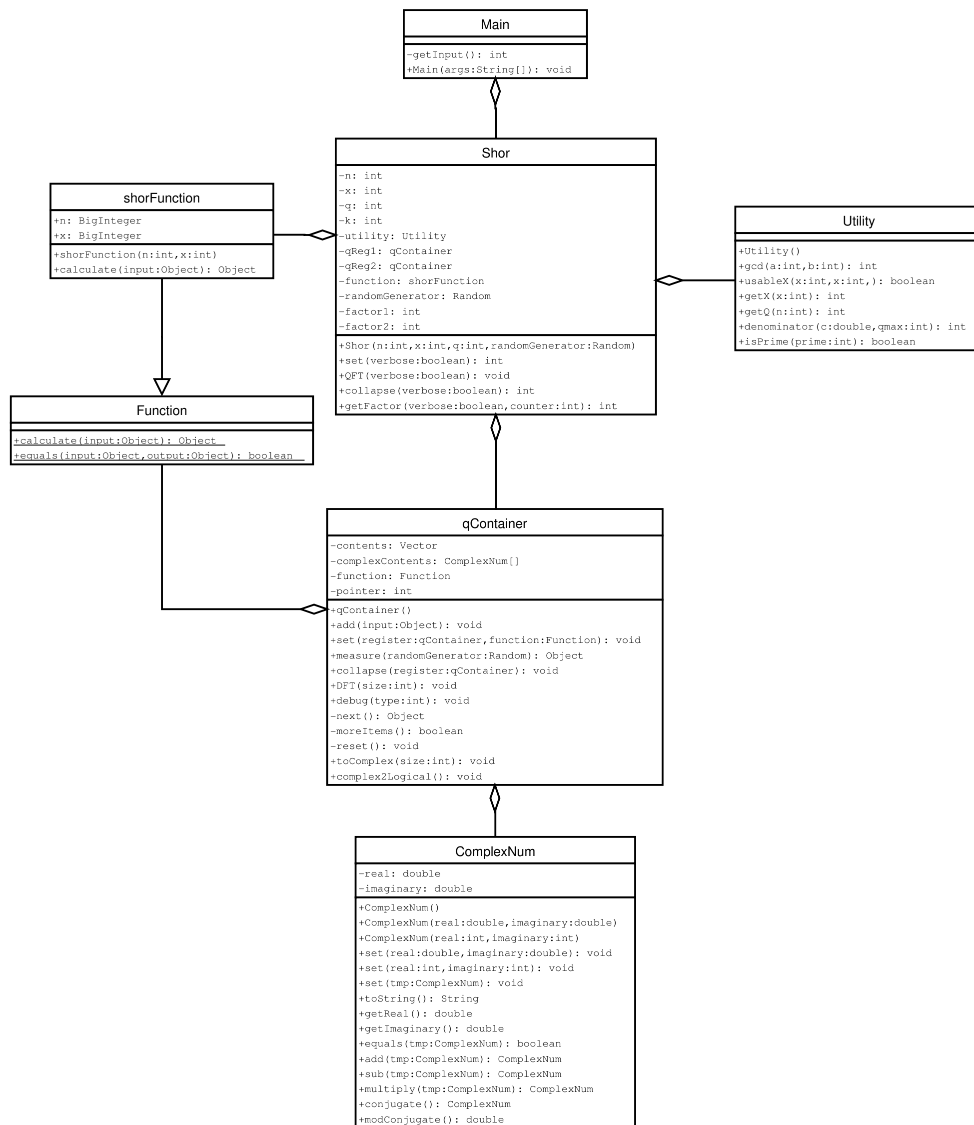


Figure 12: UML Diagram showing components used in the simulator of Shor's algorithm

6 Testing

In order to test the functionality of my simulator a number of tests will be performed on it, this section will give details of these tests and their results.

6.1 Functionality Testing

In addition to the performance tests in the previous section each class should have its own series of simple tests. These tests are outlined along with their results in figure 13.

Test	*Expected Result*	*Actual Result*	*Pass/Fail*
Addition of complex number class, 2+4i + 6-2i	-10+4i	-10+4i	Pass
Subtraction of complex number class, 2+4i + 6-2i	14+4i	14+4i	Pass
Multiplication of complex number class, 2+4i + 6-2i	-28-48i	-28-48i	Pass
Collapsing 1000 average qubits	500 qubits in the zero state +/- 20	500 in zero state	Fail
Set container with superposition of values 3, 5 and 9 and measure	One value returned	3	Pass
Debug methods warn about state collapsing	A warning message is displayed	A warning message displayed	Pass
Running program, no arguments	displays interactive interface	interface displayed	Pass
Running program, one argument	generates x value and runs algorithm	x values generated algorithm run	Pass

Test	*Expected Result*	*Actual Result*	*Pass/Fail*
Factor 15	3 and 5 outputted	3 and 5 outputted	pass
Factor 15	time ¡ 1 second	time = 0.94	pass
Factor 21	7 and 3 outputted	7 and 3 outputted	pass
Factor 33	3 and 11 outputted	3 and 11 outputted	pass
Factor 35	7 and 5 outputted	7 and 5 outputted	pass
Factor 111	3 and 37 outputted	3 and 37 outputted	pass
Factor 111	time ¡ 30 minutes	time = 19 minutes	pass
Factor 303	3 and 101 outputted	3 and 101 outputted	pass

Figure 13: Results of testing

Although the qubit class is not part of my simulation of Shor's algorithm its failings are very important. By investigating this I found that Java random number implementation is based upon the current time, if many objects each with their own random number generator are created very quickly they will all have the same random seed. The testing of the qubit class is applicable to all parts of my simulator that use a random number generator. I amended my design so that only one random number generator is created and used, the qubit class was retests and showed 498 equally weighted qubits collapsed to the zero state.

The testing also showed the exponential time inherent in quantum simulation, the smallest three digit number taking 20 minutes to factorise. The next section details more performance tests.

6.2 Performance Testing

A number of performance tests were conducted on the simulator against two rivals with varying input. M Hayward's simulator and QCL were used in the tests as their source code is freely available, allowing me to change the x values to be the same for all the simulators. To test the Java platform qAbs was run in both in a JVM (Java Virtual Machine) and compiled to byte code using GCJ mentioned earlier.
All the tests were conducted on an 1.3GHz AMD Athlon Processor with 640MB SD RAM running Linux kernel 2.6.11, processes were run at the highest priority three times and then the average taken[6].

All the simulators were run with an x value of 8, apart from QCL on factor 33, QCL would not factorise 33 with that specific x value so the simulator was reset to chose a random value. The results can be seen in Figure 14

Factor	*qAbs (JVM)*	*qAbs (Native)*	*M. Hayward*	*QCL*
15	0.9413	0.702	0.293	0.9663
21	1.5463	1.5413	1.141	2.519
33	13.951	22.229	0.846	23.539*
35	14.099	22.720	2.223	20.942

Figure 14: Simulator Performance Comparison

The table shows there is considerable overhead for using Java rather than a much lower level language like C used in M. Hayward's simulator. Considering though that qAbs uses a much more abstract design than Hayward's simulator it is not so surprising that the results obtained for qAbs are similar to those for QCL.

Another surprising result is that qAbs when run through the JVM has significant performance gains over the natively compiled version. I predicted that the native version would be much faster due to its code not having to be run through a virtual machine, this is not the case. On further investigation it appears that the Java virtual machine caches commonly used functions in an effort to speed up the execution process Philion [2005], the native version does not have this ability.

6.3 User Testing

Most testing strategies focus on the interaction of end users with the system, given that qAbs is designed for users with a background in quantum computing and Shor's algorithm it was not possible to find an end user with experience to test the system in this way.

I will however be able to conduct a number of tests on the interactive part of the interface.

Because test number six failed the user interface was fixed to correct this. The problem was that a non integer would cause the program to exit with an exception where it should have asked the user to re-enter the data.

[6] Appendix shows the full result

Test Number	*Tester*	*Task*	*Time*	*Result*
1	1	Run program	4 Minutes	Fail - Help required
2	1	Factor 15	6 seconds	Pass
3	1	Factor 12	3 seconds	Pass
4	2	Run program	1 minute	Pass
5	2	Factor 15	4 seconds	Pass
6	2	Factor 12	1 second	Fail - inputted 12w, program crashed
7	2	*Factor 12	3 seconds	Pass

Figure 15: Tests performed by users on the interactive interface

6.4 Black and white box testing

Black and white box are different methods of testing, black box treats a program, function or system as a 'black box'. In this type of testing the internal structure or programing is not known and cannot be used to influence the rests. White box testing is the reverse, the internals of the system are used to guide the testing data. Black box testing is most commonly used to test against the specification whereas white box is used to test the implementation.

Ideally each function should be tested using both methods to ensure that the specification is adhered to and that no problems are present in the final build, however given the limited time scale for project such a high level of testing is not feasible. An example white box test can be seen in figure 16 and 17.

6.5 Testing Evaluation

A number of tests have been run on the qAbs simulators, these tests showed two serious errors affecting the user interface and the construction of the simulator, both these bugs were fixed and re-tested to ensure the program passed.

Performance tests were also conducted on the simulator, Figure 14, these showed a considerable overhead using the Java language over C, it also showed that the Java virtual machine had significant gains over the native version compiled with gcj. As expected QCL was the slowest simulator tested probably due to the overhead in parsing the input file, surprisingly however it couldn't factor 33 with 8 as the random factor when the other could.

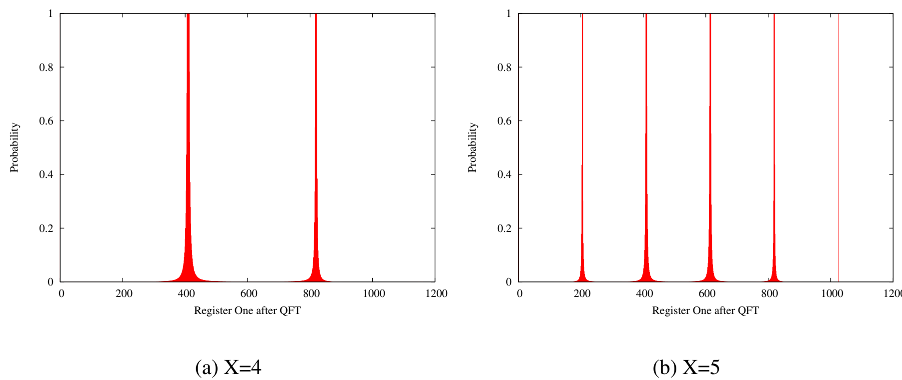
```
public void collapse(qContainer register) {
  // Collapses the current container
  // using the results in the container
  // that was passed. This function does
  // not return a state as this would infer
  // a measurement of the contents.

  Vector collapse = new Vector(0);
  while (register.moreItems()) {
    Object foreign = register.next();
    int counter = 0; // start the counter
    while (counter < contents.size()) {
      Object current = contents.elementAt(counter);
      // Shor function checks equality, returns boolean
      if (function.equals(current, foreign)) {
        collapse.add(current);
        // remove element so we don’t pick it again
        contents.removeElementAt(counter);
        // don’t inc counter
      } else {
        // was not equal, inc counter to go to next one
        counter++;
      } // end if-end
    } // end while
  } // end while
    // collapse should now hold the correct values
    contents = collapse;
}
```

Figure 16: collapse(container) method of the qContainer class

Test	*Expected Result*	*Pass/Fail*
Passing no argument	Program Halts, handled by JVM	Pass
Passing empty register	Container collapses to 0	Pass
Function not set	Container not collapsed	Failed with crash

Figure 17: Results of white box testing



(a) X=4

(b) X=5

Figure 18: Register one after Fourier transform

7 Findings

7.1 Efficiency

In Crick's paper the idea that the frequency created inside the first register was proposed as a measure of efficiency [Crick, 1999]. Although Crick used a linear simulator and did not use a Fourier transform in his work, key components of Shor's algorithm, the proposal that a smaller period makes the algorithm more probable to complete could be correct.

The two graphs shown in Figure 18 show that different periods can be obtained by altering the x value by only a small amount. As the peaks in the graphs relate to the probabilities of measuring a multiple of the period, changing the x value must affect the probability.

After the Fourier transform has been applied to register one the register contains peaks at multiples of the period, this includes the value zero, [7] this means that it is always probable that register one will return zero; if this is measured then the algorithm must be re run.

By examining 18(a) it can be seen that the probability for measuring a value for the period is $2/3$ whereas in 18(b) the probability rises to $5/6$. By changing the x value to get a smaller frequency in register one it is not only quicker to calculate any factors but also more probable that the period will be obtained, by increasing the number of peaks inside the register the probability of measuring zero is reduced.

Crick also believed that the size of the register, Figure 7.1, and frequency increased exponentially. If this is true then the total number of peaks held in superposition after the Fourier transform has taken place is not dependent on n but the random value x, i.e. it is not true that the number of repeat frequencies in the register goes up just because the register size has increased. What this means is that the random factor is the most important variable that determines the efficiency of the algorithm. More research should be carried out to see if there is a classical algorithm that can return a value of x that is not just usable for Shor's algorithm but is also the most efficient.

[7] although the fall off for this is very small and hence is not visible under the y axis

n	q
15	256
21	512
33	2048
35	2048
39	2048
51	4096
55	4096

Figure 19: Growth of q as n increases

1	9	15	3	27	12	9	15	3	27	12	9	15

Figure 20: Sequence generated in register two before measurement, n=33 x=9

7.2 The Introductory Period

When the periodic sequence is generated in register two it is possible for some of the first few values not to appear as part of the repeating sequence. This is dependant on the value of x used, for example Figure 7.2 shows the sequence generated from the first few iterations of $x^a \ mod \ n$ using n=33 and x=9.

The sequence 9 $\ldots$ 12 repeats itself inside the first register, a member of this sequence has a high probability of being measured. However there is one element at the start, the number one, that doesn't form part of the sequence; Moore calls this the introductory period and is used in his even factoring algorithm mentioned previously [Moore, 1995].

The introductory period can create a problem in register two, if a value from this set is measured then only states that could give that value would be projected into register one, the states held in superposition in register one would then have no periodicity and the algorithm would have failed. The probability of this occurring is small, usually only $1/q$, because the period is usually only one element in length, but it does account for another reason as to why Shor's algorithm fails.

More work must be done over large input size to determine if the introductory period grows at the same scale of the register. If the introductory period does not grow at the same rate of the register this means that as the input size increases the probability that algorithm will complete successfully will rise. Larger numbers will be more likely to return the correct result.

Now the idea that the efficiency or probability that the algorithm will complete successfully is dependant on two things, the measurement of register two and the value of x used to increase the number of peaks in register two.

8 Critical Assessment

Overall I believe my project was a success, I have been able to create a simulator that models the quantum states that Shor's algorithm suggests and all the primary requirements that were set in the analysis section have been completed.

I was also able to complete most of the secondary goals of my simulator with the exception of allowing auto generation of graphs, after careful investigation I found that this would depend too much on a graphical user interface and would have taken much longer to implement. Instead I was able to use the open source program gnuplot and using the results from my simulator I have been able to create all the graphs seen in this document. These graphs show results very similar to those found in Williams and Clearwater [1998], because of this I can be sure I have accurately reproduced Shor's algorithm.

The many prototypes I build as part of my spiral development model also helped in analysing the problem and what was implemented. For instance the first program I created was a linear simulator of the visualisation of Shor's algorithm, after running this program unsuccessfully a few times I found that Java does not warn the user if an integer data type overflows. As the integers used in Shor's algorithm get can become very big this caused many errors, by using Java's big integer class I was able to fix these errors and use this information in my current simulator.

Using Java as the programming language for my simulator was a good choice, already having experience with the language meant that I did not need any extra time learning another programming language. I have learnt a lot from programming a mathematical project in Java for example, I have examined the maths libraries in great detail which has extended my programming skill.

However, because my simulator is written in Java it means that all end users must have a JRE, Java runtime environment, installed. While this is not difficult to obtain the extra time needed to install the JRE properly and run my program may be too much for many people. Compiling the project using GCJ does not solve this either, the binary file need special libraries to be install which can be difficult on both windows and Linux.

Using my quantum container class instead of a register was also a success, I have shown that this class can be used in the same way as the standard quantum register but does not need complicated transforms to be run on it. Because of this my source code should be much easier to understand than the code for QCL which uses these complicated transforms.

When I first implemented the quantum container class it was made to be recursive, instead of making two separate registers and running the function to entangle them in the main body of the program I let the container set up a linked list which was stepped over once the register was measured. Although this worked for Shor's factoring algorithm I found another quantum algorithm by Deutsch through my research that could not be set this way, therefore I decided to change the way that the quantum container class would be implemented. By doing this the design of my simulator was made much more expendable.

Due to time constraints there were a number of features that were not added to either my secondary or my primary requirements for my simulator. I would have liked to have implemented a graphical user interface based on Java swing library, this would have enabled the user to see the contents of the registers as measurements are made and the Fourier transform performed. No simulator has this ability and I believe it would be a very good learning and research aid. Because I used an object orientated approach to my work this should not be difficult to include if my work was extended.

When I first started this project one of the goals of my research was to investigate the frequency over multiple iterations of $x^a \ mod \ n$, however I soon discovered that this was not possible as it required the results to be obtained from a linear simulator. The research section explains that a linear simulator does not hold a quantum state and because of this I changed the aims of my research to the ones stated in this document.

By far the most time in my project was taken up by the large amount of reading I have done, because of this it was very important that I kept to the goals set in my time plan. For the most part this has been done although the amount of research I undertook was under estimated, this could not have been avoided as the information gained from my research was required in my implementation of my simulator.

I would also have liked to implement a second algorithm in my design, the most likely being the discrete logarithm algorithm from Shor's original paper. Although I did considerable research into the algorithm it eventually became too time consuming and my knowledge of group theory wasn't enough to implement the algorithm, because of this I felt I should spent my time making my simulator more effective at simulating the factoring algorithm.

I personally consider my project very successful, I have learnt a vast amount about quantum mechanics, computing and the Java platform. The simulator I have written may also be the only one available that has been written in Java. The research I have done with has surprised myself and allowed me to learn a lot from Shor's algorithm and quantum computing in general

I will be releasing my simulator as an open source project which I hope will be continued after this project has finished.

Bibliography

Stephane Beauregard. Circuit for shor's algorithm using 2n+3 qubits. *Quantum Information and Computation*, 3(2):175–185, 2003.

David Crick. Analysis of shor's quantum factoring algorithm. Project III Stage III, Department of Computer Science, Exeter, 1999.

David Deutsch. Quantum theory the church-turing principle and the universal quantum computer. *Proceedings Royal Society London*, A400:97–117, 1985.

Richard Feynman. Quantum mechanical computers. *Optics News, Feb*, pages 11–20, 1985.

Tony Hey and Patrick Walters. *The New Quantum Universe*. Cambridge University Press, 2003.

Intel Corporation. Moore's law. www.intel.com/research/silicon/mooreslaw.htm Accessed March, 2005.

Richard Jozsa. Quantum factoring, discrete logarithms and the hidden subgroup problem. *IEEE*, 3(2), 2000.

Techworld Manek Dubash. Gordon moore speaks out. www.techworld.com/opsys/features/index.cfm?FeatureID=1353, 2005.

Gordon E. Moore. Cramming more components into integrated circuits. *Electronics*, 38(8), 1965.

Mark Moore. Quantum-inspired algorithms and a methodology for their construction. Master's thesis, Department of Computer Science, Exeter, 1995.

Ajit Narayanan, David Crick, and Julia Wallace. Exploring quantum algorithms through classical simulations. ., 1999.

Paul Philion. The java hotspot performance engine is set to break new records. www.javaworld.com/javaworld/javaone99/j1-99-hotspot.html Accessed April, 2005.

Peter Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceeding of the 35th Annual Symposium on the Foundations of Computer Science*, 1994.

Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM*, 26:1484–1509, 1997.

M Steffen, L Vandersypen, and I Chuang. Toward quantum computation: a five-qubit quantum processor. *IEEE Micro*, 21:24–34, 2001.

Julia Wallace. *Quantum Computer Software*. PhD thesis, Department of Computer Science, Exeter, 2001.

Julia Wallace. Quantum simulators. www.cs.kent.ac.uk/people/staff/jw74/qc/ Accessed November, 2004.

Colin P. Williams and Scott H. Clearwater. *Explorations in quantum computing*. Telos Publications, 1998.

Wolfram Research Inc. Euler's totient theorem. mathworld.wolfram.com/EulersTotientTheorem.html Accessed April, 2005.

Appendix A

qAbs Usage

1 Example output

qAbs can be run in a number of ways, the easier being the interactive mode. To run qAbs in interactive mode open a command line environment in you operation and change to the *bin* directory in the directory qAbs has been extracted into. This is usually done with the command 'cd'. At you prompt issue the command 'java Main' the program will then ask you for a prime factor.
The following should appear in your terminal or command environment once pressing the enter key.

```
Prime Factor: 15
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 256 as Q value
Continued Fraction Found
The following factors were found
3, 5
```

2 Compiling qAbs

Should you need to compile qAbs [1] please run the following command in the *src* directory of qAbs, 'javac Main.java -d ../bin'.
To compile with gcj firstly make sure the gcj development files have been installed correctly and then issue the command.
'gcj –main=Main Shor.java Utility.java ComplexNum/ComplexNum.java Container/qContainer.java Functions/Function.java Functions/shorFunction.java Main.java'

[1] This may be needed to you are running Blackdown JVM or Sun JVM 1.5

3 List of factors composed of two primes

Below is a list of all one, two and three digit factors made from the product of two primes. These can be used as input to qAbs or other simulators capable of running Shor's algorithm.

15 21 33 35 39 51 55 57 65 69 77 85 87 91 93 95 111 115 119 123 129 133 141 143 145 155 159 161 177 183 185 187 201 203 205 209 213 215 217 219 221 235 237 247 249 253 259 265 267 287 291 295 299 301 305 319 323 329 335 341 355 365 371 377 391 395 403 407 413 415 427 437 445 451 469 473 481 485 493 497 511 517 527 533 551 553 559 581 583 589 611 623 629 649 667 671 679 689 697 703 713 731 737 767 779 781 793 799 803 817 851 869 871 893 899 901 913 923 943 949 979 989

Appendix B

QCL Source Code

```
include "modarith.qcl";
include "dft.qcl";

procedure shor(int number) {
  int width=ceil(log(number,2));   // size of number in bits
  qureg reg1[2*width];             // first register
  qureg reg2[width];               // second register
  int qmax=2^width;
  int factor;                      // found factor
  int m; real c;                   // measured value
  int x;                           // base of exponentiation
  int p; int q;                    // rational approximation p/q
  int a; int b;                    // possible factors of number
  int e;                           // e=x^(q/2) mod number

  if number mod 2 == 0 { exit "number must be odd"; }
  if testprime(number) { exit "prime number"; }
  if testprimepower(number) { exit "prime power"; };

  {
    {                              // generate random base
      //x=floor(random()*(number-3))+2;
        x = 8;
    } until gcd(x,number)==1;
    print "chosen random x =",x;
    Mix(reg1);                     // Hadamard transform
    expn(x,number,reg1,reg2);      // modular exponentiation
    measure reg2;                  // measure 2nd register
    dft(reg1);                     // Fourier transform
    measure reg1,m;                // measure 2st register
    reset;                         // clear local registers
    if m==0 {                      // failed if measured 0
      print "measured zero in 1st register. trying again ...";
    } else {
      c=m*0.5^(2*width);           // fixed point form of m
```

```
      q=denominator(c,qmax);       // find rational approximation
      p=floor(q*c+0.5);
      print "measured",m,", approximation for",c,"is",p,"/",q;
      if q mod 2==1 and 2*q<qmax { // odd q ? try expanding p/q
        print "odd denominator, expanding by 2";
        p=2*p; q=2*q;
      }
      if q mod 2==1 {              // failed if odd q
        print "odd period. trying again ...";
      } else {
        print "possible period is",q;
        e=powmod(x,q/2,number);    // calculate candidates for
        a=(e+1) mod number;        // possible common factors
        b=(e+number-1) mod number; // with number
        print x,"^",q/2,"+ 1 mod",number,"=",a,",",
              x,"^",q/2,"- 1 mod",number,"=",b;
        factor=max(gcd(number,a),gcd(number,b));
      }
    }
  } until factor>1 and factor<number;
  print number,"=",factor,"*",number/factor;
}
```

Appendix C

Test Data

1 qAbs

1.1 JVM

```
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 256 as Q value
Continued Fraction Found
The following factors were found
3, 5

real    0m1.830s
user    0m0.475s
sys     0m0.042s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 256 as Q value
Continued Fraction Found
The following factors were found
3, 5

real    0m0.498s
user    0m0.464s
sys     0m0.026s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 256 as Q value
Continued Fraction Found
The following factors were found
```

```
3, 5

real    0m0.496s
user    0m0.460s
sys     0m0.030s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 512 as Q value
Continued Fraction Found
The following factors were found
7, 3

real    0m1.784s
user    0m1.121s
sys     0m0.034s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 512 as Q value
Continued Fraction Found
The following factors were found
7, 3

real    0m1.175s
user    0m1.106s
sys     0m0.039s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 512 as Q value
Continued Fraction Found
The following factors were found
7, 3

real    0m1.680s
user    0m1.124s
sys     0m0.038s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
3, 11
```

```
real    0m13.988s
user    0m13.850s
sys     0m0.083s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
3, 11

real    0m13.951s
user    0m13.760s
sys     0m0.146s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
3, 11

real    0m13.914s
user    0m13.800s
sys     0m0.071s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
7, 5

real    0m13.961s
user    0m13.827s
sys     0m0.110s

Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
7, 5

real    0m13.956s
```

```
user    0m13.813s
sys     0m0.080s

Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
7, 5

real    0m14.381s
user    0m13.817s
sys     0m0.067s
```

1.2 Native

```
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 256 as Q value
Continued Fraction Found
The following factors were found
3, 5

real    0m0.752s
user    0m0.487s
sys     0m0.027s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 256 as Q value
Continued Fraction Found
The following factors were found
3, 5

real    0m0.505s
user    0m0.480s
sys     0m0.024s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 256 as Q value
Measured 0 in reg1, retry
Odd period found, retry
Continued Fraction Found
```

```
Continued Fraction Found
The following factors were found
3, 5


real    0m0.849s
user    0m0.823s
sys     0m0.020s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 512 as Q value
Continued Fraction Found
The following factors were found
7, 3

real    0m1.510s
user    0m1.481s
sys     0m0.024s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 512 as Q value
Continued Fraction Found
The following factors were found
7, 3

real    0m1.513s
user    0m1.385s
sys     0m0.123s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 512 as Q value
Continued Fraction Found
The following factors were found
7, 3

real    0m1.601s
user    0m1.461s
sys     0m0.029s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
```

```
The following factors were found
3, 11

real    0m22.157s
user    0m21.925s
sys     0m0.157s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
3, 11

real    0m22.195s
user    0m22.026s
sys     0m0.080s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
3, 11

real    0m22.335s
user    0m22.067s
sys     0m0.062s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
7, 5

real    0m23.541s
user    0m21.854s
sys     0m0.061s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
7, 5
```

```
real    0m22.143s
user    0m22.012s
sys     0m0.058s
Running Shor's Algorithm Simulation
By Ross James Bevington 2005, please see readme for more details

Using 8 as X value
Using 2048 as Q value
Continued Fraction Found
The following factors were found
7, 5

real    0m22.477s
user    0m21.797s
sys     0m0.135s
```

2 M. Hayward

```
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

Input the number you wish to factor.
Found x to be 8.
Found q to be 256.
Made register 1 with register size = 9
Created register 2 of size 4
Measurement : 1
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
measured 128, approximation for 0.5 is 1 / 2
possible period is 2
8^1 + 1 mod 15 = 9,
8^1 - 1 mod 15 = 7
15 = 3 * 5

real    0m0.270s
user    0m0.026s
sys     0m0.002s
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

Input the number you wish to factor.
Found x to be 8.
Found q to be 256.
Made register 1 with register size = 9
Created register 2 of size 4
Measurement : 1
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Measured, 0 this trial a failure!
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Measured, 0 this trial a failure!
Measurement : 2
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Measured, 0 this trial a failure!
Measurement : 4
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
measured 128, approximation for 0.5 is 1 / 2
possible period is 2
8^1 + 1 mod 15 = 9,
8^1 - 1 mod 15 = 7
15 = 3 * 5

real    0m0.334s
user    0m0.090s
sys     0m0.003s
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

Input the number you wish to factor.
Found x to be 8.
```

```
Found q to be 256.
Made register 1 with register size = 9
Created register 2 of size 4
Measurement : 4
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
measured 128, approximation for 0.5 is 1 / 2
possible period is 2
8^1 + 1 mod 15 = 9,
8^1 - 1 mod 15 = 7
15 = 3 * 5

real    0m0.276s
user    0m0.027s
sys     0m0.002s
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

Input the number you wish to factor.
Found x to be 8.
Found q to be 512.
Made register 1 with register size = 10
Created register 2 of size 5
Measurement : 1
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
measured 256, approximation for 0.5 is 1 / 2
possible period is 2
8^1 + 1 mod 21 = 9,
8^1 - 1 mod 21 = 7
21 = 7 * 3

real    0m0.429s
user    0m0.187s
sys     0m0.003s
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

Input the number you wish to factor.
Found x to be 8.
Found q to be 512.
Made register 1 with register size = 10
Created register 2 of size 5
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
Measurement : 1
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
```

```
There have been five failures, giving up.

real    0m1.175s
user    0m0.929s
sys     0m0.005s

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

Input the number you wish to factor.
Found x to be 8.
Found q to be 512.
Made register 1 with register size = 10
Created register 2 of size 5
Measurement : 1
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
```

```
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
Measurement : 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 19.3738% done!
Making progress in Fourier transform, 38.9432% done!
Making progress in Fourier transform, 58.5127% done!
Making progress in Fourier transform, 78.0822% done!
Making progress in Fourier transform, 97.6517% done!
Measured, 0 this trial a failure!
There have been five failures, giving up.

real    0m1.818s
user    0m0.941s
sys     0m0.006s
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

Input the number you wish to factor.
Found x to be 8.
Found q to be 2048.
Made register 1 with register size = 12
Created register 2 of size 6
Measurement : 2
Begin Discrete Fourier Transformation!
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!
measured 1434, approximation for 0.700195 is 717 / 1024
possible period is 1024
8^512 + 1 mod 33 = 32,
8^512 - 1 mod 33 = 30
33 = 3 * 11

real    0m0.840s
user    0m0.593s
sys     0m0.003s
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

Input the number you wish to factor.
Found x to be 8.
Found q to be 2048.
Made register 1 with register size = 12
Created register 2 of size 6
Measurement : 8
Begin Discrete Fourier Transformation!
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!
measured 410, approximation for 0.200195 is 205 / 1024
possible period is 1024
8^512 + 1 mod 33 = 32,
8^512 - 1 mod 33 = 30
```

```
33 = 3 * 11

real    0m0.859s
user    0m0.608s
sys     0m0.009s
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

Input the number you wish to factor.
Found x to be 8.
Found q to be 2048.
Made register 1 with register size = 12
Created register 2 of size 6
Measurement : 4
Begin Discrete Fourier Transformation!
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!
measured 1024, approximation for 0.5 is 1 / 2
possible period is 2
8^1 + 1 mod 33 = 9,
8^1 - 1 mod 33 = 7
33 = 3 * 11

real    0m0.840s
user    0m0.594s
sys     0m0.004s
```

```
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

Input the number you wish to factor.
Found x to be 8.
Found q to be 2048.
Made register 1 with register size = 12
Created register 2 of size 6
Measurement : 22
Begin Discrete Fourier Transformation!
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!
measured 1536, approximation for 0.75 is 3 / 4
possible period is 4
8^2 + 1 mod 35 = 30,
8^2 - 1 mod 35 = 28
35 = 7 * 5

real    0m1.735s
user    0m1.468s
sys     0m0.006s

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

Input the number you wish to factor.
Found x to be 8.
Found q to be 2048.
Made register 1 with register size = 12
Created register 2 of size 6
Measurement : 1
Begin Discrete Fourier Transformation!
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!
measured 1536, approximation for 0.75 is 3 / 4
possible period is 4
8^2 + 1 mod 35 = 30,
8^2 - 1 mod 35 = 28
35 = 7 * 5

real    0m1.724s
user    0m1.475s
sys     0m0.004s
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
```

```
Input the number you wish to factor.
Found x to be 8.
Found q to be 2048.
Made register 1 with register size = 12
Created register 2 of size 6
Measurement : 1
Begin Discrete Fourier Transformation!
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!
Measured, 0 this trial a failure!
Measurement : 1
Begin Discrete Fourier Transformation!
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!
```

```
measured 512, approximation for 0.25 is 1 / 4
possible period is 4
8^2 + 1 mod 35 = 30,
8^2 - 1 mod 35 = 28
35 = 7 * 5

real    0m3.209s
user    0m2.846s
sys     0m0.102s
```

3 QCL

```
QCL Quantum Computation Language (32 qubits, seed 1113401132)
: chosen random x = 8
: measured 192 , approximation for 0.75 is 3 / 4
: possible period is 4
: 8 ^ 2 + 1 mod 15 = 5 , 8 ^ 2 - 1 mod 15 = 3
: 15 = 5 * 3

real    0m1.176s
user    0m0.456s
sys     0m0.004s
QCL Quantum Computation Language (32 qubits, seed 1113401156)
: chosen random x = 8
: measured 128 , approximation for 0.5 is 1 / 2
: possible period is 2
: 8 ^ 1 + 1 mod 15 = 9 , 8 ^ 1 - 1 mod 15 = 7
: 15 = 3 * 5

real    0m0.919s
user    0m0.459s
sys     0m0.007s
QCL Quantum Computation Language (32 qubits, seed 1113401038)
: chosen random x = 8
: measured 64 , approximation for 0.25 is 1 / 4
: possible period is 4
: 8 ^ 2 + 1 mod 15 = 5 , 8 ^ 2 - 1 mod 15 = 3
: 15 = 5 * 3

real    0m0.804s
user    0m0.456s
sys     0m0.006s
QCL Quantum Computation Language (32 qubits, seed 1113401285)
: chosen random x = 8
: measured 512 , approximation for 0.5 is 1 / 2
: possible period is 2
: 8 ^ 1 + 1 mod 21 = 9 , 8 ^ 1 - 1 mod 21 = 7
: 21 = 7 * 3
```

```
real    0m2.272s
user    0m2.245s
sys     0m0.007s
QCL Quantum Computation Language (32 qubits, seed 1113401400)
: chosen random x = 8
: measured 512 , approximation for 0.5 is 1 / 2
: possible period is 2
: 8 ^ 1 + 1 mod 21 = 9 , 8 ^ 1 - 1 mod 21 = 7
: 21 = 7 * 3

real    0m2.992s
user    0m2.250s
sys     0m0.009s
QCL Quantum Computation Language (32 qubits, seed 1113401313)
: chosen random x = 8
: measured 512 , approximation for 0.5 is 1 / 2
: possible period is 2
: 8 ^ 1 + 1 mod 21 = 9 , 8 ^ 1 - 1 mod 21 = 7
: 21 = 7 * 3

real    0m2.293s
user    0m2.229s
sys     0m0.016s
QCL Quantum Computation Language (32 qubits, seed 1113401721)
: chosen random x = 5
: measured 1229 , approximation for 0.300049 is 3 / 10
: possible period is 10
: 5 ^ 5 + 1 mod 33 = 24 , 5 ^ 5 - 1 mod 33 = 22
: 33 = 11 * 3

real    0m17.812s
user    0m15.631s
sys     0m0.078s
QCL Quantum Computation Language (32 qubits, seed 1113401859)
: chosen random x = 10
: measured zero in 1st register. trying again ...
: chosen random x = 23
: measured 2048 , approximation for 0.5 is 1 / 2
: possible period is 2
: 23 ^ 1 + 1 mod 33 = 24 , 23 ^ 1 - 1 mod 33 = 22
: 33 = 11 * 3

real    0m35.309s
user    0m33.963s
sys     0m0.122s
QCL Quantum Computation Language (32 qubits, seed 1113401941)
: chosen random x = 14
: measured 3277 , approximation for 0.800049 is 4 / 5
: odd denominator, expanding by 2
```

```
: possible period is 10
: 14 ^ 5 + 1 mod 33 = 24 , 14 ^ 5 - 1 mod 33 = 22
: 33 = 11 * 3

real    0m17.498s
user    0m16.514s
sys     0m0.020s
QCL Quantum Computation Language (32 qubits, seed 1113402085)
: chosen random x = 8
: measured 2048 , approximation for 0.5 is 1 / 2
: possible period is 2
: 8 ^ 1 + 1 mod 35 = 9 , 8 ^ 1 - 1 mod 35 = 7
: 35 = 7 * 5

real    0m20.605s
user    0m16.173s
sys     0m0.065s
QCL Quantum Computation Language (32 qubits, seed 1113402261)
: chosen random x = 8
: measured 1024 , approximation for 0.25 is 1 / 4
: possible period is 4
: 8 ^ 2 + 1 mod 35 = 30 , 8 ^ 2 - 1 mod 35 = 28
: 35 = 7 * 5

real    0m20.424s
user    0m16.558s
sys     0m0.153s
QCL Quantum Computation Language (32 qubits, seed 1113402293)
: chosen random x = 8
: measured 3072 , approximation for 0.75 is 3 / 4
: possible period is 4
: 8 ^ 2 + 1 mod 35 = 30 , 8 ^ 2 - 1 mod 35 = 28
: 35 = 7 * 5

real    0m21.798s
user    0m17.786s
sys     0m0.059s
```