Citation for published version:
Li, H 2007, The analysis and implementation of the AKS algorithm and its improvement algorithms. Computer Science Technical Reports, no. CSBU-2007-09, Department of Computer Science, University of Bath.

Publication date:
2007

Link to publication

Department of
Computer Science

UNIVERSITY OF BATH

Technical Report

Undergraduate Dissertation: The Analysis and Implementation of the AKS Algorithm and Its Improvement Algorithms

Hua Li

Technical Report 2007-09 **July 2007**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

The Analysis and Implementation of the AKS Algorithm and Its Improvement Algorithms

Hua Li

Batchelor of Science in Computer Science with Honours
The University of Bath
May 2007

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

Signed:

The Analysis and Implementation of the AKS Algorithm and Its Improvement Algorithms

Submitted by: Hua Li

COPYRIGHT

Declaration

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

Signed:

Abstract

In August 2002, three Indian researchers, Manindra Agrawal and his students Neeraj Kayal and Nitin Saxena presented a remarkable algorithm called AKS algorithm in their paper Primes is in P. It is the first deterministic primality testing algorithm in polynomial time. The project provides experimental data to suggest the complexity time of AKS algorithm is $O((\log n)^8 \log\log n)$. Afterwards, many scientists tried to improve the AKS algorithm, one of the better ones is proposed by Lenstra. R. Crandall and J. Papadopoulos has shown that Lenstras version of the AKS algorithm has a complexity time of $C(\log n)^6$ where C is a logarithm constant. Our testing supports this result and suggest C can be approximate by $\frac{\phi(r)}{(\log\log(\phi(r)))r}$ where r is the useful certificate. Further testing results have shown a better complexity time $O(\log n)^{5.73}$ for lenstra's version.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to acknowledge Professor James Davenport, my supervisor, for providing me with the idea of this project and many helpful suggestions through out the project. I would also like to thank my parents and sisters for their encouragements. Addtionally thanks to Victor Shoup, his great number theory library — NTL.

Chapter 1

Introduction

In modern cryptography, prime numbers have played an important role. Difficulties in factorizing a large number composed of two significantly large primes (RSA) or calculations of logarithms over certain finite fields of a size p prime(Diffie-Hellman) ensure that the key is computationally infeasible to compute without prior knowledge of the primenumbers used. Therefore using an good primality testing algorithm to identify prime numbers is crucial in cryptosystems.

In August 2002, three Indian researchers, Manindra Agrawal and his students Neeraj Kayal and Nitin Saxena presented a remarkable algorithm called AKS algorithm in their paper ”Primes is in P.” It is the first deterministic primality testing algorithm in polynomial time. Afterwards, many scientists did some improvements on AKS algorithm, one of the better ones is proposed by Lenstra.

Because of the importance of prime numbers in cryptography, it is worth investigating the AKS algorithm and its improvement algorithms. In this project, the above two algorithms will be implemented. After that, we will compare them, estimate the required running time, goes further to suggest some improvements on these algorithms and analyze the practicality of these two algorithms.

Chapter 2

Literature Survey

2.1 Introduction

Prime numbers have played an important role in mathematics throughout history. They form the basic building block for all positive integer numbers and therefore continuous researches are carried on to investigate the behaviour of such numbers over time. Nowadays, prime numbers are great understood and their applications are far more significant.

2.1.1 Number Theory and Algorithm Terminology

Firstly the algorithm terminology and some definitions in number theory will be presented because they are essential and helpful for understanding prime numbers and their applications.

Prime and Composite Numbers

Definition 2.1 [1]: A prime number is an integer n greater than one with the property that 1 and n are the only positive integers that divide n.

Definition 2.2 [2]: A composite number n is a positive integer $n > 1$ which is not prime.

Congruence

Definition 2.3 [3]: If two numbers a and b have the property that their difference $a - b$ is integrally divisible by a number m (i.e., $\frac{a-b}{m}$ is an integer), then a and $b|$ are said to be congruent modulo m. The number is called the modulus, and the statement a is congruent to b (modulo m) is written mathematically as

$$a \equiv b \pmod{m}.$$

If $a - b$ is not integrally divisible by m , then we say a is not congruent to b (modulo m), which is written

$$a \not\equiv b \pmod{m}.$$

Greatest Common Divisor

Definition 2.4 [4]: The greatest common divisor (GCD of two non-zero integers, is the largest positive integer that divides both numbers without remainder.

A typical algorithm for calculating GCD of two integers is the Euclidean algorithm (**?**, Section 4.5.2-4.5.3, pp.333-379): Given two natural number a and b, check if b is zero. If yes, a is the gcd. If not, repeat the process using b and the remainder after integer division of a by b (a mod b).

O-notation

Definition 2.5 [5]: $O(g(n)) = \{f(n)$: there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. If $f(n) \in O(g(n))$, then we write: $f(n) = O(g(n))$.
Similarly, $o(g(n)) = \{f(n)$: there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$. If $f(n) \in o(g(n))$, then we write: $f(n) = o(g(n))$.
Another notation is $\tilde{O}$ (read soft-O). $f(n) = \tilde{O}(g(n))$ is shorthand for $f(n) = O(g(n)\log_k g(n))$ for some k. Essentially, it is Big-O, ignoring logarithmic factors.

2.1.2 Historical Background for Prime Numbers

The study of prime numbers dates back to 300 BC, by that time an ancient Greek mathematician called Euclid used the method of contradiction to prove that there were infinitely many prime numbers. He also proved the fundamental theorem of arithmetic that is every integer can be written as a product of primes in an essentially unique way.

In 200 BC, another Greek mathematician called Eratosthenes introduced an algorithm for primality testing called the Sieve of Eratosthenes. His method [6] works like this, first take a list of integer number from 2 to n, and then mark all the even numbers except two. Then start with the next unmarked number, which in our case is three, mark all the numbers which are multiplies of three, again except three itself. Continue in this way until there are no new unmarked numbers. All the numbers remaining unmarked are prime numbers.

The prime numbers then went through a dark age until the 17th century Pierre Fermat put a great effort into it. One of his greatest achievements is the Fermat's Little Theorem which states that for any prime p and any integer a, we have $a^p \equiv a \bmod p$. Since then more and more important and famous proofs, results and properties related to prime numbers have been established. Some further details are discussed at this webpage [7].

2.1.3 Algorithm Development

Most of the mathematicians who are interested in prime numbers have two challenges:

- To find the next largest prime number relative to current one.
- To provide a fast method or improve a current available one which determines whether a given number is prime or composite.

This is due to the natural properties of prime numbers and the need of their widespread applications. The second problem is particularly important because the given number could be any magnitude. Currently the largest known prime number is $2^{24036583} - 1$ which was found by John Findley [8] in 2004. Therefore methods such as Sieve of Eratosthenes are not efficient for large numbers since their complexity of time is exponential in terms of the length of a large number. It is not feasible to determine an integer number of large magnitude via these methods even based on the latest mainframe computers. Naturally, a more efficient primality testing algorithm is urgently needed because of the important role prime numbers play in modern cryptosystems. As a result, there has been a dramatic increase in developing efficient primality testing algorithms since 1970s. Most of them are based on the Fermat's Little Theorem.[1]

In 1975, Pratt [9] has shown that a short certificate of primality always exists and hence that primes are in non deterministic polynomial time, but while his method is constructive it requires one to factor large integers and thus does not run in polynomial time. In 1976, Miller [10] obtained a deterministic polynomial time algorithm for primality testing based on the Extended Riemann Hypothesis (ERH). If this hypothesis is proved to be true, the running time of this algorithm is $O(\log n)^4$. In 1977, Solovay and Strassen [11] devised a randomized polynomial time algorithm using Fermat's Little Theorem and quadratic residues. In 1980, Rabin [12] modified Miller's algorithm and obtained an unconditonal randomized algorithm. The Miller-Rabin algorithm relies on an equality or set of equalities that hold true for prime values, and then see whether or not they hold for a number that we want to test for primality. The algorithm was able to correctly identify the composite numbers but had a probability which is less than 0.25 of error in identifying primes. In 1983, a breakthrough for primality testing was achieved by Adleman, Pomerance and Rumely [13], who gave a deterministic primality test that runs in $(\log n)^{c(\log\log\log n)}$ time which is very close to the polynomial time. In 1986, Goldwasser and Kilian [14] proved that the majority of the primes could be determined in randomized polynomial time and devised a randomized

[1] See section 1.2

algorithm based on elliptic curves. Later, Atkin and Morain[15] (1990,1993) devised a method called Elliptic curve primality proving(ECPP). It stated that given any prime n of length k, the algorithm outputs a certificate of correctness of size $O(k^2)$ which can be verified correct in $O(k^4)$ deterministic time. The running time of this algorithm is $O((\log n)^{6+\varepsilon})$ for some $\varepsilon > 0$.

2.1.4 Primality Testing and Modern Cryptography

The origin of cryptography can be dated back to the time when human beings started to communicate with each other. They need to find some ways to ensure the confidentiality of their communications. The ancient Greeks were probably the first to use some techniques to encrypt the information. They used a kind of club called scytale, consisting of a cylinder with a strip of leather wound around it on which was written a message. Cryptography was also used in the battlefield. For example, during the Second World War the British decrypted the cryptographic machines that are used by Germany military and thereby obtained the military information and took advantage of it. Cryptography has been brought into a new era since the invention of computer.

However, the traditional cryptographic techniques can be easily decrypted by the powerful computational ability of the modern computers. Therefore researches on cryptography are concerning the mathematical ways in order to devise a cryptosystem which is difficult to decrypt. In modern cryptography, prime numbers have played an important role. Difficulties in factorizing a large number composed of two significantly large primes (RSA) or calculations of logarithms over certain finite fields of a size p prime(Diffie-Hellman) ensure that the key is computationally infeasible to compute without prior knowledge of the primenumbers used. Therefore choosing an large prime numbers is crucial in cryptosystems. Primality testing is a mathematical way of finding prime numbers. The faster it works, the safer the cryptosystems are.

Cryptosystem

In general, there are two types of cryptosystems. One of them is called Private Key Cryptosystem in which the same key is used in the process of encryption and decryption, such as DES or AES cryptosystem. The other one is called Public Key Cryptosystem in which a private key and a public key are used in the process of encryption and decryption, such as RSA cryptosystem.

Generally speaking, private key cryptosystem is the traditional cryptosystem. This kind of system usually does not require a large mount of computation and the efficiency of key generation is high because the length of the key is relatively short. However, if a message encrypted in that way can be easily intercepted on the Internet, it then can be decrypted by computers using an exhausted algorithm.

As a result, it requires a secure channel for the message transmission. Conversely, in public key cryptosystem the keys they use are enough long to ensure the system cannot be decrypted in a reasonable time. Moreover, the message transmitting on the Internet is encrypted based on the receiver's public key, even if the message has been intercepted by other people, it is rather difficult for them to figure out what the private key is.

Nowadays protecting access to information for security reasons is a primary reason for using cryptography.

Morover, cryptosystem is also used for identification and authentication of individuals due to the growth of Internet. In this sense, public key cryptosystem is a sensible choice and is widely used on the Internet. We will briefly talk about some examples of public key cryptosystem in the following sections.

Deffie-Hellman Key Agreement Protocol

The Deffie-Hellman [16] key agreement protocol was published in 1976. In their system, there are two public parameters p and g which can be used by all the users. p is a prime number and g is an integer less than p. Then we have a property that for every integer number between 1 and p-1 inclusive, such that $n = g^m \bmod p$. Here n is the public value and m is the private value.

Suppose Alice and Bob[2] want to agree on a shared secret key. First Alice and Bob generate a private random integer value a and b respectively. Then they compute their public value. Alice's public value is g^a mod p and Bob's value is g^b mod p. After they exchange their public values, Alice computes $(g^b)^a \bmod p$ and Bob computes $(g^a)^b \bmod p$. Because $g^{ab} = g^{ba} = k$, Alice and Bob have a shared secret key k.

However, the Diffie-Hellman key exchange can be easily attacked if someone in the middle of them can intercept their public keys and modify them with his/her own. This vulnerability exists because the Diffie-Hellman key exchange does not authenticate the participants.

[2] Commonly used placeholders for archetypal characters in cryptography.

Despite that defect, Diffie and Hellman have introduced an important concept which is called Trapdoor One-way Function. Basically One-way Function means that it is easy to compute a function in one way but difficult in the inverse way. The Trapdoor One-way Function itself is a kind of One-way Function and just as the name implies there is a trapdoor inside the function. It is easy to compute the function in the inverse way once we know what the trapdoor is. This concept has drown a lot of attention by other people.

The R.S.A. Cryptosystem

Two years later after the publication of Diffie and Hellman, Rivest, Shamir and Adleman [17] from MIT used that concept and devised a safer algorithm which is called RSA cryptosystem. Nowadays the RSA cryptosystem has been widely used for the Internet e-commerce. Its core idea is the integer factorisation and it also uses lots of mathematical theorems such as Euler's Theorem and Fermat's Little Theorem. It works as follow.

- Find two large primes p and q.
- Suppose $n = pq$, calculate the $\phi(n) = (p-1)(q-1)$
- Find a number e which is relatively prime to $\phi(n)$, namely gcd(e, $\phi(n)$)=1
- Finally find a number d such that $ed = 1 \bmod \phi(n)$

The enciphering and deciphering function are defined as follow.
Enciphering function:$E(x) = x^e \bmod N$
Deciphering function:$D(x) = x^d \bmod N$

Theoretically speaking, the encryption and decryption in RSA is not difficult. The key thing to ensure the safety of the system is the integer factorization. Currently there are no algorithms which can factorize N in a reasonable time for large enough p and q.

2.2 The AKS Algorithm

In 2002, Manindra Agrawal, Neeraj Kayal and Nitin Saxena [18] found a deterministic primality testing algorithm in polynomial time which relies on no unproved assumptions. This is called AKS algorithm after their names. This algorithm guarantees to determine whether an input integer number is prime or not in the running time of $O(\log n)^{12}$.

2.2.1 Basic Idea

The AKS algorithm uses the following lemma which is a generalization of The Fermat's Little Theorem.
Lemma 2.1: Suppose a is coprime to n, then n is prime if and only if

$$(x-a)^n \equiv x^n - a(n). \tag{2.11}$$

To prove it, consider the coefficient of x^i in $((x-a)^n - (x^n - a))$, which is $(-1)^i \left(\begin{array}{c} n \\ i \end{array} \right) a^{n-i}$, where $0 < i < n$. Suppose n is prime then $\left(\begin{array}{c} n \\ i \end{array} \right) = 0 \pmod n$ and hence all the coefficients are zero.

An example to make it clearer, n = 7, a =1:
$(x-a)^n - (x^n - a)$
$= -7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x$
$= 7[-x^6 + 3x^5 - 5x^4 + 5x^3 - 3x^2 + x] = 0(\text{mod } 7)$. Therefore 7 is prime.

On the other hand, suppose n is not prime. It then has a prime factor q of n, the qth term will be $(\frac{n}{q})K$ instead of $(n)K$ where K denotes a cetain polynomial, hence the coefficient of x^q is not zero (mod n). Lemma 2.1 takes approximately n^2 operations involving n terms and it is not efficient for a polynomial time algorithm. To reduce the number of operations, AKS algorithm reduces the number of terms by considering (2.11) modulo a polynomial of the form x^r -1 where r is chose to be small enough. In other words, we need to test if the following equation is satisfied:

$$(x-a)^n \equiv (x^n - a)(x^r - 1, n). \tag{2.12}$$

It follows immediately that if for all primes n satisfy (2.11), it will automatically satisfy (2.12) for all values of a and r. However if n is composite, (2.12) will also be satisfied for some certain values of a and r. In other words, there exists some pairs(a, r) which will return a false result by the algorithm. So it is necessary to repeatedly test the correctness of

(2.12) for a suitable range of a's. The AKS article stated that testing a from 1 to $2\sqrt{r}\log n$ is sufficient.

2.2.2 Algorithm Overview

Below is algorithm taken from the original AKS article[27].

```
Input: Integer $n > 1$
1. if ($n$ is of the form $\mathrm{a}^b, b > 1$) output COMPOSITE;
2. $r = 2$;
3. while($r < n$){
4.         if(gcd$(n, r) \neq 1$) output COMPOSITE;
5.         if ($r$ is prime)
6.             let $q$ be the largest prime factor of $r - 1$;
7.             if $(q \geq 4\sqrt{r}\log(n)) \wedge (n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r})$
8.             break;
9.         $r \longleftarrow r + 1$;
10. }
11. for $a = 1$ to $2\sqrt{r}\log n$
12. if($(x - a)^n \not\equiv (x^n - a)(\bmod\ x^r - 1, n)$) output COMPOSITE;
13. output PRIME;
```

Figure 2.1: The original AKS Algorithm

All logarithms are to the base two throughout this dissertation.

A proof of correctness of the above algorithm is shown in the section 4 in the original AKS article[13]. The AKS algorithm can be considered as a filter for composite numbers which are detected at either at line 1, line 4 or line 12. Only prime numbers can reach line 13. The aim of the while loop is to find the smallest useful r for a given number n. Lemma 4.2 in the AKS original article [13] assures that a useful r are in the interval$[c_1(\log n)^6, c_2(\log n)^6]$ as long as n is sufficiently large. Thus the loop breaks at line 8 for no more than $c_2(\log n)^6$ iterations once r reaches n. This ensures that the algorithm works in polynomial time.

Line 1 It detects composite numbers of the form $a^b, b > 1$, where a and b are positive integers.
Line 4 It detects composite numbers which have a smaller prime factor than the useful prime[3] r.

[3] r is a useful prime when it satisfies $(q \geq 4\sqrt{r}\log(n))$ and $(n^{(\frac{r-1}{q})} \not\equiv 1 \pmod{r})$

Line 12 Composite numbers have not been detected above are detected here. This is the main detection of the algorithm.

Overall Behaviour of the Algorithm

- If n is of the form a$^b, b > 1$, then it returns composite at line 1.
- n is composite but not of the form a^b or n is prime.
    - A useful prime does not exist.
        - $*$ n is composite. It returns composite at line 4.
        - $*$ n is prime. The while loop iterated until $r = n$. The for loop iterates until $a = 2\sqrt{r}\text{log}n$. It returns prime at line 13.
    - A useful prime exists.
        - $*$ n is composite.
            - · n has a prime factor which is less than the smallest useful prime for n. It returns COMPOSITE at line 4.
            - · n does not have a prime factor. The while loop finds a useful prime r which is less than n and breaks at line 8.
It returns COMPOSITE at line 12.
        - $*$ n is prime. The while loop finds a useful prime r which is less than n and breaks at line 8. The for loop breaks when $a = 2\sqrt{r}\text{log}n$.
It returns PRIME at line 13.

2.2.3 Complexity Time

The asymptotic time complexity of the algorithm is $\tilde{O}((\text{log}n)^{12})$.
In the following, we analyse line by line to estimate the time complexity.

Line 1: if(n is of the form a^b , $b > 1$) output COMPOSITE;
This is done by checking if there is an integer in the following sequence: $n^{\frac{1}{2}}, n^{\frac{1}{3}}, ..., n^{\frac{1}{}}\text{log}n$.
Therefore the asymptotic complexity time is $O((\text{log}n)^3)$.

Line 2: $r = 2$ The complexity time is constant.

Line 3 to 10 While($r < n$){...}
Since r is in the range $[c_1(\text{log}n)^6$, $c_2(\text{log}n)^6]$ by Lemma 4.2. So the number of iterations of the while loop is O((log $n)^6$).(2.31)

Line 4: if(gcd(n, r)$\neq$ 1) output COMPOSITE: poly(log n).(2.32)

Line 5: if(r is prime)

Line 6: let q be the largest prime factor of r -1: $r^{\frac{1}{2}}$poly(log$n$)(2.33)
Line 7: if ($q \geq 4\sqrt{r}$log(n))and($n^{\frac{r-1}{q}} \not\equiv 1$ (mod r))

Line 8: break;
Line 9: $r \longleftarrow r$ +1;
poly(log$n$)(2.34)

From (2.31), (2.32), (2.33), (2.34), total complexity time for the while loop is
$O(($log$n)^6 r^{\frac{1}{2}}$poly(log$n$)) $=O(($log$n)^9$poly(log$n$)) [since $r = O(($log$n)^6)$]

Line 11: for $a = 1$ to 2 $\sqrt{r}$log$n$
Line 12: if($(x-a)^n \not\equiv (x^n - a)$(mod x^r -1, n)) output COMPOSITE;

The for loop does modular computation over polynomials. Fast Fourier Multiplication can be used then each iteration takes $O(r($log$n)^2$poly(log(r(log$n)^2$))) steps.
So the complexity time is:
$O(r($log$n)^2$poly(log(r(logn)$^2 r^{\frac{1}{2}}$log$n$)
=$O(r^{\frac{2}{3}}($log$n)^3$poly(log(r(log$n)^2$))
=$O(($log$n)^{12}$poly(log(log$n)^8$))
=$O(($log$n)^{12}$poly(8loglog$n$))
=$O(($log$n)^{12}$poly(12loglog$n$))
=$O(($log$n)^{12}$poly(log(log$n)^{12}$))
=$\tilde{O}(($log$n)^{12})$

We use the symbol $\tilde{O}$(t(n)) for O(t(n)poly(logt(n))^k, where t(n) is any function of n. For example, $\tilde{O}($log$n)^k = O(($log$n)^k$poly(loglog$n$)) $= O($log$n)^{k+\varepsilon}$ for any $\varepsilon > 0$.

2.2.4 Improved Complexity and Sophie Germain Primes

There is has been a claim in AKS-paper [26]:

”inpractice, however, ouralgorithm is likely to work much faster.”

Their justification is as follows: The AKS-paper states the following lemma:
Lemma 2.1 Let $P(n)$ denote the greatest prime divisor of n. There exist constants $c > 0$ and n_0 such that for all $x \geq n_0$

$| \{p \mid p$ is prime, $p \leq x$ and $P(p-1) > x^{\frac{2}{3}} | \geq c\frac{x}{\log x}$.

An immediate corollary is there are many primes r such that $P(r-1) > r^{\frac{2}{3}}$.
This property is used in the proof of Lemma 2.1 which determines there is a suitable $r = O(\log n)^6$. The value of r played a crucial role in determining the overall complexity of the algorithm. So if a stronger condition can be proved then a smaller value of r can be determined which will consequently reduce the running time of the algorithm.

Definition 2.6 Sophie Germain Prime
A prime r is said to be a Sophie Germain prime if $2r + 1$ is also prime.

Conjecture 1[4]
The number of Sophie Germain primes less than n is asymptotic to $\frac{Dn}{(\log n)^2}$ where D is the twin constant (approximately 0.660161).

This conjecture is widely believed to be true. A direct consequence of the proof of is for many primes $r, P(r-1) = \frac{r-1}{2}$. If this is the case then the authors of the AKS-algorithm present a simple argument which shows a suitable r of size $O((\log n)^2)$can be found. This is formalized as follows:

Lemma 2.1 Assuming the Sophie Germain conjecture (Conjecture 1) there exists suitable r in the range $64(\log n)^2$ to $a(\log n)^2$ for all $n > n_0$, where n_0 and a are positive constants.
Where a suitable r is one which satisfies conditions of (2.3) in section 2.2. A proof for the Sophie Germain Conjecture therefore finds $r = O((\log n)^2)$.

2.3 Modifications on the Original AKS Algorithm

”We will have to wait for efficient implementations of this algorithm (and hopefully clever restatements of the painful for-loop) to see how it compares to the others for integers of a few thousand digits. Until then, at least we have learned that there is a polynomial-time algorithm for all integers that both is deterministic and relies on no unproved conjectures!”

Chris Caldwell[19]

[4] Taken from Chapter 5 of [26]

Since the advent of AKS algorithm, most of researches have been attempting to reduce the complexity time of the algorithm. There are two major approaches which can achieve this:

- Working on restatement of the for loop to reduce the number of iterations.
- Finding more efficient ways to calculate the useful prime r.

2.3.1 Dan Bernstein and Lenstra's Improvements

Soon after the publication of AKS paper, Lenstra [20] modified the AKS algorithm with the observation made by C. Pomerance who observed that q need not to be the largest prime divisor of r-1. The major difference is the construction of the useful prime r. This modification reduces the number of iterations in the for loop. Hence the complexity time is reduced to $O(\log n)^6$.

```
Input: Integer $n \geq 2$
if ($n$ is of the form a$^b, b > 1$) output COMPOSITE;
$r = 2$;
while($r < n$){
    if(gcd$(n, r) \neq 1$) output COMPOSITE;
    if $o_r(n) > \lfloor \log^2(n) \rfloor$
        break;
    $r \longleftarrow r$ +1;
}
for $a = 1$ to $\phi(r) - 1$
if($(x - a)^n \not\equiv (x^n - a)(\text{mod } x^r - 1, n)$) output COMPOSITE;
output PRIME;
```

Figure 2.2: The AKS Algorithm Modified by Lenstra

Where $o_r(n)$ denotes the order of n mod r, to be the smallest number $k \geq 1$ such that $a^k \equiv 1$ (mod r).

Daniel Bernstein interpreted the modification made by Lenstra in his paper[21] and made his own improvements. His algorithm is listed below:

```
Input: Integer $n > 1$
if ($n = a^b$ for $a \in N$ and $b > 1$) output COMPOSITE;
$r = 2$;
while($r < n$){
        if(gcd$(n, r) \neq 1$) output COMPOSITE;
        if ($r$ is prime)
```

```
            let $q$ be the largest prime factor of $r$ -1;
            if $\begin{pmatrix} 2q-1 \\ q \end{pmatrix} \geq 2^{2\lfloor\sqrt{r}\rfloor logn}$ and $n^{\frac{r-1}{q}} \leq 1$
            break;
        $r \longleftarrow r$ +1;
}
for $a = 1$ to $2\sqrt{r}\log n$
if$((x-a)^n \not\equiv (x^n-a)(\text{mod } x^r-1, n))$ output COMPOSITE;
output PRIME;
```

Figure 2.3: The AKS Algorithm Modified by D. Bernstein

This algorithm also modifies the way of constructing r and hence reduces the number of iterations of the painful for-loop.

2.3.2 Other Remarkable Results

Felipe Voloch's improvement [22]
He has shown that the group considered in chapter 4 of AKS paper is bigger than claimed. As a result, a smaller range of a's need to be checked in the line 12 Fig 2.1.

Qi Cheng [23]
The reduction in time complexity is achieved by first generalizing Berrizbeitia's algorithm to one which has higher density of easily-proved primes. For a general prime, one round of ECPP is deployed to reduce its primality proof to the proof of a random easily proved prime. The algorithm has a running time of $O(\log n)^4$.

Pedro Berrizbeitia [24]
Presenting a deterministic primality tests with running time of $\tilde{O}(\log n)^6$. For integers $n \equiv 1 \pmod 4$ such that for a given integer a, $(\frac{a}{n})$ = -1 and for integers $n \equiv -1 \pmod 4$ such that $(\frac{a}{n}) = (\frac{1-a}{n})$=-1.

Dan Bernstein [25]
Generalising Berrizbeitia's improvements and presents an algorithm that, for a given ieteger n, finds and verifies the primality of n in random time $(\log n)^{4+o(1)}$.

2.3.3 AKS Later Version

In the later version of AKS paper [26] the original algorithm has been improved based on various modifications by other people. This version of algorithm which listed below eliminates the while-loop and uses the idea of Lenstra for constructing r. Hence the complexity time is reduced to $\tilde{O}(\log n)^{15/2}$.

```
Input: integer $n > 1$.
1. If ($n = a^b$ for $a \in N$ and $b > 1$), output COMPOSITE.
2. Find the smallest $r$ such that $o_r(n) > (\log n)^2$.
3. If $1 < (a, n) < n$ for some $a \leq r$, output COMPOSITE.
4. If $n \leq r$, output PRIME.
5. For a = 1 to $\lfloor \sqrt{\phi(r)} \log n \rfloor$ do
   if ($(x + a)^n \not\equiv x^n + a$ (mod $x^r$ - 1, $n$)), output COMPOSITE;
6. Output PRIME.
```

Figure 3.1 : The AKS algorithm

2.4 Summary

This literature review has discussed subjects about AKS algorithm and its relevant areas that determine the scope of the project. It began with a brief introduction of prime numbers and followed by some definitions in number theory and algorithm terminology, because they are the basic knowledge for understanding what AKS algorithm is, what it is for, why it is special and how it works. Next it was about the history of prime numbers and suggested two main research areas on prime numbers. The most important area that was identified with respect to this project is to find a faster algorithm or improve a current available one which determines that a given number is whether prime or composite. It was then followed by a brief description of primality testing algorithm development in the last 30 years. The first section ended in a brief overview of the algorithm applications —Cryptography.

It was then necessary to discuss the AKS algorithm in the second section. Firstly the basic idea of this algorithm was presented and a brief proof of its correctness was given. Then the algorithm was examined line by line to determine the overall behaviour of it, because it is helpful for implementation in the later stage of the project. Furthermore the algorithm was analysed line by line to estimate the time complexity which is $O(\log n)^{12}$. In the third section some improvements on the original AKS algorithm was presented.

The result of this review is that it would be worthwhile to implement the AKS algorithm and some improved algorithms in order to produce experimental data to support claims made regarding complexity times. Hopefully during the project, I shall come up with some further suggestions on improvements on these algorithms which will contribute to the field of primality testing.

Chapter 3

Requirements

3.1 Overview of System

This document provides the requirements for a single-user application, which is to determine whether an input number is prime or not. The requirements specification needs to be addressed properly because they are the basis to the implementation of later stage.

3.2 Requirements Analysis

3.2.1 Evaluation of Previous Systems

The previous typical methods are Rabin-Miller's method and Adleman, Pomerance and Rumely's algorithm. The rabin-miller's method has a probability of error in identifying primes so it is not guaranteed to output correct result. although the Adleman, Pomerance and Rumely's one are guaranteed to output correct result, they runs not in polynomial time.

3.2.2 Core Functionality Use Cases

Use Case: input a Number
Primary Actor: Any user
Frequency: Always
Preconditions: N/A
Post conditions:User gets an answer.
Main Success Scenario: System outputs whether the given number is prime or composite.
Extensions and Variations:

- User input valid
  - if the input is a prime number, show user the result and time taken.
  - if the input is a composte number, show user the result and time taken.
- User input invalid
  - if the input is a non possitive number, error message displayed.
  - if the input is not a number, fatal error, program terminates.

3.3 Functional Requirements Specification

3.3.1 Input Data

- The system should allow user to input data at run-time.
- The input data must only be a large positive integer which might comprise at least 20 decimal digits. Initially the requirements for the input numbers support up to 50 digits, currently I do not know how fast these algorithm works, it should be capable to identify composite up to 50 digits in a reasonable time, but probably not true for prime numbers, so the input numbers will be limited to 20 digits.
- The system should not allow user to input any illegal data.

3.3.2 Output Data

- The system should be able to output results correctly and effectively.
- The system should be able to display the output result and relevant information concisely.
- The system should be able to store all the existing data.

3.3.3 Error Handling

Since the potential errors which will only occur when ask user for input numbers, the system should then try to avoid all possible input error or handle them and give user proper error message.

3.4 System Requirements

Below are the general system requirements which will enhance the usability of our system.

- The program code should be simple, efficient and along with clear comments wherever possible.
- The system must not include unnecessary functionality

3.5 Non-Functional requirements

- The platform for system should be Microsoft windows 95 or above or Unix.
- Minimum computer memory required 32mb and available disk size 500mb.

Chapter 4

Design

It is important that the design of the final program is carefully considered in order to ensure the successful implementation of the algorithms. The main design is divided into several aspects which are justification of the programming language used, usability of the program and basic program structure.

4.1 Choosing Programming Language

The first choice was thought to be Java because it is the one I am familiar with. But when I did some search on the Internet, I found a portable library called NTL is highly recommended by others. Below is taken from Victor Shoup's homepage [27].

"(Shoup's) NTL (Number Theory Library) is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields."

NTL effectively deals with the manipulation of large integers, polynomial arithmetic and modulo arithmetic. All of these provided by NTL will be required in the implementation stage. So the program will be written using C++ because NTL is a C++ library.

4.2 Usability

As stated in the requirements chapter, Designing fancy user interfaces is not a priority of this project. However it is important that the input and output to the user are designed in a way to ensure the usability of the program.

4.2.1 Input

The program should allow user to input data at run-time. This is essential because if the program requires the input data to be explicitly stated in the source code then the user will have to change the code of the program for each number to be tested. This is time consuming and inappropriate because user will have to recompile the code and introduce possibility of errors if the code is inadvertently changed.

Some algorithms such as the Sieve of Eratosthenes (Section 1.2) perform efficiently only when providing small numbers. We have seen, especially in cryptographical areas, a good useful algorithm should accept numbers of large magnitude. Furthermore in requirements chapter, section 3.2.1 will be achieved with wider range of input numbers comprising of at least 20 decimal digits (opposed to the 9 decimal digits of a standard int type). This preliminary difficulty can be resolved by using NTL.

4.2.2 Output

Obtain a result is why users use the program, therefore the output is the most important part of the program and should be clear, concise and convey the desired information to the user. The output result should be stored to a file so that they can be recalled later for the purpose of verifying the conclusions. Consideration also needs to be paid to the information expected by the user from the algorithm. i.e. whether the input number is prime or not? What are the useful prime r and the largest prime factor q to the input number n?

4.2.3 Program Structure

The basic structure of our implementation has been mostly pre-determined because it is based on the established AKS algorithm. On observation of the AKS algorithm (Fig 2.1) it is obvious that the algorithm has clearly defined independent tests. From object oriented view, it is natural to decide to write such tests in separate head files making the final code easier and clear to read and modify. Writing the program in this way will also make our testing easier.

The program will terminate under one of the following conditions and return type is specified.

- If the input n fails under any test then n is composite and 0 is returned.
- If the input n passes all tests then n is prime and 1 is returned.

Line Number	Tests	Inputs	Return Type
1	Perfect power check	n	1/0
4	Is GCD $(n, r)! = 1$	n, r	1/0
5	Is r prime?	r	$r+1$
6	Prime factor	$r-1$	1/0
7	Conditions on q	-	-
12	Evaluate 2.12	a, r, n	1/0

Table 4.1: Table showing program structure of the AKS algorithm. Line numbers refer to Fig. 2.1

The same observations can be made for the modification of Lenstra and D.Bernstein.

Line Number	Tests	Inputs	Return Type
1	Perfect power check	n	1/0
2-8	Find r	-	-
9	Calculate Euler-Fun(r)	r	EF(r)
10	Evaluate 2.12	a, r, n	1/0

Table 4.2: Table showing program structure of the AKS algorithm modified by Lenstra. Line numbers refer to Fig. 2.2

Chapter 5

Implementation and Testing

In the implementation stage, we will look how the program is going to be developed. Some of the key tests mentioned in tables of the design stage will be discussed in the order they occur in each of the three algorithms.

5.1 The AKS Algorithm

5.1.1 Perfect Power Check

If the input number n fails in this step then the program will terminate. This test is to determine whether the input number n is a perfect power which means it checks if n can take the form a^b. Then the maximum value of b can take is $\log(n)$. Thus the problem can be reduced to just determining whether there exists an a such that $a^b = n$ for all b between 2 to $\lfloor \log(n) \ \rfloor$. Finally this can be converted to the algorithm below.

```
int root($p, b$)  /* Returns $\lfloor p^{1/b} \rfloor$. */
$a = 2^{\lfloor B(p)/b \rfloor}$; /* B($n$) = number of bits in $n$. */
    while(1) {
    $y = \lfloor ((b-1)x + p/a^{k-1})/b \rfloor$;
    if($y > a$) return $a$;
    $a = y$;
    }
}
```

Figure 5.1: Algorithm to determine if n is perfect power.

This algorithm can be further optimized, e.g., there is no need to take the b-th power if int_root() returns an even value. However, the improvement can be ignored because the time spent on this initial step of the AKS algorithm is negligible in practice.

5.1.2 The Certificate

The next stage of the algorithm (line 3 to 10 of figure 2.1) is to find a value of r which satisfies ($q \geq 4\sqrt{r}\log(n)$) and ($n^{(\frac{r-1}{q})} \not\equiv 1 \pmod{r}$).

Greatest Common Divisor

We get to this point if n is not a perfect power, but this does not mean that n does not have other non-trivial factors such as coprimality. If n and r is not coprime, then n is in fact composite so that the program will terminate at this point. The most common way of checking the coprimality is to calculate the greatest common divisor of two integers. Recall (1.1.3) the Euclidean algorithm is a good approach. However, tjhere is a predefined way of calculating GCD for ZZ class (large integers) in Shoup's NTL. Because there is already a method available and it has been developed to optimize the performance for large numbers, we will take advantage of it.

Is r Prime?

Line 5 of Fig. 2.1 requires checking whether r is a prime number. Recall in section 2.2, Lemma 4.2 in AKS original article [18] assures that r exists in the range of less then $c_2(\text{logn})^6$ (where c_2 is a constant). Therefore r is smaller enough so that we can apply an unconditional deterministic primality test algorithm in non-polynomial time.

```
Input: Integer $r \geq 2$
s= 2;
t= 4; (t=$s^2$)
while ($t <= r$){
    if $r$ mod $s = 0$
        then return $r$ is composite.
        else $s = s + 1$;
        $t = t + 2s - 1$; (note $t = s^2$)
}
return $r$ is prime.
```

Figure 5.2: Algorithm to determine if r is prime or composite.

This algorithm is based on the result that r is composite if there is at least one divisor r_1 such that $r_1 <= \sqrt{r}$. In order to ensure all values from 1 to r have been tested as possible divisors of r, we define $t = s^2$. We will implement this definition by addition and subtraction to ensure better performance. If r is prime, we move to line 6 of Fig. 2.1, otherwise the value of r will be increased and the algorithm will be repeated.

Largest Prime Factors

Line 6 of Fig. 2.1 requires a new variable q to be defined and assigned the value of the largest prime factor of r-1. In other words, for an integer $n >= 2$, let $q = \text{lpf}(r\text{-}1)$ denote the largest prime factor of r-1, i.e., the number p_k in the factorization

$$n = p_1 p_2 .... p_k$$

So the algorithm needs to derive p_k and assign this value to q.

```
Input: Integer $r - 1 \geq 2$
f= 1;
g= 2;
x = r - 1;
while (x $\neq$ 1&$\sqrt{g} <= r - 1$) {
    while ( $x \bmod g \equiv 0$ ) {
        else $x = x/g$;
        $f = g$;
    }
    $g = g + 1$;
}
return f;
```

Figure 5.3: Algorithm to determine the largest prime factor of r - 1.

In this algorithm, we divide $r - 1$ by 2 to $\lfloor \sqrt{r-1} \rfloor$ for each inner loop and save the value of g to f until x is equal to 1. Then we return f which is the value of q.

Conditions on q

The value of q has been set and we are ready to move to line 7 of Fig. 2.1 in which additional conditions of q must be met in order to move forward. The first condition is an inequality which can be written in the main program. The second condition is also an inequality and requires modular arithmetic. For integer numbers which are less than 9 digits, the division operation is efficient enough to compute the corresponding remainder. However this method becomes unequal to large numbers. Therefore, an efficient alternative for modular arithmetic needs to be employed.

The Barrett method [28] uses division and shift operations to perform modular arithmetic. The division he uses is different from naive division operation and hence less time consuming. Crandall and Papadopoulos [29] introduced a better version of this method which is an extension of the Barrett method for large numbers. In their method, only multiplication and shifting is required.

The Final Congruence

We are now in the final stage to consider line 12 of Fig. 2.1 which is the most time consuming part and surely needs necessary focus of optimization.

Arithmetic of two polynomials f and g can be calculated by reducing the problem to arithmetic of large integers. The coefficients of each polynomial are combined in a single string separated with a sufficient number of 0s in-between each value.

Binary Segmentation is a common method for efficiently multiplying large polynomials together. For example, to multiply two polynomials $f = \Sigma_i f_i x^i$ and $g = \Sigma_i g_i x^i$ each of degree (r -1), one may forge two (typically large) integers:

$$F = 0...0f_{r-1}0...0f_{r-2}0......0f_0,$$
$$G = 0...0g_{r-1}0...0g_{r-2}0......0g_0$$

The integer product will appear as a set of coefficient cells with small zero-pads between each coefficient. The zero padding prevents coefficient sums from spilling over and interfering with each other during the integer multiply. This reduction of a polynomial operation to one involving large-integer arithmetic has various advantages such as: ease of implementation, software optimisation and true exploitation of reduced complexity for squaring. On this last point, if $f = g$ then we simply do a large-integer square F^2. Crandall and Papadopoulos [29] have demonstrated that this method is even faster than Fast Fourier Transform.

NTL Polynomial Classes

It has been shown that NTL has its own polynomial routines which are more efficient than binary segmentation. The classes in NTL we used here are ZZX and ZZ_pX, the former is polynomials with large integer coefficients and the latter is polynomials with large integer coefficients mod p for some large integer p.

The left hand side of line 12 Fig. 2.1 is performed by using a routine from ZZ_pX. It calculates f^nF where f and F in our context is $x - a$ and $x^r - 1$ respectively.

The right hand side cannot be performed directly. But there is a property of modular arithmetic that we can use:
$(F_1$ (x)$+F_2$ $(x))$ mod $G(x) = (F_1$ (x) mod $G(x))+(F_2$ (x) mod $G(x))$

Letting $F_1(x) = x, x^n$ mod $x^r - 1$ can be determined using the same routine as above, denote by $F_3(x)$. Then setting $F_1(x) = ax^r - 1$ the right hand side can be determined as $F_3(x) - F_2(x)$.

5.2 AKS Algorithm - Lenstras Modification

The Lenstras modification is not completely different from the original AKS algorithm and it also starts with perfect power check.

5.2.1 The Certificate

In the certificate stage of finding a suitable value of r, the coprimality test by calculating GCD also remains the same as before. However, other conditions on r are mostly different from the original AKS algorithm.

Line 5 of Fig. 2.2 requires that the order of the input number n mod r is greater than $\lfloor \log^2(n) \rfloor$ denoted by m. We want r to be as small as possible to increase the efficiency of our algorithm. So we start with r of value 2, first ensure r is coprime to n and then calculate n^m mod r. In mod p arithmetic the order of any number is less than or equal to p, so we will only continue the process if $m \leq r$. otherwise m is set to its original value and r is increased by 1. This process stops once a suitable r is found.

5.2.2 Euler-Phi Function

Lenstras modification requires $\phi(r) - 1$ times of the final for loop where $\phi(r)$ is the Euler-Phi function.

Euler's totient function, $\phi(n)$, is defined as the number of natural numbers $\leq n$ which are relatively prime to n. It is an extremely important function in number theory. For primes p it is clear from the definition that $\phi(n) = p - 1$. For powers of a prime it also easy to see

(use induction on n) that $\phi(p^n) = p^{n-1}(p-1)$. Thus, e.g, $\phi(25) = 20$. For all other values phi can be computed by factoring n completely.
The remainder of the algorithm is identical to what we discussed for the original AKS algorithm.

5.3 Testing

After the implementation, effective testing needs to be performed in order to make sure the program performs correctly and meets the requirements.

Firstly, in order to test the system we need to compare it with initial requirements. The testing of the system involves the analysis the system in order to validate that the system meets its initial requirements. We mainly concern on testing systems functional requirements. This includes checking whether the design and the implementation meet the requirements.

Secondly, We need to test if the algorithm works correctly. Black box testing will be used instead of white box testing. Because black box testing is the functional technique of testing where the tester does not need to know how a unit works but only what function(s) it is supposed to perform. To test a part of the system using black box techniques only the description of the functionality of that part of the system are needed. Apparently, there are clear subdivisions of the algorithm in section 4.2.3. As a result, there are several places where the input number can be possibly identified as a composite and this will lead to the termination of the program. Furthermore, because these functions are independent to each other, most of the testing can be done on each individual component. This can be achieved by testing the each component with a wide range of numbers. Each test is deemed to be successful if it produces the expected result. Moreover, some suspected parts will be tested with certain inputs in order to make sure correct behaviors for all numbers.

Furthermore, the purpose our testing is not only verify the correctness of the algorithms but also analyze their performance and complexity time. Black box and isolation testing will be used. However this time we will focus on the output results, so timing analysis is an important testing target. A number of testing program will be produced in order to ease the testing.

After the validation of system requirements and individual component testing, we will start to collect data and result produced by the program in the next chapter.

Chapter 6

Results

Firstly we validate the system by checking the design and implementation meet the initial requirements, mainly the functional requirements. Secondly each individual component will be tested and the complexity time of each algorithm will be measured.

6.1 Validation of Requirements

6.1.1 Input Data

The program is capable of manipulating large integer of any size and distinguishing illegal input data.

6.1.2 Output Data

The programs output results and relevant information are correct and as expected. (See below for further testing information) All the existing results and data are stored into files.

6.1.3 Error Handling

The program handles illegal input data as expected. Any non positive integer number will be detected and error message will be displayed to ask user to input an correct number. Any illegal data will lead to program termination.

6.2 Individual Component Testing

In order to ensure our algorithms output the correct result, we need to make sure each individual component test performs correctly.

6.2.1 The Correctness of Perfect Power Check

The following table proves the correctness of perfect power check test.The numbers are intentionally selected in each different number of digits. Number denotes the number selected. In the result column, 0 means the number is not a perfect power, otherwise result of a^b will be displayed.

Bits	Number	Result
1	4	2^2
1	7	0
5	42875	35^3
5	72355	0
10	1564031349	69^5
10	8812214355	0
15	582622237229761	24137569^2
15	987687645327744	0
20	26663523546439124560 7	6436343^3
20	43454657619840381796	0
25	3334462679518153070884 93	69343957^3
25	5435657909123436543098 75	0
31	1271991467017507741703714391419	59^{17}
60	401007068543157803727680343536350900670553508041935397795649	912673^{10}

Table 6.1: Table showing the correctness of perfect power check

Our test works correctly for all the numbers we test and the time spent are within a single second. It appears that it should work for arbitrary long numbers.

6.2.2 The Correctness of Isprime Check

The numbers are intentionally selected in each different number of digits. Number denotes the number selected. In the result column, 1 means the number is prime, 0 otherwise. Since this test is used to check the primality of useful number r which is significantly smaller than input number n, so we will test up to 10 digits.

Bits	Number	Result	Number	Result
1	5	1	7	1
2	37	1	51	0
3	227	1	355	0
4	1850	0	2953	1
5	48821	1	81796	1
6	707280	0	814327	1
7	5721719	1	7467894	0
8	12324357	0	39323857	1
9	621652327	1	987678942	0
10	1450794557	1	9798822121	0

Table 6.2: Table showing the correctness of isprime check

The above table shows that the isprime test works correctly and the time spent are within a single second.

6.2.3 The Correctness of largest prime number

The numbers are intentionally selected in each different number of digits. Number denotes the number selected. Result denotes the largest prime factor of the input number.

Bits	Number	Result	Number	Result
1	5	5	9	3
2	36	3	51	17
3	121	11	355	71
4	5780	17	7744	11
5	54678	701	81796	13
6	707281	29	897654	89
7	1046529	31	5467893	20479
8	25563136	79	98654327	50411
9	405821025	79	987678942	49384471
10	4565765123	112799	9798822121	8999
11	23243549815	4648709963	42307964721	271
12	454657909123	41332537193	707678007696	6373
13	4062736078129	191	6789009812435	342965891
14	57148035975424	472477	78909845321237	11272835045891
15	104978036482641	2357	234543659210921	119995303
16	7095651301304721	228281	8909876545123254	68013871
17	13254618902157618	4391855169701	45611564418586881	101

Table 6.3: Table showing the correctness of largest prime factor

The above table shows that it calculates the largest prime factor of an input number correctly and quickly.

6.3 Useful r Testing

A test for investigating the properties of useful r is created and it is a reduce form of AKS algorithm which can be found in the directory called "Useful r test" in the accompanying cd-rom.
Another test for investigating the properties of certificate r is created and it is a reduce form of Lenstras modification to the AKS algorithm which can be found in the directory called "Lenstra certificate r" in the accompanying cd-rom.
A collection of primes from 1 bit to 18 bits were tested by the above two tests and the results are in the following table. N denotes the input number, T denotes time taken in seconds.

N	AKS useful prime r	T	Lenstra's certificate r	T
5	5	0	2	0
37	37	0	13	0
977	977	0	41	0
2909	2909	1	56	0
11699	11699	1	90	0
11701	11699	2	84	0
86599	17327	3	128	0
123457	18443	3	128	0
2284423	28607	6	208	0
96447077	45083	13	336	0
484240567	53699	19	375	0
1307135101	58727	24	416	0
435465768733	96059	64	700	0
3435465768991	111263	87	805	0
65434218790277	134867	150	970	0
700000000000051	155699	250	1202	0
7000000000000037	177323	358	1331	0
10000000000000061	181199	322	1338	0
100000000000000003	204143	412	1569	1

Table 6.4: Table showing the usful r found for AKS and Lenstra's modification

Two tests for investigating the properties of useful r is created. The AKS one can be found in the directory called "Useful r test" and the Lenstra's one can be found in the directory called "Lenstra certificate r test" in the accompanying cd-rom. Both tests found r very quickly, but Lenstra's modification is much much more faster. On average the found r by

Lenstra's algorithm is about 130 times smaller than those found in AKS. The first useful r found by AKS original algorithm is when input number n is 11701. The time taken by both tests increases very slowly in terms of the bits of the input number n. Evidence has also been produced which clearly shows r is an increasing function of the size of the input n.

6.4 Complexity Measurement - AKS Algorithm

Firstly we will investigate what stated by Lemma 2.1 which is the Sophie Germain conjecture (Conjecture 1) there exists suitable r in the range $64(\log n)^2$ to $a(\log n)^2$ for all $n > n_0$, where n_0 and a are positive constants. So there should be an constant $C \geq 64$ such that $r = C(\log n)^2$. A collection of primes from 1 bit to 18 bits was tested and the results are in the following table. N denotes the input number and C denotes the constant.

N	C
7	0.888
71	1.877
709	7.906
7103	43.393
11681	63.981
11689	64.015
71011	65.560
710009	64.060
7100003	64.378
70999997	64.106
700000001	64.270
7000000001	64.049
70000000033	64.108
700000000009	64.018
7000000000009	64.205
70000000000009	64.041
700000000000051	64.023
7000000000000037	64.002
10000000000000061	64.141
100000000000000003	64.011

Table 6.5: Table showing the value of C obtained by our test.

It is clear that the value of C has very little variance once it reaches 64. The first prime which makes C bigger than 64 is 11689, and the next prime is 11699 which is last prime that AKS algorithm cannot find a useful prime r. So we can conclude that once C reaches

64, the AKS algorithm begins to find a useful prime r and reduce the complexity of the final congruence. So we have verified that $r = 64(\log n)^2$ approximately.

6.4.1 Basic Testing

Now we just input a collection of numbers and see what results it will produce. In the following three tables, number denotes the input number, P/C denotes the whether the input number is prime or composite, C1 denotes the composite decided by perfect power check, C2 denotes the composite decided by GCD, C3 denotes the composite decided by the final congruence, r denotes the value of useful prime r if found and T denotes the time spent in seconds.

Number	P/C	C1	C2	C3	r	T
4	C	√				0
7	P				7	0
25	C	√				0
97	P				97	1
149	P				149	5
203	C		√			0
211	P				211	9
278	C		√			0
307	P				307	26
341	C		√			0
421	P				421	34
513	C		√			0
683	P				683	122
774	C		√			0
853	P				853	141
1134	C		√			0
1381	P				1381	490
1677	C		√			0
2131	P				2131	1339
3005	C		√			0
3389	P				3389	2110
4574	C		√			0
13314529	C			√	35879	132
47520727	C			√	41759	146
2111888059	C			√	61547	269
12471284621641824623	C		√			0
2138212141297461294612946927	C		√			0
12946981642961298461982461292352352	C		√			0

Table 6.6: Table showing the basic test of AKS algorithm

The AKS algorithm runs very slows for detecting primes, it is because it has not find a useful prime r while the for-loop makes up most of the running time in the whole process. When the input number is bigger than 11699, it should start becoming efficient. Most the composites were detected in the while-loop. The time taken for composite numbers which were detected by C1 and C2 were within a single sec. Initially the composite numbers which will be detected by C3 are thought to be very larger composites. However the testing results has shown that only composite numbers which are constituted by multiplying two large primes will be detected in the C3. For example 47520727 = 5413*8779. The time of the while loop is very small and can be ignored. The number of stpes of the for loop is estimated as follows:

- The number of iteration of the for loop : $O(r^{\frac{1}{2}} \log n)$
- Each iteration
  - The number of the inside iteration for $(x-a)^n$: $O(\log n)$
  - The number of multiplications of coefficients for LHS in one inside iteration : $O(r^2)$
  - Each multiplication of coefficients : $O(\log n \log\log n)$
- So the whole step is : $O(r^{\frac{5}{2}} (\log n)^3 \log\log n)$

We have verified that $r = 64(\log n)^2$, so $r^{\frac{1}{2}} = \log n$. Hence the whole step is $O((\log n)^8 \log\log n)$.

6.5 Complexity Measurement - Lenstra's Modification

For the investigation into the complexity time of the Lenstra's modification a collection of numbers were tested and the results are shown in the following table. Number denotes the input number, P/C denotes the whether the input number is prime or composite, C1 denotes the composite decided by perfect power check, C2 denotes the composite decided by GCD, C3 denotes the composite decided by the final congruence, r denotes the found certificate r, Euler denotes the Euler_phi function at r and T denotes the time spent in seconds. Any grid will be left blank if not appliable.

Number	P/C	C1	C2	C3	r	Euler	T
7	P				2	1	0.001
17	P				4	2	0.001
125	C	$\surd$					0.001
149	P				30	8	0.04
2547	C		$\surd$				0
3137	P				64	32	0.6
10133	P				88	40	1.7
32561	P				112	48	2.2
160583	P				128	64	7.6
268069	P				154	60	13
1771561	C	$\surd$					0.001
2969023	P				208	96	40
7741009	P				238	96	33
9786539	C		$\surd$				0.001
12434839	P				256	128	72
86457079	P				342	108	97
100982933	P				336	96	61
879673117	P				412	204	176
2492813497	P				456	144	314
90552556447	C			$\surd$	640	256	3
435465768733	P				700	240	725
7000000000009	P				856	424	1895
10000000000037	P				876	288	1116
100000000000031	P				1024	512	3004
984060615693223	C			$\surd$	1176	336	10.5

Table 6.7: Table showing the basic test of AKS algorithm modified by Lenstra

The results were saved in AKS_Lenstra_Prime.doc and also can be found in Appendix B. R. Crandalland J.Papadopoulos [29] showed the complexity time is $C(\log n)^6$, where C is thought to be a logarithm constant. Ideally our implementation should get the same result. To verify if it is true, we will compare $(\log n)^6$ with time taken in terms of the number of instructions computed by CPU which is seconds * 10^6 clocks.

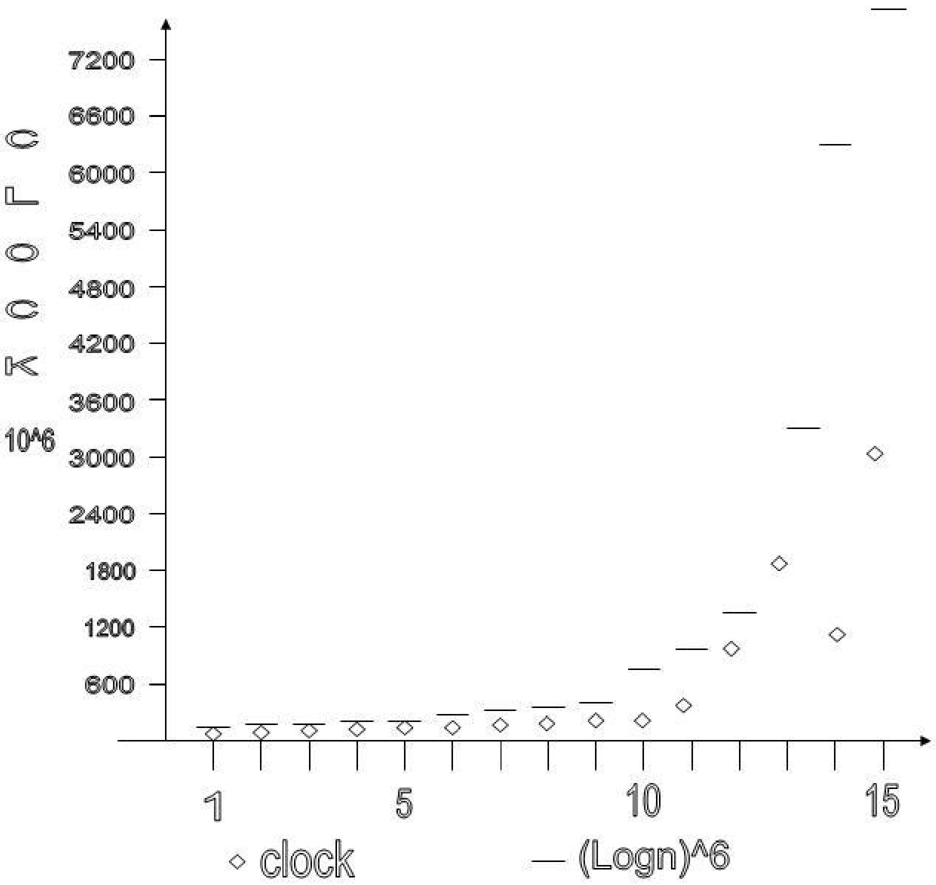

Figure 6.1: Figure illustrating the relationship between the time taken in 10^6 clocks and the expected complexity function $(\log n)^6$. The x-axis represents each n in the sequential order they appear in Appendix B. The actual numberical results can be found in Appendix B or in the file called "Lenstra_complexity_Fig 6.1.doc".

The diamond markers in Fig. 6.1 indicate the time taken by our program for each input n. The horizontal lines indicate the value of $(\log n)^6$. From the figure we can clearly see that $(\log n)^6$ provides an upper bound for the time taken to determine the primality of n. however, as n getting bigger and bigger, the difference between time and $(\log n)^6$ also increases. It seems there is still space for the improvement of the time complexity. In order to find out if it is true, an testing program called "Lenstra complexity" was written which asks for an input number a which is the power to the $\log n$, and it will output the value of $(\log n)^a$ and C (recall $C = T/(\log n)^a$). We will try different a less than 6 and estimate what is the most suitable complexity time for Lenstra's modification.

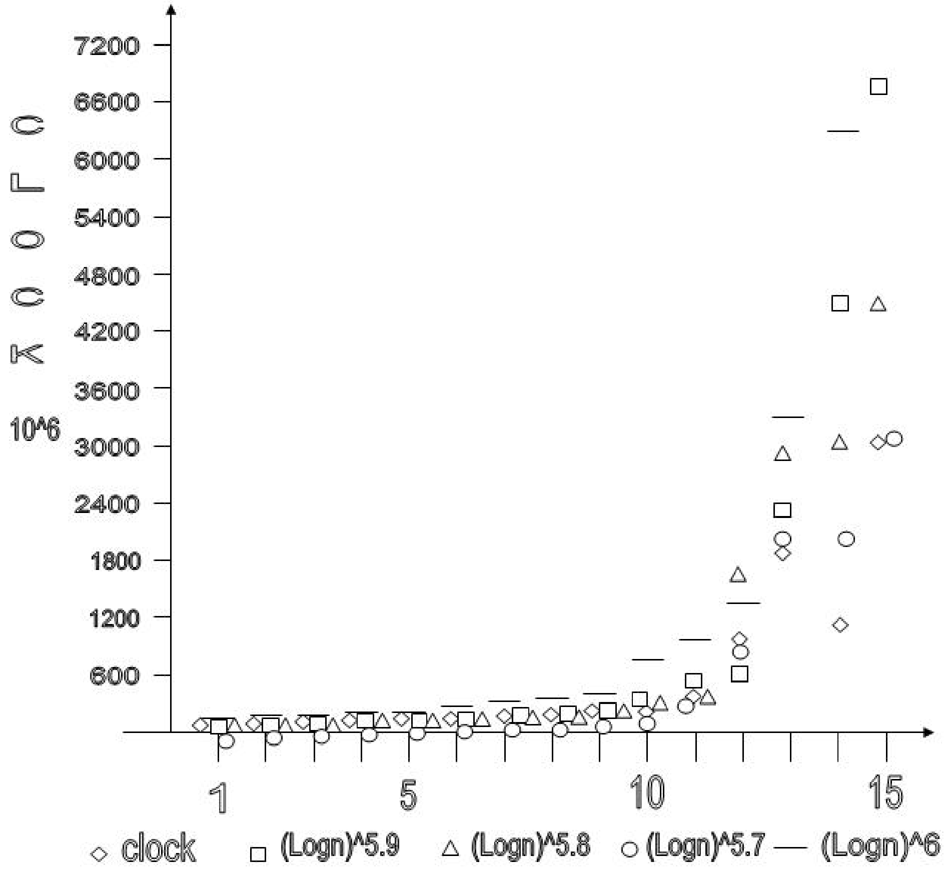

Figure 6.2: Figure illustrating the relationship between time taken in 10^6 clocks and the values of guessing complexity function $(\log n)^a$. The x-axis represents each n in the sequential order they appear in Appendix B. The actual numberical results can be found in Appendix B or in the file called "Lenstra_complexity_Fig 6.2.doc".

The results in Fig. 6.2 indeed produce a good approximation to the logarithmic function. In particular $(\log n)^{5.8}$ and $(\log n)^{5.7}$ appears to be better estimates. $(\log n)^{5.7}$ can be a good lower bound for primes less than 10 digits. So it suggests that there must be a better approximation between $(\log n)^{5.8}$ and $(\log n)^{5.7}$. After a series of trial by using the testing program Lestra complexity, the results were shown in Fig 6.3.

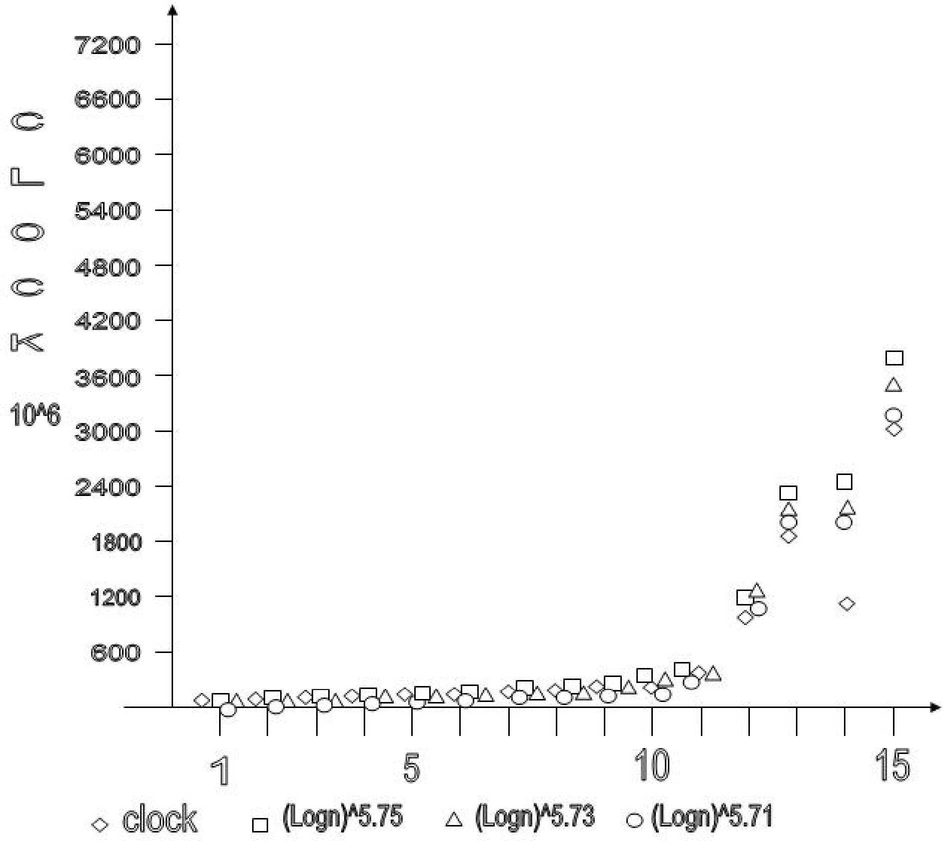

Figure 6.3: Figure illustrating the relationship between time taken in 10^6 clocks and the values of guessing complexity function $(\log n)^a$. The x-axis represents each n in the sequential order they appear in Appendix B. The actual numberical results can be found in Appendix B or in the file called "Lenstra_complexity_Fig 6.3.doc".

After careful observing the figure, we can conclude that for prime number less than 10 digits, the time complexity is $C(\log n)^{5.73}$. This is a big improvement on the previous time complexity function with an exponent of 6. For primes more than 10 digits which is 11 - 15 in x-axis, we can see that the distances between complexity $C(\log n)^a$ and actual time taken increases as n getting bigger. So there must be a better complexity time for lager primes.

Now we know T can be calculated by $C(\log n)^6$, if we knew how the value of C varies, we could actually predict the running time of the input number. From the results produced by Lenstra complexity testing program, the values of C are all positive number less than 1. On observation of Table 6.7, we can easily see that the value of $C_1 = \frac{\phi(r)}{r}$ seems to be a good approximation for C. A collection of primes which can be found in Appendix B was tested, the results were shown in Fig 6.4.

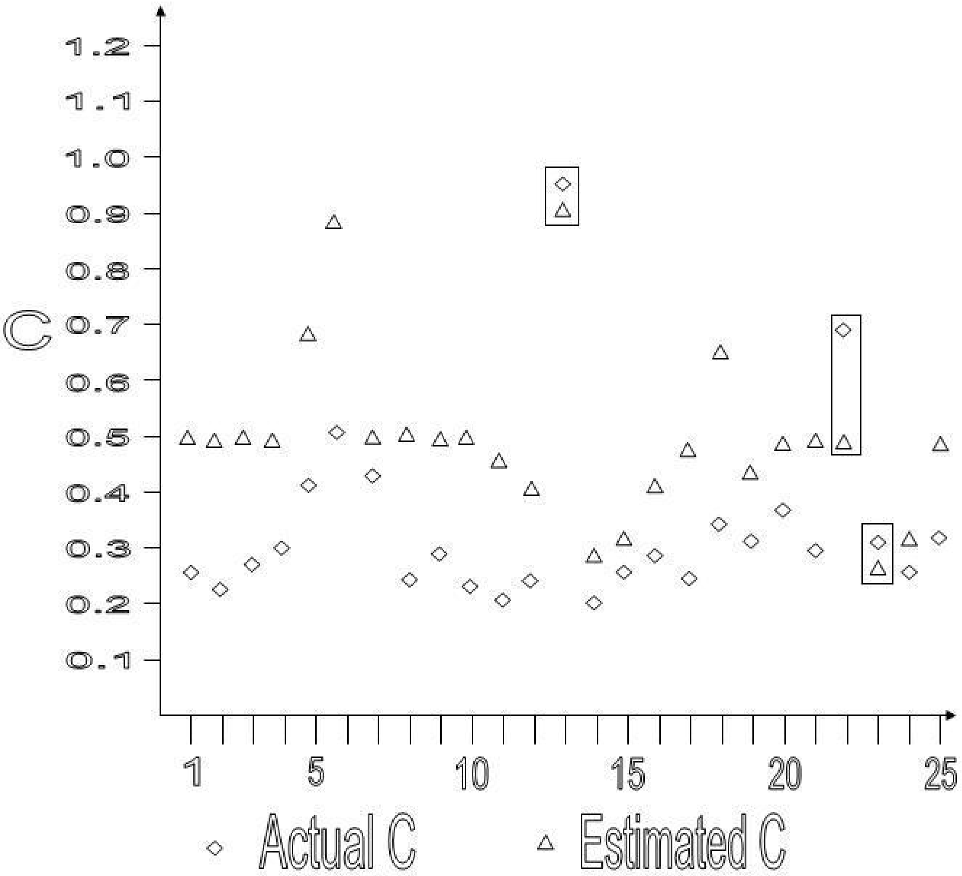

Figure 6.4: Figure illustrating the relationship between the actual value of C and first estimated value C_1.The x-axis represents each n in the sequential order they appear in Appendix B. The actual numberical results can be found in Appendix B or in the file called "Lenstra_complexity_Fig 6.4.doc".

Having observed the figure carefully, we can see our first estimation follows the shape of the actual C for most of the points. However there are three values which are underestimated. These are highlighted by rectangle in figure 6.4. Idealy an estimate should be an upper bound for C, so there must exist a better estimation in terms of more r or $\phi(r)$. By a series of experiments based on what we have got, the best estimation I found is $\frac{\phi(r)}{(\log\log(\phi(r)))r}$ denoted by C_2. The new comparison is shown in Fig 6.5.

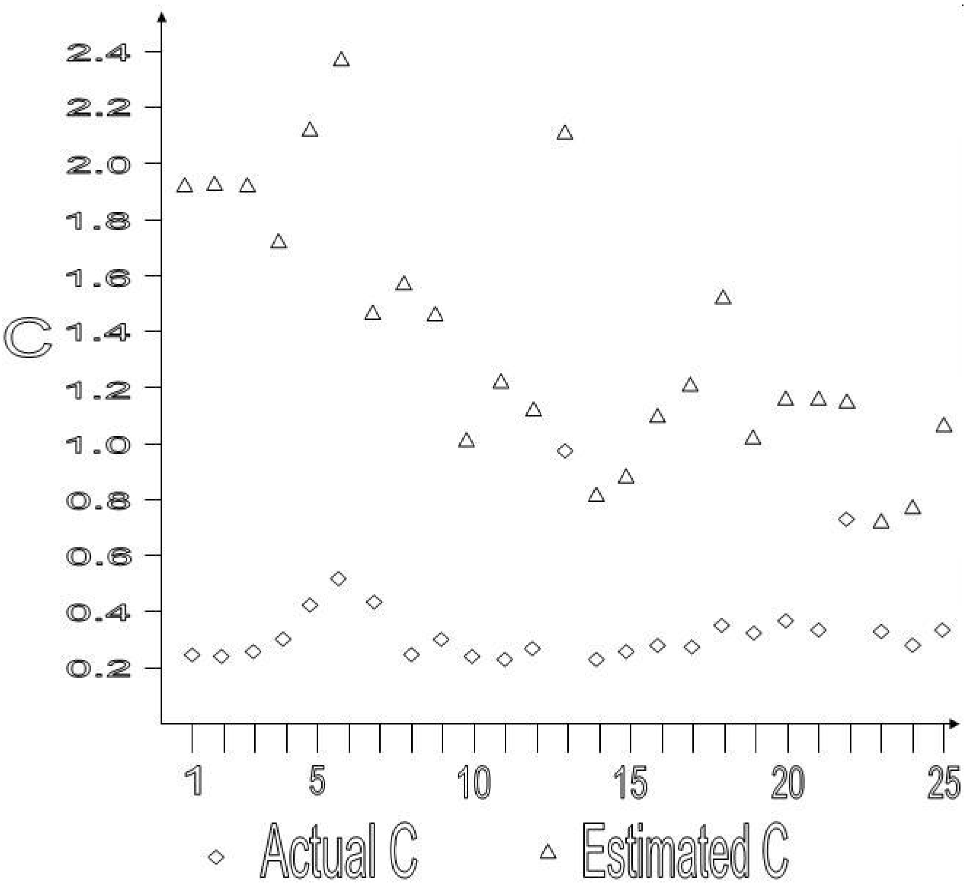

Figure 6.5: Figure illustrating the relationship between the actual value of C and first estimated value C_2.The x-axis represents each n in the sequential order they appear in Appendix B. The actual numberical results can be found in Appendix B.

Again the estimation C_2 follows the shape of the actual C, however there is quite a long distance between each pair of C_2 and C. Multiplying by a value of 0.64 should provide a good estimate C_3 for C in this range. Figure 6.6 shows the result of this modification.

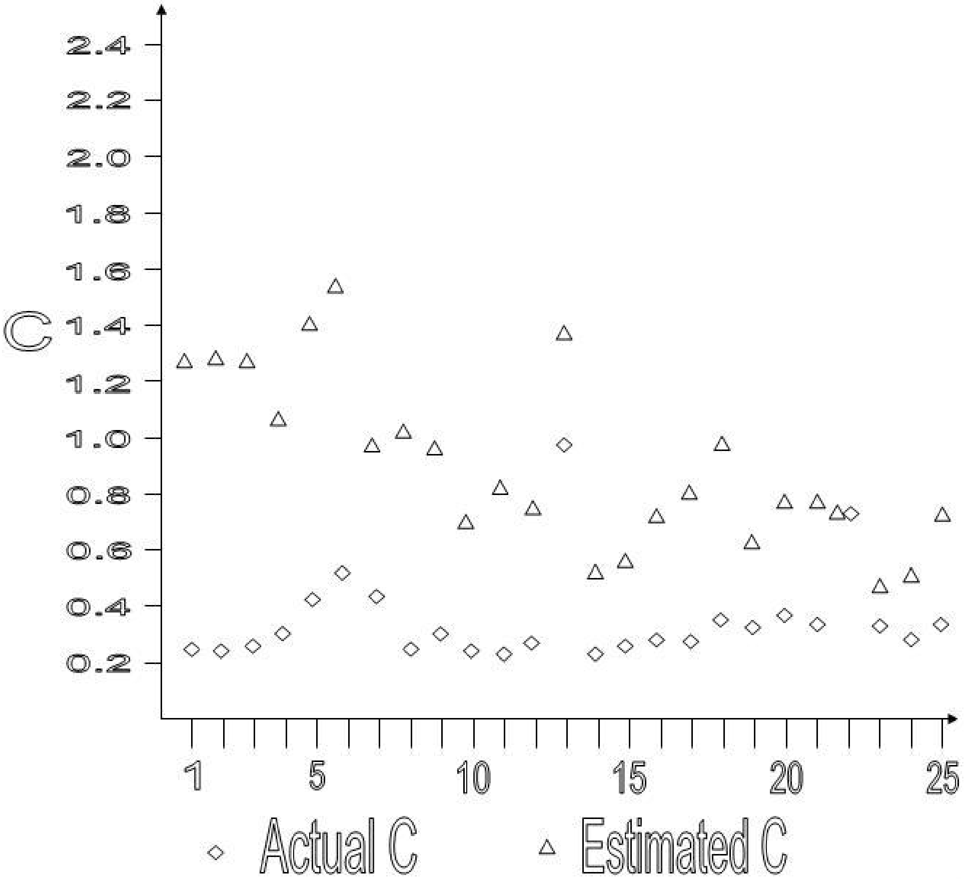

Figure 6.6: Figure illustrating the relationship between the actual value of C and first estimated value C_3.The x-axis represents each n in the sequential order they appear in Appendix B. The actual numberical results can be found in Appendix B.

Hence 0.65 * C_2 provides a better and more accurate estimation for the actual value of C for input numbers from 1 digits to 17 digits. however we cannot determine 0.65 is suitable for larger primes. So we conclude that the complexity of Lenstras modification to the AKS algorithm is

$$T = \frac{\phi(r)}{(\log\log(\phi(r)))r}(\log n)^6$$

6.5.1 Conjecture

In the analysis of Fig 6.3, a conjecture that the complexity time of Lenstra's modification improves as input number n increases was proposed. Now we also know that the complexity time is dependent on useful certificate r. So in this stage, we will investigate the conjecture that the complexity time of Lenstra's modification improves as r increases. We will proceed with the relationship between n and r.

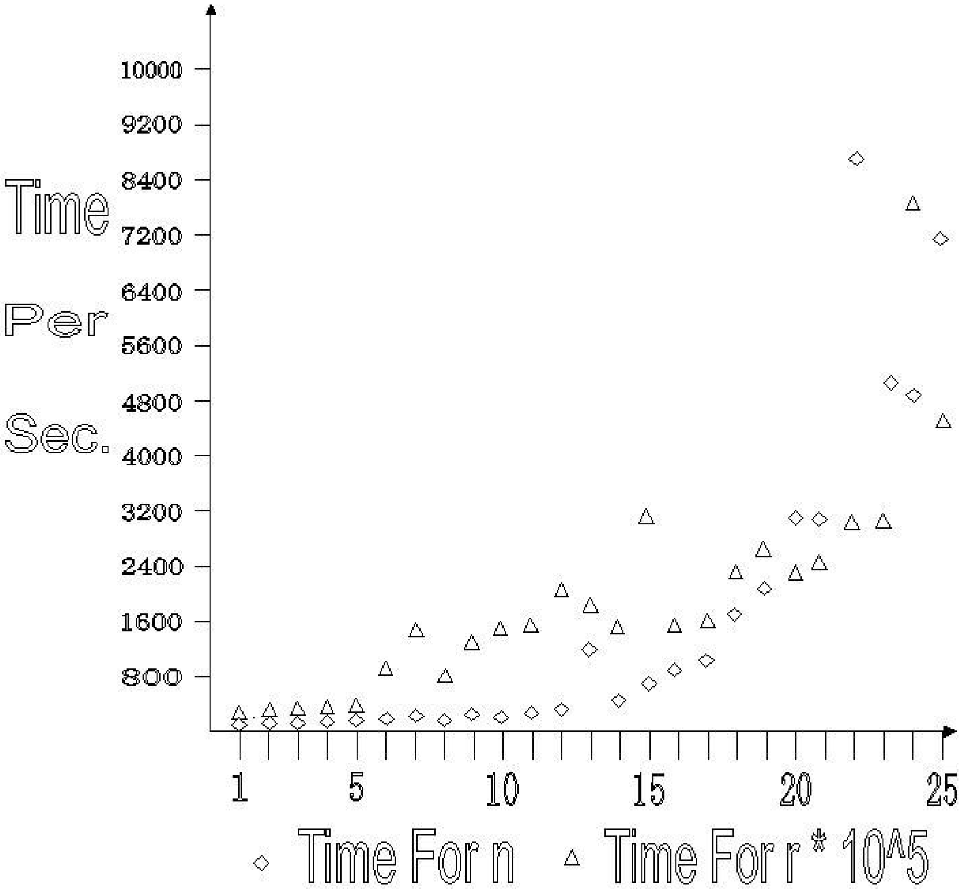

Figure 6.7: Figure illustrating the relationship between the time taken to determine the primality of n (in seconds) and the corresponding certificate of primality r (in milliseconds, note the value of r is multiplied by 10^5 to make the comparison clearer). The x-axis represents each n in the sequential order they appear in Appendix B. The actual numberical results can be found in Appendix B.

From the figure, we can see that generally two graphs follow the same trend. Therefore this result does support that the complexity time is closely related to r. If we want to determine whether this result is true for all the input numbers, we need to test more larger numbers. However the time taken for Lenstra's modification to determine the primality of a 40 digits or even 30 digits number is too long to wait. This problem can be overcomed by investigating the behaviour of certificate r for larger primes. Then we can use the result to predict the behaviour of time taken.

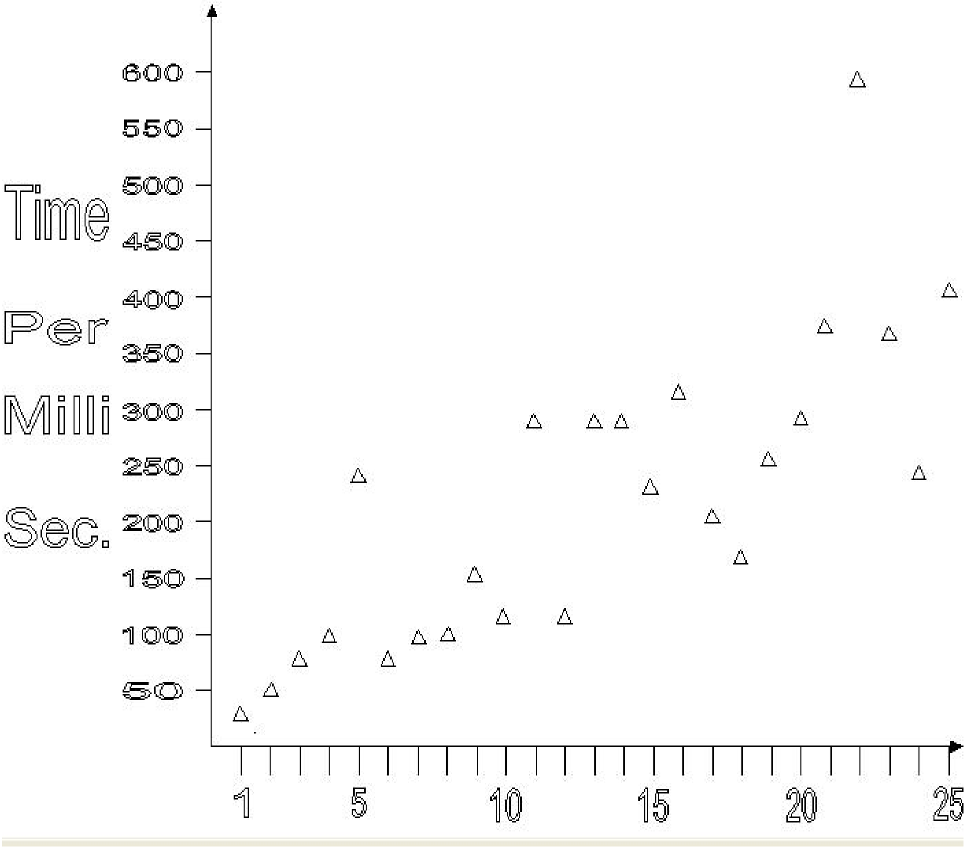

Figure 6.8: Figure illustrating the time taken to determine the certificate of primality r (in milliseconds). The x-axis represents each n in the sequential order they appear in Appendix B. The actual numberical results can be found in Appendix B.

The testing numbers are large primes from 18 digits to 42 digits. From the figure we can see that generally r stabilizes as n increases. In particular, the curve is very similar to a logarithm curve with the exception of three values. Therefore the same can be said regarding the time taken for all input number n. In conclusion the results we get indeed support the conjecture we made.

6.6 Summary

In this chapter, we have validated that our program indeed meet the initial requirements. Testing results have shown that individual component such as perfect power check, isprime and largest prime factor works correctly and efficiently. Testing results also support the Sophie Germain conjecture and the first useful prime r found when input number is 11701. The complexity time of our implementation of AKS algorithm is actually $O((\log n)^8 \log \log n)$.

The result produced by Lenstra's modification to the AKS algorithm has shown it performs much faster than AKS original algorithm. The value certificate r found is about 130 times smaller than those found in AKS. Testing results does support that $T \sim C(\log n)^6$ provides an good bound for complexity time of Lenstras modification. An argument which suggests the value of r is an important factor in determining the complexity of the algorithm. Further experiments and testing results have shown that a good approximation for C appears to be $\frac{\phi(r)}{(\log \log(\phi(r)))r}$. Moreover, we have shown the behaviour of complexity time does closely related to r. Hence as r does stabilize for larger primes up to 42 digits, it is possible to define the complexity time of Lenstra's modification to be $O(\log n)^{5.73}$.

Chapter 7

Conclusions

This project has produced both expected and surprising results based on the original objectives. Our program meets the initial requirements. Testing results suggests that the complexity time of AKS algorithm is $O((\log n)^8 \log\log n)$ instead of $O(\log n)^{12}$ based on the unproved Sophie Germain Conjecture. This is done by firstly analyzing the complexity time of finding an useful prime r and then estimate the complexity time of AKS algorithm based on that.

Testing results does support that $T \sim C(\log n)^6$ provides an good bound for complexity time of Lenstras modification. Further experiments and testing results have shown that a good approximation for C appears to be $\frac{\phi(r)}{(\log\log(\phi(r)))r}$. Finally we define the complexity time of Lenstra's modification to be $O(\log n)^{5.73}$.

Overall this project finishes smoothly, sticking to the project plan in most cases and although the initial implementation is a little bit behind the schedule (mainly due to the revision of the semester one exam) I made up the lost time and completed the project for the deadline with a number of days spare. This was due to the sensible project scheduling in which dissertation write-up was started earlier.

Addtionally, I also found literature survey is a very important part of the whole project. That is where you accumulate your fundamental knowledge of the project. It is improtant that you know what you are doing and how you are going to achieve it.

Moreover, although I was unfamiliar with C++ language before this project, it is quite easy to get on with, probably because it is similar to C and JAVA.

Possible future works:

- Prove Sophie Germain Conjecture.

- Is $O(\log n)^{5.73}$ indeed the complexity time of Lenstra's modification of AKS algorithm for all input n, or is there a better complexity time?
- Lenstra's modification is still slow for very large primes, is there any new way to replace the congruence check in the final for-loop to improve the efficiency.
- Add a user interface to the program.
- Using NTL with GMP. GMP is a library for long integer arithmetic. This will provide all of the performance benefits of GMP. Also there is build-in function for perfect power check. Using GMP will essentially improve the ifficiency of our implementation.

Bibliography

[1] D.R. Wilkins, 2005. Topics in Number Theory. [online] Available from: `http://www.maths.tcd.ie/~dwilkins/Courses/311/311NumTh.pdf` [Accessed Nov 2006]

[2] Honsberger, R. *More Mathematical Morsels.* Washington, DC: Math. Assoc. Amer., pp. 19-20, 1991.

[3] Burton, D. M. "The Theory of Congruences." Ch. 4 in *Elementary Number Theory, 4th ed.* Boston, MA: Allyn and Bacon, pp. 80-105, 1989.

[4] Donald Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition. *Addison-Wesley,* (1997). ISBN 0-201-89684-2.

[5] Nicolai Vorobjov. Algorithms. Series of Lecture Notes presented University of Bath, 2006.

[6] Anon, 1995. The Sieve of Eratosthenes. [online] Available from: `http://www.math.utah.edu/classes/216/assignment-07.html` [Accessed Nov 2006]

[7] O'Connor, J.J., and Robertson, E.F., Prime numbers. [online] Available from: `http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Prime_numbers.html` [Accessed Nov 2006]

[8] Anon, 2004. The Biggest Number. [online] Available from: `http://www.7880.com/Info/Article-3d278480.html` [Accessed Nov 2006]

[9] Pratt, V.R., "Every prime has a succinct certificate", *SIAM.J Computing 4,* (1975), 4:214-220.

[10] Miller, G.L., Riemann's hypothesis and tests for primality, *J. Computing. Sys. Sci.,* 13:300–317, 1976.

[11] Solovay, R., Strassen, V., "A Fast Monte-Carlo Test for Primality", *SIAM J. Computing* 6, 84 (1977).

[12] Rabin, M.O., Probabilistic algorithm for testing primality, *J. Number Theory,* (1980), 12:128-138.

[13] Adleman, L.M., Pomerance, C., Rumely, R.S., On distinguishing prime numbers from composite numbers, *Ann. Math.,* 117:173-206, (1983). MR0683806 (84e:10008)

[14] Goldwasser, S., Killian, J., symposium on theory of Almost all primes can be quickly ACM computing, Associate for computing Machinery[C], *New York : ACM Press,* (1986). pp.316-329

[15] Atkin, A.O.L, Morain, F., "Elliptic curves and primality proving", *Math. Computation, vol.61,* pp.29-68, 1993.

[16] W. Diffie and M.E. Hellman., New directions in cryptography, *IEEE Transactions on Information Theory 22* (1976), 644-654.

[17] Rivest, R. L., Shamir, A., Adleman, L. A.: A method for obtaining digital signatures and public-key cryptosystems; *Communications of the ACM, Vol.21, Nr.2,* (1978), S.120-126.

[18] M. Agrawal, N. Kayal and N. Saxena, 'PRIMES is in P', Preprint , 2002, 1-9.[online] Available from `http://www.cse.iitk.ac.in/news/primality.pdf` [Accessed November 2006]

[19] Chris Caldwell. Finding primes proving primality. [online] Available from: `http://primes.utm.edu/prove/prove4_3.html` [Accessed Nov 2006]

[20] Lenstra H. W. Jr. "Primality Testing with Cyclotomic Rings." Preprint. 14 Aug 2002.

[21] D Bernstein. Proving primality after Agrawal-Kayal-Saxena, 2002 [online] Available from `http://cr.yp.to/papers/aks.pdf` [Accessed December 2006]

[22] J Voloch. Improvements to aks. [online] Available from: `http://www.ma.utexas.edu/users/voloch/Preprints/aks.pdf` [Accessed November 2006]

[23] Q. Cheng, "Primality proving via one round of ECPP and one iteration in AKS," Crypto 2003, Santa Barbara, (2003) [online] Available from `http://www.cs.ou.edu/%7Eqcheng/paper/aksimp.pdf` [Accessed December 2006]

[24] Pedro Berrizbeitia. Sharpening "Primes is in P" for a large family of numbers. *Journal: Math. Comp. 74* (2005), 2043-2059

[25] D Bernstein. Proving Primality in Essentially Quartic Random Time, 2003 [online] Available from `http://cr.yp.to/primetests/quartic-20060914-ams.pdf` [Accessed December 2006]

[26] Manindra Agrawal, Neeraj Kayal and Nitin Saxena. PRIMES is in P. *Ann. of Math. 160, no. 2* (2004), 781-793

[27] VictorShoup. Victor shoup's homepage-ntl. [online] Available from `http://swww.houp.net/ntl` [Accessed Jan 2007]

[28] P. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor, *Advances in Cryptology: CRYPTO'86 (A. M. Odlyzko ed.)*, LCNS 263, Springer-Verlag, pp. 311-323, 1987

[29] R. Crandall and J. Papadopoulos. On the implementation of aks-class primality tests. March 2003 [online] Available from: `http://images.apple.com/acg/pdf/aks3.pdf`[Accessed Feb 2007]

Appendix A

User Documentation

A.1 NTL instruction

TNL is a number theory library used through out the the program. Before compling the program, you must first download, install and configure NTL.

A.1.1 Obtaining and Installing NTL for Windows and other Platforms

Full instruction can be found in `http://shoup.net/ntl/doc/tour-win.html`

A.1.2 Obtaining and Installing NTL for UNIX

Full instruction can be found in `http://shoup.net/ntl/doc/tour-unix.html`

A.1.3 Installing on Dev C++

1: Download the Dev C++ 4.9.9.2 on the website.
2: Create a new project called libntl (choose static library). And save in a specific directory.
3: click Tools → Compiler Options → select Directories → select C++ Includes → browse your downloaded `WinNTL-5_4\include` → click add → click ok.
4: compile → copy the libntl.a file to the `Dev-Cpp\lib` folder
5: click Tools → Compiler Options → tick "Add these commands to the linker command line" → type in "-lntl".

A.2 Files Explanation

There are three directories in the root directory of the cd-rom. Directory ”Code” contains final code of two algorithms and testing program. Directory ”Testing Results” contains some testing outputs, full testing results can be found in Appendix B. Directory ”Directly Open In MS VC++ 6.0” contain final code and other files generated by Microsoft Visual C++, these projects can be directly open once you put these directories into your own computer (note cannot directly open in cd-rom). In the directory there is a file called ”WINMM.LIB” which is a multimedia library for measuring time in milliseconds. Some of C++ compilers have already include this resource file, if not, include this file exactly the same way as ”NTL.lib”.

Appendix B

Prime Lists and Results Output

B.1 Prime Lists

B.1.1 A List of prime for Fig 6.1, 6.2 and 6.3.

1. 10133
2. 32561
3. 160583
4. 268069
5. 2969023
6. 7741009
7. 12434839
8. 86457079
9. 100982933
10. 879673117
11. 2492813497
12. 435465768733
13. 7000000000009
14. 10000000000037
15. 100000000000031

B.1.2 A List of prime for Fig 6.4, 6.5, 6.6, 6.7

1. 88903
2. 99397
3. 107339
4. 777619

5. 1000099
6. 4740623
7. 26933611
8. 10012333
9. 100041703
10. 103736293
11. 1001772091
12. 2958347047
13. 10015571677
14. 44426255143
15. 100020817331
16. 333267326767
17. 1000528294943
18. 3222583708567
19. 10083087720779
20. 35466059872651
21. 112272535095293
22. 281702565146179
23. 1003026954441971
24. 4467165232203397
25. 10022390619214807

B.1.3 A List of prime for Fig 6.8

1. 100055128505716009
2. 1083717775299973771
3. 29546363270378697007
4. 326070784035774767971
5. 4973004941902396102727
6. 51005856776120585677103
7. 614783152143098270145397
8. 7666569009190249923345281
9. 86084043198752959566539209
10. 912613825844053990694091143
11. 1000474617637553175973957663
12. 20476096752860587951845236929
13. 387121083116233373653498534849
14. 4313339400115792413779939218099
15. 53437079999999999999999994656293
16. 617594269939999999993824057300601
17. 7433112517127371273712736384060023
18. 81156673714614395518740388458373409

19. 974002888812604462999840811965244329
20. 1040924570757777777777777673685320703
21. 27491090500860000000000002749109050087
22. 357035643600060000000000035703564360007
23. 4160277543663151111111110695083356744797
24. 55132856630488887841876472900339410613059
25. 689960931088884849033689023336009222695077

B.2 Results Output

B.2.1 Output results for Table 6.7

n=7 r=2 Euler(2) =1 n is prime Time Taken=0

n=17 r=4 Euler(4) =2 n is prime Time Taken=0

n=125 125 is a perfect power, hence is not prime. Time Taken=1 milliseconds

n=149 r=30 Euler(30) =8 n is prime Time Taken=0

n=2547 2547 is composite. 3 is a divisor. Time Taken=0 milliseconds

n=3137 r=64 Euler(64) =32 n is prime Time Taken=1

n=10133 r=88 Euler(88) =40 n is prime Time Taken=2

n=32561 r=112 Euler(112) =48 n is prime Time Taken=3

n=160583 r=128 Euler(128) =64 n is prime Time Taken=9

n=268069 r=154 Euler(154) =60 n is prime Time Taken=14

n=1771561 1771561 is a perfect power, hence is not prime. Time Taken=0 milliseconds

n=2969023 r=208 Euler(208) =96 n is prime Time Taken=40

n=7741009 r=238 Euler(238) =96 n is prime Time Taken=33

n=9786539 9786539 is composite. 7 is a divisor

n=12434839 r=256 Euler(256) =128 n is prime Time Taken=72

n=86457079 r=342 Euler(342) =108 n is prime Time Taken=97

n=100982933 r=336 Euler(336) =96 n is prime Time Taken=61

n=879673117 r=412 Euler(412) =204 n is prime Time Taken=176

n=2492813497 r=456 Euler(456) =144 n is prime Time Taken=314

n=90552556447 r=640 Euler(640) =256 the a which fails is 1. n is not prime. Time Taken=2988 milliseconds

n=435465768733 r=700 Euler(700) =240 n is prime Time Taken=725

n=7000000000009 r=856 Euler(856) =424 n is prime Time Taken=1895

n=10000000000037 r=876 Euler(876) =288 n is prime Time Taken=1116

n=100000000000031 r=1024 Euler(1024) =512 n is prime Time Taken=3004

n=984060615693223 r=1176 Euler(1176) =336 the a which fails is 1. n is not prime. Time Taken=10525 milliseconds

B.2.2 Output results for actual time taken clocks per second for Fig 6.1, 6.2 and 6.3.

```
1. 10133 T: 2000000
2. 32561 T: 3000000
3. 160583 T: 9000000
4. 268069 T: 14000000
5. 2969023 T: 40000000
6. 7741009 T: 33000000
7. 12434839 T: 72000000
8. 86457079 T: 97000000
9. 100982933 T: 61000000
10. 879673117 T: 176000000
11. 2492813497 T: 314000000
12. 435465768733 T: 725000000
13. 7000000000009 T: 1895000000
14. 10000000000037 T: 1116000000
15. 100000000000031 T: 3004000000
```

B.2.3 Output results for Fig. 6.1

$D = (\log n)^6$

```
C =0.3602411828 D =5551836.09
C =0.264339716 D =11349032.39
C =0.3365321355 D =26743359.85
C =0.4072201489 D =34379438.34
C =0.4048000451 D =98814218.24
C =0.2297798994 D =143615695.2
C =0.4201543753 D =171365584.3
C =0.288773652 D =335903221.6
C =0.1726097242 D =353398398
```

```
C =0.2557919932 D =688059066.1
C =0.3394211837 D =925104310.2
C =0.2170265793 D =3340604650
C =0.3139375815 D =6036231760
C =0.1720531843 D =6486366436
C =0.2968864306 D =10118347250
```

B.2.4 Output results for Fig. 6.2

$D = (\log n)^{5.9}$

```
C =0.466660065 D =4285774.914
C =0.3465333734 D =8657174.835
C =0.4475210563 D =20110785.57
C =0.5437937829 D =25745053.44
C =0.5501581398 D =72706367.7
C =0.3142428464 D =105014323.7
C =0.5762898234 D =124937135.9
C =0.4005540569 D =242164567.6
C =0.2396273223 D =254561956.5
C =0.3590713117 D =490153332.4
C =0.4788234984 D =655773998.2
C =0.3127831459 D =2317899828
C =0.4569367341 D =4147182440
C =0.2507240953 D =4451107895
C =0.4358551427 D =6892198131
```

$D = (\log n)^{5.8}$

```
C =0.6045161593 D =3308430.998
C =0.454284285 D =6603794.361
C =0.5951143286 D =15123144.52
C =0.7261715292 D =19279191.54
C =0.7477123148 D =53496510.9
C =0.4297528496 D =76788321.55
C =0.7904474643 D =91087647.51
C =0.5556031562 D =174585041.3
C =0.3326652301 D =183367525.3
C =0.5040509878 D =349171024.9
C =0.6754791795 D =464855186.6
```

```
C =0.4507894685 D =1608289569
C =0.6650722669 D =2849314419
C =0.3653670939 D =3054462262
C =0.6398733181 D =4694679267
```

$D = (\log n)^{5.7}$

```
C =0.7830963355 D =2553964.192
C =0.5955392105 D =5037451.686
C =0.7913841352 D =11372479.68
C =0.969715187 D =14437228.77
C =1.01620546 D =39362118.75
C =0.5877222467 D =56148972.04
C =1.084189185 D =66409074.18
C =0.7706696809 D =125864559.6
C =0.4618261149 D =132084345.2
C =0.7075680793 D =248739315.9
C =0.9529025275 D =329519537.3
C =0.6496870039 D =1115921968
C =0.9680139223 D =1957616473
C =0.532430332 D =2096048878
C =0.9393897721 D =3197820638
```

B.2.5 Output results for Fig. 6.3

$D = (\log n)^{5.75}$

```
C =0.688036619 D =2906822.028
C =0.5201385435 D =5767694.083
C =0.686268197 D =13114406.35
C =0.8391540742 D =16683467.83
C =0.8716819013 D =45888299.32
C =0.502568712 D =65662663.06
C =0.9257400239 D =77775615.33
C =0.6543596161 D =148236531.7
C =0.3919610832 D =155627695.2
C =0.5972021344 D =294707587
C =0.8022878644 D =391380717.5
C =0.5411765508 D =1339673714
C =0.8023709951 D =2361750377
```

```
C =0.4410584123 D =2530277099
C =0.7753002324 D =3874628014
```

$D = (\log n)^{5.73}$

```
C =0.7245911219 D =2760177.346
C =0.5490799263 D =5463685.442
C =0.7265261656 D =12387716.27
C =0.8891247578 D =15745821.81
C =0.9268456734 D =43157130.84
C =0.5350398388 D =61677650.16
C =0.98613291 D =73012470.5
C =0.6986138686 D =138846370.5
C =0.4185401874 D =145744666.4
C =0.6391166086 D =275380106.9
C =0.859443884 D =365352533
C =0.5822172304 D =1245239684
C =0.8649236532 D =2190944823
C =0.475557222 D =2346720749
C =0.8371827659 D =3588224845
```

$D = (\log n)^{5.71}$

```
C =0.7630877187 D =2620930.662
C =0.5796316562 D =5175700.754
C =0.7691457532 D =11701293.24
C =0.9420711395 D =14860873.47
C =0.9855004459 D =40588515.38
C =0.5696089357 D =57934484.4
C =1.050465672 D =68541030.83
C =0.7458610302 D =130051036.4
C =0.4469216358 D =136489252.5
C =0.6839728391 D =257320159.4
C =0.9206717719 D =341055313.7
C =0.626370272 D =1157462339
C =0.9323529022 D =2032492199
C =0.5127544676 D =2176480305
C =0.904004609 D =3322991907
```

B.2.6 Output results for Fig. 6.4

Recall: Actual C is calculated by $\frac{(Timetaken*10^6)}{(\log n)^6}$. Estimated C_1 is calculated by $\frac{Euler(r)}{r}$.

n = 88903 r = 128 Euler(128) = 64 n is prime Time Taken = 5 C_1 =0.5 C = 0.253

n=99397 r=122 Euler(122) = 60 n is prime Time Taken = 5 C_1 =0.49 C = 0.24

n=107339 r=128 Euler(128) = 64 n is prime Time Taken = 6 C_1 =0.5 C = 0.28

n=777619 r=172 Euler(172) = 84 n is prime Time Taken = 17 C_1 =0.49 C = 0.3

n=1000099 r=175 Euler(175) = 120 n is prime Time Taken = 26 C_1 =0.69 C = 0.41

n=4740623 r=247 Euler(247) = 216 n is prime Time Taken = 61 C_1 =0.87 C = 0.51

n=26933611 r=316 Euler(316) = 156 n is prime Time Taken = 96 C_1 =0.49 C = 0.42

n=10012333 r=256 Euler(256) = 128 n is prime Time Taken = 37 C_1 =0.5 C = 0.23

n=100041703 r=334 Euler(334) = 166 n is prime Time Taken = 103 C_1 =0.5 C = 0.29

n=103736293 r=348 Euler(348) = 112 n is prime Time Taken = 78 C_1 =0.32 C = 0.22

n=1001772091 r=418 Euler(418) = 180 n is prime Time Taken = 147 C_1 =0.43 C = 0.21

n=2958347047 r=476 Euler(476) = 192 n is prime Time Taken = 245 C_1 =0.4 C = 0.25

n=10015571677 r=551 Euler(551) = 504 n is prime Time Taken = 1288 C_1 =0.91 C = 0.96

n=44426255143 r=588 Euler(588) = 168 n is prime Time Taken = 400 C_1 =0.29 C = 0.2

n=100020817331 r=666 Euler(666) = 216 n is prime Time Taken = 623 C_1 =0.32 C = 0.26

n=333267326767 r=686 Euler(686) = 294 n is prime Time Taken = 892 C_1 =0.43 C = 0.28

n=1000528294943 r=736 Euler(736) = 352 n is prime Time Taken = 1026 C_1 =0.48 C = 0.26

n=3222583708567 r=807 Euler(807) = 536 n is prime Time Taken = 1866 C_1 =0.66 C = 0.36

n=10083087720779 r=868 Euler(868) = 360 n is prime Time Taken = 2076 C_1 =0.41 C = 0.32

n=35466059872651 r=974 Euler(974) = 486 n is prime Time Taken = 3211 C_1 =0.5 C = 0.39

n=112272535095293 r=1024 Euler(1024) = 512 n is prime Time Taken = 3200 C_1 =0.5 C = 0.31

n=281702565146179 r=1112 Euler(1112) = 552 n is prime Time Taken = 8929 C_1 =0.5 C

= 0.73

n=1003026954441971 r=1176 Euler(1176) = 336 n is prime Time Taken = 5042 C_1 =0.29 C = 0.33

n=4467165232203397 r=1356 Euler(1356) = 448 n is prime Time Taken = 4902 C_1 =0.33 C = 0.25

n=10022390619214807 r=1328 Euler(1328) = 656 n is prime Time Taken = 7209 C_1 =0.49 C = 0.32

B.2.7 Output results for Fig. 6.5

n = 88903 C_2 = 1.94 C = 0.25

n = 99397 C_2 = 1.96 C = 0.24

n = 107339 C_2 = 1.94 C = 0.28

n = 777619 C_2 = 1.72 C = 0.3

n = 1000099 C_2 = 2.15 C = 0.41

n = 4740623 C_2 = 2.37 C = 0.51

n = 26933611 C_2 = 1.45 C = 0.42

n = 10012333 C_2 = 1.55 C = 0.23

n = 100041703 C_2 = 1.43 C = 0.29

n = 103736293 C_2 = 1.03 C = 0.22

n = 1001772091 C_2 = 1.22 C = 0.21

n = 2958347047 C_2 = 1.12 C = 0.25

n = 10015571677 C_2 = 2.11 C = 0.96

n = 44426255143 C_2 = 0.82 C = 0.2

n = 100020817331 C_2 = 0.88 C = 0.26

n = 333267326767 C_2 = 1.09 C = 0.28

n = 1000528294943 C_2 = 1.18 C = 0.26

n = 3222583708567 C_2 = 1.52 C = 0.36

n = 10083087720779 C_2 = 1.02 C = 0.32

n = 35466059872651 C_2 = 1.16 C = 0.39

n = 112272535095293 C_2 = 1.16 C = 0.31

n = 281702565146179 C_2 = 1.13 C = 0.73

$n = 1003026954441971$ $C_2 = 0.71$ $C = 0.33$

$n = 4467165232203397$ $C_2 = 0.78$ $C = 0.25$

$n = 10022390619214807$ $C_2 = 1.1$ $C = 0.32$

B.2.8 Output results for Fig. 6.6

$n = 88903$ $C_3 = 1.26$ $C = 0.25$

$n = 99397$ $C_3 = 1.27$ $C = 0.24$

$n = 107339$ $C_3 = 1.26$ $C = 0.28$

$n = 777619$ $C_3 = 1.12$ $C = 0.3$

$n = 1000099$ $C_3 = 1.4$ $C = 0.41$

$n = 4740623$ $C_3 = 1.54$ $C = 0.51$

$n = 26933611$ $C_3 = 0.94$ $C = 0.42$

$n = 10012333$ $C_3 = 1.00$ $C = 0.23$

$n = 100041703$ $C_3 = 0.93$ $C = 0.29$

$n = 103736293$ $C_3 = 0.67$ $C = 0.22$

$n = 1001772091$ $C_3 = 0.79$ $C = 0.21$

$n = 2958347047$ $C_3 = 0.73$ $C = 0.25$

$n = 10015571677$ $C_3 = 1.37$ $C = 0.96$

$n = 44426255143$ $C_3 = 0.53$ $C = 0.2$

$n = 100020817331$ $C_3 = 0.57$ $C = 0.26$

$n = 333267326767$ $C_3 = 0.71$ $C = 0.28$

$n = 1000528294943$ $C_3 = 0.77$ $C = 0.26$

$n = 3222583708567$ $C_3 = 0.99$ $C = 0.36$

$n = 10083087720779$ $C_3 = 0.66$ $C = 0.32$

$n = 35466059872651$ $C_3 = 0.75$ $C = 0.39$

$n = 112272535095293$ $C_3 = 0.75$ $C = 0.31$

$n = 281702565146179$ $C_3 = 0.73$ $C = 0.73$

$n = 1003026954441971$ $C_3 = 0.46$ $C = 0.33$

$n = 4467165232203397$ $C_3 = 0.51$ $C = 0.25$

$n = 10022390619214807$ $C_3 = 0.72$ $C = 0.32$

B.2.9 Output results for Fig. 6.7

$n = 88903$ Time Taken for $n = 5$ seconds Time Taken for $r = 3$ milliseconds

$n = 99397$ Time Taken for $n = 5$ seconds Time Taken for $r = 4$ milliseconds

$n = 107339$ Time Taken for $n = 6$ seconds Time Taken for $r = 4$ milliseconds

$n = 777619$ Time Taken for $n = 17$ seconds Time Taken for $r = 5$ milliseconds

$n = 1000099$ Time Taken for $n = 26$ seconds Time Taken for $r = 5$ milliseconds

$n = 4740623$ Time Taken for $n = 61$ seconds Time Taken for $r = 10$ milliseconds

$n = 26933611$ Time Taken for $n = 96$ seconds Time Taken for $r = 14$ milliseconds

$n = 10012333$ Time Taken for $n = 37$ seconds Time Taken for $r = 8$ milliseconds

$n = 100041703$ Time Taken for $n = 103$ seconds Time Taken for $r = 11$ milliseconds

$n = 103736293$ Time Taken for $n = 78$ seconds Time Taken for $r = 14$ milliseconds

$n = 1001772091$ Time Taken for $n = 147$ seconds Time Taken for $r = 15$ milliseconds

$n = 2958347047$ Time Taken for $n = 245$ seconds Time Taken for $r = 20$ milliseconds

$n = 10015571677$ Time Taken for $n = 1288$ seconds Time Taken for $r = 17$ milliseconds

$n = 44426255143$ Time Taken for $n = 400$ seconds Time Taken for $r = 15$ milliseconds

$n = 100020817331$ Time Taken for $n = 623$ seconds Time Taken for $r = 30$ milliseconds

$n = 333267326767$ Time Taken for $n = 892$ seconds Time Taken for $r = 16$ milliseconds

$n = 1000528294943$ Time Taken for $n = 1026$ seconds Time Taken for $r = 17$ milliseconds

$n = 3222583708567$ Time Taken for $n = 1866$ seconds Time Taken for $r = 24$ milliseconds

$n = 10083087720779$ Time Taken for $n = 2076$ seconds Time Taken for $r = 28$ milliseconds

$n = 35466059872651$ Time Taken for $n = 3211$ seconds Time Taken for $r = 23$ milliseconds

$n = 112272535095293$ Time Taken for $n = 3200$ seconds Time Taken for $r = 24$ milliseconds

$n = 281702565146179$ Time Taken for $n = 8929$ seconds Time Taken for $r = 31$ milliseconds

$n = 1003026954441971$ Time Taken for $n = 5042$ seconds Time Taken for $r = 32$ milliseconds

$n = 4467165232203397$ Time Taken for $n = 4902$ seconds Time Taken for $r = 78$ milliseconds

$n = 10022390619214807$ Time Taken for $n = 7209$seconds Time Taken for $r = 45$ milliseconds

B.2.10 Output results for Fig. 6.8

$n =$ 100055128505716009 Time Taken for r =33 milliseconds

$n =$ 1083717775299973771 Time Taken for r =50 milliseconds

$n =$ 29546363270378697007 Time Taken for r =75 milliseconds

$n =$ 326070784035774767971 Time Taken for r =100 milliseconds

$n =$ 4973004941902396102727 Time Taken for r =235 milliseconds

$n =$ 51005856776120585677103 Time Taken for r = 75 milliseconds

$n =$ 614783152143098270145397 Time Taken for r = 100 milliseconds

$n =$ 7666569009190249923345281 Time Taken for r = 101 milliseconds

$n =$ 86084043198752959566539209 Time Taken for r = 155 milliseconds

$n =$ 912613825844053990694091143 Time Taken for r = 115 milliseconds

$n =$ 1000474617637553175973957663 Time Taken for r = 280 milliseconds

$n =$ 20476096752860587951845236929 Time Taken for r = 115 milliseconds

$n =$ 387121083116233373653498534849 Time Taken for r = 280 milliseconds

$n =$ 4313339400115792413779939218099 Time Taken for r = 280 milliseconds

$n =$ 53437079999999999999999994656293 Time Taken for r = 225 milliseconds

$n =$ 617594269939999999993824057300601 Time Taken for r = 320 milliseconds

$n =$ 7433112517127371273712736384060023 Time Taken for r = 205 milliseconds

$n =$ 81156673714614395518740388458373409 Time Taken for r = 170 milliseconds

$n =$ 974002888812604462999840811965244329 Time Taken for r = 260 milliseconds

$n =$ 1040924570757777777777777673685320703 Time Taken for r = 280 milliseconds

$n =$ 27491090500860000000000002749109050087 Time Taken for r = 380 milliseconds

$n =$ 357035643600060000000000035703564360007 Time Taken for r = 600 milliseconds

$n =$ 4160277543663151111111110695083356744797 Time Taken for r = 360 milliseconds

$n =$ 55132856630488887841876472900339410613059 Time Taken for r = 260 milliseconds

$n =$ 689960931088884849033689023336009222695077 Time Taken for r = 410 milliseconds

Appendix C

Code

The following pages gives copies of the final code of AKS algorithm and Lenstra's modification. A full copy with testing program can also be found on the accompanying cd-rom. We begin with the main C++ code for AKS algorithm with the necessary header files in the subsequent subsections. The code for Lenstra modification are in the identical structure.

C.1 Program Listing

C.1.1 AKS Original Algorithm

AKS.cpp
Main source of the AKS original algorithm.

PerfectPower.h
This head contains the perfect power check function for AKS.cpp

Isprime.h
This head contains the isprime check function for AKS.cpp

Largestprime.h
This head contains the largest prime factor function for AKS.cpp

Congruence.h
This head contains the congruence check function in the final for-loop for AKS.cpp

C.1.2 Lenstra's Modification of the AKS Algorithm

AKS_Lenstra.cpp
Main source of the Lenstra's modification of AKS algorithm

PerfectPower.h

This head contains the perfect power check function for AKS_Lenstra.cpp

Euler.h
This head contains the Euler-Phi function for AKS_Lenstra.cpp

Congruence.h
This head contains the congruence check function in the final for-loop for AKS_Lenstra.cpp

C.1.3 Testing Program

Component_test.cpp
Testing the correctness of perfect power check, isprime and largest prime factor functions. Necessary head files: PerfectPower.h, Isprime.h and Largestprime.h.

Useful r test.cpp
Testing the vaule of useful prime r for AKS original algorithm. Necessary head files: PerfectPower.h, Isprime.h and Largestprime.h.

useful r test_Lenstra.cpp
Testing the vaule of useful prime r for Lenstra's modification of AKS algorithm. Necessary head files: Euler.h.

Lenstra complexity.cpp
Estimating the complexity time of Lenstra's modification of AKS algorithm.

C.2 The AKS Original Algorithm

C.2.1 AKS.cpp

```
#include "math.h"
#include "fstream.h"
#include "iostream.h"
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>
#include <time.h>
#include <NTL/ZZ.h> // NTL Libraries
#include <NTL/RR.h>
#include <NTL/ZZ_p.h>
#include <NTL/ZZ_pX.h>
#include <NTL/ZZX.h>
#include <NTL/vec_ZZ.h>
#include "PerfectPower.h" //Each Indepedent Test
#include "Isprime.h"
#include "largestprime.h"
#include "Congruence.h"
int main(int argc, char argv[]){

start:
ZZ n;
n = 0;

cout << "Enter␣a␣positive␣integer␣number␣n␣you␣want␣to␣
    be␣tested\n"; cin >> n;
  if(n<1){
     cout << "Integer␣n␣needs␣to␣be␣positive\n\n";
     goto start;
  }
  else if(n==1){
     cout << "1␣is␣neither␣prime␣or␣composite.\n\n";
     goto start;
  }
  else if(n==2){
     cout << "2␣is␣prime.\n\n";
     goto start;
  }
  else if(n==3){
     cout << "3␣is␣prime.\n\n";
     goto start;
 }
ofstream my_file("AKS_original.doc", ios::app); //
    output result into file
 cout << "n=" << n << "\n";
 my_file << "n=" << n << "\n";

 // start timing
 DWORD start, finish, duration;
 start = timeGetTime ();
 cout << "CPU␣clocks␣per␣second␣=␣" << CLOCKS_PER_SEC *
     1000 << "\n";

int PP = perfectpower(n);
       // Check if n is a perfect power
       // Returns 1 if n is a perfect power, 0 otherwise
 if(PP==1){
  my_file << n << "␣is␣a␣perfect␣power,␣hence␣is␣not␣
      prime\n\n";

  finish = timeGetTime();
  duration = finish - start;
  cout << "Time␣Taken=" << duration << "␣
      milliseconds\n\n";
  my_file << "Time␣Taken=" << duration << "␣
      milliseconds\n\n";
  my_file.close();


  int choice;
  //Give user opportunity to continue or exit program
  cout << "Press␣'1'␣to␣test␣a␣new␣number,␣'0'␣to␣exit␣
      the␣program\n"; cin >> choice;

  if(choice == 1){
    goto start;
  }
  else if(choice == 0){
    return(0);//Exit program
  }

 }
 else{
 //continue
 }
```

```
// Find a suitable r
ZZ r = to_ZZ(2);
 while(r<n){//line 3 of Fig 2.1
    ZZ r1 = GCD(r,n);
    if(r1 != 1){ // line 4 of Fig 2.1
      cout << "n␣has␣factors␣other␣than␣n␣and␣1,␣hence␣
          is␣composite\n\n";
      my_file << n << "␣is␣composite.\n";
      my_file << r1 << "␣is␣a␣divisor\n\n";
          cout << r1 << "␣is␣a␣divisor\n\n";

          finish = timeGetTime();
      duration = finish - start;
      cout << "Time␣Taken=" << duration << "␣
          milliseconds\n\n";
      my_file << "Time␣Taken=" << duration << "␣
          milliseconds\n\n";
      my_file.close();

          int choice;
      //Give user opportunity to continue or exit program
      cout << "Press␣'1'␣to␣test␣a␣new␣number,␣'0'␣to␣
          exit␣the␣program\n"; cin >> choice;

          if(choice == 1){
        goto start;
          }
      else if(choice == 0){
        return(0);//Exit program
          }
        }

        int prime = isprime(r); //line 5 of Fig 2.1
    while(prime == 0){
     r = r+1;//increase r until it becomes prime
     prime = isprime(r);
        }

     ZZ q = largestprime(r-1); //line 6 of Fig 2.1, set
         q to be largest prime factor of r-1
     ZZ_p::init(r); // mod r

     double t1 = 4*sqrt(to_long(r))*(log(n)/log(2));
     //test on lest hand side in line 7 of Fig 2.1
```

```
     ZZ t2;

     t2 = power(n,to_long((r-1)/q))%r;
       //test on right hand side in line 7 of Fig 2.1


     if(to_double(q)>=t1 && t2 !=1){

        break; //line 8 of Fig 2.1, q satsifies required
            conditions, stop while loop
          }

     else{
       r=r+1; // line 9 of Fig 2.1
          }
}

if(r>n){
  r=n;
}

cout << "r=" << r << "\n";

my_file << "r=" << r << "\n";

long a; long b =
    to_long(2*sqrt(to_long(r))*(log(n)/log(2)));

for(a=1;a<=b;a++){ // line 11 of Fig 2.1

        int f = Congruence(a,n,to_ZZ(r));// line 12 of
            Fig 2.1
        //Returns 0 if condition fails, returns 1 if
            holds
    if(f==0){

       my_file << "a␣fails␣at␣" << a << "\n";
               cout << "a␣fails␣at␣" << a << "\n";
               //Line 12 fails for an a, hence n is
                   composite
       my_file << "Hence␣n␣is␣not␣prime\n";

               finish = timeGetTime();
               duration = finish - start;
          cout << "Time␣Taken=" << duration << "␣
              milliseconds\n\n";
          my_file << "Time␣Taken=" << duration << "␣
              milliseconds\n\n";
          my_file.close();
          cout << n << "␣is␣not␣prime.\n\n";

                  int choice;
          //Give user opportunity to continue or exit
              program

          cout << "Press␣'1'␣to␣test␣a␣new␣number,␣'0'␣to␣
              exit␣the␣program\n"; cin >> choice;
          if(choice == 1){
            goto start;
                  }

          else if(choice == 0){
            return(0);//Exit program
                  }
           }
  }

  finish = timeGetTime();
  duration = finish - start;
  my_file << "n␣is␣prime\n"; // line 13 Fig 2.1
  cout << "Time␣Taken=" << duration << "␣
      milliseconds\n\n";
  my_file << "Time␣Taken=" << duration << "␣
      milliseconds\n\n";
  my_file.close();
  cout << n << "␣is␣prime.\n\n";

  int choice;
  //Give user opportunity to continue or exit program
  cout << "Press␣'1'␣to␣test␣a␣new␣number,␣'0'␣to␣exit␣
      the␣program\n"; cin >> choice;

  if(choice == 1){
      goto start;
  }
  else if(choice == 0){
      return(1);//Exit program
  }
}
```

C.2.2 PerfectPower.h

```
#include <NTL/RR.h>
// function to calculate if n = a^b
// takes input ZZ n and returns 1 if n is a perfect
    power,0  otherwise
int perfectpower(ZZ n){

    long b = 2;
     RR k = to_RR(log(n)/log(2));

    ZZ a;
 while(b <= to_long(k)){
   //b cannot be bigger than k

    long c = to_long(ceil((log(n)/log(2))/to_long(b)));
    a = pow(2,c); //assgin guess value for a

        while(power(a,b)>n){

                double d=
                    to_double(((b-1)*a+n/power(a,(b-1)))/b);//Apply
                    Integer Newton's Method
        ZZ e = to_ZZ(floor(d));
        a = to_long(e); // adjust a
        }

    if(n == power(a,b)){
    //if n is a perfect power.
     cout << n <<"␣is␣a␣perfect␣power,␣hence␣is␣not␣
         prime\n";
         cout << "n=␣a^b\n";
         cout << "b=" << b << "\n";
     cout << "a=" << a << "\n\n";
     return(1);
        }
    else{
      b = b + 1;
        }
 }
 if (n != power(a,b)){
     return(0); //n is not a perfect power.
        }
}
```

C.2.3 Isprime.h

```
//Function checks whether a given number is prime
//returns 1 if it is prime, 0 otherwise
int isprime(ZZ n){
ZZ s;
ZZ t;
t = 4;
   for(s = 2; t<=n; s++){

            if(n%s==0){ //n has factors other than n and 1,
                                   //hence is composite.
          return(0);
           }
       else{
          t=t+2*s-1;
           }
   }

   return(1); //n is prime no divisor other than n and 1
       found
}
```

C.2.4 Largestprime.h

```
//calulates the largest prime factor of an input n
ZZ largestprime(ZZ n){
ZZ f; f =1;
ZZ r; r =2;
ZZ x = n;
 while(x!=1 & sqr(r)<=n){
  while(x%r==0){
   x=x/r;
   f=r;
  }
   r=r+1;
 }
 if(x==1){
   return(f);
 }
 else{
   return(x);
 }
}
```

C.2.5 congruence.h

```
// calculates line 10 of Fig 2.2
int Congruence(long a, ZZ n, ZZ r){

   ZZ_p::init(n); //mod n
   ZZ_pX b = ZZ_pX(to_long(r),1)-1; // b = x^r-1;
   ZZ_pX c = ZZ_pX(1,1)-a; // c = x-a;
   ZZ_pX f = PowerMod(c,n,b); // f=(x-a)^n mod c,n which
        is the RHS
   ZZ_pX e = ZZ_pX(1,1);
   ZZ_pX g = PowerMod(e,n,b); // x^n mod b,n
   g = g - a; // g1 = x^n-a mod c, n.

      if(f==g){

                 return(1);//n is prime
         }

         else{

                 return(0);//n is not prime.
         }
}
```

C.3 Lenstra's Modification of the AKS Algorithm

C.3.1 File: AKS_Lenstra.cpp

```
#include "math.h" //standard libraries
#include "fstream.h"
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>
#include <time.h>
#include <NTL/ZZ.h> // NTL Libraries
#include <NTL/ZZ_p.h>
#include <NTL/ZZ_pX.h>
#include <NTL/ZZX.h>
#include <NTL/vec_ZZ.h>
```

```
#include "PerfectPower.h" //Each Indepedent Test
#include "Euler.h"
#include "Congruence.h"
 int main(int argc, char* argv[]){

 start:
  ZZ n;
  n = 0;

  cout << "Enter␣a␣positive␣integer␣number␣n␣you␣want␣to␣
      be␣tested\n"; cin >> n;
   if(n<1){
      cout << "Integer␣n␣needs␣to␣be␣positive\n\n";
      goto start;
   }
   else if(n==1){
      cout << "1␣is␣neither␣prime␣or␣composite.\n\n";
      goto start;
   }
   else if(n==2){
      cout << "2␣is␣prime.\n\n";
      goto start;
   }
   else if(n==3){
      cout << "3␣is␣prime.\n\n";
      goto start;
   }
   ofstream my_file("AKS_Lenstra.doc", ios::app); //
        output result into file
   cout << "n=" << n << "\n";
   my_file << "n=" << n << "\n";

   // start timing
   DWORD start, finish, duration;
   start = timeGetTime ();
   cout << "CPU␣clocks␣per␣second␣=␣" << CLOCKS_PER_SEC *
        1000 << "\n";

   // Test if n is a perfect power
   int PP = perfectpower(n);
   // returns 1 if n is a perfect power,0 otherwise;

   if(PP==1){
    my_file << n << "␣is␣a␣perfect␣power,␣hence␣is␣not␣
         prime\n\n";
```

```
  finish = timeGetTime();
  duration = finish - start;
  cout << "Time␣Taken=" << duration << "␣
      milliseconds\n\n";
  my_file << "Time␣Taken=" << duration << "␣
      milliseconds\n\n";
  my_file.close();

  int choice;
  //Give user opportunity to continue or exit program
  cout << "Press␣'1'␣to␣test␣a␣new␣number,␣'0'␣to␣exit␣
      the␣program\n"; cin >> choice;

  if(choice == 1){
    goto start;
  }
  else if(choice == 0){
    return(0);//Exit program
  }

}

// Find a suitable r
ZZ r = to_ZZ(2);
ZZ R;
ZZ r1;
while(r<n){ // line 3 of Fig 2.2
  ZZ R = GCD(r,n);
  if(R != 1){ // line 4 of Fig 2.2
    cout << "n␣has␣factors␣other␣than␣n␣and␣1,␣hence␣
        is␣composite\n\n";
    my_file << n << "␣is␣composite.\n";
    my_file << R << "␣is␣a␣divisor\n\n";
        cout << R << "␣is␣a␣divisor\n\n";

    finish = timeGetTime();
    duration = finish - start;
    cout << "Time␣Taken=" << duration << "␣
        milliseconds\n\n";
    my_file << "Time␣Taken=" << duration << "␣
        milliseconds\n\n";
    my_file.close();

    int choice;
    //Give user opportunity to continue or exit program
```

```
        cout << "Press␣'1'␣to␣test␣a␣new␣number,␣'0'␣to␣
            exit␣the␣program\n"; cin >> choice;

        if(choice == 1){
            goto start;
            }
        else if(choice == 0){
            return(0);//Exit program
            }

          }
    else{ // lines 5 and 6 of Fig 2.2
     ZZ v = to_ZZ(floor(power_long(to_long(log(n)),2)));
     // order of n mod r is bigger than v;
     int p = 0;
     ZZ_p::init(r); //calculate mod r
     while(v<=r){
     ZZ x = to_ZZ(power_long(to_long(n), to_long(v)));
     //calulates x =n^v
        ZZ_p z = to_ZZ_p(x);
        if(z==to_ZZ_p(1)){
           r1 = r; // store value of r;
          r = n+1;
          break;
            }
        else
        v=v+1;
     }
   }
  r=r+1; // line 7 of Fig 2.2
}
r = r1;

cout << "r=" << r << "\n";

my_file << "r=" << r << "\n";


//calulate lines 11-13 of Fig 2.2

ZZ r2 = Euler(to_long(r));

my_file <<"Euler("<<r<<")␣=" << r2 << "\n";
cout <<"Euler("<<r<<")␣=" << r2 << "\n";
```

```
long a;
 for(a=1; a<=to_long(r2-1); ++a){ // line 9 of Fig 2.2

 int f = Congruence(a,n,r);
 // line 10 of Fig 2.2, returns 1 if condition holds, 0
     otherwise
 if(f==0){
 finish = timeGetTime();
 duration = finish - start;
 cout << "the␣a␣which␣fails␣is␣" << a << "\n";
 my_file << "the␣a␣which␣fails␣is␣" << a << "\n";
 my_file << "n␣is␣not␣prime\n";
 //line 12 fails for particular a
 cout << "Time␣Taken=" << duration << "␣
     milliseconds\n\n";
 my_file << "Time␣Taken=" << duration << "␣
     milliseconds\n\n";
 cout << n << "␣is␣not␣prime.\n\n";
 my_file.close();

 int choice;
      //Give user opportunity to continue or exit program
      cout << "Press␣'1'␣to␣test␣a␣new␣number,␣'0'␣to␣
          exit␣the␣program\n"; cin >> choice;

          if(choice == 1){
        goto start;
          }
      else if(choice == 0){
        return(0);//Exit program
          }

 }
}

 finish = timeGetTime();
 duration = finish - start;
 cout << "Time␣Taken=" << duration << "␣
     milliseconds\n\n";
 my_file << "n␣is␣prime\n";
 //n must be prime if went through this stage, output
     result to file.
 my_file << "Time␣Taken=" << duration << "␣
     milliseconds\n\n";
 my_file.close();
 cout << n << "␣is␣prime.\n\n";
 int choice;
      //Give user opportunity to continue or exit program
      cout << "Press␣'1'␣to␣test␣a␣new␣number,␣'0'␣to␣
          exit␣the␣program\n"; cin >> choice;

          if(choice == 1){
        goto start;
          }
      else if(choice == 0){
        return(1);//Exit program
          }
}
```

C.3.2 PerfectPower.h

Same as Appendix C.1.2

C.3.3 Euler.h

```
//Euler's phi function
ZZ Euler(long r)
{
  long eu = to_long(1), p;

  for (p = 2; p * p <= r; p += 2)
    {
      if (r % p == 0)
        {
          eu *= p - 1;
          r /= p;
          while (r % p == 0)
            {
              eu *= p;
              r /= p;
            }
        }

      if (p == 2)
        p--;
    }

  ZZ eu1 = to_ZZ(eu);
  // now r is prime or 1
  if(r == 1){
  return eu1;
  }

  else{
  return eu1 * (r - 1);
  }
}
```

C.3.4 Congruence.h

Same as Appendix C.1.5