IMPLEMENTATION OF THE AKS PRIMALITY TESTING ALGORITHM

By
Abdullah Al Zakir Hossain

December 2007

Department of Computer Science and Engineering
The University of Asia Pacific
Dhaka – 1209, Bangladesh

IMPLEMENTATION OF THE AKS PRIMALITY TESTING ALGORITHM

By
Abdullah Al Zakir Hossain

December 2007

Department of Computer Science and Engineering
The University of Asia Pacific
Dhaka – 1209, Bangladesh

IMPLEMENTATION OF THE AKS PRIMALITY TESTING ALGORITHM

A thesis submitted to
Department of Computer Science and Engineering
The University of Asia Pacific
Dhaka – 1209, Bangladesh

in partial fulfillment of the requirement
for the degree of

Bachelor of Science in Engineering
in Computer Science and Engineering

By
Abdullah Al Zakir Hossain, Registration No: 04101009
Department of Computer Science and Engineering
The University of Asia Pacific
Dhaka, Bangladesh

Supervised by
Dr. M. Kaykobad
Professor, Department of CSE, BUET

and
Co-Supervised by
Aloke Kumar Saha
Assistant Professor, Department of CSE, UAP

December 2007

CERTIFICATE

This is to certify that this thesis work has been done by Abdullah Al Zakir Hossain, Registration No: 04101009 in Course No: CSE400, Title: Thesis and Project under the supervision of Dr. M. Kaykobad, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh and Aloke Kumar Saha, Assistant Professor, Department of Computer Science and Engineering, The University of Asia Pacific, Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted or is being submitted to anywhere else for the award of any degree or diploma.

Signature of Supervisor
Dr. M. Kaykobad

Abduallah Al Zakir Hossain

ACKNOWLEDGEMENTS

We would like to express our gratitude for the chance and all the help that we've get. We are truly grateful to the Creator Almighty for giving us the chance to be working on such a pleasurable topic.

We are grateful to the respected supervisor, Dr. M Kaykobad, Professor, Department of Computer Science and Engineering, BUET. His help and suggestion to make the solutions in every steps helped us out and made this possible to come this far. And also Aloke Kumar Saha, Assistant Professor, Department of Computer Science and Engineering, UAP; for his support and guidance. We are really grateful to M. Fayyaz Khan; Head of the Department of Computer Science and Engineering who helped us to put it all together gave his experienced suggestions to make it possible. Our Advisor Shaila Rahman, Assistant Professor, Department of Computer Science and Engineering; her kind guidance and advices have made the way smooth for us in many ways. Also thanks to Monzurur Rahman Khan for giving me the idea to calculate the log of large numbers.

We're thankful to our friends who have been a part of our life. Enough words can't be written to express our gratitude to our family and parents; you have made everything possible for us.

The University of Asia Pacific
Dhaka, December 2007.

Abdullah Al Zakir Hossain

In the name of God, the Most Beneficent, the Most Merciful

To
Our Parents

ABSTRACT

Prime numbers are particularly important in ensuring security in networks and in computer systems. While prime numbers are easily defined, discovering a polynomial time deterministic algorithm for it required thousands of years of research. There are many ways of determining whether a number is a prime or not. In cryptography and in many other problems choosing very large prime numbers is important. It is known that factoring of an integer is NP-complete. But status of primality testing was unresolved till 2002 when Manindra Agrawal, Nitin Saxena and Neeraj Kayal found an elegant deterministic polynomial time algorithm. Along with the slower algorithms a faster solution like the Fermat's little test was devised and implemented. But all of them were probabilistic although quite efficient in practice. In this research the main emphasis has been given on the work of Agrawal et al [], and the algorithm has been implemented to compare its performance with already known algorithms.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Prime numbers plays an important role particularly in number theory and generally in every section of mathematics. Since computer has been introduced as the help to the mathematicians to aid in the tedious calculations; and all the process of a computer is based on the theories of mathematics and numbers, prime numbers are an important aspect in computer science. The field of cryptography is a lot based on the analysis of numbers and the implementation of its properties. A few cryptographic algorithms require large prime numbers. Especially the study of properties of prime number can be very enjoyable and interesting.

1.1 Preliminaries:

Definition: Prime Number - An integer greater than 1 which has no divisor except 1 and the number itself is a prime number or a prime.

Definition: Composite Number - The numbers that are not prime, which means the numbers that can be expressed as a multiplication of other numbers except 1 and itself are composite numbers, 1 is considered as a composite.

Definition: Pseudoprime - If a composite number passes a probabilistic primality test then it is called a pseudoprime.

So, according to the definition an integer p can be a prime if it has no other divisor then 1 and p itself; which means to test a number if it is a prime it has to be shown that it does not have factors rather than showing it has. Fortunately all numbers are either composite or prime,

this means if it can be expressed as a multiplication of other two numbers except p and 1 then it can be chosen as a prime.

Definition Primality Test - A test to check a number if it is a prime is the primality test.

To see if p is a prime it's necessary to see if it has all possible factorization, or show the properties of a prime which composites would not comply. Based on these properties the solution mostly leads to searching for a match; which can be tedious for larger numbers. So far even when there are a number of solutions to this problem, there is not one that can efficiently check a number for primality when the number is large enough.

It has to be considered that the problem of checking a number for primality is in class co-NP: if n is not a prime it has an easily verifiable short certificate, viz., and a nontrivial factor of n. In 1974, Vaughan Pratt observed that the problem is in the class NP too and thus putting it in NP ∩ co-NP.

1.2 Prospect & Importance:

It is becoming necessary to test numbers for primality faster and with as much accuracy as possible. Many algorithms including the public key encryption cipher such as the Rivest, Shamir, and Adleman (RSA) must use a public key in very high range, at least such high range so it does not seem feasible to discover by brute force search. However, with the increase of the computational power searching in such ways are becoming easier and the encryption techniques are becoming vulnerable; checking for larger prime numbers is therefore necessary to implement in any such algorithms. Moreover, prime numbers play an important role in mathematics, not only the series of prime numbers

been an interesting topic but also a key to solve many mathematical problems.

In search for very large prime numbers a number of steps have already been taken, such as Global Internet Mersenne Prime Search (GIMPS), where a parallel computing process is being run to identify very large prime numbers of the form $2^p - 1$, where p is a prime. The AKS primality test, known as Agrawal-Kayal-Saxena primality test or cyclotomic AKS test is a deterministic primality-proving algorithm created and published by three scientists of Indian Institute of Technology Kanpur, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena on August 6, 2002 in a paper titled PRIMES is in P. It allows one step closer to identify larger primes with accuracy. Proves has been given for the correctness of the algorithm and has been implemented in show its performance in this paper.

1.3 Objective of the Thesis:

The primary goal of the research is to come up with ideas that can help to apply the mathematical theorems in reality. Many useful theorems and procedures are still not applicable because of the complexity and lack of accuracy in practice. The basic way of implementing these problems in reality is by developing an algorithm that can assure the result. The Agrawal-Kayal-Saxena Algorithm is such an elegant solution to the primality testing problem that puts forward a way to end the thousand years research in this area.

The AKS algorithm is not as fast as the former randomized algorithms such as Fermat's little test or Miller-Rabin tests, but it is accurate. The result of the algorithm is guaranteed and proved to be correct. And since the time taken to resolve a result is in polynomial so the increase of time with increase of number is practically implementable and usable. Also it leaves option to upgrade it for faster

performance; most importantly it shows the way to discover better. Besides since this solution particularly features the polynomial time complexity, so when the input is large enough the other algorithms may become slow which wouldn't happen in case of AKS algorithm.

1.4 Organization of the Thesis:

The overview of the research is arranged showing the evolution of the algorithm. To bring up the contrast of performance firstly the naïve methods to solve the problem is shown; then the most popular one which actually is used for several years in practice the probabilistic algorithm of Fermat and then the description of the AKS algorithm along with the implementation and correctness.

The chapter wise organization is detailed in this section.

Chapter 1 – Introduction: Here the historical background and importance of the problem is described along with the problem statement and preliminaries.

Chapter 2 – Naïve Methods for Primality Testing: Here the previous ways to solve the problem are described. The problems are usually used and been practiced for centuries to find small prime numbers.

Chapter 3 - A Probabilistic Algorithm Fermat's Little Test: When it comes to larger numbers a faster and reliable method is required which has been developed from famous theorem of Fermat's Little Test which can choose prime numbers fast enough but cannot assure the result.

Chapter 4 - Deterministic Polynomial Time AKS Algorithm: Finally the deterministic solution to the problem that can produce result in polynomial time is described as per the main paper 'Primes is in P'.

This paper also consists of the implementations of the theorems and algorithms described in chapters and the source codes are given in the appendixes.

Chapter 5 – Conclusion and a few notes on future works.

Appendix A – Contains all the source codes for the naïve methods for primality testing.

Appendix B – Contains the source code to implement the Fermat's little test.

Appendix C – Contains the source code for the AKS algorithm which works properly in polynomial time.

And finally a bibliographical note has been added to show the sources of resources that have been used to prepare this paper.

1.5 Remarks on Notations:

The symbols and the notations that are used to describe the algorithm and mathematical equations are described here.

The four symbols from formal logic, viz.

$$\rightarrow, \equiv, \exists, \in$$

→ is to be read as 'implies'. Thus,

$$l \mid m \rightarrow l \mid n$$

means "l is a divisor of m" implies "l is a divisor of n". Or "b | a . c | b → c | a" means "if b divides a and c divides b then c divides a".

≡ is to be read 'is equivalent to'. Thus

$$m \mid ka - ka' \equiv n \mid a - a'$$

means that the assertions "m divides ka – ka' " and " n divides a – a' " are equivalent; either implies the other.

These two symbols must be distinguished carefully from → (tends to) and ≡ (is congruent to). There can hardly be any misunderstanding, since → and ≡ are always relations between propositions.

∃ is to be read as 'there is an'. Thus

$$\exists\, l\,.1.< l < m\,.\, l \mid m$$

means 'there is an l such that (i) 1 < l < m and (ii) l divides m'.

∈ is the relation of a member of a class to the class. Thus

$$m \in S\,.n\ \in S\ \rightarrow (m\ \pm n) \in S$$

means 'if m and n are members of S then m + n and m – n are members of S'.

1.6 Literature Survey:

So far with the development in the primality testing algorithms, in broad line there are of two kinds, probabilistic and deterministic. A deterministic test is one that can prove if the number is a prime without any exception and the results of such algorithms are always correct. A probabilistic test is one that can determine if a number is a prime, but there are exceptions when the test fails. In many cases probabilistic tests are efficient enough and exceptions are too small when the number is not very large. The outcome of this kind of algorithms are said to be probable prime until it is proven deterministically.

Another way to categorize the algorithms can be based on the conditions it applies. Some algorithms are conditional and the correctness of the algorithm is dependent on other hypothesis that is not proven yet; and some algorithms are unconditional which has been proven based on all proven facts.

A number of researches have been conducted to analyze the properties of the prime numbers and check for primality; such as, Sieve of Eratosthenes, Adleman-Pomerance-Rumely Primality Test, Agrawal-Kayal-Saxena Primality Test, Elliptic Curve Primality Proving, Fermat's

Little Theorem, Fermat Pseudoprime, Fermat's Little Theorem Converse, Fermat's Theorem, Lucas-Lehmer Test, Miller's Primality Test, Pépin's Test, Pocklington's Theorem, Prime Factorization Algorithms, Proth's Theorem, Pseudoprime, Rabin-Miller Strong Pseudoprime Test, Ward's Primality Test, Wilson's Theorem etc. Among these Sieve of Eratosthenes is a way to generate all the prime numbers within a given range.

Among these only a few will be discussed in this paper. Apart from the Naïve methods which has been known from 240 B.C. one of those Naïve methods is Sieve of Eratosthenes, which can generate all the prime numbers in the given range; a faster and efficient algorithm to check for primality has been researched ever since. A fast efficient test should need only a polynomial i.e. in the size of the input which is log n number of steps.

Almost successfully such a test has been introduced by Pierre de Fermat, known as Fermat's little theorem, so far the simplest test. Pierre de Fermat first stated the theorem in a letter dated October 18, 1640 to his friend and confidant Frénicle de Bessy as the following: p divides $(a^{(p-1)} - 1)$ whenever p is a prime and a is relatively prime to p. And it works with the exception of Carmichael numbers; such numbers are composites but still it can pass the Fermat's test. It was a common practice then that he didn't disclose the proof of the theorem, he only stated, "Et cette proposition est généralement vraie en toutes progressions et en tous nombres premiers; de quoi je vous envoierois la démonstration, si je n'appréhendois d'être trop long. (And this proposition is generally true for all progressions and for all prime numbers; the proof of which I would send to you, if I were not afraid to be too long)". So, the theorem remained without proof for a long time, nearly a century, till Euler first published a proof in 1736 in a paper entitled "Theorematum Quorundam ad Numeros Primos Spectantium

Demonstratio". And Leibniz left virtually the same proof in an unpublished manuscript from sometime before 1683. Euler had also been able to generalize the theorem in terms of a new function ø(n) discovered by himself.

In 1975 another primality test The Miller-Rabin primality test has been introduced; an algorithm which determines whether a given number is prime, similar to the Fermat primality test and the Solovay-Strassen primality test. Its original version, due to Gary L. Miller, is deterministic, but it relies on the unproven generalized Riemann hypothesis; Michael O. Rabin modified it to obtain an unconditional probabilistic randomized algorithm.

Also in 1983, another deterministic version came in with `Adleman, Pomerance, Rumely' which takes about O($\log(n)^{O(\log(\log(\log(n))))}$). And in 1986 Goldwasser, Killan introduced a randomized algorithm with expected polynomial time for almost all inputs. Also another randomized algorithm came in 1992, with Adleman, Huang: randomized polynomial time.

The AKS primality test, known as Agrawal-Kayal-Saxena primality test or cyclotomic AKS test is a deterministic primality-proving algorithm created and published by three scientists of Indian Institute of Technology Kanpur, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena on August 6, 2002 in a paper titled PRIMES is in P. The authors received the 2006 Gödel Prize for this work. The last two authors were partially supported by MHRD grant MHRD-CSE-20010018. The algorithm, which was soon improved by others, determines whether a number is prime or composite and runs in polynomial time. The key significance of AKS is that it was the first published primality-proving algorithm to be simultaneously polynomial, deterministic, and unconditional. That is, the maximum running time of the algorithm can be expressed as a

polynomial over the number of digits in the target number; it guarantees to distinguish whether the target number is prime or composite rather than returning a probabilistic result; and its correctness is not conditional.

Descriptions of the popular tests are presented with its correctness, implementation and analysis.

1.7 Choice of Programming Language: Java:

Since the thesis is based on mathematical processing and over the past many years the method library of Java has been developed to support many mathematical functions it is widely used in making the numerical programs. The package defined in Java library `java.math.*' shows various useful classes and methods within. Since the main application of algorithms such AKS are for the much larger numbers so using the normal data structures as `int' or`double' is not enough. To handle the larger numbers which are over 50 or more digits long the `java.math.BigInteger' class is used which is already defined in java library. Also since these algorithms are commonly used in cryptography and cryptography is very commonly implemented in Java because of the security heavy features, I've also chose this language to show the performance of the algorithm.

2. NAÏVE METHODS FOR PRIMALITY TESTING

Some algorithms to check a number if it is a prime has been popular since very ancient time. As the definition suggests to proof a number is a prime we have to ensure that at least the numbers less than the number cannot divide it, otherwise it would be composite.

The simplest test can be, for a number n, it has to be checked if all numbers between 2 and n - 1 divides n. If n is divisible by any of the number then n is composite otherwise it's prime. But rather then testing all n - 1 we can test only to its √n. If n is composite then it can be factored into two values, at least one of which must be less than or equal to √n. Leaving it to be as given in algorithm 2.1. This algorithm returns true only when all the `i' has been checked between 2 and √n and couldn't be returned false. So it deterministically shows if the given number is a prime, but with complexity of O(√n).

```
Procedure isPrime1( int n )
{
       for i = 2 to √n
       do
              if (n mod i) == 0 then return false;
       od

       return true;
}
```

Figure 01: Algorithm 2.1

We can also improve the efficiency by skipping all even m except 2, since if any even number divides n then 2 does. We can further improve by observing that all primes are of the form 6k ± 1, with the only exceptions of 2 and 3. This is because all integers can be

expressed as (6k + i) for some k and for i =1, 0, 1, 2, 3, or 4; 2 divides (6k + 0), (6k + 2), (6k + 4); and 3 divides (6k + 3). We first test if n is divisible by 2 or 3, then we run through all the numbers of form 6k ± 1 < √n. This is 3 times as fast as the previous method. In fact, all primes are of the form c# k + i for i < c# where i represents the numbers that are relatively prime to c#. In fact, as c → ∞ the number of values that c# k + i can take over a certain range decreases, and so the time to test n decreases. For this method, all primes must be divide that are less than c. Within the range of 2 to√n, only the prime numbers can be checked for divisibility with n. This gives the algorithm 2.2.

```
/* PRIMES is the array of prime numbers. */

Procedure isPrime2( int n )
{
       i = 0;
       while PRIMES[i] <= √n
       do
              if n mod PRIMES[i] == 0 then return false;
              i++;
       od

       return true;
}
```

Figure 02: Algorithm 2.2

Observations analogous to the preceding can be applied recursively, giving the Sieve of Eratosthenes. A good way to speed up these methods is to pre-compute and store a list of all primes up to a certain bound, say all primes up to n. Such a list can be computed with the Sieve of Eratosthenes. Then to test if n is a prime we can search the list if n exist in there.

In mathematics, the Sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to a specified integer. It is the predecessor to the modern Sieve of Atkin, which is faster but more complex. It was created by Eratosthenes, an ancient Greek

mathematician. Wheel factorization is often applied on the list of integers to be checked for primality, before Sieve of Eratosthenes is used, to increase the speed. It has been introduced 240 B. C. and it is one of the oldest ways to choose prime numbers. In this procedure in a list of all numbers the composites will be sorted out leaving behind the prime numbers.

Suppose the range of numbers is n, then first composite is 1 and first even prime is 2. So keeping 2 all the multiples of 2 are listed out in the composite numbers. In this way keeping 3 all the multiples of 3 are listed in composite leaving the primes in the original list. And inducting the same logic this would have to be done up to √n.

```
/* NUMBERS is the Boolean array of size n, */
/* declared as global, and initiated true in every index */

Procedure Sieve( NUMBERS[] )
{
	NUMBERS[1] = false;

	for i = 2 to √n
	do
		if NUMBERS[i] != false then
			for j = i*i to N
			do
				NUMBERS[j] = false;
			od
			j = j + i;
		fi
		i++;
	od
}

Procedure isPrime3( int n, NUMBERS[] )
{
	if NUMBERS[n] == true then return true;
	else return false;
}
```

Figure 03: Algorithm 2.3

One simple way to do that is by keeping a Boolean array of reasonably large size according to n. And initially keeping all the

position of the array to be `true'; and eventually the indexes that are the multiple of the previous primes can be denoted as `false'.

In Algorithm 2.3 procedure Sieve all the composite indexes of the array `NUMBERS' are being set as `false'. Once the procedure is called the entire prime index of NUMBERS would become true and the composite indexes would be false. So, in first steps it takes a little time, but once it's distinguished then the later procedure isPrime3 can work pretty fast. It can test whether the given n is a prime or not by just checking if the n^{th} index of numbers is true or not.

The limitation of this algorithm is the memory limit. Usually with the popular 32-bit microcomputer can have 4 Giga bytes of system memory, and most computers don't have that. And each Boolean variable takes about a byte, and it is also possible to use a bit. So using an array of bytes the above algorithm can have an array of 4,000,000,000 bytes at most, and is not practically possible because the system itself would have been using some of the memory. And if the array can be taken in the size of a bit then the size have to be a lot less than 8 X 4,000,000,000 = 32,000,000,000. So, the limitation for the algorithm 2.3 is it cannot check a number for primality that is larger than this. Also it should be much time consuming to index such large array.

But in all the above algorithms so far the problem relies in checking n for primality is that the procedure requires to investigate all the previous prime numbers then n. These procedures cannot check a number for primality irrespective of the primes below the number.

So, it is easily observable that these methods are not feasible to be used for any numbers larger than the current 32-bit or 64-bit integer range.

3. PROBABILISTIC ALGORITHM: FERMAT'S THEOREM

One important primality testing algorithm that shows a very interesting property of prime numbers is the Fermat's Little Test. But the algorithm formulated from the Fermat's theorem is a probabilistic one, because there are exceptions, the same property of the theorem has been observed in a few composite numbers too.

3.1 Overview of Fermat's Theorem

The Fermat's theorem is one of the most famous and beautiful theorem in number theory and it's the first one to enable us to develop a polynomial time algorithm for primality testing. The discovery of this special property of prime numbers made by Fermat in 1640 and later it has been proved by Euler and Leibnitz.

Theorem 3.1: Fermat – Let p be a prime, (a, p) = 1; then $a^{p-1} - 1$ is divisible by p.

Here, (a, p) denotes the greatest common divisor of `a' and `p'. And it's true for every prime number that all $a^{p-1} - 1$ is divisible by p, where the greatest common divisor of `a' and `p' is 1. To proof this theorem we need to look up for a few more related theorems which are described here in the following section.

3.1.1 Related Theorems

The proof of the theorem requires a few other related theorems. The necessary theorems and the proofs are given here.

Theorem 3.2: Let k divide both a and b and let (k, m) = d, then

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

implies ,

$$\frac{a}{k} \equiv \frac{b}{k} (\text{mod } \frac{m}{d}).$$

Proof of Theorem 3.2: m divides (a – b). Hence m divides $\left(\frac{a}{k} - \frac{b}{k}\right)$ k. Therefore $\frac{m}{d}$ divides $\left(\frac{a}{k} - \frac{b}{k}\right)\frac{k}{d}$.

But we know that $\left(\frac{m}{d}, \frac{k}{d}\right) = 1$. It follows that $\frac{m}{d}$ divides $\left(\frac{a}{k} - \frac{b}{k}\right)$. The following is an immediate deduction from the above theorem.

Corollary if $ac \equiv bc(mod\ m)$ and $(c, m) = 1$ then

$$a \equiv b\ (mod\ m)$$

Thus it can be canceled a common factor from the two sides of a congruence if the factor is relatively prime to the modulus.

Theorem 3.3: Let p be a prime. Then,

$$\binom{p}{r} = \frac{\mathrm{p(p-1)(p-2)(p-r+1)}}{\mathrm{r!}}$$

where o < r < p, is divisible by p.

Proof of Theorem 3.3: p (p – 1) ... (p – r + 1) is the product of r consecutive integers, hence it is divisible by `r!'. But `r!' is relatively prime to p. It follows that `r!' divides (p – 1)(p – 2)...(p – r + 1). This implies,

$$\frac{\mathrm{p(p-1)(p-2)(p-r+1)}}{\mathrm{r!}}$$

is a multiple of p. So the theorem is proved that $\frac{\mathrm{p(p-1)(p-2)(p-r+1)}}{\mathrm{r!}}$ is divisible by p.

Theorem 3.4: if p is a prime then the congruence is,

$$(a + b)^p \equiv a^p + b^p \text{ (mod p)}$$

Proof of Theorem 3.4: According to the binomial theorem $(a + b)^p$ can be expressed as,

$$(a + b)^p = a^p + \binom{p}{1} a^{p-1} b + \binom{p}{2} a^{p-2} b^2 + \dots + \binom{p}{p-1} a\, b^{p-1} + b^p$$

or, $(a + b)^p = a^p + b^p$ + terms divisible by p.

According to Theorem 3.2 all the terms except a^p and b^p are divisible by p. So it follows that,

$$(a + b)^p \equiv a^p + b^p \pmod p$$

Successive application of the last theorem can prove the following theorem.

Theorem 3.5: $(a_1 + a_2 + \dots + a_n)^p \equiv a_1^p + a_2^p + a_3^p + \dots + a_n^p \pmod p$

Proof of Theorem 3.5:

$$(a_1 + a_2 + \dots + a_n)^p \equiv a_1^p + (a_2 + a_3 + \dots + a_n)^p \pmod p$$

$$(a_1 + a_2 + \dots + a_n)^p \equiv a_1^p + a_2^p + (a_3 + a_4 + \dots + a_n)^p \pmod p$$

.

.

$$(a_1 + a_2 + \dots + a_n)^p \equiv a_1^p + a_2^p + a_3^p + \dots + a_n^p \pmod p$$

So, using these theorems it is easily possible to proof the Fermat's Theorem.

3.1.2 Proof of Theorem 3.1: Fermat's Theorem

According to Theorem 3.4,

$$(a_1 + a_2 + \dots + a_n)^p \equiv a_1^p + a_2^p + a_3^p + \dots + a_n^p \pmod p$$

Letting $a_1 = a_2 = a_3 = \dots = a_n = 1$ it can be obtained,

(1) $$a_p \equiv a \pmod p$$

But (a, p) = 1, therefore by corollary of theorem 3.1 it can be canceled the common factor `a' from the two sides of the congruence (1). So it follows,

(2) $$a^{p-1} \equiv 1 \pmod p$$

This implies,

$$a^{p-1} - 1 \equiv 0 \pmod{p}.$$

The theorem is therefore proved.

3.1.3 The Idea & algorithm

There are many ways in which Fermat's theorem can be proved. Considering the congruence Fermat's theorem can be expressed in the following manner,

Theorem 3.6: Fermat's little Theorem congruence – Let p be a prime and a be any positive integer then,

Fermat's equation 3.1: $a^p \equiv a \ (mod\ p)$

and since $(a, p) = 1$ so, $a^{p-1} \equiv 1 \ (mod\ p)$

This congruence allows us to generate a probabilistic algorithm to test a number for primality. From theorem 3.6 another theorem can be derived.

Theorem 3.7: If p is a prime then the equation $x^2 \equiv 1 \ (mod\ p)$ has exactly two solutions namely 1 and n - 1.

Corollary, if the equation$x^2 \equiv 1(mod\ n)$ has roots other than 1 and $n - 1$, then n is composite.

Any integer x which is neither 1 nor n – 1 but which satisfies $x^2 \equiv 1(mod\ n)$ is said to be a nontrivial square of 1 modulo n. From these conditions, an algorithm can be derived to check if a number is a prime or composite. Having an input of n the Fermat's equation 3.1 has to be checked with several values of `a' where 1 < a < n; and if all satisfies then at least it can be said that the input number n has the property of prime that is described in Fermat's theorem.

```
Procedure isPrime1( int n )
{
	for a = 2 to n
	do
		m = modPower(a, n, n);
		if( m != a ) return false;
	od

	return true;
}
```

Figure 04: Algorithm 3.1

Using this idea an algorithm can be formulated. Primarily, since the more checking is involved there's a higher chance for the number to be a prime so it's better to check it with all the values of $1 < a < n$. As it is described in the procedure 3.1; it returns true for every input of prime number, including some composite numbers that are denoted as Carmichael numbers. The procedure `modPower(a, n, n)' that is used in 3.1, calculates the value of (a^n mod n), and assigns it in m, for any value of a, if m is not equal to a then it doesn't satisfies the Fermat's equation thus the procedure returns false; otherwise it returns true. Computing the x^n is not relative topic here, but an algorithm to compute the x^n mod b in polynomial time is given in algorithm 3.4 and discussed in the implementation section of this chapter.

The more checking with the different values of `a' is involved; there's a higher probability for the answer to be correct. But even after checking with all the values of `a' doesn't give a confirmation of a number to be a prime.

3.2 Carmichael & Pseudo-prime Numbers:

Unfortunately, still there are numbers which does satisfies the Fermat's equation but not prime. The theorem says that for each prime numbers the equation will hold, which is true, but it also holds for some composite numbers, this has been discovered by Robert Carmichael

and so named after him as Carmichael numbers which are all odd composites.

Definition: Carmichael Numbers – A positive odd composite number `n' is a Carmichael number if it can satisfy the Fermat's equation $a^{n-1} \equiv 1 \pmod{n}$ for all `a' relatively prime to n.

Definition: Pseudo-prime Numbers – A positive odd composite number `n' can said to be a pseudo-prime is it can satisfy the Fermat's equation $a^{n-1} \equiv 1 \pmod{n}$ for any value of `a'.

Actually, the Carmichael numbers satisfies the Fermat's equation for all values of `a' between 2 and n – 1. If all the values of `a' between 2 and (n–1) are checked to choose a number for primality then it should return a correct answer with the exception of Carmichael numbers. So, the algorithm in 3.1 would remain probabilistic. So, there's a practice to use randomized algorithm for this case, so it minimizes the cost and lets the user to choose how much cost and definite answer is required. One such algorithm is described in 3.2.

```
Procedure isPrime( int n, int p )
{
       for i = 1 to p
       do
              a = random() mod n;

              y = modPower( a, n, n);

              if( y != a ) then return false;
       od

       return true;
}
```

Figure 05: Algorithm 3.2

If the input `n' is a prime then algorithm 3.3 will never return an incorrect answer. The value of p has to be called by the user to ensure the probability of the returning result to be correct.

The procedure `random() mod n' returns a random value between 0 and n – 1; the one problem of the algorithm is that if the random numbers are not unique, there's a chance for trying the equation for the same value of `a' more than once which is unnecessary. A sufficiently large p ensures high probability of the correctness of $\geq 1 - n^{-p}$.

If the input n is a composite number and if the procedure returns true then there's a possibility that the number is a Carmichael number, which the given algorithm will not be able to find out even it is a composite.

To categorize the Carmichael numbers there has been a number of researches. There are about 585355 such composite numbers that are pseudo-primes, has been observed within the range of 10^{17} that satisfies the fermat's equation for at least one value.

3.3 The Korselt's Criterion

A categorization method of Carmichael numbers has been shown by Korselt in 1899, the theorem is stated below.

Theorem: Korselt's Criterion - A positive composite integer n is a Carmichael number if and only if n is square-free, and for all prime divisors p of n, it is true that $p|n - 1$. This follows the categorizations,

i) it has to be composite,

ii) it has to be square free, and

iii) for all prime p, if $p|N$, then $p - 1|N - 1$; then N can be said to be a Carmichael number.

Proof of Theorem: Korselt's Criterion – If N satisfies the criterion; then, as N is square free, we know that in order for N to divide $k^N - k$, all we require is that $p|k^N - k$ for all p|N. Fortunately, Fermat's Little Theorem tells us that, when $k \neq 0$, $k^{p-1} = 1 \pmod p$ for all k, and, as $p - 1|N - 1$, we know that $k^{N-1} = 1 \pmod p$; given that $p|k^{N-1} - 1$ for all p|N, it follows by taking the least common multiple of all such p (again taking advantage of the fact that N is square free) that $N|k^{N-1} - 1$, or that $k^{N-1} = 1 \pmod n$; multiplying by k to add the trivial case k = 0 tells us that Korselt's Criterion implies absolute pseudo primality.

Let N be an absolute pseudo prime; we take it as true that $N \mid k^N - k$ for all k. First, we prove that n must be square free; if N is not square free, then we can find some factor of N with the form c^2. Note, then, that, as $c^2|N$ and $N|c^N - c$, we have $c^2|c^N - c$. Note that, as $c^2|c^N$ trivially, this forces $c^2|c$, which is impossible. Thus, it must be that N is square free, as we expected. Now, it remains to prove that if p|N, then $p-1|N-1$; for this, we turn to some simple theory of the finite group $Z \times p$. Since this group is cyclic, and therefore a generator `a' can be chosen, which will have group order $p - 1$. We have p|N, and, by hypothesis, $N|a^N - a$, so $p|a^N - a$. And since $p \nmid a$, so therefore, we can derive that $p|a^{N-1} - 1$. It is clear from this that $a^{N-1} = 1 \pmod p$, and it follows from this that $N - 1$ must be a multiple of $p - 1$, as desired. Combining this with the result of the previous paragraph gives us the final result, Korselt's criterion for absolute pseudo primality.

There are only 15 Carmichael numbers within the range of 16-bit integer. The distribution of first 15 Carmichael numbers and its prime factors are given below. So it means that if all the primes to be generated within the range of 2 to $2^{16} - 1$ then there would be 15 different times the Fermat test would produce false result.

1	561	3 X 11 X 17
2	1105	5 X 13 X 17
3	1729	7 X 13 X 19
4	2465	5 X 17 X 29
5	2821	7 X 13 X 31
6	6601	7 X 23 X 41
7	8911	7 X 19 X 67
8	10585	5 X 29 X 73
9	15841	7 X 31 X 73
10	29341	13 X 37 X 61
11	41041	7 X 11 X 13 X 41
12	46657	13 X 37 X 97
13	52633	7 X 73 X 103
14	62745	3 X 5 X 47 X 89
15	63973	7 X 13 X 19 X 37

Figure 06: First fifteen Carmichael numbers

But it's hard to choose if a number is a Carmichael number in polynomial time without the help of further research. So the Fermat's test remains probabilistic even when the p is sufficiently large.

An Implementation and analysis of the Fermat's Test

An implementation to find out the Prime and the Carmichael numbers using the algorithm 3.1 and 3.2 is discussed and attached in appendix B. The code is divided into classes, the class `Fermat' contains the procedures for algorithm 3.1 and 3.2 and since the procedures requires algorithm 3.2 it is also included. And the methods are overloaded to use in both randomized and non-randomized ways.

If the methods are called with a probability constant then the randomized algorithm 3.2 will be processed and without the probability constant the algorithm 3.1 will be processed. The procedures have been called as the event from button, so the output should be processed by the time it is released.

The first task of implementing the code is to define the required internal procedures and organize it in object-oriented way so that the code can be shared and implemented as necessity. The important task

to code both algorithms 3.1 and 3.2 is to define the algorithm to calculate modular exponentiation. Both of the algorithms require calculating a^n mod n; fortunately a polynomial time algorithm can be formulated to calculate this.

A naïve algorithm to calculate `a^n' is to perform (n – 1) multiplications, which leads to the following algorithm in 3.3.

```
Procedure power (int a, int n)
{
       p = n;

       for i = 1 to n – 1
       do
              p = p * n;
       od

       return p;
}
```

Figure 07: Algorithm 3.3

To calculate the modular result of the exponentiation the following theorem has to be considered.

Theorem 3.8:

$$a_1\, a_2\, a_3 \ldots . a_n \equiv b_1\, b_2\, b_3 \ldots . b_n \pmod m$$

The theorem allows to modify the above algorithm and calculate a^n mod n and return the answer only by calculating p = p * n mod n.

But to develop a polynomial time algorithm to calculate the value a certain things have to be considered. It is easily observable, that if the power of `a' is a number that can be written as a power of any other number, for example the smallest as 2, then we can easily calculate a ^ 2 ^ d instead of a ^ n where n = 2 ^ d. If n cannot be written

as a power of 2 then we can subtract 1 from the power and then try to divide it to be written in form of 2 ^ d; and during this process the subtracted 1 can be calculated by multiplying that power with `a' once. Considering the Theorem 3.8 the following algorithm can be developed to calculate the modular exponentiation.

```
Procedure modPower( int x, int n, int b )
{
	m = n;
	p = 1;
	z = x;
	while( m > 0)
	do
		while( m mod 2 == 0 )
		do
			m = m/2;
			z = (z * z) mod b;
		od
		m = m – 1;
		p = (p * z) mod b;
	od

	return p;
}
```

Figure 08: Algorithm 3.4

The implementation in Appendix B uses this algorithm and calculates modular exponentiation in polynomial time. The class `FermatTest' contains all these 3.1, 3.2 and 3.4 algorithms implemented and allows users to call the methods and use accordingly.

Using these procedures it is possible to implement the algorithm described in 3.1 and 3.2. The specialty of the theorem is that all the major improvements that have been done on this problem are based on this theorem and modifications.

4. DETERMINISTIC POLYNOMIAL TIME AKS ALGORITHM

A deterministic test has to be able to choose prime numbers with complete certainty and it cannot depend on any other assumptions or hypothesis. Such an algorithm known as AKS algorithm, which was soon improved by others, that determines whether a number is prime or composite and is unconditional. After taking the hundreds of year's research a polynomial time algorithm for the problem has been proposed the AKS primality test also known as Agrawal-Kayal-Saxena primality test and cyclotomic AKS test.

4.1 PRIMES is in P: AKS Algorithm

It is a deterministic algorithm that ensures the result to be a prime without any doubt. The test has been created and published by three Indian Institute of Technology Kanpur scientists, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena on August 6, 2002 in a paper titled PRIMES is in P. The authors received the 2006 Gödel Prize for this work. Though it is much slower than other probabilistic or randomized tests but the accuracy is proven and runs in polynomial time, but still it's slower to implement where a large prime number has to be chosen faster.

The key significance of AKS is that it was the first published primality-proving algorithm to be simultaneously polynomial, deterministic, and unconditional. That is, the maximum running time of the algorithm can be expressed as a polynomial over the number of digits in the target number; it guarantees to distinguish a prime and its correctness and it is not conditional on the correctness of any subsidiary unproven hypothesis such as the Riemann hypothesis like Miller-Rabin test.

Despite the impressive progress made so far, this goal has remained elusive. In this paper a deterministic, $O\sim(\log^{15/2} n)$ time algorithm for testing if a number is prime is given. Heuristically, this algorithm does better; under a widely believed conjecture on the density of Sophie Germain primes (primes p such that 2p + 1 is also prime), the algorithm takes only $O\sim(\log^{6} n)$ steps.

The algorithm is based on a generalization of Fermat's Little Theorem to polynomial rings over finite fields. Notably, the correctness proof of this algorithm requires only simple tools of algebra; except for appealing to a sieve theory result on the density of primes p with $p-1$ having a large prime factor—and even this is not needed for proving a weaker time bound of $O\sim(\log^{21/2} n)$ for the algorithm. In contrast, the correctness proofs of earlier algorithms producing a certificate for primality are much more complex.

4.2 Overview

In following Section the basic idea behind the test is summarized and the algorithm is stated according to `PRIMES is in P'. And later the notations used are discussed then the algorithm is stated and its proof of correctness is presented.

4.2.1 Idea

The test is based on the following identity for prime numbers which is a generalization of Fermat's Little Theorem as discussed in Chapter 3.

Lemma 4.1: Let $a \in Z$, $n \in N$, $n \geq 2$, and $(a, n) = 1$. Then n is prime if and only if,

$$(x + a)^n = x^n + a \pmod n \qquad \ldots.(1)$$

Proof: The same argument can be derived as described in Theorem 3.4. Suppose n is composite. Consider a prime q that is a factor of n and let $q^k || n$. Then q^k does not divide $\binom{n}{q}$ and is coprime to a^{n-q} and hence the coefficient of x^q is not zero (mod n). Thus $((x + a)^n - (x^n + a))$ is not identically zero over Z^n.

This identity suggests a simple test for primality; as discussed in chapter 3; given an input n, choose an `a' and test whether the congruence (1) is satisfied. However, this takes time Ω(n) because we need to evaluate n coefficients in the LHS in the worst case. A simple way to reduce the number of coefficients is to evaluate both sides of (1) modulo a polynomial of the form $x^r - 1$ for an appropriately chosen small r. In other words, test if the following equation is satisfied,

$$(x + a)^n = x^n + a \pmod{x^r - 1, n} \qquad \ldots(2)$$

From the congruence it can be said all the primes n satisfies the equation for all the values of `a' and `r'. The correctness of the algorithm is in proving that for appropriately chosen r if the equation (2) is satisfied for several values of `a' then `n' must be a prime power. The number of values that have to be checked of `a' and the appropriate values of `r' are both bounded by a polynomial in log n and therefore a deterministic polynomial time algorithm for testing primality is achieved. The challenge and the deterministic behavior of the algorithm is depended on choosing the right value of `r' and evaluating the congruence.

4.2.2 The Algorithm

The algorithm is given below according to the `PRIMES is in P', the notations and preliminaries and the steps involved to evaluate the algorithm will be discussed later in this chapter. The proof of correctness is described according to this form of algorithm.

```
Procedure isPrimeAKS( int n )
{
       if(n = a^b for a ∈ N and b > 1)return false;

       find the smallest r such that o_r(n)>log^2 n.

       if(1<(a, n)<n for some a≤r) return false;

       if(n≤r) return true; *

       for a = 1 to ⌊√(φ_r) log n⌋
       do
              if((X + a)^n = X^n + a (mod X^r – 1, n))
                     return false;
       od
       return true;
}
```

Figure 09: Algorithm 4.1

4.2.3 Notations and Preliminaries

Some notations and signs of sets are used to describe the algorithm and its proof. The Z_n denotes the ring of numbers modulo n and F_p denotes the finite field with p elements, where p is a prime. Recall that if p is prime and h(x) is a polynomial of degree d and irreducible in F_p, then $F_p[x]/(h(x))$ is a finite field of order p^d. We will use the notation f(x) = g(x) (mod h(x), n) to represent the equation f(x) = g(x) in the ring $Z_n[x]/(h(x))$. We use the symbol $O\sim(t(n))$ for $O(t(n) \cdot \text{poly}(\log t(n))$, where t(n) is any function of n. For example, $O\sim(\log^k n) = O(\log^k n \cdot \text{poly}(\log \log n)) = O(\log^{k+\varepsilon} n)$ for any $\varepsilon > 0$. We use log for base 2 logarithms, and ln for natural logarithms.

Here, N and Z denote the set of natural numbers and integers respectively. Given $r \in N$, $a \in Z$ with (a, r) = 1, the order of a modulo r is the smallest number k such that $a^k = 1 \pmod r$. It is denoted as $o_r(a)$. For $r \in N$, $\phi(r)$ is Euler's totient function giving the number of numbers less than r that are relatively prime to r. It is easy to see that $o_r(a) \mid \phi(r)$ for

any a, (a, r) = 1. We will need the following simple fact about the lcm of the first m numbers.

Lemma 4.2: Let LCM(m) denote the lcm of the first m numbers. For m ≥ 7:

$$LCM(m) \geq 2m$$

4.3 Theorems and Correctness of the algorithm:

In this section the relative theorems and the proof of correctness is described according the main paper. A detailed discussion with the implementable syntax is discussed in the later sections.

Theorem 4.1: The algorithm above returns true if and only if n is a prime.

Proof: In the remainder of the section, we establish this theorem through a sequence of lemmas as described below.

Lemma 4.3: If n is a prime, the algorithm returns true.

Proof: If n is prime then steps 1 and 3 can never return false, by Lemma 4.1, the `for' loop also cannot return false. Therefore the algorithm will identify n as prime either in step 4 or in step 6. The converse of the above lemma requires a little more work. If the algorithm returns true in step 4 then n must be prime since otherwise step 3 would have found a nontrivial factor of n. So the only remaining case is when the algorithm returns PRIME in step 6. For the purpose of subsequent analysis it is assumed that this to be the case.

The algorithm has two main steps (2 and 5): step 2 finds an appropriate `r' and step 5 verifies the equation (2) for a number of a's. We first bound the magnitude of the appropriate r.

Lemma 4.4: There exists an $r \leq \max(3, \lceil log^5 n \rceil)$ such that $o_r(n) > log_2 n$.

Proof: This is trivially true when n = 2; r = 3 satisfies all conditions. So assume that n > 2. Then $\lceil log^5 n \rceil > 10$ and Lemma 4.2 applies.

Let $r_1, r_2, \ldots, r_t$ be all numbers such that either $o_{ri}(n) \leq \log^2 n$ or r_i divides n. Each of these numbers must divide the product,

$$n \cdot \prod_{i=1}^{\lfloor log^2 n \rfloor} (n^i - 1) < n^{log^4 n} \leq 2^{log^5 n}$$

By Lemma 4.2, the LCM of the first $\lceil log^5 n \rceil$ numbers is at least $2^{\lceil log^5 n \rceil}$ and therefore there must exist a number $s \leq \lceil log^5 n \rceil$ such that $s \in \{r_1, r_2, \ldots, r_t\}$. If (s, n) = 1 then $o_s(n) > \log^2 n$ and we are done. If (s, n) > 1, then since s does not divide n and $(s, n) \in \{r_1, r_2, \ldots, r_t\}$, $r = s/(s,n) \in \{r_1, r_2, \ldots, r_t\}$ and so $o_r(n) > \log^2 n$.

Since $o_r(n) > 1$, there must exist a prime divisor p of n such that $o_r(p) > 1$. We have p > r since otherwise either step 3 or step 4 would decide about the primality of n. Since (n, r) = 1 (otherwise either step 3 or step 4 will correctly identify n), p, $n \in Z^*_r$. Numbers p and r will be fixed in the remainder of this section. Also, let $l = \lfloor \sqrt{\emptyset}(r) \log n \rfloor$.

Step 5 of the algorithm verifies $l = \lfloor \sqrt{\emptyset}\,(r) \log n \rfloor$ equations. Since the algorithm does not returns false in this step, we have:

$$(x + a)^n = x^n + a \pmod{x^r - 1, n}$$

For every $0 \leq a \leq l$ the equation is satisfied including 0. This implies:

(3) $$(x + a)^n = x^n + a \pmod{x^r - 1, p}$$

For every $0 \leq a \leq l$, by Lemma 4.1:

(4) $(x + a)^p = x^p + a \pmod{x^r - 1, p}$

From equation 3 and 4 it follows that,

(5) $(x + a)^{n/p} = x^{n/p} + a \pmod{x^r - 1, p}$ for every $o \leq a \leq l$,

Thus both n and n/p behaves as a prime in the above equation, and a name has been given to it.

Definition 4.1: For polynomial f(x) and number $m \in N$, we say that m is introspective for f(x) if, $[f(x)]m = f(X^m) \pmod{x^r - 1, p}$.

It is clear from equations (5) and (4) that both n/p and p are introspective for x + a when $\propto a \leq l$. The following lemma shows that introspective numbers are closed under multiplication.

Lemma 4.5: If m and m′ are introspective numbers for f(x) then so is m·m′.

Proof: Since m is introspective for f(x) we have:

$$[f(x)]^{m \cdot m'} = [f(x^m)]^{m'} \pmod{X^r - 1, p}$$

Also, since m′ is introspective for f(x), we have (after replacing x by x^m in the introspection equation for m′),

$[f(x^m)]^{m'} = f(x^{m \cdot m'}) \pmod{x^{m \cdot r} - 1, p}$

$= f(x^{m \cdot m'}) \pmod{x^r - 1, p}$ (since $x^r - 1$ divides $x^{m \cdot r} - 1$)

Putting together the above two equations we get,

$$[f(x)]^{m \cdot m'} = f(x^{m \cdot m'}) \pmod{x^r - 1, p}$$

For a number m, the set of polynomials for which m is introspective is also closed under multiplication.

Lemma 4.6: If m is introspective for f(x) and g(x) then it is also introspective for f(x) · g(x).

Proof: We have,

$$[f(x) \cdot g(x)]^m = [f(x)]^m \cdot [g(x)]^m$$
$$= f(x^m) \cdot g(x^m) \pmod{x^r - 1, p}$$

The above two lemmas together imply that every number in the set I = $\{\left(\frac{n}{p}\right)^i \cdot p^j \mid i, j \geq 0\}$ is introspective for every polynomial in the set $p = \{\prod_{a=0}^{l}(x+a)^{e_a} \mid e_a \geq 0\}$. Now there are two groups based on these sets that will play a crucial role in the proof. The first group is the set of all residues of numbers in I modulo r. This is a subgroup of Z_r^* since, as already observed, (n, r) = (p, r) = 1. Let G be this group and |G| = t. G is generated by n and p modulo r and since $o_r(n) > \log^2 n$, $t > \log^2 n$. To define the second group, we need some basic facts about cyclotomic polynomials over finite fields. Let $Q_r(x)$ be the r^{th} cyclotomic polynomial over F_p. Polynomial $Q_r(x)$ divides $x^r - 1$ and factors into irreducible factors of degree `$o_r(p)$'. Let `h(x)' be one such irreducible factor. Since $o_r(p) > 1$, the degree of h(x) is greater than one. The second group is the set of all residues of polynomials in P modulo h(x) and p. Let G be this group which is generated by elements x, x+1, x+2, . . . , $x + l$ in the field F = F_p[x]/(h(x)) and is a subgroup of the multiplicative group of F. The following lemma proves a lower bound on the size of the group G. It is a slight improvement on a bound shown by Hendrik Lenstra Jr., which, in turn, improved a bound shown in an earlier version of the paper.

Lemma 4.7: (Hendrik Lenstra Jr.) $|G| \geq (\frac{t+l}{t-1})$

Poof: First note that since h(x) is a factor of the cyclotomic polynomial $Q_r(X)$, X is a primitive r^{th} root of unity in F. Now it can be showed that

any two distinct polynomials of degree less than t in P will map to different elements in G. Let f(x) and g(x) be two such polynomials in P. Suppose f(x) = g(x) in the field F. Let m ∈ I. We also have [f(x)]m = [g(x)]m in F. Since m is introspective for both f and g, and h(x) divides $x^r - 1$, we get: $f(x^m) = g(x^m)$ in F. This implies that x^m is a root of the polynomial Q(y) = f(y)– g(y) for every $m \in G$. Since $(m, r) = 1$ (G is a subgroup of Z_r^*), each such x^m is a primitive r^{th} root of unity. Hence there will be |G| = t distinct roots of Q(y) in F. However, the degree of Q(y) is less than t by the choice of f and g. This is a contradiction and therefore, f(x) ≠ g(x) in F.

Note that i ≠ j in F_p for 1≤ i ≠ j ≤ l since $l = \lfloor \sqrt{\emptyset}(r) \log n \rfloor < \sqrt{r} \log n < r$ and p > r. So the elements x, x + 1, x + 2, . . . , x + l are all distinct in F. Also, since the degree of h is greater than one, x + a ≠ o in F for every a, o≤ a ≤ l. So, there exist at least $l + 1$ distinct polynomials of degree one in G. Therefore, there exist at least $(\frac{t+l}{t-1})$ distinct polynomials of degree < t in G. In case n is not a power of p, the size of G can also be in upper bound.

Lemma 4.8: If n is not a power of p then $|G| \leq n^{\sqrt{t}}$.

Proof: Consider the following subset of I:

$$\hat{I} = \left\{ \left(\frac{n}{p}\right)^i . p^j \mid 0 \leq i, j \leq \lfloor \sqrt{t} \rfloor \right\}$$

If n is not a power of p then the set Î has $(\lfloor \sqrt{t} \rfloor + 1)^2$ > t distinct numbers.

Since |G| = t, at least two numbers in Î must be equal modulo r. Let these be m1 and m2 with m1 > m2. So we have,

$$x^{m_1} = x^{m_2} \pmod{x^r - 1}$$

Let f(X) ∈ P. Then,

$$[f(x)]^{m_1} = f(x^{m_1}) \pmod{x^r - 1, p}$$
$$= f(x^{m_2}) \pmod{x^r - 1, p}$$
$$= [f(x)]^{m_2} \pmod{x^r - 1, p}.$$

This implies

$$[f(x)]^{m_1} = [f(x)]^{m_2}$$

in the field F. Therefore, f(x) ∈ G is a root of the polynomial Q′ (y) = $y^{m_1} - y^{m_2}$ in the field F. As f(x) is an arbitrary element of G, the polynomial Q′(y) has at least |G| distinct roots in F. The degree of Q′(y) is m1 ≤ $(\frac{n}{p} . p)^{\lfloor\sqrt{t}\rfloor} \leq n^{\sqrt{t}}$. This shows |G| ≤ $n^{\sqrt{t}}$.

Armed with these estimates on the size of G, Now the correctness of the algorithm can be proved.

Lemma 4.9: If the algorithm returns true then n is a prime.

Proof: Suppose that the algorithm returns true. Lemma 4.7 implies that for t = |G| and $l = \lfloor\sqrt{\emptyset_r} \log n\rfloor$,

|G| ≥ $(\frac{t+l}{t-1})$

≥ $(\frac{l+1+\lfloor\sqrt{t}\ log\, n\rfloor}{\lfloor\sqrt{t}\ log\, n\rfloor})$ (since t > $\lfloor\sqrt{\emptyset_r} \log n\rfloor$)

≥ $(\frac{2\lfloor\sqrt{t}\ log\, n\rfloor+1}{\lfloor\sqrt{t}\ log\, n\rfloor})$ (since $l = \lfloor\sqrt{\emptyset_r} \log n\rfloor \geq \lfloor\sqrt{\emptyset_r} \log n\rfloor$

> $2^{\lfloor\sqrt{t}\ log\, n\rfloor+1}$ (since $\lfloor\sqrt{\emptyset_r} \log n\rfloor > \lfloor log^2 n\rfloor > 1$)

≥ $n^{\sqrt{t}}$

By Lemma 4.8, |G|≤ $n^{\sqrt{t}}$ if n is not a power of p. Therefore, n = p^k for some k > 0. If k > 1 then the algorithm will return false in the first step. Therefore, n = p. This proofs the Lemma 4.9.

4.4 Analysis and Implementation of the Algorithm

The main objective of this paper is to engineer the algorithm in real life; so the algorithm has been coded and analyzed. To simplify the process of analysis the algorithm is divided into 3 major steps. After taking the input the first step is to test whether the number can be a power i.e. if the input number is n then if it can be written as n = a^b form

for some b > 1. If in the first step it is found that it can be written as n = a^b where b > 1 then it returns false from that point without any further analysis.

In the second step a sufficiently small value would be searched out for a particular value of n in such a way so $o_r(n) > \log^2 n$, where $o_r(n)$ is the Euler's totient function giving the number of numbers less then `n' that are relatively prime to `n'. And within this search process if the value of `r' is found to be greater than or equal to n then the number can be said to be prime without any further analysis, but actually for larger numbers there would be such sufficiently small value of `r', as discussed and proved earlier in this chapter.

Then in the third step which actually is the most time consuming step, the equation (2) is checked for $2 \leq a \leq \lfloor \sqrt{\emptyset_r} \log n \rfloor$ different values. If the equation holds for every value of `a' then it is certain that the input number is a prime, otherwise if any such value of `a' is encountered where the equation doesn't hold the procedure returns false immediately indicating that the given number is composite; as it has been proved earlier that there would be no exception.

In the implementation process there are certain difficulties in processing the steps in polynomial time, a complexity analysis has been given proving the complexity to be in polynomial time but the steps are not detailed in the algorithm as it is out of scope of the paper `PRIMES is in P'. But to implement the steps had to be resolved.

4.4.1 Resolving the first step

In the first step to verify whether the input is a power or not a simple method can be followed by checking whether $\left(\left\lfloor n^{\frac{1}{b}} \right\rfloor\right)^b = n$, within the

range $2 \leq b \leq \log n$; thus making it a polynomial time algorithm. The procedure is described in algorithm 4.2.

```
Procedure isPower1( int n )
{
       for b = 2 to log n
       do
              i = ⌊n^(1/b)⌋ ;
              if(power(i, b) == n) return false;
       od
       return true;
}
```

Figure 10: Algorithm 4.2

But the problem using this algorithm is in the data structure. Since if a number is not a power, which in the most cases is even if the number is not a prime, value of `i' would require having a high precision fractional number. But the available variable types are usually `float' and `double'; with which numbers greater then $2^{63} - 1$ can't be checked, and because of the procedure `modPower' in algorithm 3.4 requires to process a square so no number greater then $2^{31.5} - 1$ can't be checked. So to implement the code to check larger numbers a different data structure has to be followed.

To implement the code for numbers larger then $2^{31.5} - 1$ the `java.math.BigInteger' class has been used. But still the class type cannot be used perfectly with fractional numbers, i.e. numbers with decimal points in it. The class contains procedures for different operations, such as, addition, subtraction, multiplication, division, remainder etc. The complexities of the operations depend on the length of the input number or numbers. So the complexity is approximately always in order of log n.

If a number `n' is not equal to the n^{th} power of any integer then $n^{\frac{1}{m}}$ is always an irrational number; the proof of the theorem is out of

the scope of this paper so it is not described here. But since so, there would always be a difficulty of calculating the fractional numbers and approximations would have to be taken.

So a different approach has to be taken, by not calculating $n^{\frac{1}{m}}$ but my calculating a^m for different values of `a' and 2≤ m ≤ log n if any value is equal to n, then the number would be of form $n = a^m$, and would return true indicating that the number is a power. Otherwise it would return false. So the checking in step 2 has to be done without any fractions value. Since the search for the root number is done in binary search it takes the complexity in polynomial order making allowing us to search in polynomial time. The algorithm 4.3 uses this approach to solve the problem.

Just like algorithm 4.2, this one also takes the values of `a' within the range of 2 to log n and checks if those can be the power of any integer to form the value of `n'. This algorithm can check whether a given number can be written as $n = a^b$ having a worst case complexity of $\log^2 n$. And thus this should be able to check the root of any powered number.

Also since it uses the number of digits in the number, it can easily be implemented on `java.math.BigInteger'. In this particular data structure the each digit of the number is kept in different variables, so it's easier to access the individual number. And it enables the way to take larger inputs.

```
Procedure isPowerOf( int n, int i )
{
       length = ⌈(number of digits of n) / i⌉ ;
       low = power(10, length – 1);
       high = power(10, length) – 1;

       while( low <= high )
       do
              mid = (low + high) / 2;
              val = power(mid, i);

              if( val < n ) low = mid + 1;
              else if( val > n ) high = mid -1;
              else return true;
              /* indicating that n^(1/i) is integer. */
       od

       return false;
       /* indicating that n^(1/i) is not integer. */
}

Procedure isPower( int n )
{
       for a = 2 to log n
       do
              if( isPowerOf(n, a) ) return true;
       od

       return false;
}
```

Figure 11: Algorithm 4.3

4.4.2 Resolving the second step

In second step a sufficiently small `r' have to be chosen to resolve the third step using equation (2). Also the search for the value has to be done in polynomial time, which is possible. The algorithm tries to find a prime `r' such that r – 1 has a large prime factor `q' such that $q \geq 4\sqrt{r}\log n$. The authors have already proved that in the algorithm, there must be such value of `r' and they are even able to establish bounds on it. They then use these bounds to establish that if `n' is prime, the algorithm returns true. To resolve the second step, the following algorithm 4.3 can be followed simply.

```
Procedure ChooseVal(int n)
{
	r = 2;

	while( r < n )
	do
		if( gcd(r, n) != 1 ) return false;
		if( isSmallPrime(r) ) then
			q = largestFactor(r – 1);
			if(q ≥ 4*√r*log n and n^((r−1)/q) ≡ 1 mod r)
				break;
			r++;
		fi
	od

	return r;
}
```

Figure 12: Algorithm 4.4

To code the algorithm each time the `gcd' of r and n have to be calculated to check if it is equal to one, and if such a case can be found when it is not then it can simply return false from that point indicating that it has a divisor thus it is composite. Since the n is in type `java.math.BigInteger' so the modified `gcd' function is used to generate the value.

One of the problems to encounter is checking the candidate value of `r' for primality. But since the value would be comparatively small and wouldn't take very long; so the Sieve of Eratosthenes has been used as discussed in chapter 2, algorithm 2.3. The algorithm generates all the primes within a certain range and when checking the value of `r' for primality it would simply take the time to check whether the numbered index is true or false. Since the number is comparatively smaller so it is possible to generate all of the primes within a certain range, but if the value of `n' is so high that it requires checking a larger value of `r' then the range would have to be increased. The notable behavior of the algorithm is to generate all the primes within a certain

range requires $\sqrt{n}$ complexity, but it only happens on the execution of the program, then all the time to test each value of `r' requires only the checking of an index in the array.

After ensuring that the value of `r' is a prime, so `r – 1' would always be a composite number and if it is a odd prime then `r – 1' would be a even number; so at least one factor would be 2 and in most cases there would be a larger factor. And in order to find the factors it's necessary to find out all the previous prime numbers, which has been done using the Sieve of Eratosthenes algorithm. So now all the primes are known and the numbers that can be a factor to `r – 1' can be chosen. Since only the largest factor is needed so within the range the in descending order the first encounter of the prime that can divide `r – 1' is the largest prime factor. The following algorithm in 4.4 can be used to do that.

```
Procedure LargestFactor(int n)
{
       if(n == 1) return 1;

       while(i > 1)
       do
              while( !isPrime(i) ) i--;
              if(n % i == 0) return i;
              i--;
       od

       return n;
}
```

Figure 13: Algorithm 4.5

Since the primes have been marked within a range with Sieve of Eratosthenes the function call of `isPrime' takes only the time to check an array index if it is true or false. So the largest factor can be chosen within a very small amount of time. And the rest of the second step can be done using the standard library functions.

4.4.3 Resolving the third step:

In third step the algorithm checks whether the given n satisfies equation (2) for all the values of `a' and `r' within a polynomial ring. After the second step the proper value for `r' is chosen, so the number of numbers for which the equation has to be checked can be calculated now, which can be `2 * √r * log n'. So the algorithm would be as follows in 4.5.

Unfortunately this step takes the most time. The exponentiation and modulo exponent calculations has been done with algorithm 3.4, so it is done in polynomial time.

```
Procedure checkEquation( int n )
{
       for a = 1 to 2√r log n
       do
              if( !(x − a)^n ≡ x^n − a mod ( x^r − n ) )
                     return false;
       od

       return true;
}
```

Figure 14: Algorithm 4.6

The only problem rises to implement this is to calculate the value of `log n'; as it is needed throughout the process several times. The library function or the conventional way to calculate log only allows dealing with small numbers, but when the numbers are in the range of 20 to 100 digits and the data structure is changed, those functions can't be used anymore; so new ways have to be followed.

If the ten base logarithm is taken on a number that can be in form of 10^m then the result of would be m. Taking the base ten logarithm of any number would almost equal to the number of digits of the target number or in other words the exponential order of the target

number. So according the rules of logarithm it can be said that for a decimal number,

$$(d_1d_2d_3...d_m) = (o.d_1d_2d_3...d_m) \text{ X } 10^m$$

So taking log in both sides,

$$\log_{10} (d_1d_2d_3...d_m) = \log_{10}(o.d_1d_2d_3...d_m) + \log_{10}(10^m)$$
$$= \log_{10}(o.d_1d_2d_3...d_m) + m$$

So this calculation leads to the algorithm 4.6 which calculates the ten base log of any large number.

```
Procedure largeLog(int n)
{
       v = number of digits of the target number;
       d = v / 10^v;
       ld = log(d);
       return ld + v;
}
```

Figure 15: Algorithm 4.7

But in practice there's a accuracy limitation, when value of `d' is taken of a large number, because of the lack of data structure to keep high precision fractional number, not many numbers would be possible to be in calculation following the decimal point. But for even larger numbers that much accuracy is enough.

And thus all the three steps have been implemented. A class contains all the required functions and constructors, which is embedded into the user interface class to calculate the numbers from a GUI. And the algorithm is implemented as an event of the button, so the calculations should be completed as soon as the button is released.

5. CONCLUSION

The paper presents the various methods for implementation of the AKS algorithm and its various variants. New methods for polynomial implementation are analyzed. The algorithms and techniques defined in the paper can be very effective to reduce the computational time.

Many researchers have been interested in the primality-testing algorithm from a theoretical standpoint. Search for a polynomial time deterministic algorithm for primality testing has been an open problem for a long time. The importance of the algorithm needs little mention. Cryptography is one the fields where the algorithm finds practical use.

Also, it opens the opportunity to develop further algorithms on the same problem as well as problems like factorization and the ones that requires prime numbers. It is advancement in the cryptography as well as any other area of computational mathematics.

APPENDIX A

The Implemented code in Java for the Naïve Methods for Primality Testing is given here.

```
////////////////////////////////////////////////////////////////////////////////
import java.io.*;
import java.awt.*;
import java.math.*;
import javax.swing.*;
import java.awt.event.*;

/*************************************************************
*                                                            *
*      Implementation of the Naive Methods for Primality Testing  *
*                                                            *
*************************************************************/

class FermatTest
{
       private int k, primes[], N, l;
       private boolean comp[];

       boolean isPrime1(long n)
       {
              long a;
              double k;

              k = Math.sqrt(n);

              for(a = 2; a < k; a++)
              {
                     if(n % a == 0) return false;
              }

              return true;
       }

       public void Sieve()
       {
              int i, j;

              N = 100000;

              comp = new boolean[N+1];
```

```
        comp[1] = true;

        for(i = 2; i * i <= N; i++)
        {
                if(comp[i] != true)
                {
                        for(j = i * i; j <= N; j += i)
                        {
                                comp[j] = true;
                        }
                }
        }
}

public void genPrimes()
{
        int i;

        primes = new int[10000];
        l = 0;

        for(i = 1; i < N; i++)
        {
                if(comp[i] == false)
                {
                        primes[l] = i;
                        l++;
                }
        }
}

boolean isPrime2(long n)
{
        int k;

        Sieve();
        genPrimes();

        for(k = 0; primes[k] * primes[k] <= n; k++ )
        {
                if(n % primes[k] == 0) return false;
        }

        return true;
}

public boolean isPrime3(int n)
{
```

```
            if(comp[n] == false) return true;
            else return false;
      }
}


public class PrimeInterface extends JFrame
{
      private long n, p;
      private JTextField ti, tp, to;
      private JRadioButton f, r, s;
      private ButtonGroup bg;
      private JButton b;
      private Container c;
      private FermatTest fr;
      private int choice;

      public PrimeInterface()
      {
            super("ZFermat's Little Test");

            ti = new JTextField(20);
            to = new JTextField(20);

            f = new JRadioButton("Slow", true);
            r = new JRadioButton("Common", false);
            s = new JRadioButton("Sieves", false);
            bg = new ButtonGroup();

            bg.add(f);
            bg.add(r);
            bg.add(s);

            b = new JButton("Primality Test");

            fr = new FermatTest();

            c = getContentPane();
            c.setLayout(new FlowLayout());
            to.setEditable(false);

            f.addItemListener(new ItemListener()
            {
                  public void itemStateChanged(ItemEvent e)
                  {
                        choice = 0;
                  }
            });
```

```
		r.addItemListener(new ItemListener()
		{
			public void itemStateChanged(ItemEvent e)
			{
				choice = 1;
			}
		});

		s.addItemListener(new ItemListener()
		{
			public void itemStateChanged(ItemEvent e)
			{
				choice = 2;
			}
		});

		b.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent e)
			{
				n = Long.parseLong(ti.getText());

				if(choice == 0)
				{
					if( fr.isPrime1(n) ) to.setText("" + n + " is a
prime");
					else to.setText("" + n + " is not Prime");
				}
				else if(choice == 1)
				{
					if( fr.isPrime2(n) ) to.setText("" + n + " is a
prime");
					else to.setText("" + n + " is not Prime");
				}
				else if(choice == 2)
				{
					fr.Sieve();
					if( fr.isPrime3((int)n) ) to.setText("" + n + " is a
prime");
					else to.setText("" + n + " is not Prime");
				}
			}
		});

		c.add(ti);
		c.add(to);
		c.add(f);
```

```
            c.add(r);
            c.add(s);
            c.add(b);

            setSize(520, 90);

            setVisible(true);
      }

      public static void main(String args[])
      {
            PrimeInterface f = new PrimeInterface();

            f.addWindowListener(new WindowAdapter()
            {
                  public void windowClosing(WindowEvent ev)
                  {
                        System.exit(0);
                  }
            });
      }
}


/*************************************************************
*          This code has been developed as a course work of       *
*             Computer Science and Engineering Department         *
*                  The University of Asia Pacific                 *
*                                                                 *
*             Abdullah Al Zakir Hossain, aazhbd@yahoo.com         *
*                   The copy is redistributable                   *
*************************************************************/
```

APPENDIX B

The Implemented code in Java for the Fermat's Little Test is given here.

```
////////////////////////////////////////////////////////////////////////////////
import java.io.*;
import java.awt.*;
import java.math.*;
import javax.swing.*;
import java.awt.event.*;

/*************************************************************************
*                                                                        *
*      Implementation of the Fermat's Little Test for Primality Testing  *
*                                                                        *
*************************************************************************/

class FermatTest
{
	long modPower(long x, long y, long n)
	{
		long m, p, z;

		m = y;
		p = 1;
		z = x;

		while(m > 0)
		{
			while(m % 2 == 0)
			{
				m = (long) m / 2;

				z = (z * z) % n;
			}
			m = m - 1;

			p = (p * z) % n;
		}

		return p;
	}

	boolean isPrime(long n)
	{
		long a, k;
```

```
		for(a = 2; a < n; a++)
		{
			k = modPower(a, n, n);

			if(k != a) return false;
		}

		return true;
	}

	boolean isPrime(long n, long p)
	{
		long i, a, k;

		for(i = 2; i <= p; i++)
		{
			a = (long) Math.floor(Math.random() * n);

			k = modPower(a, n, n);

			if(k != a) return false;
		}

		return true;
	}
}

public class PrimeInterface extends JFrame
{
	private long n, p;
	private JTextField ti, tp, to;
	private JRadioButton f, r;
	private ButtonGroup bg;
	private JButton b;
	private Container c;
	private FermatTest fr;
	private boolean rand;

	public PrimeInterface()
	{
		super("ZFermat's Little Test");

		ti = new JTextField(20);
		tp = new JTextField("10", 10);
		to = new JTextField(20);

		f = new JRadioButton("Non-Randomized", true);
```

```
		r = new JRadioButton("Randomized", false);
		bg = new ButtonGroup();

		bg.add(f);
		bg.add(r);

		b = new JButton("Primality Test");

		fr = new FermatTest();

		c = getContentPane();
		c.setLayout(new FlowLayout());
		to.setEditable(false);

		f.addItemListener(new ItemListener(){
			public void itemStateChanged(ItemEvent e)
			{
				rand = false;
			}
		});

		r.addItemListener(new ItemListener(){
			public void itemStateChanged(ItemEvent e)
			{
				rand = true;
			}
		});

		b.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent e)
			{
				n = Long.parseLong(ti.getText());
				p = Long.parseLong(tp.getText());

				if(rand)
				{
					if( fr.isPrime(n, p) ) to.setText("" + n + " is
probably a prime");
					else to.setText("" + n + " is not Prime");
				}
				else
				{
					if( fr.isPrime(n) ) to.setText("" + n + " is
probably a prime");
					else to.setText("" + n + " is not Prime");
				}
			}
```

```
            });

            c.add(ti);
            c.add(to);
            c.add(tp);
            c.add(f);
            c.add(r);
            c.add(b);

            setSize(520, 90);

            setVisible(true);
      }

      public static void main(String args[])
      {
            PrimeInterface f = new PrimeInterface();

            f.addWindowListener(new WindowAdapter()
            {
                  public void windowClosing(WindowEvent ev)
                  {
                        System.exit(0);
                  }
            });
      }
}

/*************************************************************
*         This code has been developed as a course work of       *
*            Computer Science and Engineering Department         *
*                 The University of Asia Pacific                 *
*                                                                *
*            Abdullah Al Zakir Hossain, aazhbd@yahoo.com         *
*                 The copy is redistributable                    *
*************************************************************/
```

APPENDIX C

The implemented code in Java for the AKS algorithm is given here.

```
////////////////////////////////////////////////////////////////////////////////
import java.awt.*;
import java.math.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

/**********************************************************************
*                                                                     *
*      Implementation of the AKS Algorithm for Primality Testing      *
*                                                                     *
**********************************************************************/


public class BigPolyPrimeInt extends JFrame
{
       private int k, count, facts[], primes[], N;
       private int l;
       private boolean comp[];
       private JTextField tf, lt;
       private JLabel lb;
       private JButton bt;
       private Container c;

       public boolean isSmPrime(int n)
       {
              if(comp[n] == false) return true;
              else return false;
       }

       public int largestFact(int n)
       {
              int i;

              i = n;

              if(i == 1) return i;

              while(i > 1)
              {
                     while( comp[i] == true ) i--;
                     if(n % i == 0) return i;
```

```
            i--;
        }

        return n;
}

public double bigLog(String s)
{
        String t;
        int l;
        double d, r;

        l = s.length();
        t = "." + s;
        d = Double.parseDouble(t);
        r = Math.log10(d) + l;

        return r;
}

public double bigLog(BigInteger s)
{
        String t;
        int l;
        double d, r;

        t = "." + s.toString();
        l = t.length() - 1;
        d = Double.parseDouble(t);
        r = Math.log10(d) + l;

        return r;
}

public boolean isPowerOf(BigInteger n, int i)
{
        int l;
        double len;
        BigInteger low, high, mid, res;

        low = new BigInteger("10");
        high = new BigInteger("10");

        len = (n.toString().length()) / i;
        l = (int) Math.ceil(len);

        low = low.pow(l - 1);
        high = high.pow(l).subtract(BigInteger.ONE);
```

```
	while(low.compareTo(high) <= 0)
	{
		mid = low.add(high);

		mid = mid.divide(new BigInteger("2"));

		res = mid.pow(i);

		if(res.compareTo(n) < 0)
		{
			low = mid.add(BigInteger.ONE);
		}
		else if(res.compareTo(n) > 0)
		{
			high = mid.subtract(BigInteger.ONE);
		}
		else if(res.compareTo(n) == 0)
		{
			System.out.println("res = " + res + " mid = " + mid);
			return true;
		}
	}

	return false;
}

boolean isPower(BigInteger n)
{
	int l, i;

	l = (int) bigLog(n);

	for(i = 2; i < l; i++)
	{
		if(isPowerOf(n, i))
		{
			return true;
		}
	}

	return false;
}

long mPower(long x, long y, long n)
{
	long m, p, z;
```

```
	m = y;
	p = 1;
	z = x;

	while(m > 0)
	{
		while(m % 2 == 0)
		{
			m = (long) m / 2;

			z = (z * z) % n;
		}
		m = m - 1;

		p = (p * z) % n;
	}

	return p;
}

BigInteger mPower(BigInteger x, BigInteger y, BigInteger n)
{
	BigInteger m, p, z, two;

	m = y;
	p = BigInteger.ONE;
	z = x;
	two = new BigInteger("2");

	while(m.compareTo(BigInteger.ZERO) > 0)
	{
		while( ( (m.mod(two)).compareTo(BigInteger.ZERO) ) == 0)
		{
			m = m.divide(two);

			z = (z.multiply(z)).mod(n);
		}

		m = m.subtract(BigInteger.ONE);

		p = (p.multiply(z)).mod(n);
	}

	return p;
}

public void Sieve()
{
```

```
        int i, j;

        N = 1000000;

        comp = new boolean[N+1];

        comp[1] = true;

        for(i = 2; i * i <= N; i++)
        {
                if(comp[i] != true)
                {
                        for(j = i * i; j <= N; j += i)
                        {
                                comp[j] = true;
                        }
                }
        }
}

public boolean isPrime(BigInteger n)
{
        int tr, q, tm, ai, up, o;
        BigInteger r, t, x, lh, rh, fm, yai;

        l = (int) bigLog(n);

        if( isPower(n) ) return false;

        r = new BigInteger("2");
        tr = r.intValue();

        while( r.compareTo(n) < 0 )
        {
                if( (r.gcd(n)).compareTo(BigInteger.ONE) != 0 ) return false;

                tr = r.intValue();

                if( isSmPrime(tr) )
                {
                        q = largestFact(tr - 1);

                        o = (int) (tr - 1) / q;

                        tm = (int) (4 * (Math.sqrt(tr)) * l);

                        t = mPower(n, new BigInteger("" + o), r);
```

```
                if( q >= tm && (t.compareTo(BigInteger.ONE)) != 0 ) break;
            }

            r = r.add(BigInteger.ONE);
        }

        x = new BigInteger("2");

        fm = (mPower(x, r, n)).subtract(BigInteger.ONE);

        up = (int) (2 * Math.sqrt(tr) * l);

        for(ai = 1; ai < up; ai++)
        {
            yai = new BigInteger("" + ai);
            lh = (mPower(x.subtract(yai), n, n)).mod(n);
            rh = (mPower(x, n, n).subtract(yai)).mod(n);

            if(lh.compareTo(rh) != 0) return false;
        }

        return true;
}

public BigPolyPrimeInt()
{
        super("Polynomial Time Primality Testing, Zakir");

        Sieve();

        lb = new JLabel("Enter the Number");

        tf = new JTextField(30);

        lt = new JTextField(50);

        bt = new JButton("Test Primality");

        c = getContentPane();

        c.setLayout(new FlowLayout());

        lt.setEditable(false);

        bt.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e)
            {
                BigInteger n = new BigInteger(tf.getText());
```

```
                        if( isPrime(n) ) lt.setText("The Number " + n + " is a
prime");
                        else lt.setText("The Number " + n + " is NOT Prime");
                }
            });

            c.add(lb);
            c.add(tf);
            c.add(bt);
            c.add(lt);

            setSize(600, 100);
            setVisible(true);
      }

      public static void main(String args[])
      {
            BigPolyPrimeInt bp = new BigPolyPrimeInt();

            bp.addWindowListener(new WindowAdapter()
            {
                  public void windowClosing(WindowEvent ev)
                  {
                        System.exit(0);
                  }
            });
      }
}

/*************************************************************
*         This code has been developed as a course work of       *
*            Computer Science and Engineering Department         *
*                 The University of Asia Pacific                 *
*                                                                *
*            Abdullah Al Zakir Hossain, aazhbd@yahoo.com         *
*                 The copy is redistributable                    *
*************************************************************/
```

BIBLIOGRAPHICAL NOTES

[1] Agrawal, M., Kayal, N. and Saxena, N.: Primes is in P, Annals of Math.

[2] Adleman, L., Pomerance C. and Rumely, R.: On distinguishing prime numbers from composite numbers, Annals of Math. 117 (1983) 173–206.

[3] Adleman, L. and Huang, M.-D.: Primality Testing And Two Dimensional Abelian Varieties Over Finite Fields, Lecture Notes In Mathematics 1512, Springer Verlag 1992.

[4] A. Klappenecker. An introduction to the AKS primality test. Lecture notes, September 2002.

[5] Alexandra Carvalho. "AKS Primality Algorithm".

[6] Andreas Klappenecker, "The AKS Primality Test Results from Analytic Number Theory".

[7] Miller, G.L., "Riemanns Hypothesis and Tests for Primality", Journal of Computer Systems Science, Vol. 13, No. 3, December 1976.

[8] Hardy, G. H., Wright, E. M., "An Introduction to the Theory of Numbers", Fifth Edition, 1979.

[9] Telang, S. G., "Number Theory", Edited by Nadkarni, M. G., Dani, J. S. 1999.