Implementation of the Polynomial Time Deterministic Primality Testing Algorithm

Abdullah Al Zakir Hossain*,
*Master Research in Computer and Systems Engineering, Technische Universitt Ilmenau
Student ID: 47888,
Email: aazhbd@yahoo.com

***Abstract*—Prime numbers are matter of particular interest in cryptography as well as ensuring security in networks and in computer systems. In many other problems choosing very large prime numbers is important. While prime numbers are easily defined, discovering a polynomial time deterministic algorithm for it required thousands of years of research. But status of primality testing was unresolved till 2002 when Manindra Agrawal, Nitin Saxena and Neeraj Kayal[1] found an elegant algorithm that produces correct answer with polynomial runtime in number of digits of the target number. As a part of our research along with the slower algorithms a faster solution like the Fermats test was analyzed and implemented. But an algorithm based on Fermats Little theorem is probabilistic, although quite efficient in practice. In this research the main emphasis has been given on the work of Agrawal et al.[1], and the algorithm has been implemented for large numbers with hundreds of digits.**

***Index Terms*—primality testing, prime numbers, algorithms, aks test, cyclotomic aks test**

I. Introduction

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 years 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. The authors received the 2006 Gdel Prize for this work. The AKS Algorithm is the first deterministic polynomial time primality test. A primality testing algorithm is the algorithm that checks whether a given number is prime number or not. Deterministic means that operations carried out by the algorithm are determined entirely by the algorithm and the input. In particular the algorithm does not make any random choices. Polynomial time means that there is some polynomial p such that, for every input n, the algorithm takes at most p (n) steps. This algorithm is presented in the paper "PRIMES is in P"[1] and since then many people had focused on this and put many efforts on its implementation. We have implemented the AKS algorithm and have measured it's running results.

A. PRIMES is in P, The AKS Algorithm

It is a deterministic algorithm that ensures the result to be a prime without any doubt. Though it is much slower than other probabilistic or randomized tests but the accuracy is proven. It runs in polynomial time, but still it is 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 testing 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, $O(log^{15/2}n)$ time algorithm[4] 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 $2^p + 1$ is also prime), the algorithm takes only $O(log^6 n)$ steps.

B. Overview

The algorithm is based on a generalization of Fermats 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 $O(log^{21/2}n)$ for the algorithm. In contrast, the correctness proofs of earlier algorithms producing a certificate for primality are much more complex.

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, the algorithm is stated and then its proof of correctness is presented.

C. Idea

The test is based on the following identity for prime numbers which is a generalization of Fermats Little Theorem as discussed in Chapter 3. Let $a \in Z$, $n \in N$, $n \geq 2$, and (a, n) = 1. Then n is prime if and only if,

$$(x + a)^n = x^n + a(mod n) \tag{1}$$

According to the proof of Fermats Theorem[5][6] the same argument can be derived that 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 (n q) and is co-prime 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; given an input n, choose an 'a and test whether the congruence (1) is satisfied.

However, this takes time $\Omega(n)$ 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,

$$(x+a)^n = x^n + a(mod x^r - 1, n) \quad (2)$$

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.

II. 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 section. The proof of correctness is described according to this form of algorithm.

```
/* The Agrawal-Kayal-Saxena Algorithm */

procedure isPrimeAKS( int n )
{

if(n = ab 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 ⌊√(Ø_r) log n⌋ do
if((X + a)^n = X^n
+a(mod X^r − 1, n))
return false;
od
return true;

}
```

Algorithm - 01

III. 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 Eulers 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 section.

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 doesnt hold the procedure, it 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 algorithm these steps had to be resolved.

A. 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 the following algorithm.

```
/* first method to resolve first step */

procedure isPower1( int n )
{

for b = 2 to log n
do
i = ⌊n^(1/b)⌋ ;
if( i^b == n ) return false;
od
return true;

}
```

Algorithm - 02

But one the problem of using this algorithm is in using the data structure. Since, a number is not a power, which in the most cases is even and 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 than $2^{63} - 1$ cant 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 than $2^{31.5} - 1$ 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, there would always be a difficulty of calculating the fractional numbers,and there would be approximations; so a different approach has to be taken, by

not calculating $n^{\frac{1}{m}}$ but by calculating a^m for different values of 'a' ($2 \leq m \leq logn$)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 allowing us to search in polynomial time. The algorithm 03 uses this approach to solve the problem.

Just like algorithm 02, 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 each digit of the number is kept in different variables, so its easier to access the individual number. And it enables the way to take larger inputs.

```
/* Second Method to resolve first step */

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 not
 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;

}
```

Algorithm – 03

B. 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[2][3] 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 can be followed.

```
/* Resolving the second step, choosing
the value of 'r' */

procedure ChooseVal(int n)
{

r = 2; while( r < n )

{

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

}

return r;

}
```

Algorithm 04

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 is to check for primality on the candidate values of 'r'. But since the value would be comparatively small and wouldnt take very long; so the Sieve of Eratosthenes has been used. 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 if 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, 'r - 1' would always be a composite number. And since the prime is odd number so 'r - 1' would be an even number; so at least one factor would be 2 and in most cases there would be a larger factor. r. And in order to find the factors its 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. 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.

```
/* Determining the largest factor */
procedure LargestFactor(int n)
{
if(n == 1) return 1;
i=n;
while(i > 1)
do
while( !isPrime(i) ) i--;
if(n % i == 0) return i;
i--;
od
return n;
}
```

Algorithm 05

And the rest of the second step can be done using the standard library functions and the popular polynomial time algorithm to calculate fast exponents and modular exponents.

C. Resolving the third step

In third step the algorithm checks whether the given n satisfies the equation (2) for all the values of 'a within 'r'[4]. 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 '$2\sqrt{r}logn$'. So the algorithm would be as follows.

```
/* Resolving the third step */
procedure checkEquation( int n )
{
for a = 1 to 2√r logn
do
if(!(x − a)^n ≡ (x^n − a)mod(x^r − n)) return false;
od
return true;
}
```

Algorithm - 06

Unfortunately this step takes the most time. The exponentiation and modulo exponent calculations has been done in polynomial time. 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 cant 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,
$(d_1d_2d_3d_m) = (0.d_1d_2d_3d_m) * 10^m$
$or, log_{10}(d_1d_2d_3d_m) = log_{10}(0.d_1d_2d_3d_m) + log_{10}(10^m)$
$or, log_{10}(d_1d_2d_3d_m) = log_{10}(0.d_1d_2d_3d_m) + m$

So this calculation leads to the algorithm 7 which calculates the ten base log of any large number.

```
/* Calculating log of large number */
procedure largeLog(int n)
{
v = number of digits of the target number;
d = v / 10v;
ld = log(d);
return ld + v;
}
```

Algorithm - 07

But in practice theres 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.

IV. Conclusion

In this way all the three steps have been implemented in polynomial time. 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. A comprehensive report over the full research is published online[7] where the running times can be observed. It was the objectives of this dissertation to implement the AKS Algorithm and analysis its running results. The system we have developed meets the essential demands of the requirements specification defined in the algorithm. Along with the initial objectives, we had also looked into some other algorithms related to the AKS Algorithm, and we had made some developments on them as well.

Acknowledgement

The statement of the idea described here is a part of the research done as a thesis work for bachelors degree. 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. We also thank Aloke Kumar Saha, Head of the Department, Department of Computer Science and Engineering, UAP; for his support and guidance.

References

[1] Agrawal, M., Kayal, N. and Saxena, N.:*Primes is in P*, *Annals of Math*.
[2] A. Klappenecker. *An introduction to the AKS primality test*. Lecture notes, September 2002.
[3] Alexandra Carvalho. *AKS Primality Algorithm*.
[4] Andreas Klappenecker, *The AKS Primality Test Results from Analytic Number Theory*.
[5] Hardy, G. H., Wright, E. M., *An Introduction to the Theory of Numbers*, Fifth Edition, 1979.
[6] Telang, S. G., *Number Theory*, Edited by Nadkarni, M. G., Dani, J. S. 1999
[7] Hossain, Abdullah Al Zakir. *Implementation of AKS Algorithm*. (2008) Articulatelogic.com. [Online]. Available:
http://articulatelogic.com/download/Implementation_of_AKS_Algorithm/