An Empirical Study towards Refining the AKS Primality Testing Algorithm

Lalitha Kiran Nemana, V. Ch. Venkaiah

School of Computer and Information Sciences
University of Hyderabad
Hyderabad-500046, INDIA
email: nemana.lalithakiran@gmail.com, venkaiah@hotmail.com

Abstract

The AKS (Agrawal-Kayal-Saxena) algorithm is the first ever deterministic polynomial-time primality-proving algorithm whose asymptotic run time complexity is $O(\log^{12+\epsilon} n)$, where $\epsilon > 0$. Despite this theoretical breakthrough, the algorithm serves no practical use in conventional cryptologic applications, as the existing probabilistic primality tests like ECPP in conjunction with conditional usage of sub-exponential time deterministic tests are found to have better practical running time. Later, the authors of AKS test improved the algorithm so that it runs in $O(\log^{10.5+\epsilon} n)$ time. A variant of AKS test was demonstrated by Carl Pomerance and H. W. Lenstra, which runs in almost half the number of operations required in AKS. This algorithm also suffers impracticality. Attempts were made to efficiently implement AKS algorithm, but in contrast with the slightest improvements in performance which target specific machine architectures, the limitations of the algorithm are found highlighted. In this paper we present our analysis and observations on AKS algorithm based on the empirical results and statistics of certain parameters which control the asymptotic running time of the algorithm. From this analysis we refine AKS so that it runs in $O(\log^{4+\epsilon} n)$ time.

1 Introduction

In the later part of 20th century, the primality testing problem 'PRIMES' had its long stay in the complexity classes NP-Hard, NP and ZPP. Numerous probabilistic primality tests were developed [4][5][8] and probabilistic tests based on ECPP techniques are being used even now in conventional Cryptologic applications [7][15]. In 2002 Agrawal, Kayal and Saxena proved that 'PRIMES is in P' by developing a deterministic primality test whose running time is a polynomial in the number of bits required to represent the input number n [1]. The authors essentially gave a polynomial upper bound for the number of a values, which act as witnesses for the polynomial congruence $(X+a)^n \equiv x^n+a \; mod \; (X^r-1, n)$ which is true iff n is a prime power. However, the asymptotic time complexity of the algorithm is $O\tilde{}(\log^{12} n)$ which is extremely larger than the present Elliptic Curve Primality Proving techniques[15]. In 2004, the authors reduced the time complexity to $O\tilde{}(\log^{10.5} n)$ [9] and in 2005 Lenstra [24] proposed a variant of AKS which runs in $O\tilde{}(\log^7 n)$ time. However, both of these improvements are only of theoretical interest. Even though AKS happened to be a breakthrough, it suffers impracticality due to the expensive computations involved in each step and its storage requirements[17]. This motivated us to study the second version of AKS algorithm and observe the issues in its working so as to extract the information which would help make the algorithm practical. For implementation, we used the PARI C library [18] which supports large integer arithmetic.

Section 2 presents the AKS algorithm version 2 and its time complexity. Section 3 provides a brief account of few existing implementations and the observations made by their authors. Section 4 briefly presents the details of our implementation. In Section 5 we discuss several experiments we conducted on and the results thus obtained. A variant of AKS is also presented based on our observations. We conclude with Section 6 with a brief mentioning of our future work.

2 Algorithm & its Complexity

Section 2.1 presents the second version of AKS proposed in 2004 and section 2.2 describes the time complexity. Henceforth, we use the the term 'AKS' as a substitute for 'AKS version 2' as well as the paper containing it [9].

2.1 The Algorithm

```
Algorithm 1: AKS_V2
  Data: An Integer $n \in Z$
  Result: A string "Prime" or "Composite"
1 begin
2     Step 1: If $n$ is of form $a^b$ then output "Composite";
3     Step 2: Choose smallest $r$ such that $o_r(n) \geq \lfloor \log^2 n \rfloor$;
4     Step 3: If $1 < GCD(a, n) < n$ for some $a < r$ then output "Composite";
5     Step 4: If $n \leq r$ output "Prime";
6     Step 5: for $a \leftarrow 1$ to $\lfloor \sqrt{\phi(r)} \log n \rfloor$ do
7         if $(X + a)^n \not\equiv x^n + a \ mod \ (X^r - 1, n)$ then
8             output "Composite";
9     Step 6: output "Prime";
10 end
```

Step 1 of the algorithm verifies whether $n = a^b$ for some $a, b > 2$. If it so happens then it returns "Composite". Step 2 chooses an r such that 'Order of n with respect to *mod* r', i.e., $o_r(n)$ is greater than $\lfloor \log^2 n \rfloor$. $o_r(n)$ is the least k such that $n^k \equiv 1(mod\ r)$.

AKS proves that there exists such an r in the range, $[3, \log^5 n]$ and also that step 4 returns "Prime" in the case only when $n \leq 5690034$. Step 5 verifies whether the congruence $(X+a)^n \equiv x^n + a \; mod \; (X^r - 1, n)$ is true for different $a \in [1, \lfloor \sqrt{\phi(r)} \log n \rfloor]$. Here, $\phi()$ is the Euler's Totient Function. If any candidate a is found violating this criteria, n is reported as "Composite". Step 6 returns "Prime" if $\forall a \in [1, \lfloor \sqrt{\phi(r)} \log n \rfloor]$ the above criteria holds true. ' $mod \; ((X^r - 1, n)$' in the above congruence means that modulo is taken with respect to the polynomial $X^r - 1$ and the computation of coefficients is done in the Ring, Z_n.

2.2 Asymptotic Running Time

In the calculations done below, division, multiplication and addition operations between two l bit integers take $O\tilde{}(l)$ time and the same operations on k degree polynomials with coefficients of at most l bits take $O\tilde{}(kl)$ time[10]. Also, $O\tilde{}(f(n))$ represents, $O(f(n) \cdot polynomial(\log(f(n))))$. Therefore, what essentially this notation hides is a term $\log^\epsilon n$ of the asymptotic complexity, where $\epsilon > 0$ and all other terms which are less than $\log^\epsilon n$. Even though, this notation introduced by AKS simplifies the representation, it hides a significant overhead incurred by the lower order terms. Step 1 can be implemented using at most $O\tilde{}(\log^3 n)$ operations [10][25].In step 2, we need to make $\log^5 n$ iterations for r and for succesive values of r, we should check whether $n^k \neq 1(mod \; r)$, for all $k \leq \lceil \log^2 n \rceil$. Each r needs $O(\log^2 n)$ multiplications modulo r and this takes, $O\tilde{}(\log^2 n \log r)$. Therefore, the total complexity of this step will be $O\tilde{}(\log^7 n)$. Step 3 computes GCD of r integers. Calculation of GCD takes $O(\log n)$ time. Therefore, the complexity of step 3 would be $O(\log^6 n)$. Step 4 takes $O(\log n)$ number of bit comparisons. The $for \; loop$ of step 5 iterates for at most $O(\log^{\frac{7}{2}} n)$ times to check the polynomial congruence. Each congruence requires $O(\log n)$ number of multiplications of degree r polynomials, where each coefficient is of size, $O(\log n)$. Therefore, each congruence requires $O\tilde{}(r \log^2 n)$ time for verification. So, the whole of step 5 takes $O\tilde{}(\log^{\frac{21}{2}} n)$ operations. Hence, the total asymptotic time required by the algorithm is $O\tilde{}(\log^{\frac{21}{2}} n) = O\tilde{}(\log^{10.5} n)$.

3 Related Work

Soon after the release of AKS, few implementations were made to practically verify its correctness, as well as to check its effectiveness in conventional cryptologic applications. These works also tried to analyse the algorithm and improved the performance by a little factor. Rotella[6] is the first ever published implementation of AKS Version 1[1]. The implementation is in C++, using GMP[27]. Apart from verification of its correctness, the authors also focussed on improving the performance of Step 5, by replacing the modular divisions with additions. Even though no change was reflected in asymptotic complexity of the algorithm, this modification however increased the speed of execution slightly. The authors also collected the statistics of the density of polynomials being checked for those n entering step 5. However, these statistics are collected over a small range. The modification suggested for modular division is incorporated in the latter implementations. RG

Salembier[11] implemented AKS Version 3 using NTL[26] and observed a little improvement in performance by using binary GCD which is provided by NTL. The improvement in performance can be attributed to NTL, because numerous research papers have been published on the optimization of C++ code targeting a specific machine architecture and the NTL being a C++ library, obviously has potential to achieve this performance gain in speed. Even though authors did not run the implementation on very large numbers, they provided an estimate on the number of years it would take to get the results as n grows larger and larger. They also did an implementation in LIDIA [28] and compared the performance of the two Implementations. An MPI Implementation of AKS Algorithm is done by [12]. They used method of fast polynomial exponentiation with pre-computation and parallelized AKS in order to improve its running time. Tong Jin[13] also implemented AKS using NTL, but rather than just targeting at improving the performance, it also collected few useful statistics on the number of integers entering each step of the algorithm.

It can be noticed that no sufficient statistics representing the asymptotic runtime behaviour of the algorithm are collected. Granville [3] described some interesting ideas of simulating polynomial computations in Z_n, which would reduce the complexity of the algorithm. It also presented some suggestions to convert AKS into a probabilistic algorithm. How much ever the performance gain might be, the inherent problem in the algorithm has not been solved. The storage of polynomial coefficients remains an unsolved problem. It should be noted that the coefficients belong to Z_n. Zhengjun Cao[17] describes the storage requirements for the AKS test and remarks that $1,000,000,000$ GB of storage is required for an input size of 1024 bits. Even though the current operating systems use several techniques to deal with data which is larger to fit in main memory, this remark is suggestive to be considered. Unless an alternative is devised to rule out the expensive verification of the polynomial congruence in step 5, the algorithm could never be brought into practice. In this paper we focus on gaining useful statistics which would help modify the algorithm such that a lesser asymptotic runtime is achieved. We tried to focus mainly on the parameter r which is significantly responsible for the running time of the algorithm and also to find some means to replace step 5 with some equivalent logic.

4 Implementation of AKS

Section 4.1 details the methods used in realizing each step of the AKS. Section 4.2 discusses some of the issues we faced during implementation and on how we fixed them. Section 4.1 can be skipped by the reader as we made direct use of the PARI library functions.

4.1 Implementataion

Step 1 verifies whether n is a perfect power(pure power) i.e., to check if n is of the form a^b for some $a, b \in [2, \infty)$. Joachim [10] provides a $O(\log^3 n \log\log\log n) = O\tilde{}(\log^3 n)$ time algorithm for perfect power detection.The logic essentially involves computing an integer approximation $x = n^{\frac{1}{p}}, \forall p : p \leq \log n$. This is accomplished by making a binary search for x in the range, $[2, 2^{\lfloor \frac{\log n}{p} \rfloor + 1}]$ such that $x^p = n$. Bernstein [25] presents an approach which takes essentially linear asymptotic time $O(\log^{1+\epsilon} n)$, where ϵ is a very small quantity which ensures that the running time tends towards $O(\log n)$. We used *ispurepower*() function of PARI which implements it. For implementing step 2, AKS suggested a method which takes $O\tilde{}(\log^7 n)$ operations. However, we used the library function *Fp_order*() of PARI which implements efficient algorithm for order calculaion [14] and incorporating this will not increase the asymptotic running time. Step 3 of AKS performs GCD operation $O(\log^5 n)$ times. This is also one bottleneck for the algorithm, because if GCD is implemented using traditional techniques like Eucledian GCD or Binary GCD which take quadratic time in terms of the number of input bits, it would result in $O(\log^7 n)$ time for step 3. For implementations involving arbitrary precision arithmetic, PARI provides efficient GCD routines which harness the method of Binary GCD along with fast integer multiplication algorithms[10], to make GCD computation almost linear. Therefore, we used the *ggcd*() function of PARI library to achieve running time near to $O(\log^6 n)$. Step 4 of the algorithm takes $O(\log n)$ and is straightforward. Checking the validity of the polynomial congruence in this step adds the major difficulty to the implementation of AKS algorithm. PARI provides library function *mkpoly*() to represent polynomials and *FpX_add*(), *FpX_sub*(), *FpX_mul*(), *FpX_mod*() for basic polynimial arithmetic. We made direct use of such functions. The problem with the direct use of PARI functions is nevertheless brought into picture in the case of polynomial powering. As we shall discuss in the section 4.2, functions in PARI do not perform garbage collection in order to achieve better running time. Even though *FpX_pow*() of PARI uses repeated squaring technique, it is found to immediately crash on larger inputs. Therefore, we had to do our own implementation of polynomial powering using repeated squaring technique[14]. **Computation of** $\phi(r)$ is done by first obtaining the prime factorization of r and then using the following formula.

$$\phi(r) = r \prod_{p|r} \left(1 - \frac{1}{p}\right)$$

We used $Z\_factor()$ of PARI library to obtain prime factorization of r as the order of r is comparitively very smaller than n. This implementataion does not perform best in running time, because the garbage collection techniques implemented along with the code incurs an overhead. However, the implementation is efficient in the sense that it manages memory by systematic garbage collection and hence ensuring that the program can be repeatedly run without crashing, for a reasonably long time.

4.2 Issues in implementation:

The main issue with the direct usage of PARI functions is that they do not collect garbage from the PARI stack so as to gain speedup in running time. If garbage collection is not done explicitly, the PARI stack overflows and the program abruptly terminates. Reclaiming memory from the unused variables in this way is not at all straightforward in PARI and unless it is done carefully, live variables get erased and results of operations tend to go wrong. So as to understand the mechanism of garbage collection, let us briefly discuss PARI stack management. All computations in PARI are done over the stack allocated at the begining of the application. This stack acts as a scratchboard and also as a repository for the PARI objects during runtime. The function, *void pari_init*(*size_t size*, *ulong maxprime*) initializes the PARI/GP system by allocating a stack of required size. The amount of memory allocated to the stack depends on the available main memory of the computing system. While some systems allow allocation of the total available main memory, some systems allow only half of the main memory to be utilised by stack. Figure 1 shows PARI stack organization. The stack starts from the address "bot" and ends with "top". However, computations start from "top" and proceed towards "bot". The stack contains a "current stack pointer" called "avma", which represents the next available memory address for computation. As shown in the figure, there use to be a lot of addresses corresponding to the dead variables which will not be useful in the latter part of the program. It is required to reclaim this unused memory and make it available for the remainder of the program execution. This garbage collection can be accomplished using *GEN gerepile*(*pari_sp ltop*, *pari_sp lbot*, *GENq*). Fig 1 shows the positions of these pointers before and after garbage collection. In the above funcion, *ltop* and *lbot* are the memory addresses between which garbage is situated. *gerpile*() clears all the garbage between these two pointers and sets the object q at the new *lbot* and returns q. If, there are some live variables scattered across the garbage, then either *gerpile*() fails to clear the garbage which results in stack overflow, or clears the garbage along with the live variables, resulting in inconsistent results.

```
             bot           avma   lbot          ltop     top
End of stack  |-------------[++++++[-/-/-/-/-/-/-|++++++++|  Start
                free memory            garbage
```

becomes:

```
             bot                         avma   ltop     top
End of stack  |---------------------------[++++++[++++++++|  Start
                       free memory
```

Figure 1: PARI Stack Organization

The solution to this problem is to make sure that no live variables are present across the garbage. This can be done by dividing the code into basic blocks, where each block contains only one entry point and one exit point. This means, each block of code must be such that no point in the block (except the first statement) must be the target address of any preceding or succeeding jump instruction and only one live variable must be present by the time the program control reaches the end of the block. By this, each basic block would possess only garbage along with a live variable by the time the control reaches the end of the block and this garbage can be cleared by carry forwarding the live variable to the next basic block. Now, we shall look at the following example where the garbage collection fails as $x2$ and $y2$ are live within the block.

```
GEN x2 = gsqr(x); GEN y2 = gsqr(y);
/* Block starts here. */
pari_sp ltop = avma; /* record initial avma */
··· garbage ···
.       .
.       .
.       .
··· garbage ···
pari_sp lbot = avma; /* end of the garbage pile */
GEN z = gadd(x2, y2);
/* Block ends here. */
z = gerepile(ltop, lbot, z);
```

In order to make the garbage collection successful, we need to keep the declarations of $x2$ and $y2$ after the statement which records *ltop*. This is done as follows.

```
pari_sp ltop = avma; /* record initial avma */
GEN x2 = gsqr(x); GEN y2 = gsqr(y);
··· garbage ···
.       .
.       .
.       .
··· garbage ···
pari_sp lbot = avma; /* end of the garbage pile */
GEN z = gadd(x2, y2);
z = gerepile(ltop, lbot, z);
```

By dividing the code into basic blocks and clearing the garbage at the end of each block, we were able to make the program run for sufficiently long time without crash.

5 Exploratory Analysis of AKS

This section discusses various experiments we carried out on AKS and the results thus obtained. Section 5.1 includes some of our interesting observations which highlight the possibilities of achieving lower asymptotic bound on the running time of AKS. Based on these experimental results, we present a variant of AKS and conjecture its correctness. All experiments are carried out on Ubuntu 14.04 operating system with Intel I7 3.4*ghz* processor, 16GB DDR3 RAM and 4TB Seagate hard disk. Prior to the installation of PARI/GP, GMP has been installed on the system for faster computations.

5.1 Observations:

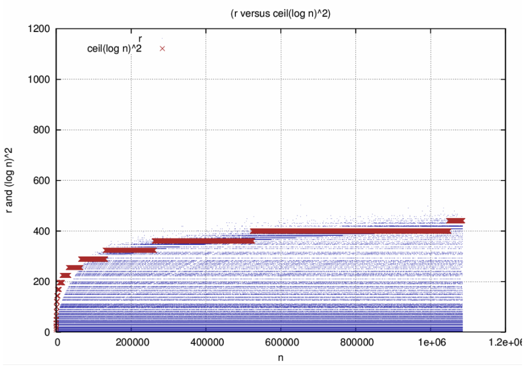
5.1.1 Lower bound for r:

The value of r determines both the time and space complexity of AKS. The larger the r, the larger is the space required to store all the $2r$ number of coefficients of the polynomials in step 5 as well as the computational time required to carry out the steps 3, 4 and 5. Therefore it is a question of interest to know whether it is possible to fix the lower and upper bounds for r. Determining the range of r also helps reducing the search space required in step 2, a second bottle neck for the algorithm.

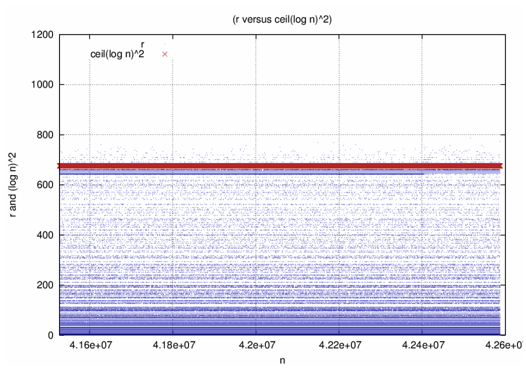
AKS [9] proves that if the Spohie Germain Primes conjecture holds true, then the maximum value of r can never be more than $O(\log^2 n)$. Consequently, the asymptotic running time of the overall algorithm comes down to $O\tilde{}(\log^6 n)$. The plot of r and $\lceil \log^2 n \rceil$ versus n over the bit range $[16, 27]$ in Figure $2(a, b, c, d)$ emphasizes this statement. It can be noted that all values of r are situated around and below the neighbourhood of $\lceil \log^2 n \rceil$. It should be observed that this fact is significant only in a theoretical sense. It does not help modify the algorithm such that the search space for r practically reduces. In reality, it only corrects the maximum bound for the value of r from $O(\log^5 n)$ to $O(\log^2 n)$ and conjectures a theoretical improvement which has no relevance to minimising the real-time computational effort needed by the algorithm, once brought into use. Let us take the following example to illustrate this. For the following 1157 bit integer n, the value of r is 1338367, which is a 21 bit integer. So, the step 2 of the algorithm has to spend time in calculating $\mathrm{o}_r(n)$ for all r in the range $[3, 1338367]$ until it encounters $r = 1338367$, which satisfies the criteria, $\mathrm{o}_r(n) > \lfloor \log^2 n \rfloor$. All values of r can be visualized to be geometrically distributed and if we generalize, it would take more than $2^{\lfloor \log r \rfloor}$ attempts to find a suitable r. This expense of computation is inevitable unless we devise a lower bound for r.

$$
\begin{aligned}
n \quad = \quad & 17653165852885465315379353151198765431236711987654312367112342 1\\
& 533456789872849673659124671123421575247919876543210000671123421\\
& 500067112342150010222202876193949596711234215101010101745967112\\
& 342152342153345678987284967365912467112342157524791987654321000\\
& 067112342150006711234215008753231537935315875323153793531587532\\
& 3153793531587532315379353158753239
\end{aligned}
$$

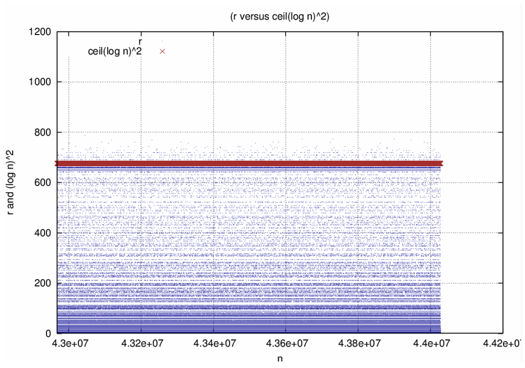
The graphs in the figures show drastic changes in the values of r and underscore the difficulty of guessing a lower bound. However by restricting our focus only on those composite numbers which enter step 5 of the algorithm, a smooth variance in the values of r is identified. The plots of r and $\lfloor \sqrt{\phi(r)} \log n \rfloor$ versus the composite numbers over the bit range $[16, 32]$ which enter the loop of step 5 are given in Figure $3(a, b, c, d)$. Irrespective of the magnitude of n, $\lfloor \sqrt{\phi(r)} \log n \rfloor$ is always found to run below r and is never noticed to intersect with the latter. This behaviour is identified even in the case of numbers with



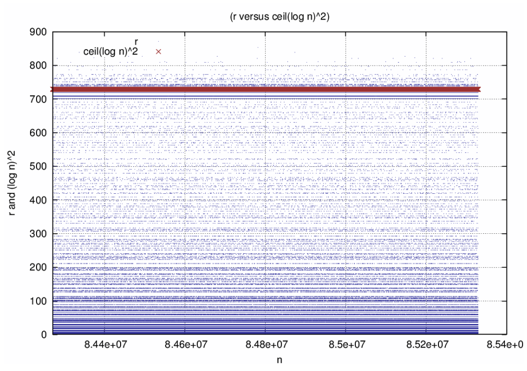
(a)



(b)



(c)



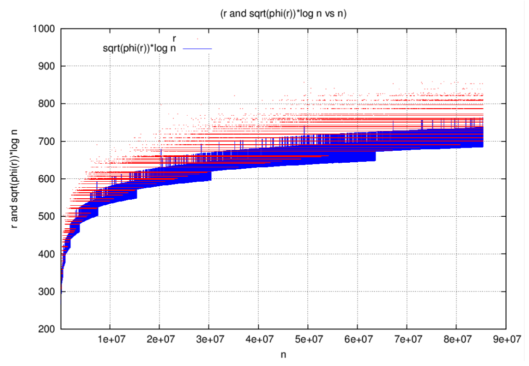
(d)

Figure 2: r and $\log^2 n$ vs n

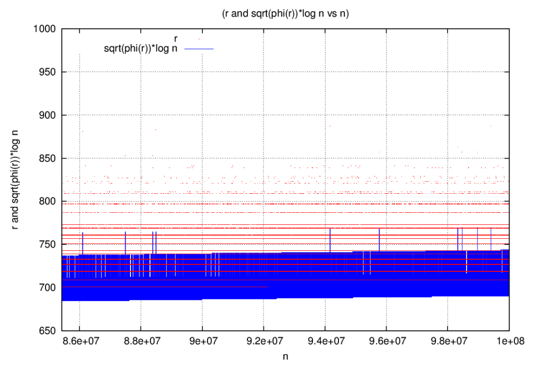
higher bit lengths. Therefore we conjecture the below inequality for all n which enter step 5.

$$
\begin{aligned}
r &> \lfloor \sqrt{\phi(r)} \log n \rfloor \\
\Rightarrow \quad r &> \lfloor \sqrt{r-1} \log n \rfloor \\
\Rightarrow \quad r &> \lfloor \sqrt{r} (\log n - \epsilon) \rfloor \\
\Rightarrow \quad r &> \sqrt{r} (\log n - \epsilon) \\
\Rightarrow r^{\frac{1}{2}} &> (\log n - \epsilon) \\
\Rightarrow \quad r &> (\lceil \log n \rceil - \epsilon')^2
\end{aligned}
$$

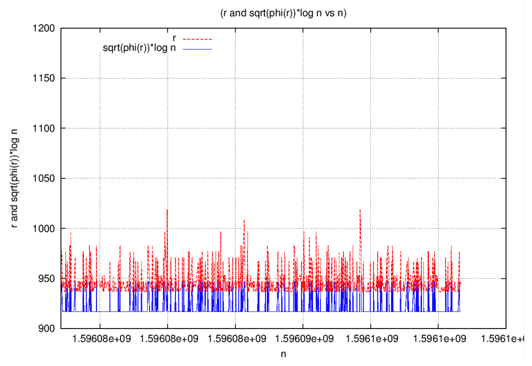
The positive real numbers ϵ and ϵ' in the below steps ensure that the inequality remains held. $\epsilon' = 1$ has been experimentally found to be sufficient. The plots of $(\lceil \log n \rceil - 1)^2$ and r versus n over the bit range $[16, 32]$ are shown in Figure $4(a, b, c, d)$.



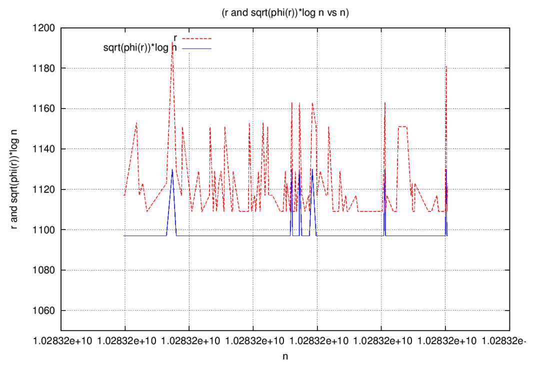
(a)



(b)



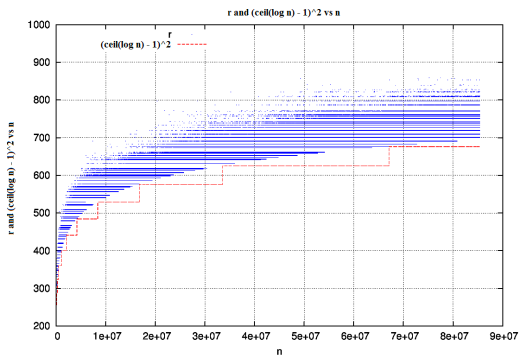
(c)



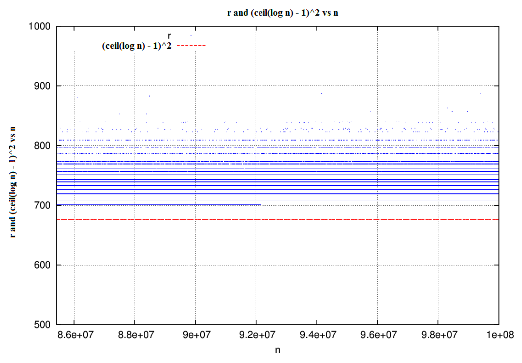
(d)

Figure 3: r, $\lfloor \sqrt{\phi(r)} \log n \rfloor$ vs n

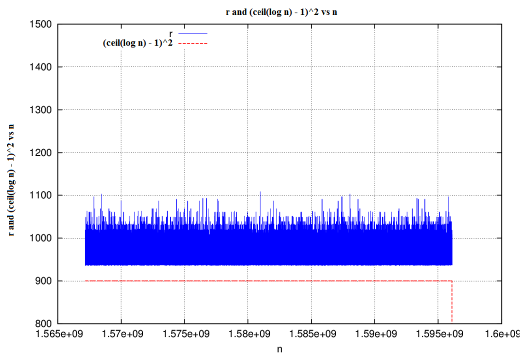
As bit length of n increases, we can notice the quantity $r - (\lceil \log n \rceil - 1)^2$ increasing. Also, the difference $r - \lfloor \sqrt{\phi(r)} \log n \rfloor$ is seen to be always positive and the maximum value of that difference is slowly increasing with bit length. The plots of $r - \lfloor \sqrt{\phi(r)} \log n \rfloor$ vs n over bit range $[2, 28]$ is given in Figure $5(a, b, c, d)$. If step 2 of the algorithm starts searching for r from $(\lceil \log n \rceil - 1)^2$, it would save $(\lceil \log n \rceil - 1)^2 - 3$ number of operations which involve the calculation of $\mathrm{o}_r(n)$. For instance, the 1157 bit integer n given in the begining of this sub section takes only 2031 order calculating operations if the search starts from $(\lceil \log n \rceil - 1)^2 = 1336336$ and saves 1336332 operations. That means, a 99.8482479% of



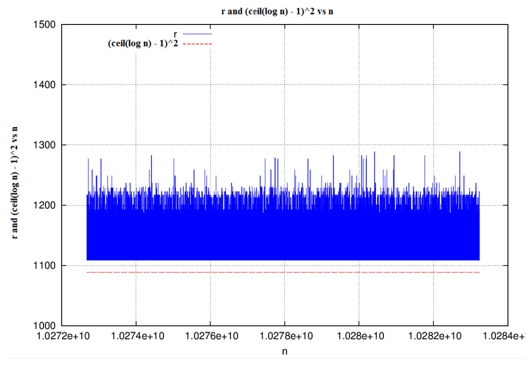
(a)



(b)



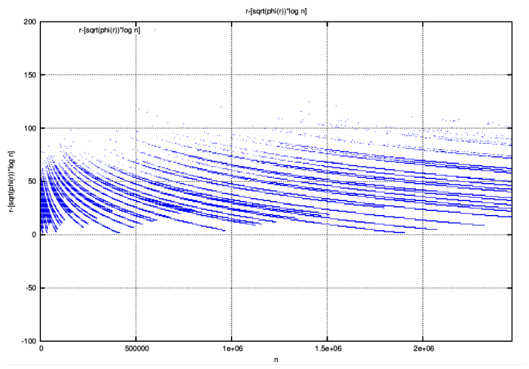
(c)



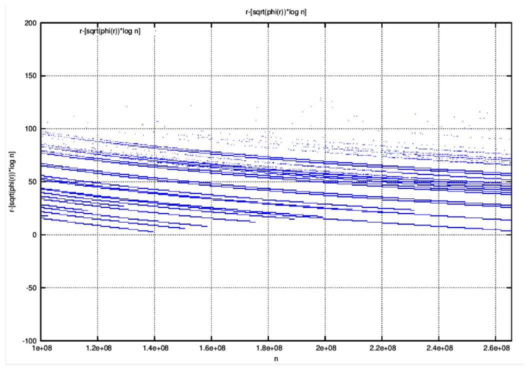
(d)

Figure 4: Lower bound of r

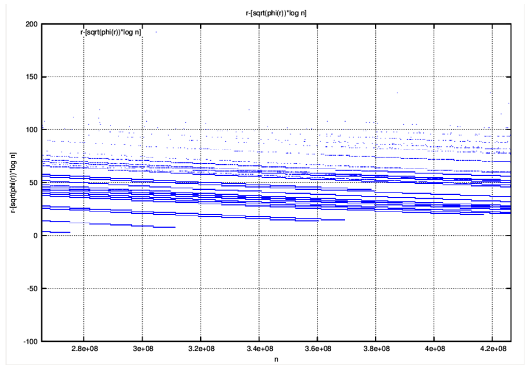
the computational effort is gained. Therefore it is quite imperative to make $(\lceil \log n \rceil - 1)^2$ as the lower bound for r, but this claim is found to be held only in the case of such composites and primes which enter step 5. For the numbers which are reported as composites at step 3, the value of r is identified to be less than $(\lceil \log n \rceil - 1)^2$. Even though AKS chooses a smallest r which satisfies the criteria, $\mathrm{o}_r(n) > \lfloor \log^2 n \rfloor$, we noted the existence of several candidates for r greater than r_{min} which serve the purpose. Now, it is a question of concern whether the algorithm does not report any false positives at step 5, if $r > (\lceil \log n \rceil - 1)^2$ and $r \neq r_{min}$. However, we observed that no such false positives are reported and in particular, the correctness of AKS does not depend on the smallest r at all. It is very pivotal to note that $\mathrm{o}_r(n) > \log^2 n$, $r < \lceil \log^5 n \rceil$, $(r, n) = 1$ and $n \neq a^b$ are the only four esential settings for the steps 5 and 6 to function correctly [9]. Therefore, we conjecture that it is always possible to find a suitable $r > (\lceil \log n \rceil - 1)^2$ and lower bound for r can be set to $(\lceil \log n \rceil - 1)^2$, to reduce the search space in step 2.



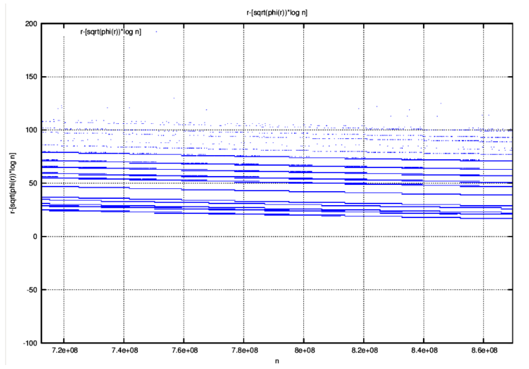
(a)



(b)



(c)

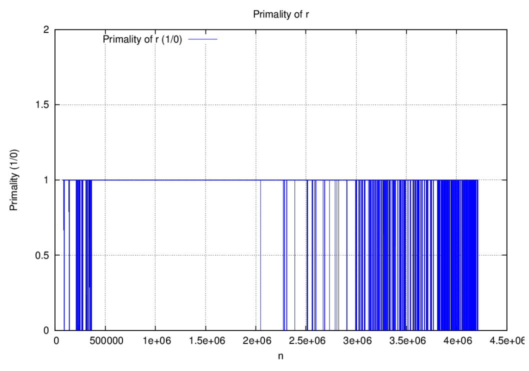


(d)

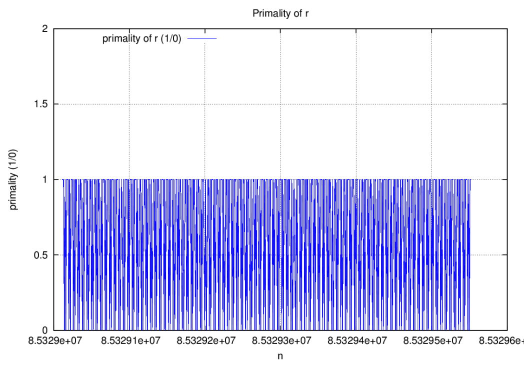
Figure 5: $r - \lfloor \sqrt{\phi(r)} \log n \rfloor$ vs n

5.1.2 Primality of r:

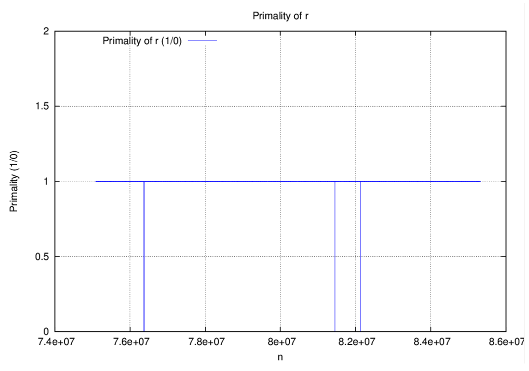
Figure $6(a, b, c, d)$ describes the probability with which r is prime, for all n over the bit range $[2, 32]$. It is vivid from the plots that r is composite for several n. So as to gain information from this distribution, we restricted our focus only on the primality of r corresponding to the composites entering step 5. The plot of primality of r against such composites is shown in the Figure $7(a, b)$ and it can be observed that r is mostly prime! In section 5.1.1, we explained that it is always possible to find r by starting the search



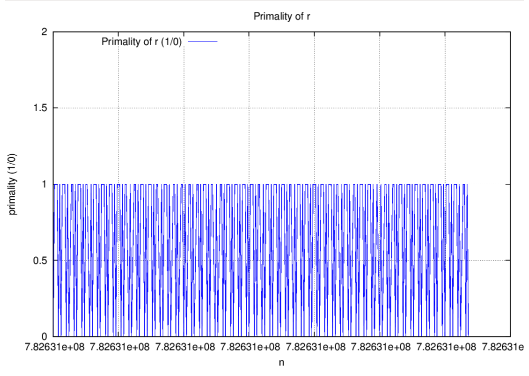
(a)



(b)



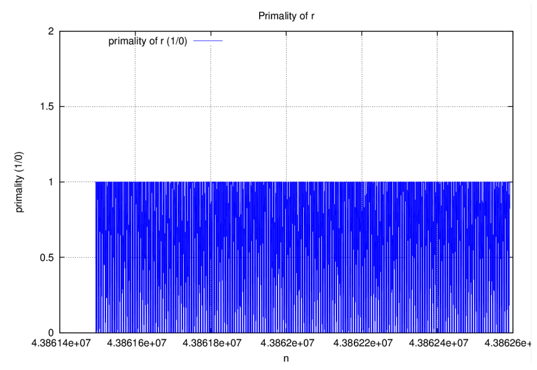
(c)



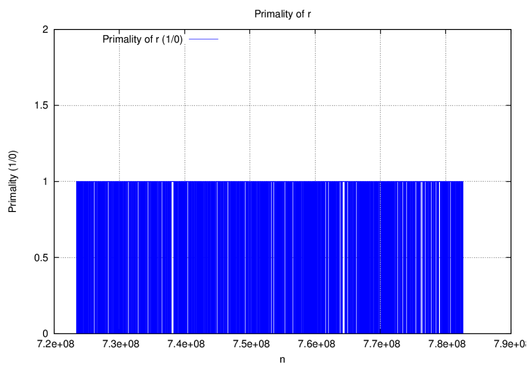
(d)

Figure 6: Distribution of prime r

from $(\lceil \log n \rceil - 1)^2$. Therefore it is anticipating to hope that instead of testing the criteria $\mathrm{o}_r(n) > \lfloor \log^2 n \rfloor$ for all r in the range $[(\lceil \log n \rceil - 1)^2, (\lceil \log n \rceil)^5$, we could choose a prime $r \in ((\lfloor \log n \rceil)^2, (\lceil \log n \rceil)^5$ with $\mathrm{o}_r(n) = r - 1$. Since, the value of r happens to be much smaller when compared to n, testing whether r is prime is easy. Bertrand's postulate states that for any integer $x \geq 2$ there is always at least one prime p such that $x < p < 2x$ [29]. Consequently, there are aproximately $O(\log^3 n)$ number of primes in $[(\lceil \log n \rceil - 1)^2, (\lceil \log n \rceil)^5$ and there could be a greater chance to find r with $\mathrm{o}_r(n) = r - 1 > \lfloor \log^2 n \rfloor$. Choosing r in this way reduces the search space in step 2. As the distribution of primes is less when compared to composites, the computation of $\mathrm{o}_r(n)$ can be bypassed for all the composite-$r \in [(\lceil \log n \rceil - 1)^2, (\lceil \log n \rceil)^5$. For example, consider the 1157 bit number n given in the section 5.1.1. By beginning the search from $(\lceil \log n \rceil - 1)^2$, it takes only 140 order calculating operations, if we include this primality check. In contrast, it requires 2031 operations if no test is performed and 1338367 operations if the search starts from $r = 3$. This saves 99.9896142% of operations. However, there is an overhead of primality check on r, which is very less when compared to order calculation.



(a)



(b)

Figure 7: Distribution of prime r step 4

5.1.3 Upper bound of r:

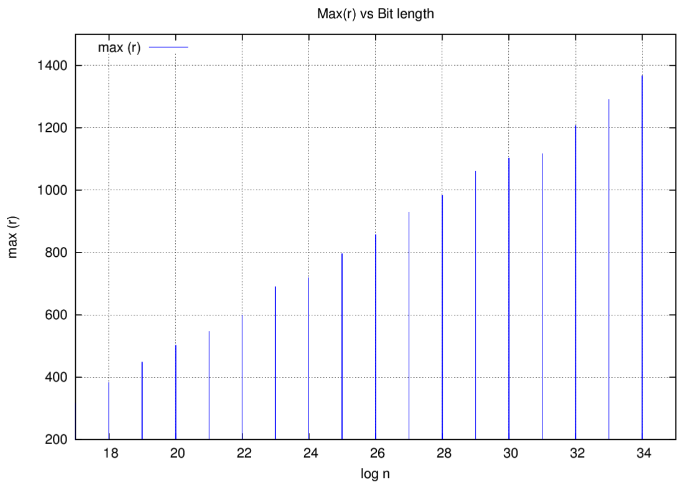
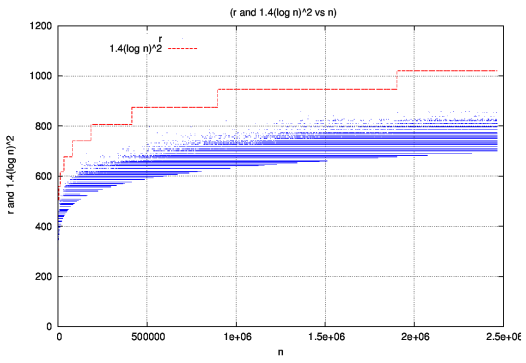
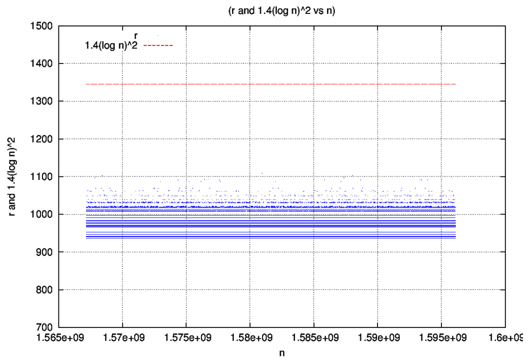


Figure 8: r_{max} vs bit length of n

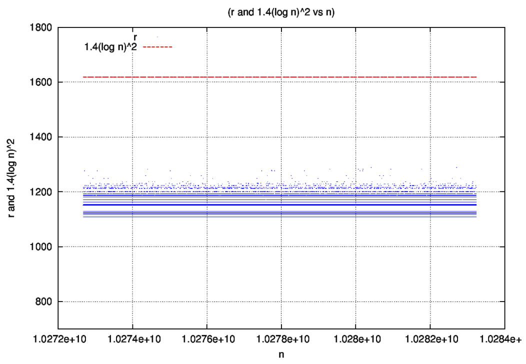
Even though this observation has not provided any immediate means to improve the algorithm, we present it as one of the useful properties of r, because it assists in correcting the asymptotic running time of the test. It is quite vivid from the graphs presented in the previous subsections that r is found to drastically vary as n increases. Even after we fix the lower bound of r to $(\lceil \log n \rceil - 1)^2$, this variation is still noticed. However, the variation is not much drastic in the latter case. It appeals as if r fluctuates in between some fixed bounds; the bounds which vary supposedly with bit length. In a quest to find these bounds, we focussed on the maximum value of r, r^i_{max}, corresponding to the numbers which possess same bit length l_i and compared it with r^j_{max} of the numbers which have successive bit length, l_j.



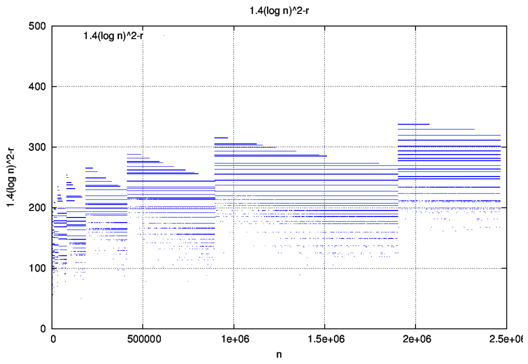
(a)



(b)



(c)



(d)

Figure 9: Upper bound of r

The empirical observations revealed that

$$\forall i,j \in [2,\infty), \quad l_i > l_j \longrightarrow r^i_{max} > r^j_{max}.$$

$$\forall i,j \in [2,\infty), \quad l_i > l_j \longrightarrow r^i_{min} > r^j_{min}.$$

$$\forall i,j \in [2,\infty), \quad l_i > l_j \longrightarrow r^i_{avg} > r^j_{avg}.$$

$$\forall i,j \in [2,\infty), \quad l_i > l_j \longrightarrow r^i_{max} - r^i_{min} > r^j_{max} - r^j_{min}.$$

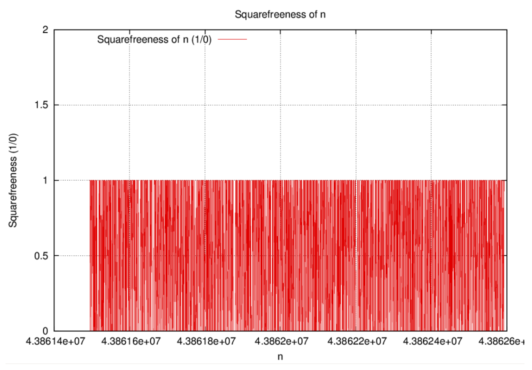
The plot of r_{max} versus bit lengths over the range $[2, 34]$ is shown in Figure 8. Similary, r_{min}, r_{avg} and $|r_{max} - r_{min}|$ are also found to increase with bit length. This made us suspect that there must be a relation between r_{max} and the bit length, $\log n$. We experimentally determined that r_{max} for a given n is always covered by the curve, $1.4(\lceil \log n \rceil)^2$. Plots in the Figure $9(a, b, c, d)$ clearly depict this phenomenon. Also, as the quantity, $1.4(\lceil \log n \rceil)^2 - r$ is greater than 0 $\forall n$ and is found to increase with bit length, it sounds well to underpin this reason to affirm that

$$\forall n, \ (\lceil \log n \rceil - 1)^2 < r < 1.4(\lceil \log n \rceil)^2.$$

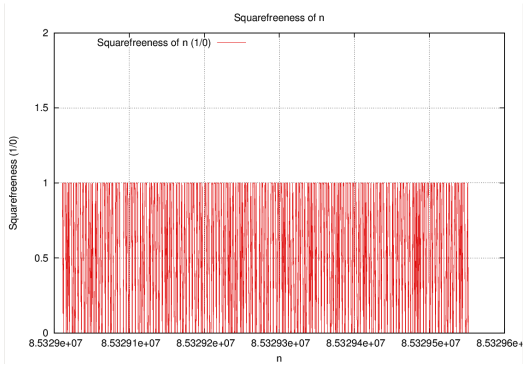
The graph of $1.4(\lceil \log n \rceil)^2 - r$ against n in the Fig 26 reveals that maximum and minimum values along y axis tend to strictly increase with bit length. This upper bound tallies with the bound conjectured by AKS, which is $O(\log^2 n)$ and from our experiments the constant factors involved in it are estimated as, $C = 1.4$ and $n_0 = 983$. In case if a prime r is chosen in step 2, $C = 2$ and $n_0 = 31$. If $P(m)$ denotes the greatest prime divisor of number m, Goldfeld [19] showed that primes q with $P(q-1) > q^{\frac{1}{2}+c}, c \approx \frac{1}{12}$, occur with positive density. Latter, Fouvry [20] has shown that there exist constants $c > 0$ and n_0 such that, $\forall x > n_0 : |\{q | q \ is\ a\ prime,\ q \leq x\ and\ P(q-1) > q^{\frac{2}{3}}\}| \geq c(x/\ln x)$ and consequently implies that $r = O(\log^3 n)$. However, our observations reinforce the correctness of Sophie Germain Prime conjecture and claim that $r = O(\log^2 n)$.

5.1.4 Numbers entering step 5 are mostly squarefree:

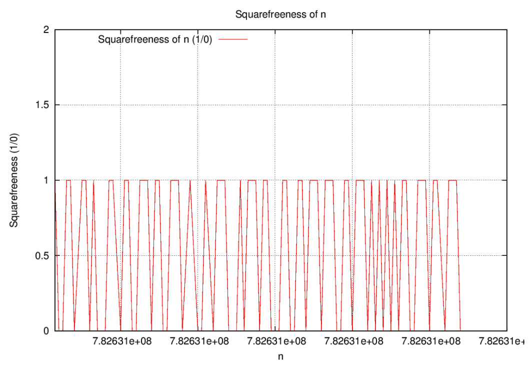
In an investigation to find out some properties of composites entering step 5, we first tried factoring them using a factoring function, $Z\_factor()$ which is available in PARI. We observed that most of them have odd number of prime factors and all prime factors of n are distinct and non-repeating! Using the function $Z\_issquarefree()$ available in PARI, we verified our observation and found most of them as square free. The plots of Figure $10(d)$ shows no fluctuation in squarefreeness. Also, we tested on what fraction of numbers reported as composites in step 3 are square free. We noted that there are several square free numbers which are detected as composites by this step. For example, out of first 300000 natural numbers, only 1450 square free composites entered step 5, while the remaining 154930 square free composites are identified by step 3. The plot in Figure $10(a, b, c)$ affirm this fact. Therefore, relatively less fraction of composites enter step 5 and among them, most are square free.



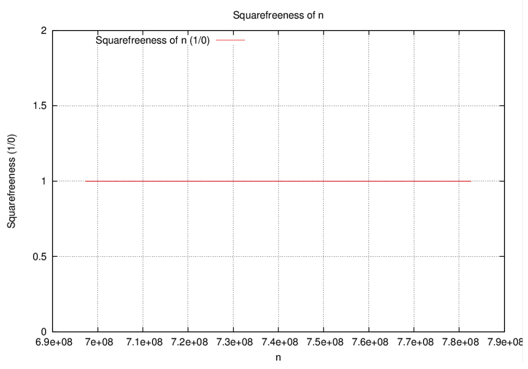
(a)



(b)



(c)



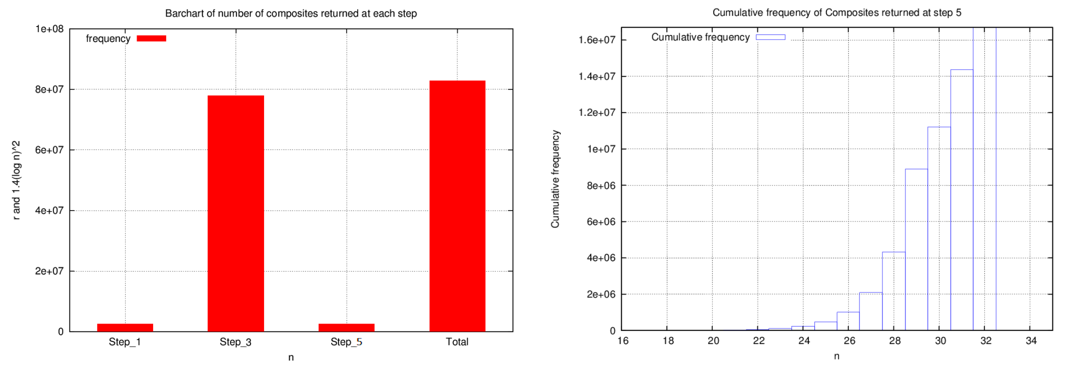
(d)

Figure 10: Distribution of squarefree n

Despite these observations, it is not straightforward to modify the algorithm. However, we hope this observation could be quiet useful in any attempts to find an alternative for step 5.

5.1.5 Frequency of composites reported at steps 1, 3 and 5:

The bar chart rendered in the Figure $11(a)$ clearly indicates that the number of composites reaching step 5 of the algorithm, say n', is less, when compared to the sum total of those reaching the steps 2 and 3. It is evident from the chart that composites returned in step 3 are in majority. The plot of cumulative density of n' against bit length given in the Figure $11(b)$ indicates that there is a little increase in n' with bit length. The observations which will be discussed in the next subsection suggests that if a number happens to be composite,

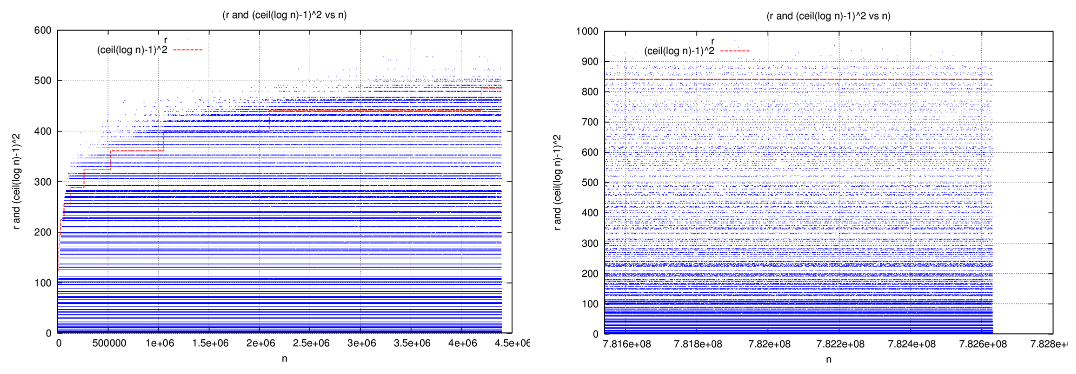


(a) (b)

Figure 11: Frequency of composites

it does not tend to spend more time in the *for loop* of step 5. This suggests that AKS is a *better compositeness tester than a primality tester.*

5.1.6 $r \leq (\lceil \log n \rceil - 1)^2$ for most of the composites identified at Step 3:



(a) (b)

Figure 12: $(\lceil \log n \rceil - 1)^2$ splits step 3

The r values of the integers which are reported as composites at step 3 are extremely small in magnitude when compared to those of the integers which pass to subsequent steps. Most of them are much smaller than the square of the bit lengths of the corresponding

n values. This motivated us to investigate on whether any such r ever assumes a value greater than $\log^2 n$. We noticed $(\lceil \log n \rceil - 1)^2$ covering the major portion of r values chosen by the algorithm. The plots in Figure $12(a, b)$ over the bit range $[2, 34]$ depicts this phenomenon how the density of r values is getting faded out with increase in the magnitude of r. Therefore it is quite beneficial to report n as composite if $r \leq (\lceil \log n \rceil - 1)^2$ before entering step 3, which computes GCD of atmost $\log^5 n$ numbers.

This modification reduces a number of composites reaching step 3 and hence eliminates the third bottle neck of the algorithm for a greater number of composites. However, this modification is possible only when the search for r starts from 3 in step 2.

5.1.7 On the number congruences verified in step 5:

For a prime number n consisting of around 350 decimal digits, it is noted that the algorithm needs to perform nearly 1548490619 polynimial multiplications. This incurs enormously high time and space complexity. Therefore, we tried to focus light on the number of iterations it takes in step 5, if n happens to be a composite. The intention behind conducting such an experiment is that, the composites which enter the loop are found to spend very less time when compared to the primes of nearly equal size. As the program crashes for larger inputs due to storage issues described in [17], we employed the following setup.

Experimental setup: A 4TB Seagate hard disk is completely made available for operating system and Oracle 11g (server) is installed in it. A Java program for AKS is written such that, whenever two coefficients of a polynomial are to be multiplied, it invokes a C program which takes two integers in the form of strings as input and gives as result, a string which contains the result of multiplication. http://Stuff.ro/ [23] illustrates the mechanism of invoking a C function from a Java program by using JNI(Java Native Interface). Then, by connecting to oracle through JDBC(Java Database Connectivity), the program stores the strings in a table representing resultant polynomial. Multiplying two polynomials in the form of tables and performing a modulo operation is not straightforward. However, we used the method suggested by Rotella[6], which replaces modular divisions with additions. Therefore, repeated squaring and multiplication with polynomials is not at all hard to implement using tables. In order to conveniently increase the default VM size of Java, we used Netbeans IDE. The program starts with two tables *T1(index, coefficient)* and *T2(index, coefficient)*, each containing two entries, 1 and a in the coefficient fields and once the product-table is obtained, *T1* and *T2* are updated with product-table. Hence, repeated squaring continues.

With the above experimental setup at disposal, we made observations on some inputs of larger bit sizes, 512 and 1024 as well. Since it is not feasible to test all the numbers, composite numbers entering the step 5 are processed in intervals, by skipping 1000000000 numbers before processing another. As a whole, it took more than 8 months of time to obtain results on 10000 numbers in the bit range, $[256, 1024]$, with each composite number taking around 30 to 60 minutes of time. We restricted our focus only on composite numbers, because the algorithm is bound to reach step 6 for primes and it takes several years to identify primes. We have made the following observations on the number of iterations taken by the *for loop*, for all composite n entering step 5.

1. **The first value of a for which the loop breaks, say a', is very small in magnitude.** Initially, it has been noticed that $a' = 1$ for all first fifty billion composite numbers. With this observation we ran the algorithm on random inputs of higher bit sizes and found that a' is still 1 even for the bit sizes like 256. a' is found to be 2 and 3 for whatever numbers we noted with bit sizes 512 and 1024. As we have not conducted the experiment on all the numbers, we do not assert that the maximum value of a' is 3 for the specified bit range, but what we would like to underscore is the fact that a' is very small when compared to the number of iterations it would take if the input happens to be a prime number. As an example, for the 100 digit prime number $n1$ given below, $r = 108587$, $\sqrt{\phi(r)}\log n1 = 108402$ and it took 35989464 iterations. When we contrast this with a similar 100 digit composite number $n2$ given below, $r = 110359$, $\sqrt{\phi(r)}\log n2 = 110288$ and it took only 1 iteration!

$$\begin{aligned} n1 &= 1537935315875323153793531587532315379353158 \\ & 7532315379353158753231537935315875323153793 \\ & 53158753231537 \end{aligned}$$

$$\begin{aligned} n2 &= 1000000000000000000000000000000000000000000 \\ & 0000000000000000000000000000000000000000000 \\ & 000069999999993 \end{aligned}$$

   This observation serves as a witness for the claim, 'AKS is a good compositeness tester, but can not be used as a primality tester for larger inputs'. This difference in the number of iterations is quite amazing and it demands to search for an alternative which would take equal amount of time for both compositeness and primality testing. We also suspect that the value of a' increases monotonically with respect to n. The reason being, at no instance it so happened that the value of a' is found to decrease, as the input size n increases. However, it is a fact that even as $n \to \infty$, a' never converge to $\sqrt{\phi(r)}\log n$, because AKS has been proven to be true. Supposedly, they both might become asymptotic to each other as n grows without bound. Therefore, the essence of this observation is that there is an enormously high difference between the number of iterations and hence the running times, for compositeness and primality testing using AKS.

2. **The quantity, a' splits the range $[1, \lfloor\sqrt{\phi(r)}\log n\rfloor]$ into two disjoint subsets $S1 : [1, a')$ and $S2 : [a', \lfloor\sqrt{\phi(r)}\log n\rfloor]$, where $(\forall x \in S1$ and $\forall y \in S2) : (x < y)$.** Since, a' happens to be extremely small, the behaviour of the remaining $\lfloor\sqrt{\phi(r)}\log n\rfloor - a'$ values of a remains to be an issue of interest. For all composites entering step 5, it has been observed that

$$\forall a \in [1, a'),\ (X + a)^n \equiv x^n + a(X^r - 1, n)$$

and

$$\forall a \in [a', \lfloor\sqrt{\phi(r)}\log n\rfloor],\ (X+a)^n \not\equiv x^n + a(X^r-1,n)$$

This observation is found to hold true even on larger inputs and infers that $a = \lfloor\sqrt{\phi(r)}\log n\rfloor$ could itself serve the purpose of the whole loop. The candidates $[a', \lfloor\sqrt{\phi(r)}\log n\rfloor]$ for different n can be located to be sandwiched between the curves $\lfloor\sqrt{\phi(r)}\log n\rfloor$ and $a'-1$. The necessity of making $\lfloor\sqrt{\phi(r)}\log n\rfloor$ iterations can be bypassed using only one polynomial test with $a = \lfloor\sqrt{\phi(r)}\log n\rfloor$, i.e., if

$$(X+\lfloor\sqrt{\phi(r)}\log n\rfloor)^n \not\equiv x^n + \lfloor\sqrt{\phi(r)}\log n\rfloor(X^r-1,n),$$

then we report n as composite, else prime. The truthfulness of this check is based on the fact that $a' = \lfloor\sqrt{\phi(r)}\log n\rfloor$ for primes or else AKS is false.

In section 5.1.4 we demonstrated a phenomenon that n is mostly squarefree if it enters step 5. A squarefree number has a property that, under divisibility operation, all its positive divisors form a boolean lattice. In contrast, all positive divisors of a composite number form a poset. This seems to have a connection with number if iterations in step 5, because a' also splits the range S into two equivalence classes $S1 = [1]_{\doteq}$ and $S2 = [a']_{\doteq}$ based on the success and failure of the polynomial check. We shall see how it can be. For a given n, let us define $f(a)$ and $\doteq(a,b)$ as follows:

$$f(a) = \begin{cases} 1 & \text{if, } \forall a \in [1,a'),\ LHS \equiv RHS; \\ 0 & \text{if, } \forall a \in [a', \lfloor\sqrt{\phi(r)}\log n\rfloor), LHS \not\equiv RHS \end{cases}$$

and

$$\doteq(a,b) = \begin{cases} 1 & \text{if, } f(a) = f(b); \\ 0 & \text{if, } f(a) \neq f(b)\ . \end{cases}$$

The following propositions are found to hold true $\forall(a,b) \in S1$:

1) $a \doteq b$	(Reflexivity)
2) if $a \doteq b$ then $b \doteq a$	(Symmetry)
3) if $a \doteq b$ and $b \doteq c$ then $a \doteq c$	(Transitivity)

The above propositions hold true $\forall(a,b) \in S2$ also. Hence, $\doteq$ is an equivalence relation. And the set, $S' = \{f(a),\ \forall a \in [1, \lfloor\sqrt{\phi(r)}\log n\rfloor]\}$ consists of only two elements $\{0,1\}$ and hence $(S', \leq)$ is a boolean lattice. The essence of the observation is that the composites entering the step 5 divides the range $[1, \lfloor\sqrt{\phi(r)}\log n\rfloor]$ into two equivalence classes, thereby suggests a possibility to replace the *for loop* with a single polynomial test. As per the rigorous tests we made on the inputs over varied ranges, we have not found any false positives with this modification. As the attempts[3] are being made to replace the polynomial arithmetic with integer arithmetic and linear recurrences, there is a good hope that an efficient deterministic primality test which can be used in practice, will soon be realized.

5.1.8 An attempt to replace step 5

As mentioned earlier, there is an enormously high requirement for time and space to carry out step 5 even if this step is modified based on observation 5.1.7.2 to perform the verification of congruence for $a = \lfloor\sqrt{\phi(r)}\log n\rfloor$ alone. The reason for this can be attributed to the fact that coefficients of polynomials belong to Z_n and all these coefficients must be in main memory while polynomial arithmetic is performed. This very much highlights the impracticality of AKS. For large n, it is not even possible to break this computation into parts so as to claim the garbage. Therfore, we tried the following means based on the estimation of $\omega(n)$, number of distinct prime factors of n. As the numbers entering step 5 are mostly square free, if $\omega(n) = 1$, then n is mostly prime. However, $\omega(n)$ can not be calculated with this precision in polynomial time. Therefore, we fix the running time to a polynimial and leave the error probability of the resulting randomized algorithm for investigation to make it deterministic.

Hardy, G. H. and Ramanujan [16] provides an estimate of $\omega(n)$ as follows:

$$\omega(n) \approx \ln\ln n + B_1 + \sum_{k=1}^{\infty}\left(-1 + \sum_{j=0}^{k-1}\frac{\gamma_j}{j!}\right)\frac{(k-1)!}{(\log n)^k}$$

where B_1 is 'Mertens constant' and is given by

$$B = t + B_1' + \frac{1}{6}\pi^2$$

where B_1' is calculated from the $var(\omega(n))$ as

$$B_1' = 1.83588427\ldots$$

and

$$t = \sum_{k=1}^{\infty}\frac{1}{p_k^2} = 0.452247$$

The constants γ_j in the above equation are known as Stieltjes Constants and are obtained from the expansion of Riemann zeta function $\zeta(z)$ about $z = 1$. This expansion gives

$$\zeta(z) = \frac{1}{z-1} + \sum_{n=0}^{\infty}\frac{(-1)^n}{n!}\gamma_n(z-1)^n.$$

The constants γ_n are given by

$$\lim_{m\to+\infty}\left[\sum_{k=1}^{m}\frac{(\ln k)^n}{k} - \frac{(\ln m)^{n+1}}{n+1}\right]$$

As these expressions for $\omega(n)$ include the terms which tend to ∞, they seem to be infeasible to compute. However, we tried to approximate it by replacing ∞ with $\sqrt{\phi(r)}\log n$. So,

$\omega(n)$ and γ_n are now modified as

$$\omega'(n) \approx \ln\ln n + B_1 + \sum_{k=1}^{\lfloor\sqrt{\phi(r)}\log n\rfloor} \left(-1 + \sum_{j=0}^{k-1} \frac{\gamma_j}{j!}\right) \frac{(k-1)!}{(\log n)^k}$$

and

$$\gamma_j' = \left[\sum_{k=1}^{\lfloor\sqrt{\phi(r)}\log n\rfloor} \frac{(\ln k)^j}{k} - \frac{(\ln \lfloor\sqrt{\phi(r)}\log n\rfloor)^{j+1}}{j+1}\right].$$

We shall now look at the computational effort needed to implement $\omega'(n)$. For the sake of simplicity, let us not consider the time required for multiplications and divisions. We shall multiply their running time to the asymptotic complexity of computing $\omega'(n)$, at the end. The outer summation requires $\lfloor\sqrt{\phi(r)}\log n\rfloor$ number of iterations. From the observations presented in the section 5.1, we have $\lfloor\sqrt{\phi(r)}\log n\rfloor = O(\log^2 n)$. Let this cost be, A. $(k-1)!/(\log n)^k$ can be computed in at most k iterations. Since, k can be at most $\lfloor\sqrt{\phi(r)}\log n\rfloor$, these k iterations take $O(\log^2 n)$ time. Let this cost be, B. The inner summation takes at most k iterations and hence takes $O(\log^2 n)$ time. Let this be, C. The inner summation involves computation of $j!$ and γ_j'. Peter [21] describes a method for computing $n!$, which takes $O(n(\log n \log\log n)^2)$ time. This also includes the time taken to multiply two $\lceil \log n\rceil$ bit numbers. As j runs till k, $j = O(\log^2 n)$. So, the time required to compute $j! = O(\log^2 n(\log\log n \log\log\log n)^2)$. Let this be, D. Coming to the calculation of γ_j', the summation needs $O(\log^2 n)$ iterations. The subtrahend term inside this summation requires a maximum of $O(\log j)$ multiplications. The minuhend term of the summation also needs $O(\log n)$ multiplications. Since, $j = O(\log^2 n)$, cost of computing γ_j' is $O(\log^2 n \log\log n)$. Let this be, E.

So the total cost of computation, including the time required for multiplications and divisions is,

$$A \cdot (B + C \cdot (D + E)) \cdot O(\log n \log\log n \log\log\log n)$$

which is equal to

$$O(\log^7 n(\log\log n \log\log\log n)^3).$$

Since, we are not computing the terms to ∞, but up to $\sqrt{\phi(r)}\log n$, the value of $\omega'(n)$ is not found to be exactly 1 for primes. But we are investigating on the range of values for $\omega'(n)$, where n is prime. We also suspect that the range is variable with bit length.

If we compare the implementation of $\omega'(n)$ with that of polynomial arithmetic involved in step 5 of AKS, the earlier appears to be quite feasible. The advantage in implementing $\omega'(n)$ is that we need not store the intermediate values in physcical memory. We are free to break the implementation into blocks such that the PARI stack can be made free of garbage, on reaching the end of each block of computation. Where as in the case of the latter, at any cost we ought to store the coefficients of the polynomials on the stack.

When implemented in a traditional way, we found that $\omega'(n)$ was never 1 for several primes and as n increases, $\omega'(n) > 1$ for almost all primes, but we observed a noticeable difference between the estimates of $\omega'(n)$ for primes and those of composites for a given bit length and this difference is found to increase with bit length. Efficient ways to compute Stieltjes constants can also be found in the recent literature [22], which do not involve computing terms up to infinity. Therefore there is a greater scope for $\omega'(n)$ to be computed with higher accuracy in a reasonable time or minimum difference between the estimates of $\omega'(n)$ for primes and those of composites for a given bit length can be found experimentally.

5.2 Proposed Variant of AKS Test:

Based on the observations made from the empirical results which we discussed in the previous section, we modified AKS algorithm to give the following variant which gave no false positives on varried bit ranges. Since, $r = \theta(\log^2 n)$, steps $2, 3, 5$ take $O\tilde{}(\log^4 n)$, $O(\log^3 n)$ and $O\tilde{}(\log^4 n)$ times respectively. Therefore, the overall running time is $O\tilde{}(\log^4 n)$. However, its correctness has to be investigated theoretically.

Algorithm 2: AKS_V3

```
  Data: An Integer $n \in Z$
  Result: A string "PRIME" or "COMPOSITE"
1 begin
2     Step 1: If $n$ is of form $a^b$ then output "COMPOSITE";
3     Step 2: Choose a prime $r$: $r \in ((\lceil \log n \rceil - 1)^2, 2(\lceil \log n \rceil)^2)$ and $o_r(n) \geq \lfloor \log^2 n \rfloor$;
4     Step 3: If $1 < GCD(a, n) < n$ for some $a < r$ then output "COMPOSITE";
5     Step 4: If $n \leq r$ output "PRIME";
6     Step 5: If $(X + \lfloor \sqrt{\phi(r)} \log n \rfloor)^n \not\equiv x^n + \lfloor \sqrt{\phi(r)} \log n \rfloor \ mod\ (X^r - 1, n)$ then
7             output "COMPOSITE";
8     Step 6: output "PRIME";
9 end
```

6 Conclusion

The reason for the impracticality of AKS is attributed to the enormous requirement of space and time. We tried to address this problem by collecting the statistics of parameter r and the number of congruences verified in step 5. Empirical results provided in section 5.1 helped us deriving conclusions on the upper and lower bounds of r. Alternative to the verification of the polynomial congruence in the step 5 of AKS[9] is also suggested in section 5.2. The observations described in section 5.1 provide confidence in carrying out theoretical work to prove the facts affirmed by the results we obtained. We take these observations as foundations for our future work on providing a theoretical upper bound for AKS and making it convinient for practical use.

References

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. *PRIMES is in P*. Preprint (http://www.cse.iitk.ac.in/news/primality v3.ps), February 2003.

[2] Gietzfelbinger, M. *Primality Testing in Polynomial Time: From Randomized Algorithms to PRIMES is in P*. Lecture Notes in Computer Science. Springer, 2004.

[3] Granville, A. *It is easy to determine whether a given integer is prime*. Bull. Amer. Math. Soc Bull. Amer. Math. Soc.

[4] Miller, G. *Riemanns hypothesis and test for primality.* J. Comput. Syst. Sci. 13 (1976), 300317.

[5] Rabin, M. *Probabilistic algorithm for testing primality.* J. Number Theory. 12 (1980), 128138.

[6] C. Rotella, *An efficient implementation of the aks polynomial-time primality proving algorithm,*. School of Computer Science-Carnegie Mellon University. Pittsburgh-Pennsylvania-USA, 2005.

[7] S. Goldwasser and J Kilian, *Almost all primes can be quickly certified.* In Proceedings of Annual ACM Symposium on the Theory of Computing, pages 316329, 1986.

[8] R. Solovay and V. Strassen, *A fast Monte-Carlo test for primality.* SIAM Journal on Computing, 6:8486, 1977.

[9] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, *PRIMES is in P.* Preprint (https://www.cs.auckland.ac.nz/ msta039/primality_v6.pdf).

[10] Joachim von zur Gathen and J rgen Gerhard, *Modern Computer Algebra.* Cambridge University Press, 1999.

[11] RG Salembier, P Southerington, *An implementation of the aks primality test.* Computer Engineering, 2005.

[12] T. Jayasimha, MPI *Implementation of AKS Algorithm.* serc.iisc.ernet.in, 2005

[13] Tong Jin, *Researching and Implementing on AKS Algorithms.* University of Bath, 2005.

[14] A. Menezes, P. OOshot, S. Vanstone, *Handbook of Applied Cryptography.* CRC Press 1996. pp. 606-608.

[15] L. M. Adleman and M.-D. Huang, *Primality testing and two dimensional Abelian varieties over finite fields.* Lecture Notes in Mathematics, 1512, 1992.

[16] Hardy, G. H. and Ramanujan, S. Quart. J. Math. 48, 76-92, 1917. Hardy, G. H. and Wright, E. M. *"The Number of Prime Factors of n" and "The Normal Order of* $\omega(n)$" 22.10 and 22.11 in An Introduction to the Theory of Numbers, 5th ed. Oxford, England: Clarendon Press, pp. 354-358, 1979.

[17] Zhengjun Cao, *A Note On the Storage Requirement for AKS Primality Testing Algorithm.* International Association for Cryptologic Research, (https://eprint.iacr.org/2013/449.pdf)

[18] *PARI/GP computer algebra system,* PARI/GP Development Headquarters (pari.math.u-bordeaux.fr/).

[19] M. Goldfeld. *On the number of primes p for which p+a has a large prime factor.* Mathematika, 16:2327, 1969.

[20] E. Fouvry. *Theoreme de Brun-Titchmarsh; application au theoreme de Fermat.* Invent. Math., 79:383407, 1985.

[21] PB Borwein, *On the Complexity of Calculating Factorials.* Department of Mathematics, Statistics, and Computing Science, Dalhousie University. 1985.

[22] F Johansson, *Rigorous high-precision computation of the Hurwitz zeta function and its derivatives. Johannes Kepler University.* Sep 2013.

[23] *Calling C code from Java using JNI.* (http://stuf.ro/calling-c-code-from-java-using-jni)

[24] H. Lenstra, C. Pomerance, *Primality Testing with Gaussian Periods.* Manuscript 2003.

[25] DJ BERNSTEIN, *Detecting perfect powers in essentially linear time.* American Mathematical Society. 1998.

[26] *NTL: A Library for doing Number Theory.* (www.shoup.net/ntl/)

[27] *The GNU MP Bignum Library.* (https://gmplib.org/)

[28] LiDIA: *A C++ Library For Computational Number Theory.* (www3.cs.stonybrook.edu/ algorith/implement/lidia/implement.shtml)

[29] P. Tchebychev. *Mmoire sur les nombres premiers.* Journal de mathmatiques pures et appliques, Sr. 1(1852), 366-390. (Proof of the postulate: 371-382). Also see Mmoires de l'Acadmie Impriale des Sciences de St. Ptersbourg, vol. 7, pp.15-33, 1854