CS 6150 - ADVANCED ALGORITHMS
FALL 2011
SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Primality Testing

Arijit Banerjee, Suchit Maindola, Srikanth Manikarnike
December 9, 2011

1 PROBLEM STATEMENT

The aim of this project is to implement a very widely studied problem of Mathematics – primality testing – as efficiently as possible. In saying so, our main focus is to successfully implement a breakthrough achieved in 2002 by Agrawal, Kayal and Saxena (AKS) [1] [2] who came up with the "first published primality-proving algorithm to be simultaneously general, polynomial, deterministic and unconditional". Although, this is a worst-case polynomial time algorithm that works on any general number deterministically and is not dependent on yet unproven hypothesis, it does not necessarily give the best running time for all possible inputs. There are other non-determinisitic, approximate, polynomial time algorithms like the Miller Rabin test, the Lucas-Lehmer test for Mersenne numbers and Pepin's test but these algorithms are "constrained" in some way or the other.

2 MOTIVATION

Prime numbers are used extensively in a broad spectrum of fields.

- **Public Key Cryptography**
  With a pair *public* and *private* keys generated using prime numbers, cryptographic algorithms generate cipher text from plain text and vice versa.

$$Plaintext = (Cyphertext^{e}) \mod k$$

  where, e is a product of two large primes p and q and k is the public key.

- **Modular Arithmetic**
  Hash functions are a good example of use of prime numbers for modular arithmetic.

$$h_{a,b}(x) = [(ax+b) \mod n] \mod k$$

  where, n is a prime number

- **Pattern Recognition**
  Finger printing words of an alphabet, used in pattern recognition, can be done using prime numbers. For an alphabet $A = \{x_0,..,x_m\}$, p_i, the i^{th} prime greater than m is generated. Using weighted averages $\forall x_i \in A, w_i = x_i / p_i$, fast finger-printing can be performed [3].

- **Pseudo Random Number Generation**
  Recurrence of the form:

$$Z_k + 1 = (a \cdot Z_k + r) \mod k$$

  where, r and k are relatively prime, known as Linear Congruential Generator Algorithm is used to generate pseudo random numbers.

3 Literature Review

A natural number is said to a prime number if it is greater than one and has no other divisors other than 1 and itself. [4] Though widely studied, it took about 2300 years to come up with a deterministic polynomial time algorithm. While it was always believed that one could produce a polynomial deterministic algorithm, "one reason for the excitement within the mathematical community is not only does this algorithm settle a long-standing problem, it also does so in a brilliantly simple manner. Everyone is now wondering what else has been similarly overlooked".[5]

Existing algorithms to test for primality of a number can be categorized as probabilistic and deterministic. While probabilistic algorithms could result in false positives i.e., determining a composite number as prime, they do not produce false negatives i.e., determining a prime number as composite. The probability of a false positive is quite small for numbers that are not very large. Deterministic algorithms, however, compute the primality or compositeness of a number correctly every time.

3.1 Characteristics of Primality Testing Algorithms

Specificity vs. Generality There exist extremely fast primality tests such as - the Lucas–Lehmer test for Mersenne numbers, and Pepin's test for Fermat's number. Though fast, these tests are specific for a small subset of numbers. Although these algorithms could be used in our implementation, a general primality testing algorithm is needed to handle any number $\in N$.

PROBABILISTIC VS. DETERMINISTIC Deterministic algorithms such as cyclotomy test have a running time that can be proven to be $O((\log n)^{c \log\log\log n})$. [8] Determinism is a necessary characteristic for a primality testing algorithm. Nonetheless, probabilistic algorithm are so much faster than deterministic ones that, one often finds himself incorporating a probabilistic algorithms in his primality tester. Low probability of false negatives, zero false positives and ability to repeat the test for a given input are some of the reasons why. Some tests that we considered were Miller–Rabin primality test, Solovay–Strassen primality test.

POLYNOMIAL VS. NON-POLYNOMIAL The choice between a polynomial algorithm over a non-polynomial algorithm, though glaringly obvious, needs significant consideration. An algorithm with a polynomial running time is one whose running time can be expressed a polynomial of the length of the input, and hence the number bits in it. This however, does not mean they are the fastest solutions around. Non-polynomial algorithms, such as Fermat's Theorem, is deterministic but this test requires a partial factorization of $n-1$ the running time is still quite slow in the worst case. Faster algorithms like the probabilistic Miller-Rabin Test runs in polynomial time over all inputs, but its correctness depends on the truth of the yet unproven generalization of the Reimann hypothesis. [6] And lastly the AKS algorithm - both polynomial and deterministic can be painfully slow for smaller inputs.[8]

Our choice of algorithms to implement the fastest primality tester for this course project must be able to find a sweet spot between determinism, polynomial run time and speed for smaller inputs that polynomial algorithms take too long to process. We will discuss our approach in detail in the Implementation section.

3.2 EXISTING PRIMALITY TESTING ALGORITHMS

Abundant and continuous research is being conducted to analyze the unique characteristic of natural numbers - primality. Based on the application, prime number generators target speed, correctness and polynomial runtime. We describe the following algorithms and our choices in this section.

3.2.1 SIEVE OF ERATOSTHENES - A NAÏVE APPROACH

Named after the ancient Greek mathematician, Eratosthenes of Cyrene, this algorithm is claimed to be the oldest known algorithm to generate prime numbers, originating around 220 B.C. E. The sieve literally "sieves out" every second number after two – the smallest prime – (or multiples of two), then moves to the next available (3) and continues to "sieve out" every multiple of 3 and so on. This algorithm is still important today in number research theory [9] and is found to be the most efficient method to find all small primes (less than 10 million) [10].

Algorithm 1 Sieve of Eratosthenes

```
1: Create a list of consecutive integers from 2 to $n$ : $(2, 3, 4, ...., n)$.
2: $p \leftarrow 2$
3: Compute integral multiples of $p$ and mark every number strictly greater than $p$ in the list.
4: Find the first number greater than $p$ in the list that is not marked
5: Let $p$ equal this number (the next prime).
6: If there are no more numbers marked in the list, stop. Else, repeat step 3.
```

3.2.2 Fermat's Theorem - The Elegant

Pierre de Fermat stated his theorem on October 18, 1640, as

"p divides $a^{p-1} - 1$ whenever p is prime and a is coprime to p"

This algorithm so elegantly states a fact of number theory. This special property stated by Fermat was a stepping stone for researchers around the world to produce a polynomial time algorithm for primality testing.[11] Mathematically, Fermat's theorem can be written as,

$$a^p \equiv 1 \mod p$$

Using this hypothesis, we can modify the Sieve of Eratosthenes and reduce its running time to a mere polynomial of the input n.

Algorithm 2 Fermat's Test

```
1: for $i = 2$ to $n$ do
2:     if $n$ is of the form $i^n \equiv 1 \mod n$ then
3:         return false
4:     end if
5: end for
6: return true
```

As with any theorem, there are exceptions. In 1910, Robert Carmichael found the first smallest number 561 that doesn't obey this rule. He went on define these numbers as "a composite positive integer n which satisfies the congruence

$$b^{n-1} \equiv 1 \mod n$$

for all integers b which are relatively prime to n[12]. These numbers are pseudo-prime or probably-prime [13] and are known as Carmichael Numbers.

3.2.3 Miller-Rabin Algorithm - Mr. Flash

Miller–Rabin primality test is an algorithm which determines whether a given number is prime deterministically. Its determinism, however relies on the unproven generalized Riemann hypothesis (GRH). The original version proposed by Gary L. Miller, in 1975, is deterministic based on GRH. Michael O. Rabin modified it to obtain an unconditional probabilistic algorithm.

The Miller–Rabin test is based on the claim that if we can find an a such that

$$a^d \not\equiv 1 \mod n$$

and

$$a^{2^r d} \not\equiv -1 \mod n \text{ for all } 0 \leq r \leq s-1$$

then n is not prime. The algorithm for Miller-Rabin Test is given below.

Algorithm 3 Miller-Rabin Test

```
Input: n > 3, an odd integer to be tested for primality;
Input: $k$, a parameter that determines the accuracy of the test
Output: composite if $n$ is composite, otherwise probably prime
write $n-1$ as $2^s \cdot d$ with $d$ odd by factoring powers of 2 from $n-1$
loop
  repeat $k$ times
  pick a random integer $a$ in the range $[4, n-2]$
  $x \leftarrow a^d \mod n$
  if $x = 1$ or $x = n-1$ then
    do next loop
  end if
  for $r = 1$ to $s-1$ do
    $x \leftarrow x^2 \mod n$
    if $x = 1$ then
      return COMPOSITE
    end if
    if $x = n-1$ then
      do next loop
    end if
  end for
  return COMPOSITE
end loop
return PROBABLY PRIME
```

The running time of this algorithm is $O(k \log^3 n)$, where k is the number of different values of a we test; hence this is an efficient, polynomial-time algorithm. The probability of getting a correct result improves with the number of bases a we test with. For any odd composite n, at least $\frac{3}{4}$ of the bases a are witnesses for the compositeness of n. If n is composite, then the Miller-Rabin test declares n probably prime with at most 4^{-k} probability[6]. The error bound of Miller-Rabin is $4^{-1} = 25\%$ in the worst case.The probability of a false negative of

Solovay-Strassen test is 2^{-k} and hence its error bound is $\frac{1}{2}$.

DETERMINISTIC VARIANT OF MILLER-RABIN TEST Miller-Rabin can be made deterministic by trying all possible a below a certain limit i.e. The subset of numbers containing the witness to test for the compositeness of n must be present in the set of numbers less than $O((\log n)^2)$ as noted by Miller. The constant of Big-oh notation was later reduced to 2 by Bach. This reduces the condition of primality testing to try out all possible a's between $[2, \min(n-1, \lfloor 2(\log n)^2 \rfloor)]$.

TIME COMPLEXITY The running time of this algorithm is $\tilde{O}((\log n)^4)$. Pomerance, Selfridge and Wagstaff and Jaeschke have verified that when the number n is small (of the order of 10^{15}, it is not necessary to try out all $a < 2(\log n)^2$. This drastically improves the efficiency of the algorithm.

3.2.4 THE AKS ALGORITHM - THE NEW KID ON THE BLOCK

M. Agarwal, N. Kayal and N. Saxena on August 6, 2002 published a deterministic polynomial-time primality-proving algorithm in their paper "Primes is in P". Main characteristics of the algorithm include -

- **General:** The algorithm verified the primality of any general number, unlike the Lucas-Lehmer test, Pépin Test.
- **Polynomial:** The maximum running time of the algorithm can be expressed as a polynomial over the number of digits in the input to be tested unlike the cyclotomy test (APR) [8].
- **Deterministic:** The algorithm is guaranteed to distinguish between primes and composites unlike Miller-Rabin test.
- **Unconditional:** The correctness of AKS is not based on any subsidiary unproven hypothesis unlike the Miller test.

The AKS Algorithm can be explained through 6 theorems [14]-

Theorem 3.1. *Extended Fermat's Theorem*
An integer $n(\geq 2)$ is prime if and only if the polynomial congruence relation

$$(x-a)^n \equiv (x^n - a)(\mod n)$$

holds for all integers a coprime to n

Theorem 3.2. *AKS Theorem*
Suppose that for all

$$1 \leq a, r \leq O(\log^{O(1)} n),$$

theorem 3.1 holds, and a is cop rime to n. Then n is either prime, or a power of a prime.

Theorem 3.3. *AKS Theorem – Key Step*
Let r be cop rime to n, and the residues $(n^i \mod r)$ for $1 \leq i \leq \log^2 n$ are distinct. Suppose that for all $1 \leq a \leq O(r\log^{O(1)} n)$ and theorem 3.1 holds, and a is cop rime to n, then n is either a prime or a power of a prime.

Theorem 3.4. *Existence of good r*
There exists $r = O(\log^{O(1)} n)$ coprimes to n, such that n has order greater than $\log_2{}^2 n$ in multiplicative group $(\mathbb{Z}/r\mathbb{Z})^\times$.

Theorem 3.5. *Lower bound on G*
Let $G \subset F^\times$ be the multiplicative group generated by the quantities $x + a$ for $1 \leq a \leq O(r\log^{O(1)} n)$ then $\forall z \in G, z^m = \phi_m(z)$ and t be the number of such quantitites in F, then

$$|G| \geq 2^t$$

Theorem 3.6. *Upper bound on G*
Suppose that there are exactly t residue classes modulo r of the form $p^i(n/p)^j \mod r$ for $i, j \geq 0$. Then,

$$|G| \leq n^{\sqrt{t}}$$

Algorithm 4 AKS Algorithm

```
if $n = a^b$ for integers $a > 0$ and $b > 1$ then
  return COMPOSITE
end if
Find the smallest $r$ such that $o_r(n) > \log_2{}^2(n)$
if $1 < \gcd(a, n) < n$ for some $a \leq r$ then
  return COMPOSITE
end if
if $n \leq r$ then
  return PRIME
end if
for $a = 1$ to $\lfloor\sqrt{\varphi(r)}\log_2(n)\rfloor$ do
  if $(X + a)^n \neq X^n + a( \mod X^r - 1, n)$ then
    return COMPOSITE
  end if
end for
return PRIME
```

where,
$o_r(n)$ is the multiplicative order of n modulo r
$\varphi(r)$ is Euler's totient function of r.

TIME COMPLEXITY The authors proved that the asymptotic time complexity of this algorithm to be $\tilde{O}(\log^{12}(n))$. The time taken by the algorithm, in the worst case, is of the order the 12^{th} power of the number of digits in n times a polylogarithmic factor.

VARIANTS OF THE AKS ALGORITHM A significant improvement to the initial AKS algorithm was demonstrated in 2005. H. W. Lentsra, Jr. and C. Pomerance proved a variant of AKS could run with a time complexity of $\tilde{O}(\log^6(n))$.

4 IMPLEMENTATION

The strategy chosen to implement this project was an incremental one. We aimed at creating a completely functioning version of the algorithms we chose (discussed in Section 4.2) before tweaking around with the code to accommodate for large integers, conforming to problem specification (such as input handling, exceptions, platform constraints), dealing with bitwise operations leading to data corruption, optimizing for better memory allocation and storage, and efficient algorithms to compute logarithm, GCD, powerOf, etc. We started our initial implementation in C/C++ regardless of the algorithm chosen.

To accommodate for larger inputs, we used GMP libraries such as Library for Number Theory (NTL), Library for Number Theory in C++ — LiDIA, java.math.BigInteger library of java. We optimized the loops for storing values of sieve such that the worst case complexity would be $\tilde{O}(n^{1.5})$. Once we started dealing with larger integers, logarithm, GCD and powerOf computations were also optimized to work quickly with larger inputs.

4.1 ALGORITHMS CONSIDERED

Apart from those algorithms listed in Section 3, we considered Solovay-Strassen test, Lucas-Lehemer test and Pepin's Test. Detailed analysis of runtime, algorithm type and significant literature survey was carried to decide on the final algorithms that we would implement to form a part our primality tester. The key characteristics that helped us determine the final algorithms that would be implemented are listed below.

1. Sieve of Eratosthenes
   – a naïve, general, deterministic and non-polynomial algorithm
   – it uses excessive memory, has a complexity of $O(n(\log n)(\log\log n))$ bit operations

2. Fermat's Test
   – an elegant, specific, non-deterministic and polynomial time algorithm
   – results in false positives for Carmichael Numbers and has an order of $O(k \times \log^2 n \times \log\log n \times \log\log\log n)$ where k is a random number of times we test a random a

3. Miller-Rabin Test
   – a non-deterministic and polynomial time algorithm – works exceedingly well for numbers of the order of 10^{15}, has a time complexity of $O(k \log^3 n)$ and an error bound of 4^{-k}, depends on yet unproven GRH

4. Solovay-Strassen Test
   – another fast, specific (well, almost), non-deterministic and polynomial time algorithm
   – works well, has a time complexity of $O(k \log^3 n)$ and an error bound of 2^{-k} and hence loses out to Miller-Rabin Test

5. Lucas-Leher Test
   – a fast (for a small range of numbers), highly specific, non-deterministic and polynomial time algorithm
   – works only for Mersenne primes which are 47 in number and extremely difficult to represent, has a time complexity of $\tilde{O}(p^2)$ where p is the number of bits in the number.

6. The AKS Algorithm
   – a slow, general, deterministic and polynomial time algorithm
   – slow yet efficient, works well only for extremely large numbers, no dependencies, has a time complexity of $\tilde{O}(\log^6(n))$

7. Pepin's Test
   – a fast, specific, non-deterministic, polynomial time algorithm.
   – can determine whether a Fermat number is prime, which however cannot be used as it highly non-polynomial.[15]

Based on the above characteristics we chose to implement Fermat's Test for primality – to compare our algorithm with a much slower benchmark, Miller–Rabin Test and Lucas-Lehmer Test, as they were extremely fast, had the same time complexity but varied only in their error bound, which was probabilistic and the AKS Algorithm as we needed a deterministic algorithm to default in a situation where none of the other algorithms could be used to test for primality.

4.2 Challenges

Although the choice of algorithms for implementing our primality tester were well weighed and thought out, we did run into roadblocks and hurdles that made us deviate from the prior chosen path. We decided to drop certain algorithms that we had thought were best suited for this implementation. Here is a brief account of what we experienced and how we handled them.

4.2.1 Mersenne Primes - Lucas –Lehmer Test

Mersenne numbers are positive integers that is one less than a power of two and be represented as

$$M_p = 2^p - 1$$

If M_p is prime then p is prime as well. As of October 2009, 47 Mersenne primes are known. The largest known Mersenne prime is also the largest prime number known to man, which is, $2^{43,112,609} - 1$. The search for next Mersenne prime still continues and a prize of \$100,000 would be awarded if one were to discover the next one.

Although ideally 34.3 billion digit numbers can be represented by known operating systems, programming languages limit them to $8.906 \times 10^{19,566}$ which accounts for only the first 27 Mersenne numbers. And coding up the algorithm that worked only for such a small fraction of the input seemed like a large overhead with almost limited or no returns. Hence we decided to not cache Mersenne numbers or implement the Lucas–Lehmer Test.

4.2.2 AKS WITH GMP

Salembier and Southerington [18] describe optimizations for implementing AKS Primality Test using LiDLA library in C++. They use dgcd() and bgcd() functions for Euclidian division and binary classification respectively. Our approach was similar and we would later suggest optimizations. We implemented a crude version the AKS algorithm using inputs and ideas from [1] and [18] using the NTL library. NTL is a high–performance, portable C++ library providing data structures and algorithms for manipulating arbitrary length integers and implementing time-?intensive algorithms [19]. Additionally, we made use of the GMP to provide support for large integers [20]. The running time of AKS algorithm was extremely large as compared to any of the other algorithms. This was expected as this was a initial version of our algorithm before optimizations.

As per the requirements of this project, we were required to ensure that our project would run on the University CADE Lab machines. Despite trying for a long time, we were unable to get these libraries to run on those machines. Hence we decided to move away from this implementation and ported our entire code to java. Java on the other hand is very easy to use and exposes libraries that help us manipulate large integers conveniently.

4.3 FINAL IMPLEMENATAION

For our implementation of the primality tester, we have decided to use both Miller-Rabin Test and the AKS Algorithm. Based on the variant of the Miller-Rabin Test proposed by Pomerance, Selfridge and Wagstaff and Jaeschke, we use the upper limit until which this algorithm behaves deterministically. Values beyond this limit are fed to the AKS Algorithm.

4.3.1 AKS WITH BIGNUMBER

To code up the analysis of the AKS algorithm in Section 3.4.2, we subdivided the algorithm into snippets based on the theorems 3.1 through 3.6. This simplified our task at hand and we were able to reduce the large task of translating the algorithm to code into generating stub code that solved these theorems individually and later integrated them.

Firstly, if there existed any number n that could be expressed in the form $n = a^b$, for some $b > 1$, the we immediately conclude that the number is COMPOSITE without proceed further. In order to perform this check, we needed to compute whether the given number n could be expressed in the form of a^b. This is nothing but checking if $t^k = n$ for k in the range

$2 \le b \le \log n$ and $t = \lfloor n^{\frac{1}{k}} \rfloor$. It is essential that the precision of t be higher than double as n can be greater than $2^{63} - 1$ and any compromise in the precision of t cannot be tolerated[16].

Alternatively, the same step can be performed by calculating a p^m for various values of p and $2 \le m \le \log n$. If any of the values return n, then n would be of the form p^m.The search p is a modified binary search algorithm and hence, its complexity is polynomial.

The second step in solving this algorithm is to find a small r such that $r-1$ has a large prime factor x and $x \ge 4\sqrt{r}\log n$. This is done by computing the GCD of r and n and checking if it equals unity. An occurrence of such a value means that n is COMPOSITE. Values of r are generated using Sieve of Eratosthenes as r is rather small and generating primes within a given fixed range is of the order of the square root of the size of the range. The numbers generated by the sieve are sorted in decreasing order, the largest is chosen which forms the largest prime factor x that can divide $r-1$.

Next, we compute all those for which we need to check for compositeness using extended Fermat's Theorem. These values range from 1 through $2\sqrt{r}\log n$. The exponentiation and modulo operations, however, take a considerable amount of time.

Computation of logarithm of large numbers is done as follows. For a base b and a number w with w_D digits,

$$log_b w = log_b k + w_D \text{ ; where, } k = \frac{w}{b^{w_D}}$$

Finally if none of these methods report the number as COMPOSITE, it is indeed PRIME

4.3.2 Miller–Rabin Test

Implementation of the Miller Rabin Algorithm is fairly easier. For a given number n, we compute $d = l >> 1$, where l is the position of the lowest set bit of n. With $l-1$ as the the limit of a loop, we iterate from 1 with unit increments, checking if either

$$a^d \not\equiv 1 \mod n$$

and

$$a^{2^r d} \not\equiv -1 \mod n \text{ for all } 0 \le r \le s-1$$

If either result true, then the number is COMPOSITE else the number is PRIME.[17] The values of a are chosen from the table –

- if $n <$ 1,373,653, it is enough to test $a = 2$ and 3
- if $n <$ 9,080,191, it is enough to test $a = 31$ and 73
- if $n <$ 4,759,123,141, it is enough to test $a = 2$, 7, and 61

- if n < 2,152,302,898,747, it is enough to test a = 2, 3, 5, 7, and 11
- if n < 3,474,749,660,383, it is enough to test a = 2, 3, 5, 7, 11, and 13
- if n < 341,550,071,728,321, it is enough to test a = 2, 3, 5, 7, 11, 13, and 17

4.4 Integration

To integrate, the two algorithms, we have created a separate class that provides a means for the tester to decide on which algorithm to choose based on the length of the input given. Apart from this, this part of the code

- Conforms to problem statement
- Handles expections
- Reads from standard input in a manner specified in the requirements
- Caches smaller primes upto 17 to instantaneously produce results
- Chooses the value of a for Miller-Rabin test based on the input provided

4.5 Language & Tools

We have chosen java as our programming language for our implementation. Our incremental strategy for implementing this project led us to use java as it a portable language. Algorithms written C/C++ could easily be ported to java.

The sieve function stores a large amount of data. The garbage collector in java performs an efficient task of managing these large chunks of memory.

As per the requirements of the project, we are expected to handle inputs unto 200 digits in length. A regular integer or long variable can store upto 2^{64} i.e. 20 digit. Since our input is significantly beyond this range, we have chosen to use the BigInteger library provided by java.

Libraries in java, unlike that NTL or LiDiA, could easily be included and operated upon. We ran into issues with installing the NTL library in our Univerisity's CADE Lab.

5 Experiments & Results

In this section we present the results of our implementation. We have experimented to large numbers (both prime and composites) and run both Miller-Rabin and AKS algorithms. In our implementation, we have used a combined approach where we choose Miller-Rabin when the number is in such a range such that Miller-Rabin produces deterministic results. For numbers beyond that range, we the AKS algorithm. Our algorithm is disjoint depending on the input it

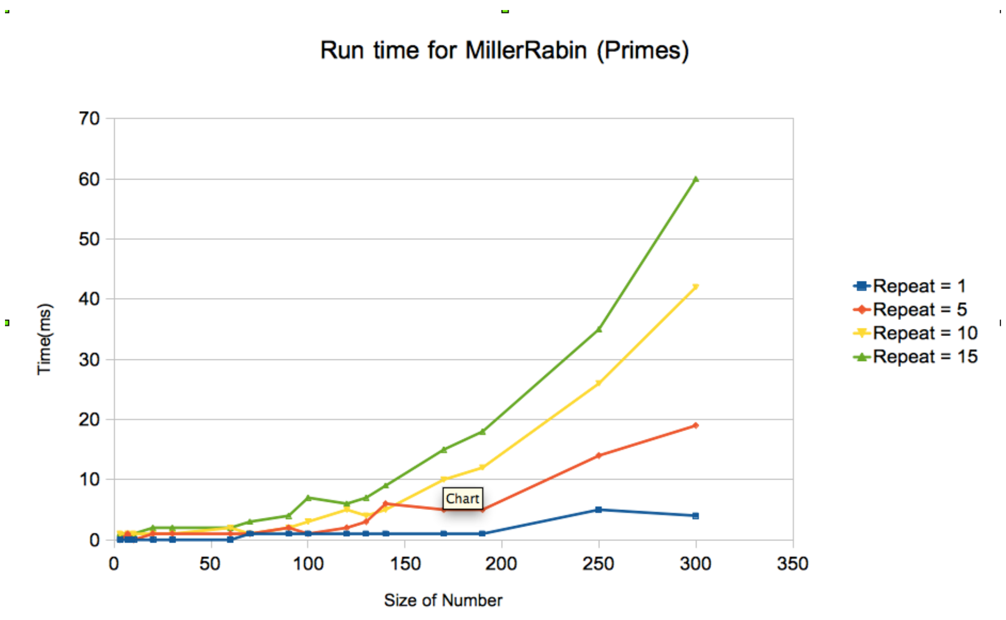

Figure 5.1: Runtime for Miller Rabin (prime)

receives.

Hence separate plots for AKS and Miller-Rabin Tests are presented here that captures the flavor of our algorithm. Since Miller-Rabin is a nondeterminisitc algorithm, we have experimented with extremely large primes and composite inputs separately to look out for any false positives. i.e. when a composite number is falsely reported prime.

In figure 5.1 and figure 5.2 presents the runtime for Miller-Rabin test for prime and composite numbers respectively.

The probability of reducing error in Miller-Rabin Test can be done by varying the number of iterations (values of a) that the test is preformed for every input. We have tried the number of iterations from the set (1, 5, 10, 15). And the result is presented. As expected, runtime increases as we increase the number of iterations. But still for all cases, the execution time is very small even for very large numbers. The maximum execution time that we obtain for a 300 digit number is of the order of 60ms.

In figure 5.3, we present the run time of the AKS algorithm that we implemented. As we see from the figure, the running time is very slow as the number of digits in the input increases. For an input of 300 digits, the AKS takes 80,000 seconds to converge. Even for inputs of smaller length the run time of AKS is orders of magnitude more than that of Miller-Rabin test.

In figure 5.4, we present the results of the AKS implementation using the NTL and GMP

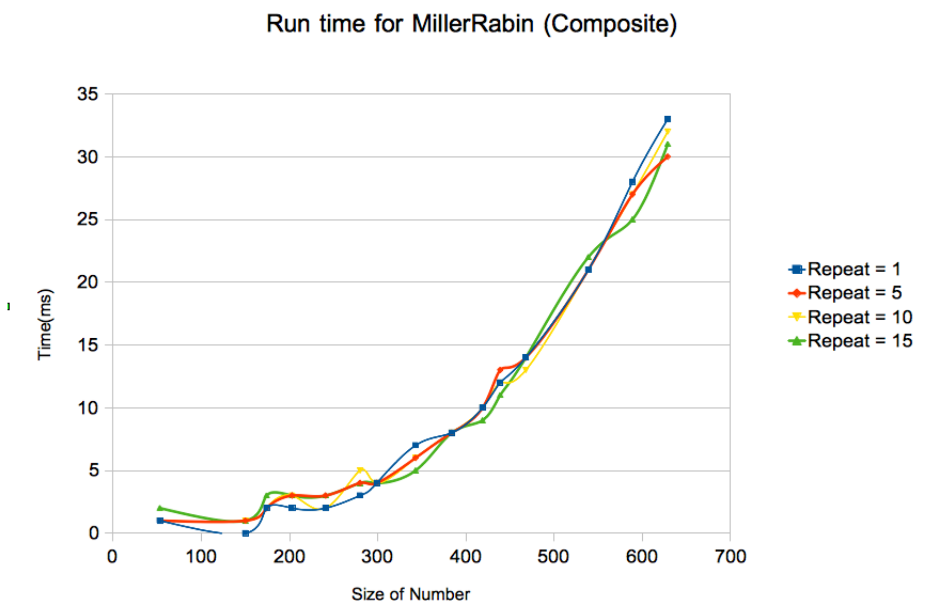

Figure 5.2: Runtime for Miller Rabin (composite)

libraries. As seen from the figure, this implementation is much slower and takes 8000 seconds for inputs as small as 30 digits in length.

So we see that our implementation using java and the BigInteger library is much more efficient as compared to the existing implementation in C++ using GMP and NTL libraries.

Configuration of Machine used
Processor: Intel Core i7 2.8 GHz
Memory: 4Gigabytes

6 Conclusion

In this project we have explored different methods for primality testing and tried to come up with an efficient approach that can test for large primes. After exploring different avenues of the current state of the art of primality testing, we focused on mainly two different algorithms - Miller-Rabin which as a probabilistic algorithm and AKS which is the only polynomial time deterministic algorithm to test primality to date. Our experiments show that though AKS is deterministic and theoretically runs in polynomial time, for large numbers its execution time is painfully slow. Whereas, the non-deterministic Miller-Rabin algorithm executes very quickly for even large numbers. Though Miller-Rabin never tells a prime number as composite, it can give false positives,i.e. flag one composite number as prime. However, by adjusting the

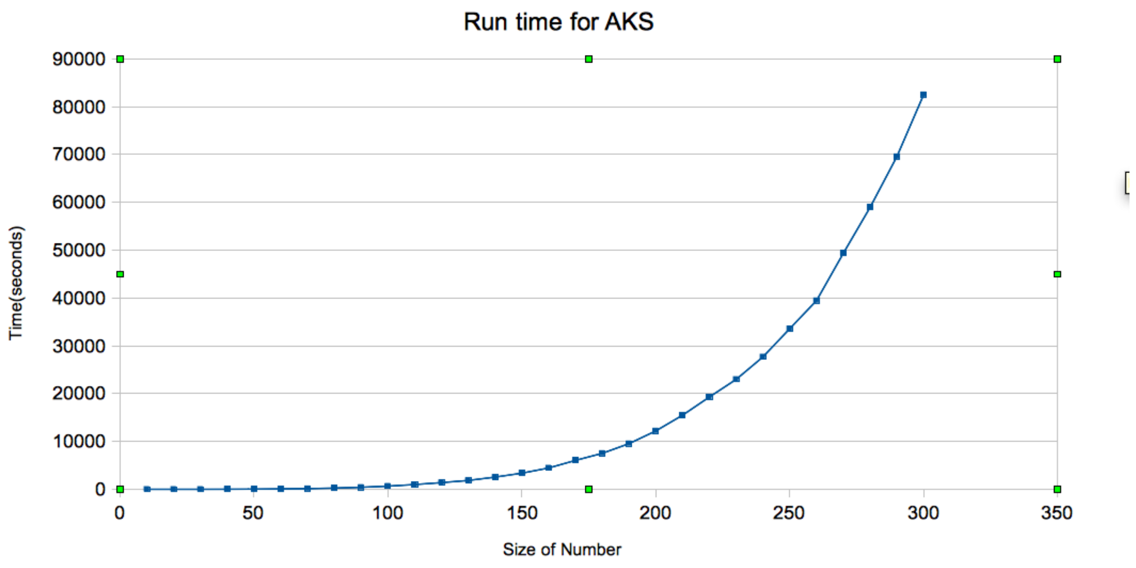

Figure 5.3: Runtime for AKS (our implementation)

number of trials we can make the probability of false alarms negligibly small. In fact, for all the experiments we run we never encountered any false alarms. So, we believe that until and unless a complete deterministic primality testing is required , for all practical purposes one should use Miller Rabin test , adjusting the probability of false positive as required. It should be noted that, for numbers below certain range Miller Rabin test can be made fully deterministic. So though we have presented a combined approach of AKS and Miller-Rabin as a deterministic test for primality, we recommend that based on the application need one should try to use Miller-Rabin as much as possible.

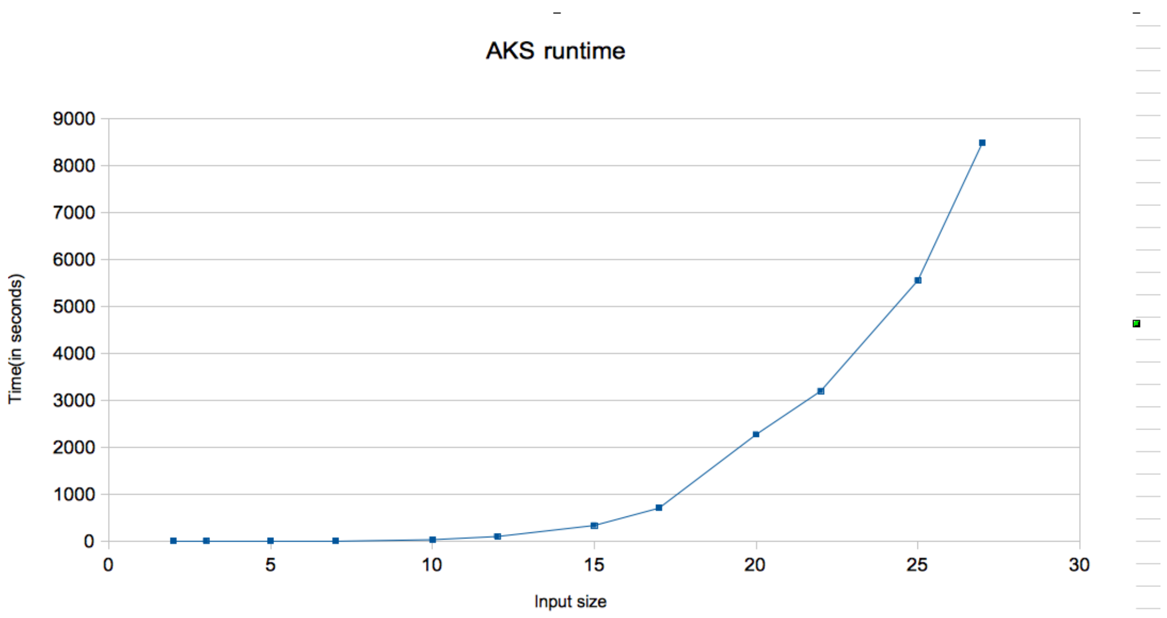

Figure 5.4: Runtime for AKS (using NTL)

7 References

[1] - Agrawal, Kayal and Saxena. Primes is in p. Annals of Mathematics (2004), 781– 793.
[2] - Advanced Algorithms, University of Utah, Fall 2011. `https://learn-??uu.uen.org/courses/48426/files/3870539/download?wrap=1`.
[3] - `http://users.auth.gr/~karanika/Ergasies/string1.pdf`
[4] - Wikipedia. Prime number. `http://en.wikipedia.org/wiki/Prime_number`.
[5] - Crandall and Papadopoulos 2003
[6] - `http://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test`
[7] - `http://en.wikipedia.org/wiki/AKS_primality_test`
[8] - `http://en.wikipedia.org/wiki/Adleman-Pomerance-Rumely_primality_test`
[9] - `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`
[10] - `http://www.math.twsu.edu/history/men/eratosthenes.html`
[11] - `http://www.pballew.net/FermLit.html`
[12] - `http://en.wikipedia.org/wiki/Fermat's_little_theorem`
[13] - `http://en.wikipedia.org/wiki/Carmichael_number`
[14] - `http://terrytao.wordpress.com/2009/08/11/the-aks-primality-test/`
[15] - `http://en.wikipedia.org/wiki/P%C3%A9pin's_test`
[16] - Abdullah Hosain, Implementation of the AKS Primality Testing Algorithm, December 2007.
[17] - `http://en.literateprograms.org/Special:Downloadcode/Miller-Rabin_primality_test_%28Java%29`
[18] - Salembier and Southerington. An Implementation of AKS Primality Test. 2005.

[19] - NTL: A library for doing Number Theory. `http://www.shoup.net/ntl/`
[20] - The GNU Multi Precision Arithmetic Library. `http://gmplib.org/`