

University of Connecticut

Computer Science and Engineering Department

CSE 6512 Randomization in Computing

Spring 2016

Randomized and Deterministic Primality Testing

Proposal of Course Project

Abdullah Alenizi, Dina Abdelhafiz

Email : {abdullah.alenizi@uconn.edu, dina.abdelhafiz@uconn.edu}

Supervised by

Prof :-Sanguthevar Rajasekaran

Contents

Abstract.....	4
1. Problem Definition.....	5
2. Introduction	5
3. Characteristics of Primality Testing Algorithms	7
4. Existing Primality Testing Algorithms	8
4.1. Sieve of Eratosthenes.....	9
4.2. Fermat’s Theorem.....	9
4.3. Miller-Rabin Algorithm.....	10
4.4. AKS Algorithm	13
4.4.1. Complexity analysis of AKS algorithm.....	15
4.5. Modified AKS V3 Algorithm.....	16
4.5.1. Time Complexity analysis for modified AKS V3.....	17
5. Implementation.....	18
5.1. Hardware:.....	18
5.2. Software:	20
5.3. Usage:.....	22
5.4. Method of testing:	22
6. Conclusion	26
References	26
Appendix A:.....	28
Appendix B.....	40

Table of figures

Figure 1 the specification of the virtual machine	19
Figure 2 the specification of the virtual machine	19
Figure 3 shows the result of "top" command after starting the test.	20
Figure 4 shows a screen shots of the bash scripts:.....	21
Figure 5 shows a screen shots of the bash scripts:.....	21
Figure 6 shows a comparison between AKS & Miller-Rabin for 1000 random prime numbers 9 digits each. .	23
Figure 7 testing 100 numbers 50 digit each.....	23
Figure 8 we tested 5000 digit 100 numbers.	24
Figure 9 shows random 5000 digit numbers with AKS method.	25

Abstract

The aim of this project is to implement a very widely studied problem of Mathematics primality testing as efficiently as possible. There currently exist many different primality tests, each with their distinct advantages and disadvantages. These tests fall into two general categories, probabilistic and deterministic.

Probabilistic algorithms determine with a certain probability that a number is prime, while deterministic algorithms prove whether a number is prime or not. Probabilistic algorithms are fast but contain error. Deterministic algorithms are much slower but produce proofs of correctness with no potential for error. Furthermore, some deterministic algorithms don't have proven upper bounds on their run time complexities, relying on heuristics.

In saying so, our main focus is to implement one major primality tests in each category and compare their primality test results. The first Algorithm is the Miller–Rabin test, which is widely used probabilistic primality test. The second test is a breakthrough achieved in 2002 by Agrawal, Kayal and Saxena (AKS) they presented unconditional deterministic polynomial-time algorithm that determines whether an input number is prime or composite.

1. Problem Definition

A prime number is a positive integer p having exactly two positive divisors, 1 and p . while a composite number is a positive integer $n > 1$ which is not prime. In course of the project, we want to explore the fastest possible way to determine whether a number is prime or not, AKS[1], Miller Rabin [2] algorithms will definitely be our main focus area. Primality test's output is binary: either 1: PRIME or 0: COMPOSITE.

2. Introduction

Prime numbers are of fundamental importance in mathematics in general, and number theory in particular. So it is of great interest to study different properties of prime numbers. Of special interest are those properties that allow one to determine efficiently if a number is prime. Such efficient tests are also useful in practice: a number of cryptographic protocols need large prime numbers and is a very useful in the Pattern Recognition. For example, Prime numbers are used extensively in a broad spectrum of fields like the following:

- **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.

$$\text{Plaintext} = (\text{Cypher text}^e) \bmod k$$

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

- **Pseudo Random Number Generation**

Recurrence of the form:

$$Z_{k+1} = (a * Z_k + r) \text{ mod } k$$

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

- **Modular Arithmetic**

Hash functions are a good example of use of prime numbers for modular arithmetic.

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

Where, n is a prime number

The search for a polynomial time deterministic algorithm for primality testing has been an open problem for a long time. Consequently, many researchers have provided various algorithms. There are some very efficient algorithms such as the Solvay - Strassen [3] or Miller Rabin algorithm [2]. They are termed as “probabilistic” polynomial time primality testing algorithms. The term probabilistic implies that they can make mistakes some times. In 1983, Adleman, Pomerance, and Rumely [4] achieved a major breakthrough by giving a deterministic algorithm that runs in $(\log n)O(\log \log \log n)$ time, though correctness proofs of this deterministic algorithm is very complex.

It was finally proved by M. Agrawal, N. Kayal and N. Saxena that the set of primes is in the complexity class P. The name of this algorithm is called AKS Algorithm[1], [5]. They named after these three inventors. For any given integer n , the AKS algorithm can determine the number n is prime number or not in the running time of $O(\log^{12} n)$, while the best deterministic algorithm known before has polynomial time complexity. The AKS Algorithm is based on the de randomization of a polynomial identity testing. It includes

many iterations of polynomial modular exponentiation. Since the algorithm has been presented many efforts are made to implement the algorithm. The major challenge is to implement the congruence, which is the most time consuming step in the algorithm. Most of the efforts are focused on speeding up this congruence check.

3. Characteristics of Primality Testing Algorithms

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.

- **Specificity VS. Generality**

There exist extremely fast primality tests such as - the Lucas–Lehmer test [6] for Mersenne numbers, and Pepin’s test [7] for Fermat’s number. Though fast, the tests are specific for a small subset of numbers.

- **Probabilistic VS. Deterministic**

Deterministic algorithms such as cyclotomy test have a running time that can be proven to be $O((\log n)^c \log \log \log n)$ [4]. 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.

- Polynomial Vs. Non-Polynomial

The choice between a polynomial algorithms 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[2]. And lastly the AKS algorithm – both polynomial and deterministic can be painfully slow for smaller inputs [1], [5].

In this project we are concentrating on implementing of the AKS algorithm and Miller Rabin Algorithm and their run time measurements and to discover any improvement during the development process.

4. 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.

4.1. Sieve of Eratosthenes

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) [8]-[10]

Algorithm1 Sieve of Eratosthenes

- 1: Create a list of consecutive integers from 2 to n : $(2, 3, 4, \dots, n)$.
 - 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.
-

4.2. Fermat's Theorem

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,

$$a^{p'} \equiv 1 \pmod{p}$$

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

Algorithm2 Fermat's Test

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

In 1910, Robert Carmichael [12] 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

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

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

4.3. Miller-Rabin Algorithm

Miller-Rabin primality test is an algorithm which determines whether a given number is prime deterministically [2]. 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

$$a^d \neq 1 \pmod{n}$$

and

$$a^{2^r d} \neq -1 \pmod{n} \text{ for all } 0 \leq r \leq s-1$$

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

Algorithm3 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 $a^s \cdot d$ with d odd by factoring powers of 2 from $n-1$

loop

 repeat k times

 pick a random integer a in the range $[4, n/2]$

$x \leftarrow a^d \pmod{n}$

if $x = 1$ or $x = n-1$ **then**

 do next **loop**

end if

for $r=1$ to $s-1$ **do**

$x \leftarrow x^2 \pmod{n}$

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

```

4.3.1. Complexity Miller-Rabin Test

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 [2]. The error bound of Miller-Rabin is $4^{-1} = 25\%$ in the worst case. The probability of a false negative of Solovay-Strassen [3] test is 2^{-k} and hence its error bound is $\frac{1}{2}$.

4.3.2. 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 $[2, \min(n-1, 2(\log n)^2)]$.

TIME COMPLEXITY The running time of this algorithm is $\tilde{O}((\log n)^4)$. Pomerance, Selfridge And Wagstaff and Jaeschke [14] 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.

4.4. AKS Algorithm

AKS Algorithm Original This algorithm was first published in August 2002 by three India computer scientists named M. Agrawal, N. Kayal, and N. Saxena. This was first time in the history that to determine a number whether or not a prime in deterministic polynomial time complexity[1]. Main characteristics of the algorithm include -

- **General:** The algorithm verified the primality of any general number, unlike the Lucas-Lehmer test[6], Pépin Test[7].
- **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).
- **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 4 theorems:

Theorem 1. Extended Fermat's Theorem

An integer $n > 2$ is prime if and only if the polynomial congruence relation

$$((x - a)^n \equiv x^n - a \pmod{n})$$

Holds for all integers a coprime to n

Theorem 2. AKS Theorem

Suppose that for all

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

Theorem 1 holds, and a is coprime to n. Then n is either prime, or a power of a prime.

Theorem 3. AKS Theorem – Key Step

Let r be coprime to n, and the residues $(n^i \pmod{r})$ 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 coprime to n, then n is either a prime or a power of a prime.

Theorem 4. Existence of good r

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

The AKS Algorithm can be written in three main steps and they are listed below:

- a) Perfect power checking the input n
- b) Find r, $q = \text{LargestPrimeFactor}(r)$ that $q \geq 4\sqrt{r} \log n$ & $n^{\frac{r-1}{q}} \not\equiv 1 \pmod{r}$
- c) $a=1$ to $2\sqrt{r} \log n$, check $(x - a)^n \not\equiv x^n - a \pmod{x^r - 1, n}$

The most time consuming step is at the last step of the for loop to compute the congruence modulo. We will discuss its time complexity in the next section just below the algorithm.

The algorithm is obtained from the 'PRIMES is in P' paper [1].

Algorithm4 AKS

Input: Integer $n > 1$

1. if (n is of the form $a^b, b > 1$) output COMPOSITE;
 2. $r = 2$;
 3. while ($r < n$) {
 4. if ($\gcd(n, r) \neq 1$) output COMPOSITE;
 5. if (r is prime)
 6. let q be the largest prime factor of $r - 1$;
 7. if ($q \geq 4\sqrt{r} \log n$ & $n^{\frac{r-1}{q}} \neq 1 \pmod{r}$)
 8. break;
 9. $r \leftarrow r + 1$;
 10. }
 11. for $a = 1$ to $2\sqrt{r} \log n$
 12. if $(x - a)^n \neq x^n - a \pmod{x^r - 1, n}$ output COMPOSITE;
 13. Output PRIME;
-

4.4.1. Complexity analysis of AKS algorithm

Below are the three main steps required in the AKS Algorithm:

- a. Perfect power checking the input n
- b. Find $r, q = \text{LargestPrimeFactor}(r)$ that $q \geq 4\sqrt{r} \log n$ & $n^{\frac{r-1}{q}} \neq 1 \pmod{r}$
- c. $a = 1$ to $2\sqrt{r} \log n$, check $(x - a)^n \neq x^n - a \pmod{x^r - 1, n}$

The first step of the algorithm takes asymptotic time $O(\log^3 n)$.

And the while loop in step b makes $O(\log^6 n)$ iterations.,

The for loop at last step takes asymptotic time $r^{3/2} \log^3 n = O(\log^{12} n)$, The GCD can be evaluated using Euclid's Algorithm. This takes $\text{poly}(\log \log r)$ steps. The check for primality of r and finding the largest factor q can be done by within $O(\sqrt{r} \text{poly} \log r)$ steps.

The total complexity of the while loop is $O(r \sqrt{r} \text{poly}(9n \log r)) = O \log^9 n$. Though more efficient and sophisticated algorithm can be used but even a very straightforward and slow implementation of these does not effect the time taken by the algorithm as most of the time is spent in the painful congruence for loop from step 11 to 13.

The for loop iterates $2\sqrt{r} \log n$ times, in one iteration of the loop, if repeated squaring and multiplication is used then $O(\log n)$ multiplication and squaring of polynomials of degree less than r are done. By using FFT for these polynomial and integer operations, a single iteration takes $O(r \log^2 n)$ time. Hence the time taken by the for loop $O(r \log^2 n)$ becomes the total complexity of the algorithm.

4.5. Modified AKS V3 Algorithm

The algorithm was published in 2004[5]. It has been improved from its original AKS Algorithm. And its time complexity is faster than its original. This algorithm also has three main steps to compute the input n . The algorithm listed below:

a) Perfect power checking the input n

b) Find $r, O_r(n) \geq 4 \log^2 n$

c) $a=1$ to $[2 \sqrt{\varphi(r)} \log n]$ check $(x - a)^n \not\equiv x^n - a \pmod{x^r - 1, n}$

The modified AKS Version algorithm also needs three main steps to determine the input data. But there are some changes in the step 2 and 3, which increase its computation time. Therefore, its time complexity is faster than original AKS Algorithm. The more detailed time complexity analysis is stated after the algorithm below. The algorithm shown below is modified AKS [5]:

Algorithm 5 modified AKS

Input: Integer $n > 1$

1. if $(n = a^b, \text{ for } a \in \mathbb{N} \text{ and } b > 1)$ output COMPOSITE;
 2. Find the smallest r such that $O_r(n) \geq 4\log^2 n$
 3. If $1 < (a, n)$ for some $a \leq r$, output COMPOSITE.
 4. If $n \leq r$, output PRIME.
 5. For $a = [2\sqrt{\varphi(r)\log n}]$ do
 If $(x + a)^n \neq x^n + a \pmod{x^r - 1, n}$, output COMPOSITE;
 6. Output PRIME;
-

4.5.1. Time Complexity analysis for modified AKS V3

In step 1, it is same as original AKS Algorithm, so it requires $O(\log^3 n)$.

In step 2, we find an r with $O_r(n) \geq 4\log^2 n$. This can be done by trying out successive values of r and testing if $n^k \neq 1 \pmod r$ for every $k \leq 4\log^2 n$. For a particular r , this will involve at most $O(n \log^2 n)$ multiplications modulo r and so will take time $O(\log^2 n \log r)$. We know that only $O(\log^5 n)$ different r 's need to be tried. Thus the total time complexity of step 2 is $\log^7 n$.

The third step involves computing GCD of r numbers. Each GCD computation takes time $O(\log n)$ and therefore, the time complexity of this step is $O(\log n) = O(\log^6 n)$.

The time complexity of step 4 is just $O(\log n)$.

In step 5, we need to verify $\sqrt{\varphi(r) \log n}$ equations. Each equation requires $O(\log n)$ multiplications of degree r polynomials with coefficients of size $O(\log n)$. So each equation can be verified in time $O(r \log^2 n)$ steps. Thus the time complexity of step 5 is $O(r \sqrt{\varphi(r)} \log^3 n = O(r^{3/2} \log^3 n = O(\log^{10.5} n))$. This time dominates all the other and is therefore the time complexity of the algorithm.

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 $\tilde{O}(\log^6(n)[4])$.

5. Implementation

In this Implementation, we tried to test the execution time for AKS and Miller-Rabin primality testing methods to see the difference in execution time.

5.1. Hardware:

In this project, we relied on the cloud to perform our tests. The reason is because in some cases using a virtual machine would not have tasks other than the our scripts which makes it more reliable when it comes to execution time. Another reason is that a virtual machine can be left running the tests without worrying about it would updates, reboots, sleeps, ...etc. Figures [1:2] below shows the specification of the virtual machine:

```

root@ubuntu-512mb-ncat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 62
model name    : Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz
stepping      : 4
microcode    : 0x1
cpu MHz       : 2399.998
cache size   : 15360 KB
physical id   : 0
siblings     : 1
core id      : 0
cpu cores    : 1
apicid       : 0
initial apicid : 0
fpu          : yes
fpu_exception : yes
cpuid level  : 13
wp           : yes
flags        : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge m
ca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp
lm constant_tsc arch_perfmon rep_good nopl eagerfpu pni pclmulqdq vmx s
sse3 cx16 pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave
avx f16c rdrand hypervisor lahf_lm xsaveopt vnmi ept fsgsbase tsc_adjust
smep erms
bogomips     : 4799.99
clflush size : 64
cache_alignm : 64
address sizes : 40 bits physical, 48 bits virtual
power management:

```

Figure 1 the specification of the virtual machine.

```

[root@ubuntu-512mb-nyc3-01:~# free -m
              total        used         free       shared    buffers     cached
Mem:           490          444           45           5           57          270
-/+ buffers/cache:
Swap:            0             0             0

```

Figure 2 the specification of the virtual machine

Figure 3 shows the result of “top” command after starting the test:

```

top - 22:10:24 up 8 days, 5:59, 2 users, load average: 0.08, 0.03, 0.05
Tasks: 84 total, 2 running, 82 sleeping, 0 stopped, 0 zombie
%Cpu(s): 10.8 us, 0.7 sy, 0.0 ni, 88.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 501780 total, 457444 used, 44336 free, 59052 buffers
KiB Swap: 0 total, 0 used, 0 free, 278440 cached Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 7431 root        20   0  12452   1532  1268  S  13.0   0.3   0:00.83 bash
    7 root        20   0     0     0     0  S   6.5   0.0   0:34.21 rcu_sched
    1 root        20   0  33500   2212   860  S   0.0   0.4   0:03.25 init
    2 root        20   0     0     0     0  S   0.0   0.0   0:00.00 kthreadd
    3 root        20   0     0     0     0  S   0.0   0.0   0:00.52 ksoftirqd/0
    5 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 kworker/0:0H
    6 root        20   0     0     0     0  S   0.0   0.0   0:03.69 kworker/u2:0
    8 root        20   0     0     0     0  S   0.0   0.0   1:13.75 rcuos/0
    9 root        20   0     0     0     0  S   0.0   0.0   0:00.00 rcu_bh
   10 root        20   0     0     0     0  S   0.0   0.0   0:00.00 rcuob/0
   11 root        rt    0     0     0     0  S   0.0   0.0   0:00.00 migration/0
   12 root        rt    0     0     0     0  S   0.0   0.0   0:05.49 watchdog/0
   13 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 khelper
   14 root        20   0     0     0     0  S   0.0   0.0   0:00.00 kdevtmpfs
   15 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 netns
   16 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 writeback
   17 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 kintegrityd
   18 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 bioset
   19 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 kworker/u3:0
   20 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 kblockd
   21 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 ata_sff
   22 root        20   0     0     0     0  S   0.0   0.0   0:00.00 khubd
   23 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 md
   24 root         0 -20     0     0     0  S   0.0   0.0   0:00.00 devfreq_wq
   27 root        20   0     0     0     0  S   0.0   0.0   0:00.28 khungtaskd
   28 root        20   0     0     0     0  S   0.0   0.0   0:03.59 kswapd0

```

Figure 3 shows the result of “top” command after starting the test.

5.2. Software:

We used the implementation found in this repository: <https://github.com/phillipm/ecpp-aks-primality-proving.git> in github. It was downloaded using this command:

```

root@ubuntu-512mb-nyc3-01:~# git clone https://github.com/phillipm/ecpp-aks-primality-proving.git

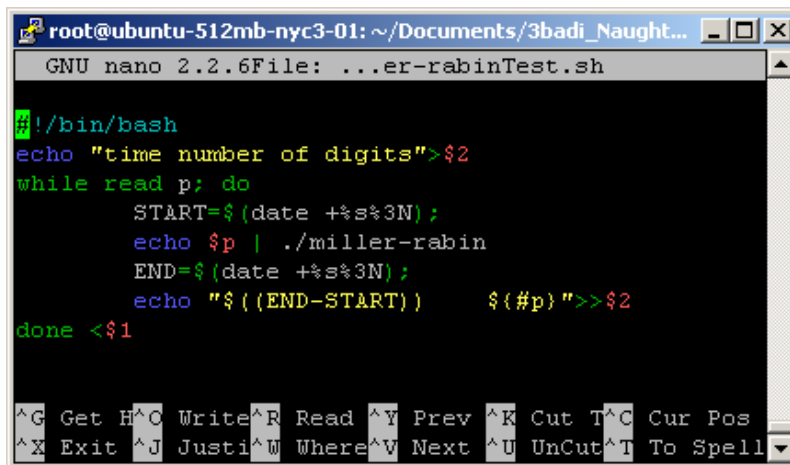
```

After compiling it using “make”, we had two main executable files:

1. aks
2. miller-rabin

They are straightforward to use. Once the command is executed, the number can be inputted through the terminal the result is outputted. The output is either 0 if the inputted number is composite and 1 if the inputted number is prime.

For our requirements, we wrote two bash scripts that would execute the tests automatically as well as calculating the execution time which was done by taking the difference between the times before and after doing the test in milliseconds. Figure 4,5 show a screen shots of the bash scripts:

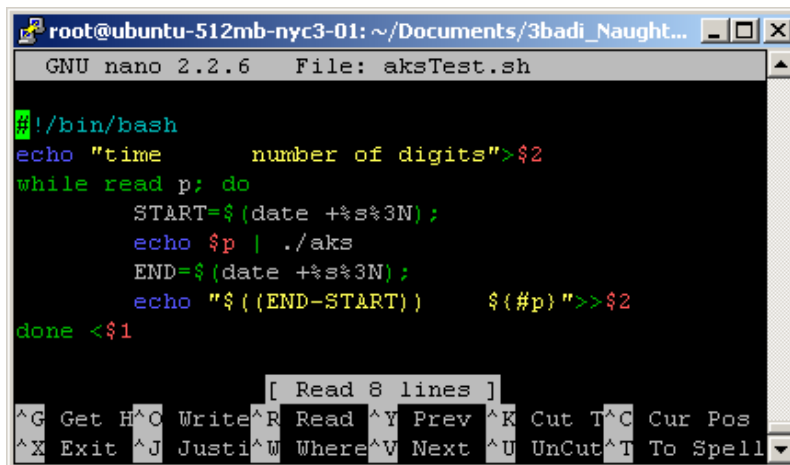


```
GNU nano 2.2.6File: ...er-rabinTest.sh

#!/bin/bash
echo "time number of digits">$2
while read p; do
    START=$(date +%s%3N);
    echo $p | ./miller-rabin
    END=$(date +%s%3N);
    echo "${(END-START)}    ${#p}">>$2
done <$1

^G Get H^O Write^R Read ^Y Prev ^K Cut T^C Cur Pos
^X Exit ^J Justi^W Where^V Next ^U UnCut^T To Spell
```

Figure 4 shows a screen shots of the bash scripts:



```
GNU nano 2.2.6 File: aksTest.sh

#!/bin/bash
echo "time    number of digits">$2
while read p; do
    START=$(date +%s%3N);
    echo $p | ./aks
    END=$(date +%s%3N);
    echo "${(END-START)}    ${#p}">>$2
done <$1

[ Read 8 lines ]
^G Get H^O Write^R Read ^Y Prev ^K Cut T^C Cur Pos
^X Exit ^J Justi^W Where^V Next ^U UnCut^T To Spell
```

Figure 5 shows a screen shots of the bash scripts:

5.3. Usage:

- root@ubuntu-512mb-nyc3-01:~# bash aksTest.sh input.txt output.txt
- root@ubuntu-512mb-nyc3-01:~# bash miller
- -rabinTest.sh input.txt output.txt

5.4. Method of testing:

Measuring execution time for testing prime numbers:

We downloaded few lists of ONLY prime numbers from: <https://primes.utm.edu> . Then, we had to sort them so that each number in a separate line which is required by the bash script. From these lists we chose 1000 numbers randomly using the Linux command:

```
cat p50.txt | sort --random-sort | head -1000 > p50random.txt
```

After that we executed the bash scripts above and analyzed the output file using excel.

Figure 6 below shows a comparison between AKS & Miller-Rabin for 1000 random prime numbers 9 digits each.

In this experiment the execution time was around 3ms in average for Miller-Rabin and 35ms for AKS as in table 1.

Testing random prime numbers 9 digit each

Table 1 Testing random prime numbers 9 digit each for Miller-Rabin and AKS Alg.

	Average	Min	Max
Miller-Rabin	2.748	2	8
AKS	35.644	22	86

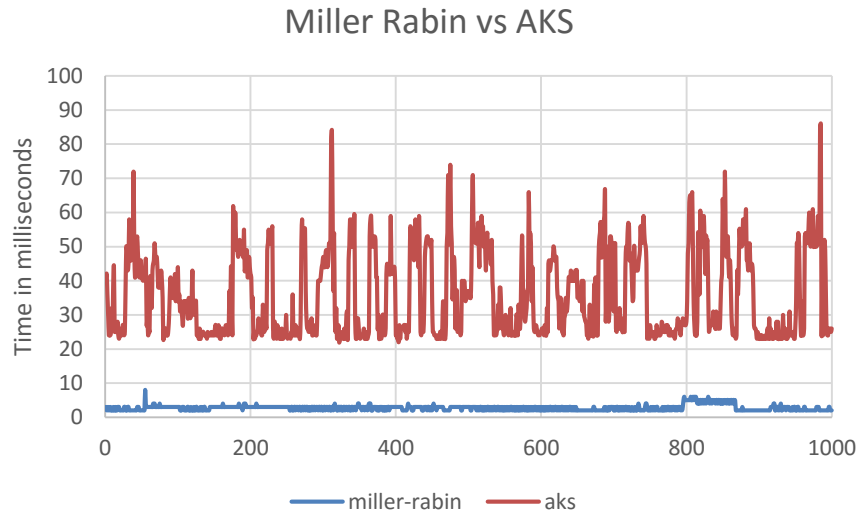


Figure 6 shows a comparison between AKS & Miller-Rabin for 1000 random prime numbers 9 digits each.

Measuring execution time for testing random numbers

For this test we developed a short java program that outputs random numbers with the required length. We chose that the least significant number would be odd to increase the possibility that this number is prime. We used 50 digit 100 numbers and the result as shown figure 7.

Testing random numbers 50 digit each

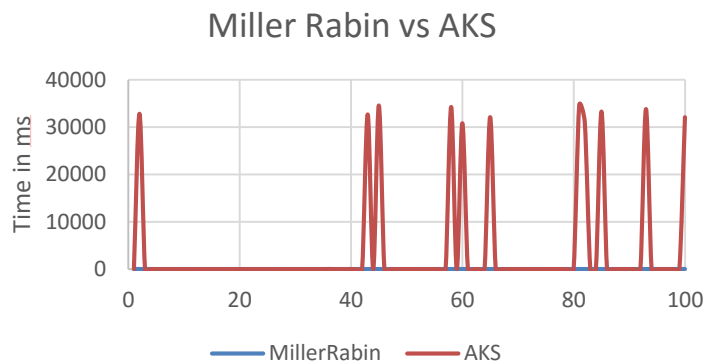


Figure 7 testing 100 numbers 50 digit each.

We clearly see that Miller Rabin in this case is not showing a big difference in execution time than in the previous experiment. However, AKS is. The execution time has around 1000 times the previous one as in table 2.

Table 2 shows average execution times for both algorithms.

	Average	Min	Max
Miller-Rabin	6.97	2	15
AKS	3634.57	7	34584

Testing random numbers 5000 digit each

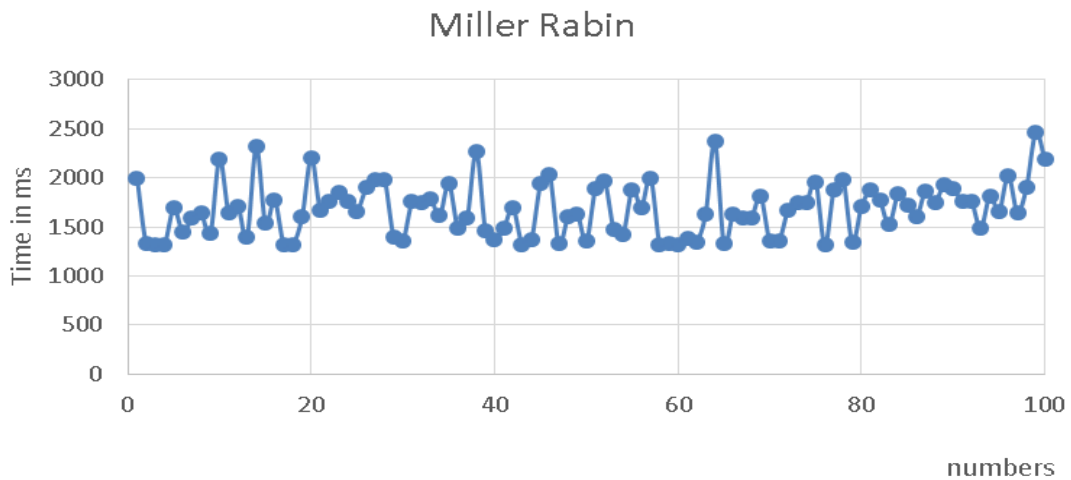


Figure 8 we tested 5000 digit 100 numbers.

	Average	Min	Max
Miller-Rabin	1683.55	1314	2458

In this experiment we tested 5000 digit 100 numbers as in figure 8. Miller-Rabin showed an average execution time around 1.5 seconds which about 100 times the previous result (i.e 50 digit numbers).

However, figure 9 shows the result using the same random 5000 digit numbers with AKS method. The execution time was too long that we had to abort it. It ranged between 5 minutes to 90 minutes per number.

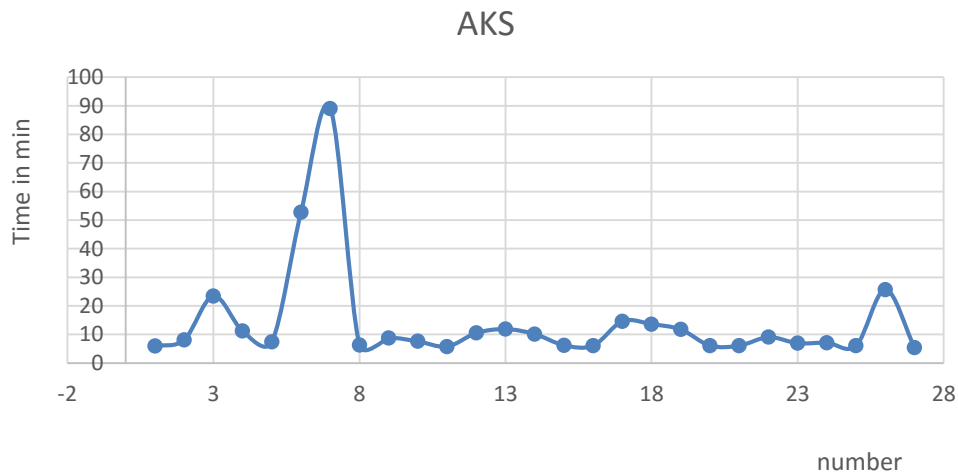


Figure 9 shows random 5000 digit numbers with AKS method.

	Average	Min	Max
Miller-Rabin	14.26	5.42	89.11

However, it is worth mentioning that the Miller-Rabin showed pretty closer results than AKS. The highest execution time in Miller-Rabin was 2 times the lowest. AKS on the other hand, the highest was 18 times the lowest.

Finally, there is no doubt that Miller-Rabin is much faster in execution. However, AKS gives fast results for most composite number although they are not as fast as Miller Rabin.

6. Conclusion

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.

References

- [1] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," 2002 PRIMES is in P , Preprint, IIT Kanpur, August 2002. (<http://www.cse.iitk.ac.in/news/primality.pdf>)...
- [2] M. O. Rabin, "Probabilistic algorithm for testing primality," *J. number theory*, vol. 12, no. 1, pp. 128–138, 1980.
- [3] R. Solovay and V. Strassen, "A Fast Monte-Carlo Test for Primality," *SIAM J. Comput.*, vol. 6, no. 1, pp. 84–85, Mar. 1977.
- [4] L. M. Adleman, C. Pomerance, and R. S. Rumely, "On Distinguishing Prime Numbers from Composite Numbers," *Ann. Math.*, vol. 117, no. 1, p. 173, Jan. 1983.
- [5] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," *Ann. Math.*, vol. 160, pp. 781–793, 2004.
- [6] E. W. Weisstein, "Lucas-Lehmer Test." Wolfram Research, Inc.
- [7] "Pépin's test - Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/P%C3%A9pin%27s_test. [Accessed: 02-May-2016].
- [8] C. Bays and R. H. Hudson, "The segmented sieve of eratosthenes and primes in arithmetic progressions to 1012," *BIT*, vol. 17, no. 2, pp. 121–127, Jun. 1977.

- [9] T. A. Peng, "One Million Primes Through the Sieve," *BYTE*, pp. 243 – 244, 1985.
- [10] D. Gries and J. Misra, "A linear sieve algorithm for finding prime numbers," *Commun. ACM*, vol. 21, no. 12, pp. 999–1003, Dec. 1978.
- [11] "Fermat's little theorem - Encyclopedia of Mathematics." [Online]. Available: https://www.encyclopediaofmath.org/index.php/Fermat_little_theorem. [Accessed: 02-May-2016].
- [12] R. D. Carmichael, "Note on a new number theory function," *Bull. Am. Math. Soc.*, vol. 16, no. 5, pp. 232–238, 1910.
- [13] "http://en.wikipedia.org/wiki/Carmichael_number."
- [14] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff, "The pseudoprimes to $25 \cdot 10^9$," *Math. Comput.*, vol. 35, no. 151, pp. 1003–1003, Sep. 1980.
- [15] <https://github.com/phillipm/ecpp-aks-primality-proving.git>
- [16] <https://cloud.digitalocean.com/droplets>

Appendix A:

Miller Rabin

```
#include "miller-rabin.h"
#include <gmp.h>
#define COMPOSITE 0
#define PRIME 1
#define DEFINITELY_PRIME 2
#define UNDECIDED 3

#define DEFINITELY_PRIME_LIMIT 1000000

/**
 * Set s and d such that  $2^s \cdot d$  is equal to the given n.
 */
void factor_powers_of_2 (mpz_t s, mpz_t d, const mpz_t n) {
    mpz_set_ui(s, 0);
    mpz_set(d, n);
    while (mpz_divisible_ui_p(d, 2)) {
        mpz_add_ui(s, s, 1);
        mpz_divexact_ui(d, d, 2);
    }
}

/**
 * If the number is small enough, perform a sieve test, testing all
 * numbers to up to the sqrt of n to see if they are factors.
 */
int is_definitely_prime(const mpz_t n) {
    if (mpz_cmp_ui(n, DEFINITELY_PRIME_LIMIT) < 0) {
        mpz_t i, limit;
        mpz_init(i);
        mpz_init(limit);
        mpz_sqrt(limit, n);
        for (mpz_set_ui(i, 2); mpz_cmp(i, limit) <= 0; mpz_add_ui(i, i, 1)) {
            if (mpz_divisible_p(n, i)) {
                return COMPOSITE;
            }
        }
        mpz_clear(i);
        mpz_clear(limit);
        return DEFINITELY_PRIME;
    }
}
```

```

return UNDECIDED;
}

/**
 * Run the Miller-Rabin primality test k times. If the number is composite,
 * 0 will be returned. If the number is probably prime, 1 will be returned with
 * a probability of error of  $1 / 4^k$ .
 */
int miller_rabin_is_prime(const mpz_t n, unsigned int k) {
    int retval, i;
    mpz_t s, d, a, x, r, n_minus_one, n_minus_four;

    // Check for simple cases
    if (mpz_cmp_ui(n, 2) == 0 || mpz_cmp_ui(n, 3) == 0)
        return DEFINITELY_PRIME;
    if (mpz_cmp_ui(n, 1) <= 0 || mpz_divisible_ui_p(n, 2))
        return COMPOSITE;

    // Perform a sieve test if possible
    retval = is_definitely_prime(n);
    if (retval == DEFINITELY_PRIME || retval == COMPOSITE)
        return retval;

    gmp_randstate_t state;
    gmp_randinit_default(state);

    mpz_init(s);
    mpz_init(d);
    mpz_init(a);
    mpz_init(x);
    mpz_init(r);
    mpz_init(n_minus_one);
    mpz_init(n_minus_four);

    mpz_sub_ui(n_minus_one, n, 1);
    mpz_sub_ui(n_minus_four, n, 4);

    // Compute s and d
    factor_powers_of_2(s, d, n_minus_one);

    retval = PRIME;
    for (i = 0; i < k; i++) {
        // Pick a random a in the range [2, n - 2]
        mpz_urandomm(a, state, n_minus_four);
        mpz_add_ui(a, a, 2);
        // Set x to  $a^d \pmod n$ 

```

```

mpz_powm(x, a, d, n);
// If x is 1 or n - 1 then get another random number
if (mpz_cmp_ui(x, 1) == 0 || mpz_cmp(x, n_minus_one) == 0) {
    continue;
}

// Test values of r from 1 to s - 1
for (mpz_set_ui(r, 1); mpz_cmp(r, s) < 0; mpz_add_ui(r, r, 1)) {
    // Set x to x ^ 2 % n
    mpz_powm_ui(x, x, 2, n);
    // If x is 1 then the number is composite
    if (mpz_cmp_ui(x, 1) == 0) {
        retval = COMPOSITE;
        break;
    }
    // If x is n - 1 we need to get another random number
    if (mpz_cmp(x, n_minus_one) == 0) {
        retval = UNDECIDED;
        break;
    }
}

if (retval != UNDECIDED) {
    retval = COMPOSITE;
    break;
}
if (retval == UNDECIDED) {
    retval = PRIME;
}

gmp_randclear(state);

mpz_clear(s);
mpz_clear(d);
mpz_clear(a);
mpz_clear(x);
mpz_clear(r);
mpz_clear(n_minus_one);
mpz_clear(n_minus_four);

return retval;
}

```

```
#include "aks.h"
#include <gmp.h>
#include <mpfr.h>
#include <stdlib.h>
#define FALSE 0
#define TRUE 1
#define COMPOSITE 0
#define PRIME 1

int aks_debug = 0;

/**
 * Wrapper function to find the log of a number of type mpz_t.
 */
void compute_logn(mpz_t rop, mpz_t n) {
    mpfr_t tmp;
    mpfr_init(tmp);
    mpfr_set_z(tmp, n, MPFR_RNDN);
    mpfr_log(tmp, tmp, MPFR_RNDN);
    mpfr_get_z(rop, tmp, MPFR_RNDN);
    mpfr_clear(tmp);
}

/**
 * Wrapper function to find the square of the log of a number of type mpz_t.
 */
void compute_logn2(mpz_t rop, mpz_t n) {
    mpfr_t tmp;
    mpfr_init(tmp);

    mpfr_set_z(tmp, n, MPFR_RNDN);
    mpfr_log(tmp, tmp, MPFR_RNDA);
    mpfr_pow_ui(tmp, tmp, 2, MPFR_RNDA);
    mpfr_ceil(tmp, tmp);

    mpfr_get_z(rop, tmp, MPFR_RNDA);
    mpfr_clear(tmp);
}

/**
 * Finds the smallest r such that order of the a modula r which is the
 * smallest number k such that  $n^k = 1 \pmod{r}$  is greater than  $\log(n)^2$ .
 */
void find_smallest_r(mpz_t r, mpz_t n) {
```

```

mpz_t logn2, k, tmp;
mpz_init(logn2);
mpz_init(k);
mpz_init(tmp);

// Compute  $\log(n)^2$  in order to do the comparisons
compute_logn2(logn2, n);
// R must be at least  $\log(n)^2$ 
mpz_set(r, logn2);

int found_r = FALSE;
while (!found_r) {
    found_r = TRUE;
    // Check several values of k from 1 up to  $\log(n)^2$  to find one that satisfies the equality
    for (mpz_set_ui(k, 1); mpz_cmp(k, logn2) <= 0; mpz_add_ui(k, k, 1)) {
        // Compute  $n^k \pmod r$ 
        mpz_powm(tmp, n, k, r);
        // If it is not equal to 1 than the equality  $n^k = 1 \pmod r$  does not hold
        // and we must find test a different value of k
        if (mpz_cmp_ui(tmp, 1) == 0) {
            found_r = FALSE;
            break;
        }
    }
    // All possible values of k were checked so we must start looking for a new r
    if (!found_r) {
        mpz_add_ui(r, r, 1);
    }
}

mpz_clear(logn2);
mpz_clear(k);
mpz_clear(tmp);
}

/**
 * Return true if there exists an a such that  $1 < \gcd(a, n) < n$  for some  $a \leq r$ .
 */
int check_a_exists(mpz_t n, mpz_t r) {
    mpz_t a, gcd;
    mpz_init(a);
    mpz_init(gcd);

    int exists = FALSE;

    // Simply iterate for values of a from 1 to r and see if equations hold for the gcd of a and n

```



```

for (mpz_set_ui(a, 1); mpz_cmp(a, r) <= 0; mpz_add_ui(a, a, 1)) {
    mpz_gcd(gcd, a, n);
    if (mpz_cmp_ui(gcd, 1) > 0 && mpz_cmp(gcd, n) < 0) {
        exists = TRUE;
        break;
    }
}

mpz_clear(a);
mpz_clear(gcd);

return exists;
}

/**
 * Return the totient of op, which is the count of numbers less than op which are coprime to
 * op.
 */
void totient(mpz_t rop, mpz_t op) {
    mpz_t i, gcd;
    mpz_init(i);
    mpz_init(gcd);
    mpz_set_ui(rop, 0);

    // Simply iterate through all values from op to 1 and see if the gcd of that number and op
    // is 1.
    // If it is then it is coprime and added to the totient count.
    for (mpz_set(i, op); mpz_cmp_ui(i, 0) != 0; mpz_sub_ui(i, i, 1)) {
        mpz_gcd(gcd, i, op);
        if (mpz_cmp_ui(gcd, 1) == 0) {
            mpz_add_ui(rop, rop, 1);
        }
    }

    mpz_clear(i);
    mpz_clear(gcd);
}

/**
 * Returns sqrt(totient(r)) * log(n) which is used by step 5.
 */
void compute_upper_limit(mpz_t rop, mpz_t r, mpz_t n) {
    mpz_t tot, logn;
    mpz_init(tot);
    mpz_init(logn);

```

```

totient(tot, r);
if (aks_debug) gmp_printf("tot=%Zd\n", tot);
mpz_sqrt(tot, tot);

compute_logn(logn, n);
mpz_mul(rop, tot, logn);

mpz_clear(tot);
mpz_clear(logn);
}

/**
 * Multiplies two polynomials with appropriate mod where their coefficients are indexed
 into the array.
 */
void polymul(mpz_t* rop, mpz_t* op1, unsigned int len1, mpz_t* op2, unsigned int len2,
mpz_t n) {
    int i, j, t;

    for (i = 0; i < len1; i++) {
        for (j = 0; j < len2; j++) {
            t = (i + j) % len1;
            mpz_addmul(rop[t], op1[i], op2[j]);
            mpz_mod(rop[t], rop[t], n);
        }
    }
}

/**
 * Allocates an array where each element represents a coefficient of the polynomial.
 */
mpz_t* init_poly(unsigned int terms) {
    int i;
    mpz_t* poly = (mpz_t*) malloc(sizeof(mpz_t) * terms);
    for (i = 0; i < terms; i++) {
        mpz_init(poly[i]);
    }
    return poly;
}

/**
 * Frees the array and clears each element in the array.
 */
void clear_poly(mpz_t* poly, unsigned int terms) {
    int i;
    for (i = 0; i < terms; i++) {

```

```

    mpz_clear(poly[i]);
}
free(poly);
}

/**
 * Test if  $(X + a)^n \neq X^n + a \pmod{X^r - 1, n}$ 
 */
int check_poly(mpz_t n, mpz_t a, mpz_t r) {
    unsigned int i, terms, equality_holds;

    mpz_t tmp, neg_a, loop;
    mpz_init(tmp);
    mpz_init(neg_a);
    mpz_init(loop);

    terms = mpz_get_ui(r) + 1;
    mpz_t* poly = init_poly(terms);
    mpz_t* ptmp = init_poly(terms);
    mpz_t* stmp;

    mpz_mul_ui(neg_a, a, -1);

    mpz_set(poly[0], neg_a);
    mpz_set_ui(poly[1], 1);

    for (mpz_set_ui(loop, 2); mpz_cmp(loop, n) <= 0; mpz_mul(loop, loop, loop)) {
        polymul(ptmp, poly, terms, poly, terms, n);
        stmp = poly;
        poly = ptmp;
        ptmp = stmp;
    }

    mpz_t* xMinusA = init_poly(2);
    mpz_set(ptmp[0], neg_a);
    mpz_set_ui(ptmp[1], 1);
    for (; mpz_cmp(loop, n) <= 0; mpz_add_ui(loop, loop, 1)) {
        polymul(ptmp, poly, terms, xMinusA, 2, n);
        stmp = poly;
        poly = ptmp;
        ptmp = stmp;
    }
    clear_poly(xMinusA, 2);

    equality_holds = TRUE;
    if (mpz_cmp(poly[0], neg_a) != 0 || mpz_cmp_ui(poly[terms - 1], 1) != 0) {

```

```

    equality_holds = FALSE;
}
else {
    for (i = 1; i < terms - 1; i++) {
        if (mpz_cmp_ui(poly[i], 0) != 0) {
            equality_holds = FALSE;
            break;
        }
    }
}
}

clear_poly(poly, terms);
clear_poly(ptmp, terms);

mpz_clear(tmp);
mpz_clear(neg_a);
mpz_clear(loop);

return equality_holds;
}

/**
 * Run step 5 of the AKS algorithm.
 */
int check_polys(mpz_t r, mpz_t n) {
    mpz_t a, lim;
    mpz_init(a);
    mpz_init(lim);

    int status = PRIME;
    if (aks_debug) gmp_printf("computing upper limit\n");
    compute_upper_limit(lim, r, n);
    if (aks_debug) gmp_printf("lim=%Zd\n", lim);
    // For values of a from 1 to sqrt(totient(r)) * log(n)
    for (mpz_set_ui(a, 1); mpz_cmp(a, lim) <= 0; mpz_add_ui(a, a, 1)) {
        if (!check_poly(n, a, r)) {
            status = COMPOSITE;
            break;
        }
    }
}

mpz_clear(a);
mpz_clear(lim);

return status;
}

```

```

int aks_is_prime(mpz_t n) {

    // Perform simple checks before running the AKS algorithm
    if (mpz_cmp_ui(n, 2) == 0) {
        return PRIME;
    }

    if (mpz_cmp_ui(n, 1) <= 0 || mpz_divisible_ui_p(n, 2)) {
        return COMPOSITE;
    }

    // Step 1: Check if n is a perfect power, meaning  $n = a^b$  where a is a natural number and
    b > 1
    if (mpz_perfect_power_p(n)) {
        return COMPOSITE;
    }

    // Step 2: Find the smallest r such that  $or(n) > \log(n)^2$ 
    mpz_t r;
    mpz_init(r);
    find_smallest_r(r, n);

    if (aks_debug) gmp_printf("r=%Zd\n", r);

    // Step 3: Check if there exists an a <= r such that  $1 < (a,n) < n$ 
    if (check_a_exists(n, r)) {
        mpz_clear(r);
        return COMPOSITE;
    }

    if (aks_debug) gmp_printf("a does not exist\n");

    // Step 4: Check if  $n \leq r$ 
    if (mpz_cmp(n, r) <= 0) {
        mpz_clear(r);
        return PRIME;
    }

    if (aks_debug) gmp_printf("checking polynomial equation\n");

    // Step 5
    if (check_polys(r, n)) {
        mpz_clear(r);
        return COMPOSITE;
    }
}

```

```
mpz_clear(r);

// Step 6
return PRIME;
}
```

Prime Numbers

2637155093855448570938472380222706770724032300690222174268886576847799
3502106023033054133313527480707193900521496593412042151923156710669524
9539904735396139061088635385740995937345785513679594618961226387523617
0318795134273481432527070687247510614508784835081794376527418779967081
2728687941533655420783986217464273073452151853539860061889978438992120
0512721760392067642122784928177090842157844859789408286526992685913987
6177450902923782844336477175056440134903116960986898377982693474283025
2487656646413398123494571263273773332343627385186978077156118468120130
0492965994675324604818209093611917316894597113897100760536269910684681
0974806048375647698291140982545670981499914586671229511548772003754040
3890434090747607367175686736151151323727908304566954998750941416929683
8890299104566466816216720153900061518634563523592079034028443149663200
2205445885507584940654298624210671530604325387038782380914960598549899
0937113294715949033121664189746049555023910586310424353449126107093581
8337767469147954797743676159452115902380586067016649156812528874803647
6140345058263111084365983560977195316815903885523336333911086935479590
5423054162139177836170483660714885131032110587995635673029843471133728
3471109420076256839892602868551847798075630807160127986958780830317363
1811042642889713849373147632276540618659525259698517544219776178245902
9053180104435856117990241409768101484527497939373197617258939395030300
6953824814828568253596738055190162189783664953104304785033181349423903
242876247571673409022912327627

1912224376975798017412972537839234556299282436200249018470363668484368
4576558748040571261417728968644919375430791040965203039973879931669266
7906116208200303429177415541112738120688056158254584636768157378552709
1395182718711173909093104513223043359753949043793832660644364252586280
1830834492010007905848879513390820143862995442492735997910253406834165
8872193416031279796260516544906288466240671104651988665806233690286736
6151569921374200502469990256914624210967314960937749861238011914660550
7240320492976224459047399010708035928237897614340247350234823857826463
3591098744417667299359301835778151217129887784017029802499018933741097
9391967455653590464108500407238737221127606724167043454732630670650996
8570238429190702488473483121303594063013149525116210606474011566005508
2328671475511036495102429548358263069090603479481382373319600314932354

3349394181944505542087282285137833245753224608195638619296979668366223
2627735402634586861740259240751992513488752060495576026450958788554734
3667675075895807496506808035811082777326173051514056516820311381030512
1463418534382898604606472945560314910008629236906926568592697611713402
1629239369283061331454542091851568759298791297834212310171547602842391
8100381963933367041467658469410304410865365421186877470509895235622540
0813312877508759155028213852232008127949864768024395676162689065214049
7427366412687852400265007075698227867959496059743637403404635892003761

Appendix B

5000 digit random odd number generator:

```
package primeTest;
import java.util.Random;
public class PrimeTest {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        for(int j=0; j<100; j++){
            System.out.print(randomint(1,9,false));
            for(int i=0; i< 4998; i++){
                System.out.print(randomint(0,9,false));
            }
            System.out.println(randomint(1,9,true));
        }
    }
    //random.nextInt(max - min + 1) + min
    private static int randomint(int min, int max, boolean odd) {
        // TODO Auto-generated method stub
        int result;
        Random rn = new Random();
        result = min + rn.nextInt(max-min+1);
        if(odd){
            if(result % 2 == 0){
                result++;
                if(result > max){
                    result = result - 2;
                }
            }
        }
        return result; }
}
```