Abstract— The most awaited solution to an old problem of finding whether a given number is prime was presented in August 2002 when three Indian researchers named Manindra Agrawal, Neeraj Kayal, and Nitin Saxena presented deterministic polynomial-time algorithm AKS in their Paper “PRIMES is in P”. The algorithm is first of its kind and gives only two results whether the given number is prime or composite. Up until now, every test for primality has either been non-deterministic (probabilistic) or of exponential complexity.

The project is targeted towards understanding of AKS Algorithm and its implementation. It also analyzes the obtained results and concludes with further possible improvements.

Index Terms-AKS, JAVA, Primes

I. INTRODUCTION

PRIMES is in P:- the title itself indicates that the algorithm falls in class P. That is given an input the algorithm states with 100% accuracy whether the number is prime or composite in polynomial time. The complexity must be polynomial, that is to say: as $n$ increases, the time it takes to come up with an answer is less than $n^c$ where $c$ is a constant. The time complexity of the algorithm is known to be $O(\log^{12} n)$. There are various variants of this algorithm which appeared giving more efficient version of AKS algorithm.

I used JDK1.4.2 to implement this algorithm. Although Java has various classes to make this project easy to implement there is no library for polynomial calculation and makes the program very slow. Various implementations issues are discussed further based on the results obtained using the program. Different ways to optimize the code are discussed and then concluded.

II. PRIMES IN PUBLIC CRYPTOGRAPHY SYSTEMS

RSA, and all other asymmetric ciphers, depend upon the existence of some kind of trapdoor function. This function is supposed to be practically irreversible, although reversibility might be possible in theory.

The RSA cipher uses the fact that, while it is easy to compute the product of two primes, factorizing a large number into its prime factors seems to be essentially impossible. There are factorization methods available but all of these take a long time for large numbers. As the time necessary to factor a large number is longer than the usefulness of the data encrypted we consider this problem unsolvable [4].

It is this property of prime numbers that have occupied various brains for decades.

III. EXPLANATION OF CLASS P AND OTHER COMPLEXITY CLASSES

A. What is P?

P stands for a class of decisional problems with only two solutions: yes or no, for which any instance of the problem can be solved using deterministic algorithm in polynomial time depending on the size of input N. The size of input is measured in bits for the problem of testing for Primality. The number of bits needed to represent n is $\log(n)$, where $\log$ represents logarithm to the base 2.

Time here is not defined in seconds, nor in computer cycles of a particular computer processor, but rather in an abstract way as the number of elementary operations needed to execute an algorithm.

The notion of step is defined in a way that is as machine-independent as possible, assuming only a single-processor machine. Each operation takes single step time and each line in the code takes constant time to execute.

For a routine to be of complexity P it has to have 2 qualities:

a. The routine must be deterministic, which is to say it ALWAYS returns the exact right answer. It is much easier to find routines that are non-deterministic, which are right 99.999999% of the time, good enough for most uses. Such routines belongs to class NP.

b. The complexity must be polynomial i.e. as $n$ increases, the time it takes to come up with an answer is less than $n^c$ where $c$ is a constant. The need for the second one is evident when we consider number of bits growing. See how quickly $2^n$ grows as $n$ gets bigger (2, 4, 8, 16, 32, 64...) with the growth of $n^2$ (1, 4, 9, 16, 25, 36,), while in the early stages they are...
about the same, by the time \( n \) reaches value of 6 the former is already nearly twice as big, and it only gets worse from there. (49 vs. 128, 64 vs. 256, etc.)

B. Other Complexity Classes

- **BPP** (bounded probabilistic polynomial-time) is the class of problems which can be solved using a randomized polynomial-time algorithm with at most an exponentially small error of probability on every input.
- **RP** (Randomized Polynomial-Time): RP is a class similar to BPP, except that the algorithms that solve the problems in this class error only on inputs for which the answer is really supposed to be "yes".
- **coRP** (Complementary Randomized Polynomial-Time): coRP is the analogous class of problems for which algorithms that solve the problems only error on inputs which are really suppose to be "no". It is complementary to RP class.
- **ZPP** (zero-error probabilistic polynomial-time) is the class of problems which can be solved with zero error, but for which the algorithm may, with low probability, run for a long time.
- **NP** (Non deterministic Polynomial): the class of problems that can be solved in polynomial time by a non-deterministic algorithm.

C. Relation between different classes

Let \( A \subseteq B \) denote the fact that A is a subset of the set B. A hierarchy exists among the different complexity classes mentioned:

- \( P \subseteq \text{ZPP} \subseteq \text{RP} \subseteq \text{NP} \)
- \( P \subseteq \text{ZPP} \subseteq \text{co-RP} \subseteq \text{BPP} \)

For example this says that any problem which is in P is also in ZPP, RP, co-RP and NP. However a problem that is in NP is not necessarily in RP, co-RP, ZPP and P. There is no known direct relation between BPP and NP, however one can prove for example that if P = NP then BPP = P, thus if we can one day prove that P=NP, the whole hierarchy collapses.

A problem is said to be in a certain class if the best known algorithm for solving the problem satisfies the criteria of that class. Thus through time, more efficient algorithms for problems can be found and the problem may be classified in a different class.

A long standing problem is to decide whether or not \( P = NP \), that is to determine whether all the problems that are in NP are in fact in P as well. A mathematical proof demonstrating whether P is equal to NP or not is worth one million dollars: http://www.claymath.org/prizeproblems/index.htm [5].

D. Brief History

The starting point for many methods to find prime numbers has been Fermat’s Little Theorem. It says that for every prime number \( n \) and every number \( a \) co-prime to \( n \) one has the relation

\[ a^n \equiv 1 \mod n \]

It is no wonder, then, that refinements of this criterion are the basis of important algorithms.

An elementary probabilistic algorithm of Miller and Rabin from 1976 makes use of a random number generator and shows after \( k \) runs either that the number is certainly composite or that the number is prime with high probability. In practice the algorithm is very fast, and it finds application in cryptography and e-commerce for the production of “industrial-grade primes” (belongs to co-RP).

A deterministic algorithm of Adleman, Pomerance, and Rumely from 1983, which uses much more theory and a generalization of Fermat’s Little Theorem to integers in cyclotomic fields, completely characterizes the prime numbers. The best deterministic algorithm prior to August 2002, it has running time of super polynomial order \( (\log n)^{o(\log \log \log n)} \). The triple logarithm in the exponent grows so slowly, however, that concrete versions of the algorithm have had excellent success in the past producing primes with thousand of decimal digits.

Another class of modern algorithms uses elliptic curves. Adleman and Huang, in a very difficult and technical 1992 monograph, were able to give a probabilistic algorithm with polynomial running time which after \( k \) iterations either gives a definitive answer (with no possibility of error) or gives no answer in the latter case (belongs to ZPP).

With this background, and in view of the level of difficulty that had been reached and the absence of further successes in over ten years, it was hardly to be expected that there could be a short, elegant resolution of the question that would be understandable by everyone [10].
IV. EXPLANATION OF AKS ALGORITHM

A. AKS Algorithm

1. Input: Integer n > 1
2. if (n has the form $a^b$ with $b > 1$) then output composite
3. $r = 2$
4. while ($r < n$)
   5. If $\gcd(n, r) \neq 1$ then output composite;
   6. if ($r$ is prime > 2)
      7. let $q$ be the largest factor of $r - 1$;
      8. if ($q > 4 \sqrt{r \log n}$) and ($n \equiv 1 \pmod{r}$)
         9. break.
   10. $r = r + 1$
5. for ($a = 1$ to $2 \sqrt{r \log n}$)
   12. if ($x - a \equiv (x^n - a) \pmod{x^n - 1, n}$)
      13. output composite;
   14. output prime;

B. Explanation

The algorithm has 3 filters. Line 1 checks whether $n$ is of the form $a^b$, line 5 checks whether GCD of $r$ and $n$ is 1 if not i.e. there is a common factor between them proving that $n$ is not a prime and line 14 which finds out all the composite numbers and only prime numbers pass this test of negative congruence. There are two loops: While loop finds a useful prime $r$ and for loop tests negative congruence.

Line 1
It might take input $n$ as parameter and check it for Primality or generate a random number $n$ of the specified size and check that for Primality.

Line 2
Line 2 determines if $n$ is a power of an odd prime (the only case that the rest of the routine will fail on) and returns COMPOSITE if it is.

If $r$ is sufficiently large, the only composite numbers that will pass the test at line 12 are powers of odd primes which are tested here at line 2 using a much easier test also in $P$.

Line 3-10
It starts by initializing $r$ to 2. The while loop is executed until a small prime $r \in (\log^n n)$ is found such that $r - 1$ has a large prime divisor $q$, which divides the multiplicative order $n$ modulo $r$. It also checks whether $r$ is prime. Any method could be used to check Primality of $r$ as it is a small number. Even Brute force method of dividing $r$ by $a$ ($a = 2$ to $\sqrt{r}$ ) and checking the remainder, will not increase the execution time by much. Then the algorithm finds Sophie Germain pair of primes and sets $r$ to highest of 2. They (AKS team) proved that if you find a Sophie Germain pair of primes $q$ and $2q + 1$ such that $q > 4(\sqrt{2q + 1})$ log $n$, then $r$ does not need to exceed $2(\sqrt{2q + 1})$ log $n$. The negative congruence check in for loop uses $r$ as its upper bound. Thus, $r$ assures that the algorithm works in $P$.

Line 11-14
This is based on the Fermat’s little Theorem:

Old Idea: $(x - a)^p \equiv (x^p - a) \pmod{p}$

This is true if and only if a and p are relatively prime and p is prime. If $p$ is prime, then $p$ divides the binomial coefficients $\binom{r}{r}$ for $r = 1, 2, ..., p - 1$. If p is prime and a is relatively prime to p then all coefficients of p will be divisible by p except the first term $x^p$ and last term $-a^p$. The problem is that it still takes exponential time to determine if p isprime using this test, because we need to calculate $p - 1$ coefficients and divide all by p to see if there is a remainder producing time complexity $O(n) = O(2^{\log_2 n})$

New Idea: $(x - a)^p \equiv (x^p - a) \pmod{x^n - 1, n}$

In 1999, Dr. Agrawal proposed a Monte Carlo test. Based on this theorem to shorten the test and bring it in $P$, the polynomial $(x - a)^p$ would be shortened by applying modulo $x^n - 1$, which is just a fancy but short way of saying "lets just look at the last $r$ terms of $(x - a)^p". If $r$ is sufficiently large, the only composite numbers that will pass are powers of odd primes that can be tested for separately using a much easier test, also in $P$. Line 2 of the algorithm exactly does that. The new idea could be implemented using:

- Fast Fourier Transform.
- Square and multiply.
- Creating and using Java API with functions such as mod and power on BigPolynomials [6].

V. IMPLEMENTATION

My implementation of AKS algorithm is developed using Java 2 Standard Development Kit version 1.4.2.0 on laptop with following configuration.

Processor: AMD Athlon (2.4 GHZ)
Ram: 256 MB
Os: Windows XP.

All the results were obtained on the same;

Java programs are platform independent and can run on variety of computers using a range of operating systems. Java program runs on a standardized
A hypothetical computer that is called the Java Virtual Machine, which is emulated inside the computer.

A Java compiler converts the Java source code into a binary program consisting of byte codes. Byte codes are instructions for JVM.

An interpreted java program would typically run at only one tenth of the speed of an equivalent program implemented using native machine instructions.

Java has a class java.math which further contains 2 classes BigInteger & BigDecimal, which are immutable arbitrary large but very slow. BigInteger class further contains methods for finding GCD of two numbers, isProbablePrime() checking whether given number is probable prime with given amount of certainty. The probability that this BigInteger is prime exceeds $1 - \frac{1}{2^{1000}}$.

Java also contains java.security.SecureRandom class. This class provides a cryptographically strong pseudo-random number generator (PRNG). Like other algorithm-based classes in Java Security, SecureRandom provides implementation-independent algorithms, whereby a caller (application code) requests a particular PRNG algorithm and is handed back a SecureRandom object for that algorithm. It is also possible, if desired, to request a particular algorithm from a particular provider. My implementation uses SHA1PRNG supplied by the SUN provider. It computes the SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used. The random object generated needs a seed as a parameter. If the same seed is provided to generate two prime numbers it will give a pair of primes with same size which could be used for public key systems like RSA.

Java does not provide functions for performing power and log operations over BigInteger. These functions were free and made available by www.optimtika.se (package org.ojalgo). Java does not provide class for polynomial calculations and thus square and multiply method was used to check congruence which makes the system very slow and impractical. If in future java or any third party provides Polynomial API [6] the working of AKS could be improved by a great amount and AKS could be of some importance to Cryptography.

VI. TIME COMPLEXITY AND ANALYSIS OF OBTAINED RESULTS

A. Time Complexity

Complexity theory is a field in theoretical computer science which attempts to quantify the difficulty of computational tasks and tends to aim at generality while doing so. "Complexity" is measured by various natural computing resources, such as the amount of memory needed, communication bandwidth, time of execution, etc. For the problem of determining the Primality of an integer, the resource we examine is the time of execution.

In 2000, Kayal and Saxena used an unproven conjecture to determine that $r$ does not need to exceed $4(\log^2 n)$. Thus, the time complexity of their routine is an amazing $O(\log^3 n)$ and easily in $P$. Still, an unproven conjecture is an unproven conjecture. Instead of trying to prove the conjecture, what they did was use a proven prime theorem relating to Sophie Germain Primes. They proved that if you find a Sophie Germain pair of primes $q$ and $2q+1$ such that $q > 4\sqrt{2q + 1} \log N$, then $r$ does not need to exceed $2\sqrt{2q + 1} \log N$. (Prime $P$ is said to be a sophie Germain prime if both $P$ and $2P+1$ are prime). The biggest negative of this fix, is that the resulting test for primes becomes a recursive test. In order to test for prime $p$ we have to test for prime $r$, etc. This causes little $O(\log^3 n)$ test to grow to an unwieldy but still polynomial $O(\log^{12} n)$.

Times according to steps [8]:

- Line 2 is used to check whether $n$ is of the form of $a^b$ takes $O(\log(n)^3)$ time.
- The while loop used to find $r$ takes $O(\log(n)^6)$ time.
- Time taken by for loop does modular computation over polynomials and takes $O(\log(n)^{12})$ time.

B. Useful Prime $r$

The for loop uses $r$ to decide the upper bound. Now some results obtained from the program clearly show that even if $n$ doubles in the number of bits, $r$ just grows by few digits. It is $r$ that assures that the algorithm runs in $P$.

<table>
<thead>
<tr>
<th>$K$</th>
<th>$N=2^k$</th>
<th>$R$ Useful prime</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>$2^{64}$</td>
<td>262127</td>
</tr>
<tr>
<td>128</td>
<td>$2^{128}$</td>
<td>1047587</td>
</tr>
<tr>
<td>256</td>
<td>$2^{256}$</td>
<td>4192547</td>
</tr>
<tr>
<td>512</td>
<td>$2^{512}$</td>
<td>16773479</td>
</tr>
<tr>
<td>1023</td>
<td>$2^{1023}$</td>
<td>66970103</td>
</tr>
</tbody>
</table>

Fig. 1. Useful Prime $r$ for k bits.

Note: $r$ does not exist for small $N$ in the range of 0 to 11700 and from the range of 1170 to 11808. Thus,
lower bound for $N$ is 11809., cf. Table 3.1[7]. This is due unavailability of $q$ which is the largest factor of $r-1$ such that $q \geq 4\sqrt{r \log n}$.

How small can $r$ be expected?

Now $q \geq 4\sqrt{r \log n}$

$$\rightarrow \frac{q}{4\sqrt{r}} \geq \log n$$

If $r$ is co-Sophie German prime and $q$ is $(r-1)/2$ than

$$\rightarrow \frac{r/2}{4\sqrt{r}} \geq \log n$$

$$r \geq 64(\log n)^2$$

C. AKS CPU times for different bits $N$

The running time of AKS algorithm is expected to be very different between when it deals with a prime number or a composite number. When the algorithm checks a prime number, it always iterates in the for loop until the upper bound $a=2\sqrt{(r)\log n}$. On the other hand, a composite number is detected far before the for loop reaches the upper bound of $a$ or even before that during power of odd primes check done at line 2 or the gcd check.

While loop for finding $r$ takes only fraction of the total time. Most of the time is taken by the for loop. It uses so many modular calculations of polynomials that the algorithm runs for a very long time.

<table>
<thead>
<tr>
<th>K(bits)</th>
<th>$N=2^k$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.06(secs)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.06(secs)</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>0.09(secs)</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>0.13(secs)</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>0.29(secs)</td>
</tr>
<tr>
<td>6</td>
<td>61</td>
<td>0.97(secs)</td>
</tr>
<tr>
<td>7</td>
<td>127</td>
<td>5.57(secs)</td>
</tr>
<tr>
<td>8</td>
<td>251</td>
<td>31.99(secs)</td>
</tr>
<tr>
<td>9</td>
<td>499</td>
<td>206.41(secs)</td>
</tr>
<tr>
<td>10</td>
<td>1019</td>
<td>1304.74(secs)</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>Years</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>Years</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>Years</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>Years</td>
</tr>
<tr>
<td>1023</td>
<td></td>
<td>Years</td>
</tr>
</tbody>
</table>

The results show that time grows exponentially as $k$ grows by one bit. I could have used Forecast function in Ms Excel to get a linear trend using existing values, but it did not gave correct results due to lack of existing data. The prediction that it will take years to produce prime numbers with bits greater than 64 could be verified with 3.4.2 Measurement of CPU time [7]. This is not due to AKS algorithm but due to Java and lack of availability of proper API.

Fig. 1. “Speed of AKS in Java in seconds”

VII. AKS VARIANTS

After the original paper was published various variants with improvements on AKS algorithm appeared [5]. The general trend is the lowering of the complexity exponent $k$, which began as an unconditional $k = 12+\varepsilon$ with the original AKS paper and now
stands at $k = 4^{+\varepsilon}$ for certain input integers, or a similar low exponent for general input integers.

Early in March 2003, Agrawal, Kayal, and Saxena posted on the Web a revision of their preprint: It contains the improvements and culminates in the new time-complexity bound $O^*(\log^{7.5} n)$, cf. Theorem 5.3.[10]

VIII. CONCLUSION

Thus, AKS algorithm is a very important one in complexity theory, but probably has no (practical) impact in cryptography right now. Lower-quality “industrial-grade primes” with 512 binary digits can be produced in a fraction of a second using the Miller-Rabin test on an 2GHz PC. If required, their Primality can actually be proved in a couple of seconds with the ECPP- method of Atkin-Morain based on elliptic curves. On the other hand, because of the high time cost of the AKS algorithm, same will take years.

Various known classical algorithms along with the knowledge of what is known in the past of long integers could be employed to get an ultimately fast variant of AKS[9].

Since the AKS is so unexpectedly new, we may confidently anticipate improved capabilities after further maturation of the algorithm.

ACKNOWLEDGMENT

My Special thanks to Kris Gaj for providing such a challenging task and his continuous help in solving it.

REFERENCES


