BACHELOR

Deterministic primality testing

Weenink, T.J.

Award date:
2015

Deterministic Primality Testing

Tim Weenink
0775602
t.j.weenink@student.tue.nl

May 18, 2015
Bachelor eindproject: 2WH40
TU Eindhoven

Chapter 1

Deterministic primality testing

1.1 Introduction

Prime numbers have been fascinating people for a really long time. The numbers, only divisible by one and itself, are used in many fields of mathematics. One of those fields is cryptology, which uses large numbers for computing encryption keys. What's nice about prime numbers, is that they are relatively easy to compute but that large numbers are much harder to factor into primes. This makes prime numbers extremely useful. But how would we know if numbers are prime or not?

The ancient Egyptians started wondering about ways to find prime numbers. One of the most intuitive ways to find prime numbers is to cancel out all numbers that cannot be prime. This method is also known as the sieve of Eratosthenes: it simply cancels out all multiples of 2, 3, 5, $\ldots$, $\sqrt{n}$ where n is the number of integers you want to test. You can imagine that this method will give you 100% certainty whether or not an integer is prime, but also that it will take a lot of time to compute when n is big.

Many years later, important discoveries about the identities of prime numbers were made. One of these discoveries is also known as Fermat's little theorem: $a^p \equiv a \pmod{p}$ for a a positive integer and p prime. Because of the fact that this theorem holds for every prime it can be used to exclude numbers from potential primes. However, the theorem only holds one way. This means that this test cannot tell with certainty whether or not a number is prime. There are more probabilistic primality tests, like the Miller-Rabin test. This test also relies on an identity that holds for primes but this identity also holds for some composite numbers. This means that the accuracy of the test can become high, but never 100%. The same holds for the Solovay-Strassen test, which is based on a theorem proved by Euler.

In the 1930s Lehmer improved a primality test that had been invented by Lucas before: the Lucas-Lehmer test was born. Although this primality test is deterministic, it only works for a small group of numbers: the so-called Mersenne numbers $M_p = 2^p - 1$ where p is a known prime number. We can use this test to find a group of primes, but we prefer to know all prime numbers. This test has been improved by Riesel to improve the set of possible primes that can be found with this test to numbers of the form $h2^p - 1$. This is an improvement to the Lucas-Lehmer test, but we would preferably see a test for which the input number does not have to meet strict requirements.

About 50 years later, the elliptic curve algorithm was published. Although this algorithm is not deterministic, it is used very often in primality proving. The main reason for this is that the algorithm is the fastest for general numbers, i.e. there are no restrictions on the input number other than having to be a positive integer. The algorithm has been improved, so that it will return a prime certificate which can be verified. However, this made the algorithm much slower and thus less attractive to use in practice. [7]

In the 21^{st} century however, there was a breakthrough. It was only a few years ago, in 2002, that a document (”*PRIMES is in P*”) was published in which a new primality testing algorithm was introduced. This algorithm was named after the 3 Indians who had produced it: Agrawal, Kayal and Saxena. Unlike the prime tests mentioned earlier, this algorithm can tell with 100% certainty whether or not a number n is prime. The nice things about this algorithm are that it is relatively simple, it works for all integers greater than 0 and it runs in polynomial time. The latter is a huge improvement to the previously mentioned sieve of Eratosthenes, which runs in exponential time.

The main topic of this paper will be the AKS test. You can find the general algorithm, the correctness and the complexity of the algorithm. Also, there is an implementation with corresponding results in which you can see the actual run time compared to other implementations of the AKS algorithm. In the end, two other algorithms are explained and analysed for their complexity: the Lucas-Lehmer test and the Pocklington test.

1.2 The AKS test

Quite recently (2002) a deterministic test on primality testing has been published by Agrawal, Kayal and Saxena [1]. The main idea of this test is to check whether or not a number n satisfies an identity for prime numbers:

$$(X + a)^n = X^n + a \pmod{n} \tag{1.1}$$

where $a \in \mathbb{Z}$, $n \in \mathbb{N}$, $n \geq 2$ and $\gcd(n, a) = 1$. The proof of this identity can be found as the proof of theorem 1.1.

1.2.1 The algorithm

```
Input: integer $n > 1$

1. If $n = a^b$ for integers $a > 1$ and $b > 1$, return composite.
2. Find the smallest integer $r$ such that the multiplicative order of $n \pmod{r}$ is greater than $\log_2^2 n$.
3. If $1 < \gcd(a, n) < n$ for any integer $a \leq r$, return composite.
4. If $n \leq r$, return prime.
5. For $a = 1$ to $l = \left\lfloor \sqrt{\phi(r)} \log_2 n \right\rfloor$, if $(X + a)^n \neq X^n + a \pmod{X^r - 1, n}$, return composite.
6. Return prime.
```

The entire algorithm to test the primality of input number n consists of 6 steps.

In the first step, we check if n is a so-called perfect power, i.e. $n = a^b$ for $a \in \mathbb{N}$ and $b > 1$. We can do this by simply taking the b^{th} root of n for $b = 2$ until $\log_2 n$ and flooring it. Call this floored root a. Now check whether or not $a^b = n$. If so, we have found that n is a perfect power.

If n is not a perfect power, we proceed to the second step in which we search for the smallest number r, such that the multiplicative order of n modulo r is bigger than $\log_2^2 n$. This number r is used in step 3, 4 and 5. In step 3, we calculate the GCD of n and a, where $a \in \mathbb{N}^+, a \leq r$, and check if it's greater than 1. If so, n is composite. The 4^{th} step is to check if $n \leq r$. If this holds, n is prime. The 5^{th} consists of checking identity (1.1) for $a = 1$ to $\left\lfloor \sqrt{\phi(r)} \log_2 n \right\rfloor$. If it does not hold for all a, n is composite. If it does hold however, we can say with 100% certainty that n is prime.

At first sight, the algorithm seems a bit long which makes you ask yourself if all steps are really necessary. Concerning the first step of the algorithm, it seems that perfect powers will also be detected in step 3: if you can write n as a^b with a minimal, then you can also write n as a a^{b-1}. However, $a \leq \sqrt{n}$, and the $r \leq \max\{3, \lceil \log_2^5(n) \rceil\}$. So when $r < a$, perfect powers will not be sorted out in step 3. This means that in step 1 $b < \frac{\log n}{\log r}$.

The r calculated in the second step is used in all remaining steps. In the next section, you can read why this specific value of r is necessary. Step 3 and 4 obviously filter out some composite numbers and primes. However, not all numbers can be tested on primality in these steps. Step 5 and 6 do give a result for every number, so step 3 and 4 would actually be redundant if it weren't for the run time.

1.2.2 Correctness

The underlying idea of the AKS algorithm is a specific property that only holds for prime numbers. When we look at Pascal's triangle, it seems as if all numbers in the p^{th} row are multiples of p, except for the first and last number, for p prime. In fact, this is true and we can prove it.

Theorem 1.1. $(X+a)^n = X^n + a \pmod{n} \forall a > 0$ *holds only for n prime.*

Proof. The expansion of $(X+a)^n$ is given by $\sum_{k=0}^{n} \binom{n}{k} X^{n-k} a^k = X^n + \sum_{k=1}^{n-1} \binom{n}{k} X^{n-k} a^k + a^n$. We distinguish between two possibilities:

1: n is prime. We see that $(X+a)^n = X^n + a^n + \sum_{k=1}^{n-1} \binom{n}{k} X^{n-k} a^k$, which holds for all n. We also know for $k = 1, \ldots, n-1$ that $\binom{n}{k} \equiv 0 \pmod{n}$ for n is prime, because $\binom{n}{k} = \frac{n(n-1)...(n-k+1)}{k(k-1)...1}$ is a natural number, in which the numerator contains n and the denominator does not. Also, according to Fermat's little theorem, $a^n = a \pmod{n}$. So the identity holds.

When n is composite, we see that Fermat's little theorem also holds for Fermat liars. Furthermore, we see that $(X+a)^n \not\equiv X^n + a \pmod{n}$, because $\binom{n}{k} \not\equiv 0 \pmod{n}$: let p be the smallest prime factor of n, so that $n = pk$. Then $\binom{n}{p} = \frac{n(n-1)...(n-p+1)}{p(p-1)...1} = \frac{k(n-1)...(n-p+1)}{(p-1)(p-2)...1}$. This is not congruent with 0 (mod pk), because otherwise n would divide the numerator, so $(n-1)\ldots(n-p+1)$ would be divisible by p. Since p divides n, it does not divide $(n-1), (n-2), \ldots, (n-p+1)$ individually and hence p does not divide the product. □

Now that we know that the main idea is correct, we want to prove that our algorithm gives the correct output for every positive integer n. The rest of this section will be filled with lemmas and theorems that complete the proof of our algorithm's correctness.

In step 5 we want to check whether the polynomial $(X-a)^n - (X^n - a)$ is equal to zero for different values of a. Computing these equations for one a would take $\mathcal{O}(\log(n))$ multiplications (using exponentiation by squaring) in which the number of coefficients grows exponentially, so we would like to reduce the number of coefficients we have to multiply every time. This can be done by reducing our polynomial modulo $X^r - 1, n$ for some number r. There is a specific name for this action in our terminology:
Let l be the same number as in step 5 of the algorithm.
If $\forall a : 1 \leq a \leq l$: $(X-a)^m = X^m - a \pmod{X^r - 1, p}$, we call m introspective.

Lemma 1.2. *If m_1 and m_2 are introspective, then $m_1 m_2$ is also introspective.*

Proof. We know that $(X-a)^{m_2} = (X^{m_2} - a) \pmod{X^r - 1, p}$. We can also write this as $(X-a)^{m_2} - (X^{m_2} - a) = (X^r - 1) f(X)$ with $f(X)$ a polynomial. Likewise, we see that $(X^{m_1} - a)^{m_2} - (X^{m_1 m_2} - a) = (X^{m_1 r} - 1) f(X^{m_1})$, which equals 0 $\pmod{X^r - 1, p}$. So $(X-a)^{m_1 m_2} = (X^{m_1} - a)^{m_2} = X^{m_1 m_2} - a \pmod{X^r - 1, p}$. □

Suppose we have a number n which is composite, but fails the test in step 5 $\forall a : 1 \leq a \leq s$ for some bound s. We obviously don't want to let the output be *prime*, so we will try to find a bound such that only prime numbers will pass step 5. If we have 2 numbers n (composite) and p (prime) which are both introspective, each m of the form $p^i n^j$ is also introspective.

We need to prove one more property of introspective numbers.

Lemma 1.3. *If m is introspective for $f(X)$ and $g(X)$, then m is also introspective for $f(X)g(X)$.*

Proof. $f(X)^m = f(X^m) \pmod{X^r - 1, p}$ and $g(X)^m = g(X^m) \pmod{X^r - 1, p}$, so $(f(X)g(X))^m = f(X)^m g(X)^m = f(X^m)g(X^m) \pmod{X^r - 1, p}$. □

When we combine the last two lemmas, we see that the group $I = \{(\frac{n}{p})^i p^j | i, j \geq 0\}$ is introspective for every polynomial in the set $P = \{\prod_{a=0}^{l}(X + a)^{e_a} | e_a \geq 0\}$.

In order to complete our proof for correctness, we need to look at two groups. The first group G consists of all residues of numbers in I modulo r, which is a subgroup of $\mathbb{Z}_r^*$. Let t be the order of G, i.e. $t = |G|$. Since the element of a finite group divides the order, and since $o_r(n) > \log_2^2 n$, we know that $t > \log_2^2 n$.

The second group has to do with cyclotomic polynomials over F_p. Let $Q_r(X)$ be the r^{th} cyclotomic polynomial, i.e. the unique irreducible polynomial that only divides $X^r - 1$ and thus does not divide $X^k - 1$ for any $k < r$. $Q_r(X)$ also factors $X^r - 1$ into irreducible polynomials of $o_r(p)$. Let $h(X)$ be such a polynomial. We know that the degree of $h(X)$ is greater than 1, since $o_r(p) > 1$. Let $\widetilde{G}$ be our second group, consisting of all residues of the polynomials from $P \pmod{h(X), p}$. This means $\widetilde{G}$ is generated by $X, X + 1, \ldots, X + l \pmod{h(X), p}$ and it is a subgroup of $F = F_p(X)/(h(X))$. Now we want to find a lower bound for the order of $\widetilde{G}$.

Lemma 1.4. $|\widetilde{G}| \geq \binom{t+l}{t-1}$.

Proof. We want to show that there are no distinct polynomials of degree ¡ t in P that map to the same element in $\widetilde{G}$. Let $f(X)$ and $g(X)$ be two distinct polynomials in P and suppose they do map to the same element in F, so $f(X) = g(X)$ and thus $(f(X))^m = (g(X))^m$ in F for $m \in I$. Since m is introspective for both $f(X)$ and $g(X)$ and since $F = F_p(X)/(h(X))$, where $h(X)$ divides $X^r - 1$, we see that $(f(X))^m = f(X^m) \pmod{X^r - 1, p} = f(X^m) = g(X^m)$ in F. This means that X^m is a root of the polynomial $Q(Y) = f(Y) - g(Y) \forall m \in G$. Since G is a subgroup of Z_r^*, we know that $\gcd m, r = 1$, so all X^m are primitive roots, which means that $Q(Y)$ will have $|G| = t$ distinct roots in F, but since we chose both the degree of $f(X)$ and $g(X)$ to be smaller than t, the degree of $Q(Y)$ is also smaller than t. This means that we have a contradiction, so all distinct polynomials in P will map to different elements in F.

We know that all elements $X, X+1, \ldots, X+l$ are distinct in F because $l = \left\lfloor \sqrt{\phi(r)} \log_2 n \right\rfloor < \sqrt{r} \log_2 n < r < p$. We also know that the degree of $h(X)$ is greater than 1, so $X + a \neq 0$ in $F \forall a : 0 \leq a \leq l$. This means that we have at least $l + 1$ distinct polynomials of degree one in $\widetilde{G}$. In order to make a polynomial of degree $< t$, we look at the product of all distinct elements in F: $(X+a_0)^{b_0}(X+a_1)^{b_1}\dots(X+a_l)^{b_l}$ where $b_0+b_1+\dots+b_l=n$. This is also known as the egg colouring problem. The number of solutions for this equation is given by $\binom{t-1+l+1}{t-1}=\binom{t+l}{t-1}$. □

We can also say something about the size of $\widetilde{G}$ if n is not a power of p.

Lemma 1.5. *If n is not a power of p, then $|\widetilde{G}| \leq n^{\sqrt{t}}$.*

Proof. For this proof, we take a look at the subset of I: $\hat{I}=\{(\frac{n}{p})^i p^j | 0 \leq i,j \leq \lfloor\sqrt{t}\rfloor\}$. If n is not a power of p, then there are $(\lfloor\sqrt{t}\rfloor+1)^2 > 1$ distinct numbers in $\hat{I}$. We know that at least 2 numbers in $\hat{I}$ must be equal modulo r, because $|G|$, the number of residues of numbers in I modulo r, is t. Let m_1 and m_2 be such numbers with $m_1 > m_2$, such that: $X^{m_1}=X^{m_2} \pmod{X^r-1}$. Let $f(X)\in P$, then $(f(X))^{m_1}=f(X^{m_1}) \pmod{X^r-1,p}$, but this is the same as $f(X^{m_2}) \pmod{X^r-1,p}=(f(X))^{m_2}$. So $f(X)\in\widetilde{G}$ is a root of $Z(Y)=Y^{m_1}-Y^{m_2}$ in F. $Z(Y)$ has at least $|\widetilde{G}|$ distinct roots in F and the degree of $Z(Y)$ is $m_1 \leq (\frac{n}{p}p)^{\lfloor\sqrt{t}\rfloor} \leq n^{\sqrt{t}}$, so $|\widetilde{G}| \leq n^{\sqrt{t}}$. □

When we combine the two lemmas above, we can prove that our algorithm gives the correct output.

Theorem 1.6. *If the algorithm returns* prime *then n is prime.*

Proof. Suppose that the algorithm returns *prime*. We still use the same t and l from the lemmas above and we see that $|\widetilde{G}| \geq \binom{t+l}{t-1} \geq \binom{l+1+\lfloor\sqrt{t}\log_2 n\rfloor}{\lfloor\sqrt{t}\log_2 n\rfloor}$, because $t > \lfloor\sqrt{t}\log_2 n\rfloor$. Now $\binom{l+1+\lfloor\sqrt{t}\log_2 n\rfloor}{\lfloor\sqrt{t}\log_2 n\rfloor} \geq \binom{2\lfloor\sqrt{t}\log_2 n\rfloor+1}{\lfloor\sqrt{t}\log_2 n\rfloor}$, since $l=\left\lfloor\sqrt{\phi(r)}\log_2 n\right\rfloor \geq \lfloor\sqrt{t}\log_2 n\rfloor$. Because $\lfloor\sqrt{t}\log_2 n\rfloor > \log_2^2 n \geq 1$, we see that $\binom{\lfloor\sqrt{t}\log_2 n\rfloor+1}{\lfloor\sqrt{t}\log_2 n\rfloor} > 2^{2\lfloor\sqrt{t}\log_2 n\rfloor+1} \geq n^{\sqrt{t}}$.

From the lemma above, we know that $|\widetilde{G}| \leq n^{\sqrt{t}}$ if n is not a power of p, so $n=p^k$ for some positive k. But if $k>1$, the algorithm would already have returned *composite* in the first step (perfect prime testing). Therefore, $n=p$. □

To be even more complete, we can prove that theorem 1.6 also works the other way around:

Theorem 1.7. *If n is prime, the algorithm returns* prime.

Proof. Suppose n is prime. This means that n has no divisors, so $\gcd(n,z)=1$ for $z=1,\dots,n-1$. Also suppose that the algorithm returns "*composite*". This can only happen in step 1, 3 and 5. Since n has no divisors, it can never be a perfect power so step 1 will never return "*composite*". If the r computed in step 2 is smaller than n, all a's in step 3 will also be smaller than n and thus have no common divisor bigger than 1 with n. So step 3 will never return "*composite*" either. We also know that $(X+a)^n=X^n+a \pmod{n}$ holds for n prime and $a=1,\dots,l$, so it will never return "*composite*". Therefor, the algorithm will return "*prime*" in step 4 or 6. □

1.2.3 Complexity

We'll retrieve the complexity of the AKS test step by step.
Testing whether or not n is a perfect power has to be done by calculating $\lceil\sqrt[b]{n}\rceil$ (call this number a) for $b = 2$ to $b = \log_2 n$ and checking if a^b equals n. In order to determine the bth root of n, we use an algorithm that computes $\left\lfloor n^{\frac{1}{b}}\right\rfloor$ in time $\mathcal{O}(\log_2^2 n)$. Computing it at most $\log_2 n$ times, shows us that the total order of this step is $\mathcal{O}(\log_2^3 n)$.

The calculation of r in step 2 can be done quite efficiently. We are looking for the smallest r for which the smallest k implies $n^k \equiv 1 \pmod r$, where $k > \log_2^2(n)$. This means that we only have to calculate $n^k \pmod r$ for $k \leq \log_2^2(n)$, starting with $r = 2$. As soon as we find a $k \leq \log_2^2(n)$ such that $n^k \equiv 1 \pmod r$, we compute all powers again for $r+1$. If every output is greater than 1, we know that the order k is higher than $\log_2^2(n)$, which means we have found our r. In the worst case, we have to compute $n^k \pmod r$ for every r. Computations can be made using the right-to-left method, which are $\mathcal{O}((\log(\log n))^3)$, and according to lemma 1.9 $r < \log_2^5 n$, so the total order is $\mathcal{O}((\log n)^{5+\epsilon})$.

Calculating the GCD of two numbers a and b has complexity $\mathcal{O}(\log\max\{a,b\})$. In step 3, we have to calculate GCD(a,n) $r-1$ times. Since $r < \log_2^5 n$, the complexity is given by $\mathcal{O}(\log\max\{a,n\}\log_2^5 n) = \mathcal{O}(\log_2^6 n)$. Note that step 4 only takes 1 calculation, so $\mathcal{O}(\log_2 r)$.

In step 5, we have to check $\left\lfloor\sqrt{\phi(r)}\log_2 n\right\rfloor$ equations. $\mathcal{O}(\log_2 n)$ multiplications have to be done for each equation, the polynomials are of order r and all coefficients in the equations are $\mathcal{O}(\log_2 n)$. Hence the total order of step 5 is $\mathcal{O}(r\log_2^2 n\sqrt{\phi(r)}\log_2 n) = \mathcal{O}(r^{\frac{3}{2}}\log_2^3 n) = \mathcal{O}(\log_2^{\frac{21}{2}} n)$.

When we add all complexities, we find that the total complexity of our algorithm is $\mathcal{O}(\log_2^{\frac{21}{2}+\epsilon} n)$.

Lemma 1.8. *Let $LCM(m)$ denote the least common multiple of first m numbers. For $m \geq 7$: $LCM(m) \geq 2^m$.*

Lemma 1.9. *There exists an $r \leq \max\{3, \lceil\log_2^5(n)\rceil\}$ such that $o_r(n) > \log_2^2(n)$.*

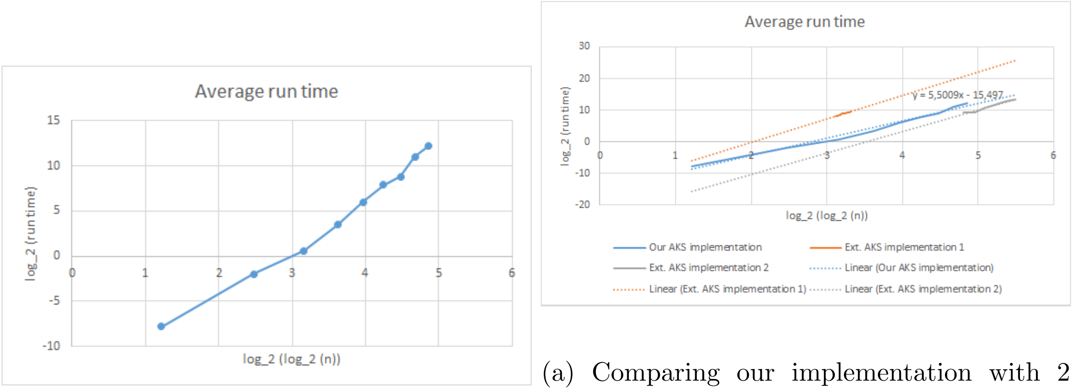
Proof. For $n = 2$, we see that the equation holds for $r = 3$. For $n > 2$, we see that $\lceil\log_2^5(n)\rceil > 10$. We now use the LCM lemma 1.8 (with $m = \lceil\log_2^5 n\rceil$). The largest value for k to satisfy $m^k \leq B = \lceil\log_2^5(n)\rceil, m \geq 2$ is $\lfloor\log_2 B\rfloor$. Now take a look at the smallest number r that does not divide $n^{\lfloor\log_2 B\rfloor}\prod_{i=1}^{\lfloor\log_2^2 n\rfloor}(n^i-1)$. The GCD of r and n cannot be divisible by all prime divisors of r at the same time, because otherwise r would divide n and also $n^{\lfloor\log_2 B\rfloor}$. This means that $\frac{r}{GCD(r,n)}$ will not divide $n^{\lfloor\log_2 B\rfloor}$ either. We know that r is the smallest number not dividing the product, so $GCD(r,n) = 1$. From the second part of the product we know that r does not divide any of the $n^i - 1$ for $1 \leq i \leq \left\lfloor\log_2^2 n\right\rfloor$, so the order $o_r(n) > \log_2^2 n$. Now we want to find an upper bound for the product: $n^{\lfloor \log_2 B \rfloor} \prod_{i=1}^{\lfloor \log_2^2 n \rfloor} (n^i - 1) < n^{\lfloor \log_2 B \rfloor + \frac{1}{2} \log_2^2 n (\log_2^2 n - 1)} \leq n^{\log_2^4 n} \leq 2^{\log_2^5 n \leq 2^B}$. By filling in B for the LCM lemma, we see that $LCM(B) \geq 2^B$, so $r \leq B$. □

Note that this lemma is only necessary for step 4 when $n \leq 5690034$. When $n > 5690034$, we see that $r \leq \lceil \log_2^5 n \rceil < n$.

1.2.4 Run time

In order to confirm our predicted run time, we test the algorithm for a number of primes. The results can be found in the table and graph below.

Prime number n	Run time (sec.)
47	0.257
499	1.542
4999	11.185
49999	63.871
499979	236.324
4999999	461.847
49997999	2106.061
499979999	4695.661



(a) Comparing our implementation with 2 other versions found online. The solid lines consist of the gathered data. The dashed lines are trend lines made using the data points.

Figure 1.1: The average run time: each number has been tested three times.

Comparing our implementation of the AKS algorithm to others, we find out that there is a huge difference in run times an that our implementation is neither slow nor fast. For some examples of other implementations, please see [2] and [3]. The slope of the line of the results of our implementation is 5.5, which is slightly different from what we expected in our complexity analysis (10.5). This difference can be explained by the small amount of test numbers I used and secondly by the fact that the complexity is an upper bound, which describes the run time in the worst case. Our test numbers are not necessarily the worst cases, so that's why the run time that we found is smaller than the complexity.

1.3 Improvements

In July 2005, Hendrik Lenstra Jr. and Carl Pomerance published a draft version of an article in which they showed how the run time of the AKS algorithm could be reduced. In 2009, they published the final version of the article [4] in which they claimed to have reduced the algorithm's complexity to $\mathcal{O}(\log_2 n)^6(2 + \log_2 \log_2 n)^c$ for some effectively computable real number c. This is a great improvement compared to the run time proposed in the document by Agrawal, Kayal and Saxena: $\mathcal{O}(\log_2 n)^{\frac{21}{2}}(2 + \log_2 \log_2 n)^c = \mathcal{O}(\log_2 n)^{\frac{21}{2}+\epsilon}$. The main difference between this algorithm and the one published by Agrawal, Kayal and Saxena is that Lenstra and Pomerance have their rings generated by Gaussian periods rather than by roots of unity.

Another algorithm that can be used to reduce the run time was published in 1998 by Daniel Bernstein [6]. This algorithm is about detecting perfect powers, with run time $\mathcal{O}(\log_2 n)^{1+o(1)}$.

1.4 Lucas-Lehmer test

Another algorithm to test an integer's primality is described by Édouard Lucas and later on it has been improved by Derrick Henry Lehmer. Even though there is also a Lucas-Lehmer test which only works for Mersenne numbers, this test works for every positive integer.[7]

1.4.1 The algorithm

Input: integer $n > 1$. Let $1 < a < n$ and q be positive integers.

1. Check if $a^{n-1} \equiv 1 \pmod{n}$. If not, return ”*composite (step 1 (a))*”.
2. Check if $a^{\frac{n-1}{q}} \not\equiv 1 \pmod{n}$ for every prime $q | n-1$. If so, return ”*prime (a)*”. If not, repeat the algorithm for another a if possible, otherwise return ”*composite (step 2 (a))*”.

If an appropriate a is found to meet the conditions in step 2, this a can be used as a prime certificate. This certificate is a very short proof of the input number's primality, in the shape of an a that suffices both equations. This way, one can use the certificate to verify the equations very efficiently for the given a. The proof of the algorithm will complete the input number's primality proof.

Furthermore, it has to be said that this test for n is only useful when the factorisation of $n-1$ is known. This is due to the need of knowing all prime divisors of $n-1$ in step 2. The problem of not knowing can be solved by invoking the algorithm recursively, i.e. running the algorithm for possible prime divisors q_i of $n-1$ when checking for n's primality, and then checking for possible prime divisors of $q_i - 1$ by running the algorithm again for q_i, etc. This affects the complexity of the algorithm drastically: in the worst case, you would have to check primality for about $\frac{\sqrt{n}}{2}$ numbers. However, you can also use a prime factorisation algorithm to find the prime divisors of $n-1$, which has a smaller complexity than recursively invoking the algorithm $\frac{\sqrt{n}}{2}$ times.

1.4.2 Correctness

From the first step of the algorithm, we see that the order of a in $\mathbb{Z}_n^*$ is a divisor of $n-1$. However, in the second step we see that the order of a is not a divisor of $n-1$, which means that it is equal to $n-1$. Since the order of an element in a group divides its group's order we see that $n-1$ divides $\phi(n)$, so $n-1 \leq \phi(n)$. Let's take a look at the definition of $\phi(n)$. This function represents the number of integers relatively prime to n. Now suppose that n is composite and it has prime factor p. This would mean that p and n are not coprime to n, so $\phi(n) \leq n-2$. This contradicts $n-1 \leq \phi(n)$, which means that n is prime.

1.4.3 Complexity

In the worst case we have to check the 2 steps for all prime divisors of $n-1$, which means that the factorisation of $n-1$ has to be known. The corresponding complexity is

$\mathcal{O}(\exp(C(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}))$. The first step can be computed with the right-to-left method, which is $\mathcal{O}((\log n)^3)$. Checking the equation in the second step has to be done for every prime factor q of $n-1$. This means that the complexity of this step is $\mathcal{O}((\exp(C(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}))(\log a)^3)$. So the total complexity of this algorithm is $\mathcal{O}((\exp(C(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}))((\log n)^3 + (\log a)^3)) = \mathcal{O}((\exp(C(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}))((\log n)^3))$.

1.4.4 Test for Mersenne primes

There is also a test especially for Mersenne numbers. A Mersenne number is of the form $M_p = 2^p - 1$ where p is a known prime number. The main idea of this variant is to create a recurrence relation and use it to check equalities. The test is as follows:

Input: prime number p.

1. Compute $M_p = 2^p - 1$.
2. Construct the sequence $\{s_i\}$ as follows:
   $s_i = s_{i-1}^2 - 2$ with $s_0 = 4$ until $i = p - 2$.
3. If $s_{p-2} \equiv 0 \pmod{M_p}$, return "*$M_p$ prime*". Else, return "*M_p composite*".

Note that the sequence is always the same, but the number of terms that have to be computed depends on the size of p. This means that in step 2, we start with $s_0 = 4$ and repeat $s_i = s_{i-1}^2 - 2 \pmod{M_p}$ $p-2$ times.

Correctness

In order to prove that $s_{p-2} \equiv 0 \pmod{M_p} \iff M_p$ prime, we define $\omega = 2 + \sqrt{3}$ and $\overline{\omega} = 2 - \sqrt{3}$. We now define L_n as $\omega^{2^n} + \overline{\omega}^{2^n}$, such that $L_0 = \omega^1 + \overline{\omega}^1 = 4$. Furthermore, $L_{n+1} = \omega^{2^{n+1}} + \overline{\omega}^{2^{n+1}}$. Note that $\omega\overline{\omega} = (2+\sqrt{3})(2-\sqrt{3}) = 1$, so the last equation can be rewritten as $\omega^{2^{n+1}} + \overline{\omega}^{2^{n+1}} + 2\omega^{2^n}\overline{\omega}^{2^n} - 2 = (\omega^{2^n} + \overline{\omega}^{2^{n+1}})^2 - 2 = L_n^2 - 2$. This means L_n and s_i must be the same sequence. Now we can use this L_n to prove both sides of the theorem.

"$\Leftarrow$": Consider the group of numbers of the form $a + b\sqrt{3} \pmod{M_p}$ where $M_p = 2^p - 1$. We know from the AKS algorithm's proof that $(1+\sqrt{3})^{M_p} \equiv 1 + \sqrt{3}^{M_p} \pmod{M_p} = 1 + \frac{\sqrt{3}}{\sqrt{3}}\sqrt{3}^{M_p} \pmod{M_p} = 1 + (\sqrt{3})3^{\frac{M_p-1}{2}} \pmod{M_p}$. From quadratic reciprocity we know that $3^{\frac{M_p-1}{2}} \equiv \pm 1 \pmod{M_p}$. Now we have to find out what the correct sign is. M_p and 3 are congruent to $-1 \pmod 4$, so the law of quadratic reciprocity tells us that either M_p is a quadratic residue $\pmod 3$ or that 3 is a quadratic residue $\pmod{M_p}$. We know that $M_p \equiv 1 \pmod 3$, so M_p is a quadratic residue $\pmod 3$ and as a result we see that $3^{\frac{M_p-1}{2}} \equiv -1 \pmod{M_p}$. This means that $(1+\sqrt{3})^{M_p} \equiv 1 - \sqrt{3} \pmod{M_p}$. We now multiply both sides with $(1+\sqrt{3})$, which gives us $(1+\sqrt{3})^{M_p+1} \equiv -2 \pmod{M_p}$. We also know that $(1+\sqrt{3})^2 = (4+2\sqrt{3}) = 2\omega$, which we use to rewrite $(1+\sqrt{3})^{M_p+1}$ as $(2\omega)^{\frac{M_p+1}{2}} = 2^{\frac{M_p+1}{2}}\omega^{\frac{M_p+1}{2}} = 2^1 2^{\frac{M_p-1}{2}}\omega^{\frac{M_p+1}{2}} \equiv -2 \pmod{M_p}$. We know that $2^{\frac{M_p-1}{2}} \equiv 1 \pmod{M_p}$ because of the fact that 2 is a quadratic residue of primes congruent to $\pm 1 \pmod 8$. The last equation can now be rewritten as

$2\omega^{\frac{M_p+1}{2}} \equiv -2 \pmod{M_p}$. The inverse of 2 $\pmod{M_p}$ is $\frac{M_p+1}{2}$, so by multiplying both sides with this inverse, we get $\omega^{\frac{M_p+1}{2}} \equiv -M_p - 1 \equiv -1 \pmod{M_p}$. We can write the left-hand side of the equation as $\omega^{\frac{2^p-1+1}{2}} = \omega^{2^{p-1}} = \omega^{2^{p-2}}\omega^{2^{p-2}}$, which is equivalent to $-1 \pmod{M_p}$. Now we multiply both sides with $\overline{\omega}^{2^{p-2}}$, which results in $\omega^{2^{p-2}} + \overline{\omega}^{2^{p-2}} \equiv s_{p-2} \equiv 0 \pmod{M_p}$.

"$\Rightarrow$": M_p divides s_{p-2} implies that there exists an integer C such that $L_{p-2} = \omega^{2^{p-2}} + \overline{\omega}^{2^{p-2}} = CM_p$. By multiplying with $\omega^{2^{p-2}}$, we can rewrite this equation to $\omega^{2^{p-1}} = CM_p\omega^{2^{p-2}} - 1$. By squaring both sides, we get $\omega^{2^p} = (CM_p\omega^{2^{p-2}} - 1)^2$. Assume that M_p is composite, which means that there is a prime divisor $q < \sqrt{M_p}$. We now look at the group G of all numbers $a + b\sqrt{3}$ which are invertible mod q and we see that $|G| \leq q^2 - 1$ (because 0 does not have an inverse). We can now look back at the equations with ω, only this time mod q. Recall that q divides M_p, so we derive that $\omega^{2^{p-1}} = -1$ and $\omega^{2^p} = 1 \pmod{q}$ which means that the order of ω in G is 2^p. We also know that the order of an element divides the order of the group, which means that the order of an element is at most the order of the group. However, we see that $2^p \leq q^2 - 1 < q^2 \leq M_p = 2^p - 1$. This contradiction completes our proof.

Complexity

In order to check s_{p-2}, we have to compute $M_p = 2^p - 1$ once and then use modular exponentiation and substraction $p - 2$ times. The corresponding complexity is $\mathcal{O}(\log(2)^3 + (p - 2)(\log(M_p)^3 + log(M_p))) = \mathcal{O}((p-2)(\log(M_p)^3) + \epsilon)$.

1.5 Pocklington primality test

Another way of testing numbers for primality is called the Pocklington(-Lehmer) primality test [7]. As the name suggests, the test was devised by Henry Cabourn Pocklington and Derrick Henry Lehmer. Although this test is used to check whether or not an input number is prime, it has a different result from the AKS test. Where the AKS test provided ”*prime*” or ”*composite*” as output, the Pocklington test will return a prime certificate: a proof that the input number is prime or not. That is, if the input is not prime the output will be the step number which can not be met. The algorithm is as follows:

1.5.1 The algorithm

Input: integer $n > 1$. Let $1 < a < n$ and q be positive integers.

1. If there does not exist a prime $q > \sqrt{n} - 1$ such that $q | n-1$, return ”*indeterminate* (step 1)”.
2. Check if $a^{n-1} \equiv 1 \pmod{n}$. If not, return ”*composite* (step 2: a)”.
3. Check if $\gcd\{a^{\frac{n-1}{q}} - 1, n\} = 1$. If so, return ”*prime (a)*”. If not, repeat the algorithm for another q.

1.5.2 Correctness

Suppose n is composite. This implies that n has a prime divisor $p \leq \sqrt{n}$. Since we choose $q > \sqrt{n}$, we know that $q > p - 1$ and therefore $\gcd\{q, p-1\} = 1$, which means that q has a modular multiplicative inverse x modulo $p - 1$: $xq \equiv 1 \pmod{p-1}$. Now suppose that we find an a such that $a^{n-1} \equiv 1 \pmod{n}$. (Note that if we find an a such that $a^{n-1} \not\equiv 1 \pmod{n}$, this is a Fermat witness for n being composite.) Since p divides n, we see that $a^{n-1} \equiv 1 \pmod{n} \equiv 1 \pmod{p}$. We can rewrite the first part of the equation as follows: $a^{n-1} \equiv (a^{n-1})^x \equiv a^{x(n-1)} \equiv a^{xq\frac{n-1}{q}} \equiv (a^{xq})^{\frac{n-1}{q}}$. We also know from Fermat's little theorem and using that $xq \equiv 1 \pmod{p-1}$ that a^{xq} can be rewritten in an easier form. xq is of the form $c(p-1)+1$, so $a^{xq} = a^{c(p-1)+1} = (a^{p-1})^c a^1 \equiv 1^c a \equiv a \pmod{p}$. This means we get: $(a^{xq})^{\frac{n-1}{q}} \equiv a^{\frac{n-1}{q}} \pmod{p}$, so $a^{\frac{n-1}{q}} \equiv 1 \pmod{p}$. We now know that p divides n and p divides $a^{\frac{n-1}{q}} - 1$, so the greatest common divisor in step 3 is not 1. This completes the proof.

1.5.3 Complexity

In the worst case, we have to loop through the algorithm for every q that divides $n - 1$. This means that we have to compute the factorisation of $n - 1$, which has complexity $\mathcal{O}(\exp(C(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}))$. The computation in step 2 can be done using the right-to-left modular exponentiation with complexity $\mathcal{O}(\log(n)^3)$. Calculating the GCD in step 3 is $\mathcal{O}(\log(n)^3 \log(n)^2) = \mathcal{O}(\log(n)^5)$. So the total complexity of this algorithm is
$\mathcal{O}((\exp(C(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}))(\log(n)^{5+\epsilon}))$.

1.5.4 Prime number generation

Once we know that an input number n_0 is prime we can use it again in the algorithm, only this time as q_1. This means that we are able to use the certificate of n_0's primality (together with its corresponding q_0) to construct bigger prime numbers. The new input number n_1 has some limitations though. Since it uses q_1 and $q > \sqrt{n} - 1$, we infer that $n < q^2 + 2q + 1$.

An example: we know that 3 is prime. This means that we can use 3 as q_0 in our quest to find bigger primes, so $n_0 < q_0^2 + 2q_0 + 1 = 16$. An n_0 that also satisfies $q_0|n_0 - 1$ is 13. We verify step 2 for $a_0 = 2$ and use this same a_0 to see that we can conclude from step 3 that n_0 is indeed prime. So our certificate would look like $\{q_0, n_0\} = \{3, 13\}$. Next, we can use n_0 as q_1 in order to find more prime numbers. This way, we accumulate the certificates which can all be verified extremely easy. An implementation of this algorithm in Mathematica returns the following start of a sequence of prime numbers: 3, 13, 157, 24179, 583535987. The Mathematica code in the appendix can be used to find even more prime numbers!

1.6 Conclusion

The AKS primality test has its pros and cons. The nice thing about it, is that it is a deterministic test that works for every input number and runs in polynomial time. It gives you 100% certainty about the primality of the number you want to test, contrary to other prime tests like Miller-Rabin, which relies on the unproven generalised Riemann hypothesis.

However, timewise it seems more efficient to use other tests. The original AKS algorithm's complexity is $\mathcal{O}(r^{\frac{3}{2}} \log_2^6 n \log_2^1 5r)$ and its improved version has complexity $\mathcal{O}(\log_2 n)^6(2 + \log_2 \log_2 n)^c$, whereas Miller-Rabin's complexity is $\mathcal{O}(R(\log n)^3)$. So in practice the AKS algorithm will not be very useful, as you can tell by the results from our own implementation.

The Lucas-Lehmer test can be useful in practice, but only if you know the factorisation of $n-1$. In that case, the algorithm has a complexity of $\mathcal{O}(n^2 \log^2(a) + n \log^3(n))$. If the factorisation is not known, the algorithm will invoke itself recursively, resulting in a much higher complexity. The Lucas-Lehmer test for Mersenne numbers has a lower complexity, namely $\mathcal{O}((p-2)(\log(M_p)^3) + \epsilon)$. This variant's main drawback is that the amount of input numbers is much more restricted, as they have to be of the form $2^p - 1$ with p prime.

When the prime factorisation is not known, the Pocklington test is a better way to test for a number's primality as the complexity is $\mathcal{O}(n(\log(n)^{5+\epsilon}))$. This method returns prime certificates, which can be accumulated when used in order to find new prime numbers: a nice result, as two consecutive prime numbers that are found with the algorithm can differ hugely in length. This way, you can construct large prime numbers relatively easy.

Test	**Input number type**	**Complexity**
AKS	$\mathbb{Z}_+$	$\mathcal{O}(r^{\frac{3}{2}} \log_2^6 n \log_2^1 5r)$
Improved AKS	$\mathbb{Z}_+$	$\mathcal{O}(\log_2 n)^6(2 + \log_2 \log_2 n)^c$
Lucas-Lehmer	$\mathbb{Z}_+$	$\mathcal{O}((\exp(C(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}))((\log n)^3)$
Lucas-Lehmer for Mersenne primes	$M_p = 2^p - 1$, p prime	$\mathcal{O}((p-2)(\log(M_p)^3) + \epsilon)$
Pocklington	$\{n \in \mathbb{Z}_+ \mid \exists q : q \mid n-1\}$	$\mathcal{O}((\exp(C(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}))(\log(n)^{5+\epsilon}))$

Table 1.1: A schematic overview of the tests discussed in this paper.

For further information about the AKS primality test, please see the original document by Agrawal, Kayal and Saxena [1] or Shoup's book on number theory and algebra [5]. Also, for the AKS test, the Lucas-Lehmer test, the Pocklington test and everything you want to know about prime numbers, I'd like to refer to the book "*Prime Numbers: A Computational Perspective*" by Crandall and Pomerance [7].

Bibliography

[1] M. Agrawal, N. Kayal, and N. Saxena. *PRIMES is in P.* Annals of Mathematics, 160(2):781793, 2004.

[2] R.G. Salembier, and P. Southerington. *An Implementation of the AKS Primality Test.*

[3] C. Rotella. *An Efficient Implementation of the AKS Polynomial-Time Primality Proving Algorithm.*

[4] H. W. Lenstra jr. and C. Pomerance. *Primality testing with Gaussian periods.*

[5] V. Shoup. "Deterministic primality testing". In *A Computational Introduction to Number Theory and Algebra (Version 2).*

[6] D. Bernstein, *Detecting perfect powers in essentially linear time.* Mathematics of computation, Volume 67, Number 223, July 1998, Pages 1253-1283, S 0025-5718(98)00952-1

[7] R. Crandall, and C. Pomerance. "Recognizing primes and composites" and "Primality proving". In *Prime Numbers. A Computational Perspective (Second Edition).*

1.7 Appendix

1.7.1 Java code

Please note that I split the code into two separate classes: AKS.java and Polynomial.java.

```
import java.math.BigInteger;
import java.util.Scanner;
import javax.swing.Timer;

public class AKS {
    BigInteger n, input, m, g, B, z, polPower;
    int length, k;
    Boolean isPerfectPower, stap34, notComp;

    static BigInteger convert(int x) {
        return BigInteger.valueOf(x);
    }

    public static double logBigInteger(BigInteger val, int e) {
        double doubN = val.doubleValue();
        return Math.log(doubN)/Math.log(e);
    }

    public BigInteger BinSqrt(BigInteger n) {
        length = (int) Math.floor(Math.log(n.doubleValue())/Math.log(2))+1;
        k = (int) Math.floor((length-1)/2);
        m = convert(2).pow(k);
        for(int i=k-1; i>=0; i--){
            if((m.add(convert(2).pow(i)).pow(2)).compareTo(n) <= 0){
                m.add(convert(2).pow(i));
            }
        }
        return m;
    }

    public BigInteger ESqrt(BigInteger n, int e){
        length = (int) Math.floor((logBigInteger(n, e)))+1;
        k = (int) Math.floor((length-1)/2);
        m = convert(2).pow(k);
        for(int i=k-1; i>=0; i--){
            if((m.add(convert(2).pow(i)).pow(e)).compareTo(n) <= 0){
                m.add(convert(2).pow(i));
            }
        }
        return m;
    }
```

```
public BigInteger GCD(BigInteger a, BigInteger b){
    if(b.equals(convert(0))) return a;
    return GCD(b, a.mod(b));
}

public int totient(BigInteger n){
    int count=0;
    for(int a=1; n.compareTo(convert(a))>0; a++){
        if(GCD(n,convert(a)).equals(convert(1))){
            count++;
        }
    }
    return(count);
}

public BigInteger R2L(BigInteger x, BigInteger a, BigInteger m){
    BigInteger z = BigInteger.ONE;
    BigInteger s = x;
    BigInteger a1 = a;
    while(a1.compareTo(BigInteger.ZERO) > 0){
        if(a1.mod(convert(2)).equals(BigInteger.ONE)){
            z = z.multiply(s).mod(m);
        }
        a1 = ((a1.divide(convert(2))));
        if(a1.compareTo(BigInteger.ZERO) > 0){
            s = s.multiply(s).mod(m);
        }
    }
    return z;
}

public void isPerfectPower(BigInteger n){
    isPerfectPower = false;
    outerloop:
    for(int k=2; k<=Math.floor(Math.log(n.doubleValue())/Math.log(2)); k++){
        BigInteger wortel = ESqrt(n,k);
        if(wortel.pow(k) == n){
            isPerfectPower = true;
            System.out.println(n+" = "+wortel+"^"+k);
            break outerloop;
        }
    }
}

public void multOrd(BigInteger n){ //Bereken de kleinste r met o_r(n) > log^2(n)
    System.out.println("Stap 2");
    int log2n = ((int) Math.pow(logBigInteger(n,2),2));
```

```
        int log5n = ((int) Math.pow(logBigInteger(n,2),5)); //Volgens lemma 4.3
        System.out.println("log^2("+n+") = "+log2n+"; log^5("+n+") = "+log5n);
        outerloop:
        for(int r = 2; r <= log5n ; r++){
            int k = 2;
            while(R2L(n,convert(k), convert(r)).compareTo(convert(1)) > 0){
                k++;
                if(log2n == k && R2L(n,convert(k), convert(r)).
                compareTo(convert(1)) > 0){
                    System.out.println("r = "+r);
                    stap34(n, convert(r));
                    break outerloop;
                }
            }
        }
    }

    public void stap34(BigInteger n, BigInteger r){
        System.out.println("Stap 3/4");
        stap34 = false;
        if(n.compareTo(r) > 0){
            loop:
            for(int a = 1; r.compareTo(convert(a)) >= 0; a++){
                if((GCD(convert(a), n).compareTo(convert(1)) > 0) &&
                (GCD(convert(a), n).compareTo(n) < 0)){
                    stap34 = true;
                    System.out.println("Samengesteld (stap 3): "+n+" =
                    "+GCD(convert(a), n)+"*"+n.divide(GCD(convert(a), n)));
                    break loop;
                }
                if(convert(a).compareTo(r) == 0 && !(GCD(convert(a), n).
                compareTo(convert(1)) > 0) && (GCD(convert(a), n).compareTo(n) < 0)){
                    stap5(n, r);
                }
            }
        }
        else if(n.compareTo(r) <= 0){
            stap34 = false;
            System.out.println("Priem! (stap 4)");
        }
    }

    public void stap5(BigInteger n, BigInteger r){
        notComp = true;
        System.out.println("Stap 5");
        int a = 2;
        BigInteger sqrtot = convert((int) (Math.sqrt(totient(r))*
        (logBigInteger(n,2)))); //Floor
        System.out.println(Math.sqrt(totient(r))*(logBigInteger(n,2)));
        System.out.println(notComp);
        System.out.println(convert(a).compareTo(sqrtot));
        System.out.println(GCD(convert(a), n).compareTo(BigInteger.ONE));
        while(notComp = true && convert(a).compareTo(sqrtot)<=0 &&
        GCD(convert(a), n).compareTo(BigInteger.ONE) == 0){
            polMult(convert(a),r,n);
            a++;
            if(a>Math.floor(Math.sqrt(totient(r))*(logBigInteger(n,2)))){
                if(notComp = true){
                    System.out.println("Priem! (stap 5)");
                }
                else {
                    System.out.println("Samengesteld (stap 5)");
                }
            }
        }
    }

    public void polMult(BigInteger a, BigInteger r, BigInteger n){
        notComp = true;
        BigInteger power = n;
        Polynomial p1 = new Polynomial(convert(1));
        Polynomial pkeer = new Polynomial(convert(1),a);
        while(power.compareTo(convert(1)) > 0){
            if(power.mod(convert(2)).compareTo(convert(1)) == 0){
                //Vermenigvuldig
                p1 = p1.multiply(pkeer).mod(r, n);
            }
            power = convert((int) Math.floor(power.divide((convert(2))).
            doubleValue()));
            pkeer = pkeer.multiply(pkeer); //Kwadrateer
            pkeer = pkeer.mod(r, n);
        }

        p1 = p1.multiply(pkeer).mod(r, n);

        //Controleer coefficienten
        BigInteger p1coeff[] = p1.getCoeff();
        int countPol = 0;
        outerloop:
        for(int i = 0; r.compareTo(convert(i)) > 0; i++){
            if(p1coeff[i].compareTo(convert(0)) != 0){
                if(p1coeff[i].compareTo(convert(1)) == 0){
                    countPol++;
                }
                else if(p1coeff[i].compareTo(a) == 0){
                    countPol++;
                }
                else {
                    System.out.println("Coefficient ongelijk aan 1 of
                    "+a+" gevonden.");
                    notComp = false;
                    break outerloop;
                }
            }
        }
        if(countPol != 2){
            System.out.println("Aantal coefficienten ongelijk aan 0
            is niet gelijk aan 2.");
            notComp = false;
        }
        System.out.println(p1);
        System.out.println("Not composite: "+notComp);
    }

    public void AKS(BigInteger n){
        //Stap 0: Kijk of n even is
        if(n.mod(convert(2)).compareTo(convert(0)) == 0){
            System.out.println(n+" is even (stap 0)");
        }
        else{
            //Stap 1: Kijk of n een perfect power is
            System.out.println("Stap 1");
            isPerfectPower(n);
            if(isPerfectPower){
                System.out.println("Samengesteld (stap 1)");
            }
            else{
                //Stap 2 t/m 6
                multOrd(n);
            }
        }
    }

    public void main(){
        Scanner reader = new Scanner(System.in);
        System.out.println("Voer een getal in om te testen: ");
        input = reader.nextBigInteger();
        long startTime = System.nanoTime();
        AKS(input);
        long estimatedTime = System.nanoTime() - startTime;
        System.out.println("Looptijd programma: "+estimatedTime+" nanosec.");
    }

    public static void main(String[] args){
        new AKS().main();
    }
}

import java.math.BigInteger;
import java.util.Arrays;

public class Polynomial {
    private final BigInteger[] coeff;

    static BigInteger convert(int x) {
        return BigInteger.valueOf(x);
    }

    public Polynomial(BigInteger... coeff) {
        this.coeff = coeff;
    }

    @Override
    public String toString() {
        return Arrays.toString(coeff);
    }

    public BigInteger[] getCoeff(){
        return coeff;
    }

    public Polynomial multiply(Polynomial polynomial) {
        int totalLength = coeff.length + polynomial.coeff.length - 1;
        BigInteger[] result = new BigInteger[totalLength];
        for(int k=0; k<totalLength; k++){
            result[k] = BigInteger.ZERO;
        }
        for (int i = 0; i < coeff.length; i++)
            for (int j = 0; j < polynomial.coeff.length; j++) {
                result[i + j] = result[i + j].add(coeff[i].
                multiply(polynomial.coeff[j]));
            }
        return new Polynomial(result);
    }

    public Polynomial mod(BigInteger r, BigInteger n) {
        int totalLength = r.intValue();
        BigInteger[] result2 = new BigInteger[totalLength];
        for(int k=0; k<totalLength; k++){
            result2[k] = BigInteger.ZERO;
        }
        for (int i = coeff.length-1; i >= 0 ; i--){
            result2[((i-coeff.length)%totalLength+totalLength)%totalLength]
            = result2[((i-coeff.length)%totalLength+totalLength)%totalLength].
            add(coeff[i].mod(n));
            result2[((i-coeff.length)%totalLength+totalLength)%totalLength]
            = result2[((i-coeff.length)%totalLength+totalLength)%totalLength].
            mod(n);
        }
        return new Polynomial(result2);
    }
}
```

1.7.2 Implementation of Pocklington

```
q = 3;
n0 = 0;
n = 0;
i = q^2 + 2 q;
While[n == 0,
 While[n0 == 0 , If[MemberQ[Divisors[i - 1], q], n0 = i]; i--];
 If[PowerMod[2, n0 - 1, n0] == 1 && GCD[2^((n0 - 1)/q) - 1, n0] == 1,
  n = n0; Print[n], n0 = 0; i--]]
```