Ecole Polytechnique Fédérale de Lausanne

Bachelor semester project: Randomized and Deterministic Primality Testing

Monica Perrenoud

Under the direction of

Professor Mohammad Amin Shokrollahi
Responsible assistant Ghid Maatouk

Fall 2009

Abstract

This project presents some randomized and deterministic algorithms of number primality testing, and, for some of them, their implementation in C++.
The algorithms studied here are the naive algorithm (deterministic), the Miller-Rabin algorithm (randomized), the Fermat algorithm (randomized), the Solovay-Strassen algorithm (randomized) and the AKS algorithm (deterministic).
These algorithms are presented with the number theory they need to be understood and with some proofs of the theorems they use.

Contents

1 Introduction

For many applications (as cryptography), it is very useful to find large prime numbers. But the problem is to find these large prime numbers. Some algorithms have been developed to check if a given large number is prime or not. Some of them make the study of this project.
Some of these algorithms do not give a sure answer, they only give it with a certain probability (which can be very high if we iterate a lot the algorithm). But we can study how many errors they make (in comparison to the naive algorithm) and what can be done to make these errors as small as possible. This is the case for the Miller-Rabin, the Fermat and the Solovay-Strassen algorithm.
The AKS algorithm is interesting in an other way. It is more theoretical than practical. The problem of finding if an algorithm of primality testing can be in P (so which runs in polynomial time) was resolved a little time ago. This algorithm is the AKS algorithm which is presented here with the explanation of its running time and correctness.
All of the algorithms of primality testing are based on some elements of number theory. Here are presented some of these elements that are used in the studied algorithms and are necessary to understand them.
Note that this subject makes sense only in $\mathbb{N}$. So all the numbers, unless otherwise specified, are taken in $\mathbb{N}$.

2 General informations about the algorithms

2.1 Deterministic and randomized algorithms

There are two types of algorithms: deterministic and randomized.
A **deterministic algorithm** is an algorithm that behaves predictably. We can run it as many times we want, with the same input, it will always do the same steps and give the same answer.
But a **randomized algorithm** is an algorithm that uses some random values in hope to verify a property in the average case. Since it is not possible to have random numbers, the randomized algorithms use a pseudorandom generator of numbers. If the probability to find a number verifying the property is rather good, we say that it is a good randomized algorithm.

2.2 Density of prime numbers

To ensure that the randomized algorithms can work in a little time, we have to look at the density of prime numbers.

Definition 2.2.1
*The **distribution function of prime numbers** is written $\pi(n)$ and specifies the quantity of prime numbers less than or equal to n.*

Theorem 2.1 (Prime number theorem)
A good approximation of $\pi(n)$ is given by:

$$\lim_{n\to\infty} \frac{\pi(n)}{\left(\frac{n}{\ln n}\right)} = 1.$$

So we can estimate that a number $a \in \mathbb{N}$ randomly chosen has a probability of $\frac{1}{\ln a}$ to be prime. If we randomly take a number a, we will have to test about $\ln a$ numbers around a to find a prime number of the same length of a.

2.3 Output of the algorithms

The output of the algorithms seen in this project is, for an input $n \in \mathbb{N}$, if n is prime or composite.
But not in all algorithms the output is right at 100%. Some of them are based on some criterion that ensures with a certain probability that the answer is right. In our case, when the algorithm gives the answer that a number is composite, then it is really composite. But if the algorithm gives the answer that this number is prime, this means that this number is prime with a certain (high) probability (but it can be composite). It is the case for the Miller-Rabin, the Fermat and the Solovay-Strassen algorithms. With them, we can determine if a number is composite, but only say that

a number have a certain probability to be prime because it passes a certain number of tests. These tests are criteria for the number not to be prime. The goal of these algorithms is to have a high probability for a number to be prime in a few tests.
The naive algorithm and the AKS algorithm give a "sure" answer. So if such an algorithm gives us the answer that a certain number is prime (respectively composite), then this number is really prime (that is sure) (respectively composite).

2.4 Running time

In this project, the running time of the algorithms is analyzed. They are given in function of the binary length of their input $n \in \mathbb{N}$ (written $len(n)$).

3 The naive algorithm

First, we present the naive algorithm, which is easy to understand and implement, but is really slow to run.

3.1 General idea

The naive algorithm is very simple. It consists of dividing the given number $n \in \mathbb{N}$ by a number $d \in \{2, \ldots, \lfloor\sqrt{n}\rfloor\}$. If d divides n, then the algorithm stops and declares that n is composite. If not, it continues with another number in $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$. So the algorithm runs until it reaches a divisor of n. If it has checked all the numbers in $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ and none of them has divided n, then n is prime.
The more common way is to check the numbers (possible divisors) in increasing order.

3.2 More precisely

This algorithm is composed by one principal function:

1. the **is_prime** function.

The **is_prime** function is the function that determines if a number is prime in the way explained above. To be a little faster, it first checks if the input is an even number. If it is equal to two, it declares that n is prime. If it is even but not equal to two, then it says that n is composite. Otherwise, it checks if at least one of the odd numbers in $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ divides n.
Here follows its pseudocode.

Algorithm 1 is_prime(n)

Input: $n \in \mathbb{N}$
Output: *true* for n prime, $false$ for n composite

```
1: if n = 2 then
2:     return true
3: end if
4: if n ≡ 0 (mod 2) then
5:     return false
6: end if
7: for i = 1, 3, 5, ..., √n do
8:     if n ≡ 0 (mod i) then
9:         return false
10:    end if
11: end for
12: return true
```

3.3 Running time

The advantage of this algorithm is that it never gives a false answer, but for large numbers, this algorithm is really slow. For exemple, if n is an integer of length β bits (which means that $\beta = \lceil \log_2 n \rceil$), then the algorithm has a running time in $O(\sqrt{n}) = O(\sqrt{2^\beta}) = O(2^{\frac{\beta}{2}})$, which is exponential in the length of n.

4 The Miller-Rabin algorithm

To understand and implement this algorithm, I based my researches on [1].

4.1 Number theory for the Miller-Rabin algorithm

Definition 4.1.1
We define $\mathbb{Z}_n$ *by*

$$\mathbb{Z}_n = \{0, 1, \ldots, n-1\}.$$

And

$$\mathbb{Z}_n^+ = \mathbb{Z}_n \setminus \{0\}.$$

Then if n is prime, we have $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$, where $\mathbb{Z}_n^*$ is the set of the invertible elements of $\mathbb{Z}_n$.
We can also see that with the usual addition and multiplication $\mathbb{Z}_n^+$ is a ring and $\mathbb{Z}_p^+$ is a field for p prime.

Theorem 4.1 (Chinese remainder theorem)
Let $n = n_1 \cdot \ldots \cdot n_k$ *be an integer where the* n_i *are pairwise relatively prime. Then*

$$\mathbb{Z}_n \cong \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}. \tag{1}$$

Proof. We can say that $n = n_1 \cdot m$ with $m = n_2 \cdot \ldots \cdot n_k$ and m and n_1 relatively prime. So we can prove the theorem for $k = 2$ and then prove the rest by induction over k.
We suppose that $n = n_1 \cdot n_2$ with n_1 and n_2 relatively prime. That means that $n_1\mathbb{Z} + n_2\mathbb{Z} = \gcd(n_1, n_2)\mathbb{Z} = \mathbb{Z}$. So there exist some u and v in $\mathbb{Z}$ such that $un_1 + vn_2 = 1$ (this result is called Bezout's theorem). To simplify we call $b_1 = un_1$ and $b_2 = vn_2$.
Let $\pi : \mathbb{Z} \to \mathbb{Z}_{n_1}, a \mapsto \bar{a}$ and $\rho : \mathbb{Z} \to \mathbb{Z}_{n_2}, a \mapsto \tilde{a}$ be the two canonic homomorphisms. We consider now the homomorphism

$$\phi : \mathbb{Z} \to \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2}, a \mapsto (\bar{a}, \tilde{a}).$$

We clearly see that $\ker \phi = n_1 n_2 \mathbb{Z} = n\mathbb{Z}$.
We want to show that ϕ is surjective. Let $(\bar{x}, \tilde{y}) \in \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2}$. And let $a = yb_1 + xb_2$ (where x is a preimage of $\bar{x}$ by the canonic homomorphism π and y is a preimage of $\tilde{y}$ by the canonic homomorphism ρ). Then working in $\mathbb{Z}_{n_1}$, we have

$$\begin{aligned} \bar{a} &= \overline{yb_1 + xb_2} = \bar{y}\bar{b_1} + \bar{x}\bar{b_2} = \bar{y}\bar{u}\bar{n_1} + \bar{x}\bar{v}\bar{n_2} = \bar{y}\bar{u}\bar{0} + \bar{x}\bar{v}\bar{n_2} = \bar{x}\bar{b_2} \\ &= \bar{x}\overline{(1 - b_1)} = \bar{x}\overline{(1 - un_1)} = \bar{x}(\bar{1} - \bar{u}\bar{n_1}) = \bar{x} - \bar{x}\bar{u}\bar{n_1} = \bar{x}. \end{aligned}$$

Similarly, working in $\mathbb{Z}_{n_2}$, we have $\tilde{a} = \tilde{y}$ too. So we have shown that $(\bar{a}, \tilde{a}) = \phi(a) = (\bar{x}, \tilde{y})$, which implies that ϕ is surjective.
We can now apply the first isomorphism theorem, which tells us that

$$\mathbb{Z}_n \cong \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2}.$$

□

We can repeat this theorem into the following corollary.

Corollary 4.2
Let n_1 and n_2 be two integers which are relatively prime. Then for some given a and b, we can solve the following congruence relations:

$$\begin{cases} x & \equiv & a & (\text{mod } n_1) \\ x & \equiv & b & (\text{mod } n_2). \end{cases}$$

And the solution is unique modulo $n_1 n_2$.

Corollary 4.3
Let n_1 and n_2 be two integers which are relatively prime and such that $n = n_1 n_2$. Then for some given x and a, we have:

$$\begin{cases} x & \equiv & a & (\text{mod } n_1) \\ x & \equiv & a & (\text{mod } n_2) \end{cases}$$

if and only if

$$x \equiv a \pmod{n}.$$

These two corollaries can be generalized for $n_1, \ldots, n_k$ some integers where the n_i are pairwise relatively prime.

Definition 4.1.2
*The **Euler totient function** $\phi(n)$ is the number of integers less than or equal to n which are relatively prime to n.*

Properties 4.4
The totient function has the following properties:

- $\phi(p) = p - 1$ *for each p prime;*
- $\phi(p^e) = p^{e-1}(p-1)$ *for each p prime and each e integer;*
- $\phi(mn) = \phi(m) \cdot \phi(n)$ *for each m and n relatively prime.*

Theorem 4.5 (Euler's theorem)
Let n be an integer. Then for every $a \in \mathbb{Z}_n^+$ with a and n relatively prime, n verifies the following equation:

$$a^{\phi(n)} \equiv 1 \pmod{n}. \tag{2}$$

Proof. Since $\gcd(a, n) = 1$, we have (using again Bezout's theorem seen in the proof of the Chinese remainder theorem, theorem 4.1) that a is an element of $\mathbb{Z}_n^*$. And the reciprocal is true too (with the same arguments). This implies that $\phi(n) = |\mathbb{Z}_n^*|$.

Let $\langle a \rangle$ be the subgroup of $\mathbb{Z}_n^*$ generated by a and let d be its order. We know that d divides $|\mathbb{Z}_n^*|$ (result called Lagrange's theorem). So there exists a $k \in \mathbb{N}$ such that $d \cdot k = |\mathbb{Z}_n^*|$.
By the definition of the order, we have $a^d = 1$. So we have

$$a^{\phi(n)} = a^{|\mathbb{Z}_n^*|} = a^{d \cdot k} = (a^d)^k = 1^k = 1$$

(all equations done in $\mathbb{Z}_n$). □

Theorem 4.6 (Fermat's little theorem)
Let p be a prime number. Then for every $a \in \mathbb{Z}_p^*$, p verifies the following equation:

$$a^{p-1} \equiv 1 \pmod{p}.$$

Proof. Since a lies in $\mathbb{Z}_p^*$, we have that $\gcd(a, p) = 1$. Then by the properties of the totient function, we know that p prime implies that $\phi(p) = p-1$. Then using Euler's theorem, we have

$$a^{p-1} = a^{\phi(p)} \equiv 1 \pmod{p}.$$

□

Theorem 4.7
Let n be an integer. Then

$$\{a \in \mathbb{Z}_n^* : a^{n-1} \equiv 1 \pmod{n}\} \leq \mathbb{Z}_n^*,$$

where $\leq$ means "to be a subgroup of".

Proof. Let $a, a_1, a_2 \in (\mathbb{Z}/n\mathbb{Z})^*$ be such that $a^{n-1} \equiv 1 \pmod{n}$, $a_1^{n-1} \equiv 1 \pmod{n}$ and $a_2^{n-1} \equiv 1 \pmod{n}$.
We have now to verify the criteria to be a subgroup:

- $1^{n-1} \equiv 1 \pmod{n}$;
- $(a_1 \cdot a_2)^{n-1} \equiv a_1^{n-1} \cdot a_2^{n-1} \equiv 1 \cdot 1 \equiv 1 \pmod{n}$;
- let a^{-1} be such that $a^{-1} \cdot a \equiv 1 \pmod{n}$, then

$$\begin{aligned} (a^{-1})^{n-1} &\equiv (a^{-1})^{n-1} \cdot 1 \equiv (a^{-1})^{n-1} \cdot a^{n-1} \\ &\equiv (a^{-1} \cdot a)^{n-1} \equiv 1^{n-1} \equiv 1 \pmod{n}. \end{aligned}$$

So

$$\{a \in \mathbb{Z}_n^* : a^{n-1} \equiv 1 \pmod{n}\}$$

is a subgroup of $\mathbb{Z}_n^*$. □

The equation

$$a^{n-1} \equiv 1 \pmod{n} \tag{3}$$

for $n \in \mathbb{N}$ is the criterion of the Fermat algorithm and one of the criterion of the Miller-Rabin algorithm.

4.1.1 Carmichael numbers

There exist some numbers that verify equation (3) for a given $a \in \mathbb{Z}_p^+$. They are called **pseudo-primes to the base** a. Furthermore, some of these pseudo-primes verify equation (3) for all $a \in \mathbb{Z}_p^+$.

Definition 4.1.3
The composite numbers that verify equation (3) *for all* $a \in \mathbb{Z}_p^+$ *are called the* ***Carmichael numbers****.*

So Fermat's little theorem is not a characterization of prime numbers, but only a criterion they verify.

Theorem 4.8
If n *is a Carmichael number, then* n *does not have any square factor.*

Proof. By contradiction, we suppose that there exists a prime number p such that p^2 divides n. This means that there exist two integers m and $k \geq 2$ such that $n = p^k m$ with $\gcd(m, p^k) = 1$. Now, let $g \in \mathbb{Z}_n^*$ be a generator of $\mathbb{Z}_{p^k}^*$ (which exists since $\mathbb{Z}_{p^k}^*$ is cyclic). By definition of a generator, we have $|\langle g \rangle| = |\mathbb{Z}_{p^k}^*| = \phi(p^k) = p^{k-1}(p-1)$. Since $n = p^k m$ with $\gcd(m, p^k) = 1$, we can apply the Chinese remainder theorem which implies that there exists an $a \in \mathbb{Z}_n$ such that

$$\begin{cases} a & \equiv & g & (\text{mod } p^k) \\ a & \equiv & 1 & (\text{mod } m). \end{cases}$$

The first equation implies that a is a generator of $\mathbb{Z}_{p^k}^*$ too. So a is invertible in $\mathbb{Z}_{p^k}$ (so a lies in $\mathbb{Z}_{p^k}^*$). And this means that $\gcd(a, p^k) = 1$, so $\gcd(a, p) = 1$, which implies that p does not divide a. The second equation implies that $a \equiv 1 \pmod{q_i}$ for every q_i prime dividing m. This means that q_i does not divide a. So none of the divisors of n divides a. So $\gcd(a, n) = 1$, which means that a lies in $\mathbb{Z}_n^*$.
Now we suppose that $a^{n-1} \equiv 1 \pmod{n}$. Since p^k divides n, we have that $a^{n-1} \equiv 1 \pmod{p^k}$. But we know that $a^{n-1} \equiv g^{n-1} \pmod{p^k}$. Together, this implies that $g^{n-1} \equiv 1 \pmod{p^k}$. By Euler's theorem, we know that $g^{p^{k-1}(p-1)} \equiv 1 \pmod{p^k}$ (where $p^{k-1}(p-1)$ is the smallest power which verifies this property). So we must have that $p^{k-1}(p-1)$ divides $n-1$. But since p^k divides n, we have that p divides n, which implies that $n \equiv 0 \pmod{p}$ and this is equivalent to $n - 1 \equiv -1 \pmod{p}$. So since $k - 1 \geq 1$, we obtain that $-1 \equiv n - 1 \equiv p^{k-1}(p-1)l \equiv 0 \pmod{p}$ (for some $l \in \mathbb{N}$). Which is a contradiction! So $a^{n-1} \not\equiv 1 \pmod{n}$.
We have shown that there exists an a in $\mathbb{Z}_n^*$ such that $a^{n-1} \not\equiv 1 \pmod{n}$. So n is not a Carmichael number. □

Theorem 4.9
If n *does not have any square factor, then it is a Carmichael number if and only if* $p - 1$ *divides* $n - 1$ *for every* p *dividing* n*.*

Proof. First, we prove the undirect way. We suppose that for every p dividing n we have that $p-1$ divides $n-1$. Let a be in $\mathbb{Z}_n^*$. This is equivalent saying that $\gcd(a,n)=1$. Using Fermat's little theorem, we now calculate $a^{n-1} \equiv a^{(p-1)k} \equiv 1^k \equiv 1 \pmod{p}$. We obtain that $a^{n-1}-1 \equiv 0 \pmod{p}$ for every p dividing n. This implies that p divides $a^{n-1}-1$ for every p dividing n and then n divides $a^{n-1}-1$ (it is the case since n does not have any square factor). So $a^{n-1}-1 \equiv 0 \pmod{n}$, which implies that $a^{n-1} \equiv 1 \pmod{n}$ for every $a \in \mathbb{Z}_n^*$. So n is a Carmichael number.
Now we show the direct way. We do it by proving the contrapositive. We suppose that there exists a p dividing n such that $p-1$ does not divide $n-1$. Let g be a generator of $\mathbb{Z}_p^*$. We apply now the same method as in theorem 4.8. We know by the Chinese remainder theorem that there exists an $a \in \mathbb{Z}_n$ such that

$$\begin{cases} a & \equiv & g & \pmod{p} \\ a & \equiv & 1 & \pmod{m}, \end{cases}$$

where $n = p \cdot m$ (since n does not have any square factor, p and m are relatively prime and we can apply the Chinese remainder theorem). As in the proof of theorem 4.8, the first equation implies that a lies in $\mathbb{Z}_p^*$. And this means that $\gcd(a,p)=1$. By the same reasoning as before we obtain that $\gcd(a,n)=1$, so a lies in $\mathbb{Z}_n^*$. With the first equation we obtain that $a^{n-1} \equiv g^{n-1} \pmod{p}$ too. But since g is a generator, we have that $|\langle g \rangle| = p-1$. This implies that $g^{p-1} \equiv 1 \pmod{p}$. So by definition of a generator, $g^s \equiv 1 \pmod{p}$ for a given s if and only if there exists a $k \in \mathbb{N}$ such that $s = k(p-1)$. But by hypothesis $p-1$ does not divide $n-1$. This means that $g^{n-1} \not\equiv 1 \pmod{p}$. So $a^{n-1} \not\equiv 1 \pmod{p}$, which implies that $a^{n-1} \not\equiv 1 \pmod{n}$.
So we have shown that there exists an a in $\mathbb{Z}_n^*$ such that $a^{n-1} \not\equiv 1 \pmod{n}$. So n is not a Carmichael number. □

Theorem 4.10
The number of prime factors in the decomposition in prime factors of a Carmichael number is at least 3.

Proof. Let n be a Carmichael number. Then by theorem 4.8 we know that n does not have any square factor. We will use the fact that it is a Carmichael number if and only if $p-1$ divides $n-1$ for every p dividing n.
We show that a Carmichael number n is product of at least three prime factors.
By contradiction, we suppose that it is not the case. Since n is not prime and does not have any square factor, there exist some p and q with $p < q$ such that $n = p \cdot q$. This implies that p divides n and since n is a Carmichael number this means that $p-1$ divides $n-1$. So $n-1 \equiv 0 \pmod{p-1}$. We can do the same reflexion about q and we obtain that $n-1 \equiv 0 \pmod{q-1}$ too. But $n-1 = pq-1 = p(q-1+1)-1 = p(q-1)+p-1 \equiv p-1 \pmod{q-1}$.

Since $p < q$, $p-1 < q-1$ and $p-1 \neq 0$ since $p > 1$. So $p-1 \not\equiv 0 \pmod{q-1}$. And this is a contradiction!
We have shown that if n is a Carmichael number, then n is product of at least three prime factors. □

We can see that Carmichael numbers are not very common, but the following theorem (stated without proof) tells us something important about them.

Theorem 4.11
There is an infinity of (different) Carmichael numbers.

Definition 4.1.4
*We say that a number $x \in \mathbb{N}$ is a **nontrivial square root of** $1 \pmod{n}$ if $x \neq \pm 1$ and $x^2 \equiv 1 \pmod{n}$.*

Definition 4.1.5
*The **discrete logarithm** of a modulo n to the base g is the number z such that $g^z \equiv a \pmod{n}$ where g is a primitive element of $\mathbb{Z}_n^*$ and a a number in $\mathbb{Z}_n^*$. This z is written $ind_{n,g}(a)$.*

Theorem 4.12 (Discrete logarithm theorem)
Let g be a primitive element of $\mathbb{Z}_n^*$. We have that

$$g^x \equiv g^y \pmod{n} \text{ is satisfied } \iff x \equiv y \pmod{\phi(n)} \text{ is satisfied .}$$

Theorem 4.13
Let p be an odd prime number and $e \geq 1$. Then the equation

$$x^2 \equiv 1 \pmod{p^e} \tag{4}$$

has exactly two solutions, which are $x = 1$ and $x = -1$.

Proof. We know that for every $p > 2$ and for all $e \geq 1$ $\mathbb{Z}_{p^e}^*$ is a cyclic group. This implies that there exists $g \in \mathbb{Z}_{p^e}^*$ such that g is a primitive element of $\mathbb{Z}_{p^e}^*$. We have that $|\langle g \rangle| = |\mathbb{Z}_{p^e}^*| = \phi(p_e) = p^{e-1}(p-1)$.
So for every $a \in \mathbb{Z}_{p^e}^*$ there exists a $k \in \{0, \ldots, p^{e-1}(p-1)-1\}$ such that $a \equiv g^k \pmod{p^e}$. This k is written $ind_{p^e,g}(a)$. We can now rewrite equation (4) as

$$\left(g^{ind_{p^e,g}(x)}\right)^2 \equiv g^{ind_{p^e,g}(1)} \pmod{p^e}. \tag{5}$$

By the theorem of the discrete logarithm (theorem 4.12), we obtain

$$ind_{p^e,g}(x) \cdot 2 \equiv ind_{p^e,g}(1) \equiv 0 \pmod{\phi(p^e)}. \tag{6}$$

But since p is an odd prime number, we have that $\gcd(2, p^{e-1}(p-1)) = 2$. Since 2 divides 0 we have that $2 \cdot ind_{p^e,g}(x) \equiv 0 \pmod{\phi(p^e)}$ has a solution. But we know that an equation of type $ax \equiv b \pmod{n}$ has no solution or

exactly $\gcd(a, n)$ distinct solutions. But since we have shown that equation (6) has a solution, it has exactly $\gcd(2, \phi(p^e)) = 2$ solutions.
By the theorem of discrete logarithm, we have that equation (5) is satisfied if and only if equation (6) is satisfied. So (5) has exactly two solutions.
Clearly, $x = 1$ and $x = -1$ are the two unique solutions of equation (4). □

The contrapositive of this theorem is used to show the following corollary, which is a criterion of the Miller-Rabin algorithm.

Corollary 4.14
Let n be in $\mathbb{N}$. If there exists a nontrivial square root of $1 \pmod n$, then n is composite.

Theorem 4.15
Let n be an odd integer. Then

$$|\{a \in \mathbb{Z}_n^* : a^2 \equiv 1 \pmod n\}| = 2^{\sharp n},$$

where $\sharp n$ is equal to the number of different prime factors of n.

Proof. Let $n = p_1^{\alpha_1} \cdot \ldots \cdot p_t^{\alpha_t}$ the decomposition of n in prime factors, where p_i is prime for all $i \in \{1, \ldots, t\}$, $p_i \neq p_j$ for all $i \neq j$ and α_i is an integer for all $i \in \{1, \ldots, t\}$. Since n is an odd integer, we have that $p_i \neq 2$ for every $i \in \{1, \ldots, t\}$.
By the Chinese remainder theorem, we have

$$a^2 \equiv 1 \pmod n \iff a_i^2 \equiv 1 \pmod{p_i^{\alpha_i}} \ \forall i \in \{1, \ldots, t\},$$

where $(a_1, \ldots, a_t)$ is defined by the isomorphism of the proof of the Chinese remainder theorem (theorem 4.1). So we have the correspondence $a \leftrightarrow (a_1, \ldots, a_t)$ for every a in $\mathbb{Z}_n^*$.
But we know that the second equation above has exactly two solutions for each $i \in \{1, \ldots, t\}$ which are $a_i = 1$ and $a_i = -1$. So there are 2^t possible vectors $(a_1, \ldots, a_t)$ which verify the equation above. And, because of the isomorphism, this implies that the first equation has exactly $2^t = 2^{\sharp n}$ solutions. □

Theorem 4.16
Let n be an integer. Then the set

$$\left\{ \begin{array}{ll} a \in \mathbb{Z}_n^* : & a^{n-1} \equiv 1 \pmod n \text{ and} \\ & a^{2k} \equiv 1 \pmod n \Rightarrow a^k \equiv \pm 1 \pmod n \ \forall k \in \{1, \ldots, \frac{n-1}{2}\} \end{array} \right\}$$

is contained in a subgroup of $\mathbb{Z}_n^*$.
If n is composite, then this set is contained in a proper subgroup of $\mathbb{Z}_n^*$.

Proof. To simplify the notation, we call S the set above.
When n is prime, by theorems 4.6 and 4.13, every $a \in \mathbb{Z}_n^*$ verifies the properties of S. So we obtain that $S = \mathbb{Z}_n^*$. For n prime the result is clear.
We have now to show that, if n is composite, S is contained in a proper subgroup of $\mathbb{Z}_n^*$. So suppose now that n is composite.
First, remark that S is the set of the non-witnesses of n. Clearly the two properties are the one characterizing the non-witnesses of n and we show that the non-witnesses lie in $\mathbb{Z}_n^*$. In fact, if $a \in \mathbb{Z}_n^+$ is a non-witness then it has to verify $a^{n-1} \equiv 1 \pmod{n}$, which we can write as $a \cdot a^{n-2} \equiv 1 \pmod{n}$. This implies that there exists a solution for equation $ax \equiv 1 \pmod{n}$. But since we know that equation $ax \equiv b \pmod{n}$ has a solution for x if and only if $gcd(a, n)$ divides b, we have that $gcd(a, n)$ divides 1. So $gcd(a, n) = 1$. And this is equivalent to say that $a \in \mathbb{Z}_n^*$.
We will find now a proper subgroup B of $\mathbb{Z}_n^*$ which contains every non-witnesses of n, so such that $S \subseteq B$. We divide this research into two cases: if n is not a Carmichael number and if it is.

- Suppose that n is not a Carmichael number. So this implies that there exists a $x \in \mathbb{Z}_n^*$ such that $x^{n-1} \not\equiv 1 \pmod{n}$. Let

  $$B = \{b \in \mathbb{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\}.$$

  Since $1 \in B$, B is not empty. We have already shown that B is a subgroup of $\mathbb{Z}_n^*$ (theorem 4.7). Remark that every non-witness is in B, so $S \subseteq B$. But since $x \in \mathbb{Z}_n^* \setminus B$, we have that B is a proper subgroup of $\mathbb{Z}_n^*$.

- Now suppose that n is a Carmichael number. This implies that we have $x^{n-1} \equiv 1 \pmod{n}$ for all $x \in \mathbb{Z}_n^*$. We know that n does not contain any square factor (theorem 4.8) and is the product of at least three primes (theorem 4.10). So we can write n as $n = p_1 \cdot \ldots \cdot p_k$ with $k \geq 3$. So there exist n_1, n_2 odd integers relatively prime such that $n = n_1 n_2$ (we can take for example $n_1 = p_1$ and $n_2 = p_2 \cdot \ldots \cdot p_k$). Let u be an odd integer and t be a positive number such that $n - 1 = 2^t u$. Now we call a pair (v, j) of integers **acceptable** if $v \in \mathbb{Z}_n^*$, $j \in \{0, 1, \ldots t\}$ and

  $$v^{2^j u} \equiv -1 \pmod{n}.$$

  There exist acceptable pairs (since u is odd the pair $(n-1, 0)$ is acceptable). Now we choose the largest j such that there exists a pair (v, j) acceptable and we fix v to have a pair (v, j) acceptable. Let

  $$B = \{x \in \mathbb{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

  Since j is fixed, B is clearly a subgroup of $\mathbb{Z}_n^*$. Now take a in S. Then if it does not exist j' such that $a^{2^{j'} u} \equiv -1 \pmod{n}$ then $a^{2^k u} \equiv 1 \pmod{n}$

for all $k \in \{0, 1, \dots t\}$, so $a^{2^j u} \equiv 1 \pmod{n}$. If there exists j' such that $a^{2^{j'} u} \equiv -1 \pmod{n}$, then by maximality of j, we have $j' \leq j$, so $a^{2^j u} \equiv \pm 1 \pmod{n}$. So we have shown that $S \subseteq B$.
Now we show that there exists $w \in \mathbb{Z}_n^* \setminus B$. Using corollary 4.2, we know that there exists $w \in \mathbb{Z}_n$ such that

$$\begin{cases} w & \equiv & v & (\text{mod } n_1) \\ w & \equiv & 1 & (\text{mod } n_2). \end{cases}$$

But since $v^{2^j u} \equiv -1 \pmod{n}$, by corollary 4.3 we have $v^{2^j u} \equiv -1 \pmod{n_1}$. So we obtain

$$\begin{cases} w & \equiv & -1 & (\text{mod } n_1) \\ w & \equiv & 1 & (\text{mod } n_2). \end{cases}$$

By corollary 4.3, $w \not\equiv 1 \pmod{n_1}$ implies that $w \not\equiv 1 \pmod{n}$ and $w \not\equiv -1 \pmod{n_2}$ implies that $w \not\equiv -1 \pmod{n}$. So $w \not\equiv \pm 1 \pmod{n}$, which means that $w \notin B$. We have now to show that $w \in \mathbb{Z}_n^*$. Since $v \in \mathbb{Z}_n^*$, we have $gcd(v, n) = 1$, which implies that $gcd(v, n_1) = 1$ and $gcd(v, n_2) = 1$. And since $w \equiv v \pmod{n_1}$, $gcd(w, n_1) = 1$. But since $w \equiv 1 \pmod{n_2}$, we have that $gcd(w, n_2) = 1$ (by the same argument as in the beginning of the proof). So we obtain that $gcd(w, n) = 1$, which means that $w \in \mathbb{Z}_n^*$. Finally we have $w \in \mathbb{Z}_n^* \setminus B$, which implies that B is a proper subgroup.

In each case, for n composite, we have that $S \subseteq B$ and $B < \mathbb{Z}_n^*$. □

This theorem will help us to calculate the number of witnesses of n (by observations on the cardinality of the proper subgroup B). This is done in theorem 4.17.

4.2 General idea

The Miller-Rabin algorithm detects if a number is prime or not by testing two criteria.
First recall Fermat's little theorem (theorem 4.6). It says that if p is a prime number, then for every $a \in \mathbb{Z}_p^+$, p verifies $a^{p-1} \equiv 1 \pmod{p}$. So if, for a certain number $n \in \mathbb{N}$, we can find an $a \in \mathbb{Z}_p^+$ such that n does not verify equation (3), then we can ensure that n is composite. And this is the first criterion used in the Miller-Rabin algorithm.The goal of Miller-Rabin algorithm is to check equation (3) for a certain number of a, which are randomly chosen in $\{1, \dots, n\}$, such that the probability, for n composite, of determining that n is composite increases.
Recall corollary 4.14. The other criterion used in this algorithm is to detect a non trivial square root of 1 $(\text{mod } n)$. If there is one, then we can ensure that n is composite.
We can easily see that the Miller-Rabin algorithm is a randomized algorithm.

4.3 More precisely

Let $n \in \mathbb{N}$ be the input of our algorithm. We want to know if n is prime or composite.
The Miller-Rabin algorithm uses a random function that gives it a random number $a \in \mathbb{Z}_n^+$. This algorithm is composed by three principal functions:

1. the **witness** function;
2. the **modular_exponentiation** function;
3. the **miller_rabin** function.

The **witness** function tells us if a is a "witness" that n is composite or not (with the criterion of Fermat's little theorem).
First we can find an odd integer u and a positive number t such that $n-1 = 2^t u$. So the binary representation of $n-1$ is the binary representation of u followed by t zeros. So we have $a^{n-1} \equiv (a^u)^{2^t} \pmod{n}$. We can now calculate $a^{n-1} \pmod{n}$ by calculating $a^u \pmod{n}$ and then taking t times the square of the answer. To calculate $a^u \pmod{n}$ we use the **modular_exponentiation** function.
If in one of the squaring steps a non trivial square root of 1 is found, then the function stops and says that n is composite (because of corollary 4.14). And if at the end of the procedure $a^{n-1} \not\equiv 1 \pmod{n}$, then the function says that n is composite (because of the contraposition of Fermat's little theorem).
If we're not in one of these cases, then the function says that n is prime.

The **modular_exponentiation** function calculates $a^u \pmod{n}$. We can calculate it directly, but if a and u are large, then a^u will be really large. So it is better to calculate that in steps and taking the value modulo n in each step. So we will work with smaller numbers.
The function consists of doing repeated squaring steps and using the binary representation of u. We can write $u = \Sigma_{i=0}^{k} \alpha_i 2^i$ with $\alpha_i \in \{0, 1\}$. So the list $\langle \alpha_k, \ldots, \alpha_0 \rangle$ is the binary representation of u. It now calculates $a \pmod{n}$, $a^2 \pmod{n}$, $a^4 \pmod{n}$, ... (each time squaring the value obtained at the previous step). Then it takes all the values $a^{2^i} \pmod{n}$ obtained in the steps where $\alpha_i = 1$ and multiplies them. Since we have

$$a^u = a^{\Sigma_{i=0}^{k} \alpha_i 2^i} = \Pi_{i=0}^{k} a^{\alpha_i 2^i} = \Pi_{i=0}^{k} (a^{2^i})^{\alpha_i},$$

this result is equal to $a^u \pmod{n}$.
So using the binary representation of u, the **modular_exponentiation** function calculates $a^u \pmod{n}$.

The **miller_rabin** function does the following operations with a certain number $a \in \mathbb{Z}_p^+$ (randomly chosen, in the algorithm by a function called

random(1,n-1)) and then if it does not stop before, it iterates the same operation s times (s chosen by the user) with other numbers.
If the **witness** function of a and n returns that n is composite, then the algorithm stops and says that n is composite. If not, then it tries again with another number in $\mathbb{Z}_p^+$. Again, if the **witness** function of this number and n returns that n is composite, then the algorithm stops and says that n is composite. And so on, until the algorithm has made s iterations. If none of the s numbers tried by the algorithm gives the answer that n is composite, then the **miller_rabin** function declares that n is prime.

We see that, in this algorithm, if n is declared composite, then it sure is. But if n is declared prime, it can be prime or composite.

The following pseudocodes are the ones of these functions.

Algorithm 2 modular_exponentiation(a, u, n)

Input: $a, u, n \in \mathbb{N}$
Output: $a^u \pmod n$

```
1: d ← 1
2: ⟨α_k, ..., α_0⟩ the binary representation of u
3: for i = k, ..., 0 (decreasing) do
4:     d ← d · d (mod n)
5:     if α_i = 1 then
6:         d ← d · a (mod n)
7:     end if
8: end for
9: return d
```

4.4 Running time

The **modular_exponentiation** function costs $O(len(n))$ for the $len(n)$ successive squaring steps. Since it deals only with numbers of lenght $len(n)$ (because all numbers are taken modulo n) every multiplication between two numbers in $\mathbb{Z}_n$ costs $O(len(n)^2)$. Finally the **modular_exponentiation** function costs $O(len(n)^3)$ bits operations. This implies that the Miller-Rabin algorithm costs $O(s \cdot len(n)^3)$ operations.

4.5 Analysis of the errors

We have seen that if n is declared composite, then it sure is. But if n is declared prime, it can be prime or composite. So we have to understand why the algorithm declares n prime when it is composite. This depends on the random numbers a tested and on s. Recall that a is called a "witness" of

Algorithm 3 witness(a, n)

Input: $a, n \in \mathbb{N}$

Output: $true$ for a witness of n, $false$ for a non-witness of n

```
1: n − 1 = 2^t u, with t ≥ 1 integer and u odd integer
2: x_0 ← modular_exponentiation(a, u, n)
3: for i = 1, ..., t do
4:    if x_i = 1 and x_{i−1} ≠ 1 and x_{i−1} ≠ n − 1 then
5:       return true
6:    end if
7: end for
8: if x_t ≠ 1 then
9:    return true
10: end if
11: return false
```

Algorithm 4 miller_rabin(n, s)

Input: $n, s \in \mathbb{N}$

Output: $true$ for n prime, $false$ for n composite

```
1: for j = 1, ..., s do
2:    a ← random(1, n − 1)
3:    if witness(a, n) then
4:       return false
5:    end if
6: end for
7: return true
```

n if $a^{n-1} \not\equiv 1 \pmod{n}$ (so this implies that n is composite). The following theorem tells us something important.

Theorem 4.17
If n is an odd composite number, the number of witnesses of n is at least $\frac{n-1}{2}$.

Proof. We have shown in theorem 4.16 that the set (called S) of non-witnesses of n for the Miller-Rabin algorithm is contained in a proper subgroup of $\mathbb{Z}_n^*$, called B. By the Lagrange theorem, we know that the order of a subgroup divides the order of the group. So $|B|$ divides $|\mathbb{Z}_n^*| = \phi(n) \leq n-1$. So, since $S \subseteq B$, we have $|S| \leq |B| \leq \frac{|\mathbb{Z}_n^*|}{2} \leq \frac{n-1}{2}$. This means that the number of non-witnesses is at most $\frac{n-1}{2}$. So the number of witnesses is at least $\frac{n-1}{2}$. □

So each time that we test the **witness** function, we have the probability of at least $\frac{1}{2}$ to obtain true. So we have the following theorem (stated without proof).

Theorem 4.18
For every odd integer $n > 2$ and every positive integer s (number of random numbers tested if they are witnesses or not), the probability of error of the Miller-Rabin algorithm is at most 2^{-s}.

In practice, we can run the algorithm for $s = 50$ to have a good probability that if n is composite we catch a witness.

5 The Fermat algorithm

The Fermat algorithm is based on the same principles as the Miller-Rabin algorithm, so uses the Fermat's little theorem as principal criterion. But it is less restrictive because it does not check the square roots when doing the calculus. Its criterion is the one in 4.7.

5.1 General idea

The Fermat algorithm is exactly the same algorithm as the Miller-Rabin algorithm, except for the checking of square root of 1 (mod n).

5.2 More precisely

Let $n \in \mathbb{N}$ be the input of our algorithm. We want to know if n is prime or composite.
The Fermat algorithm is composed by exactly the same functions as the Miller-Rabin algorithm:

1. the **witness** function;
2. the **modular_exponentiation** function;
3. the **miller_rabin** function.

The only thing that changes is the **witness** function. It does not use the corollary 4.14, but only bases its criterion on the Fermat's little theorem (theorem 4.6).
The **witness** function in this algorithm tells us if a is a "witness" that n is composite or not (with the criterion of Fermat's little theorem).
We calculate $a^{n-1} \pmod{n}$ with the **modular_exponentiation** function.
If $a^{n-1} \not\equiv 1 \pmod{n}$, then the function says that n is composite (because of the contraposition of Fermat's little theorem).
If we're not in this case, the function says that n is prime.

Here follows the pseudocode of the **witness** function for the Fermat algorithm.

5.3 Running time

Since this algorithm is quite the same as the Miller-Rabin algorithm (it only checks less things, but performs the same calculations), it has the same running time as the Miller-Rabin algorithm.

Algorithm 5 witness(a, n)

Input: $a, n \in \mathbb{N}$

Output: $true$ for a witness of n, $false$ for a non-witness of n

1: $n - 1 = 2^t u$, with $t \geq 1$ integer and u odd integer
2: $x_0 \leftarrow$ modular_exponentiation(a, u, n)
3: **if** $x_t \neq 1$ **then**
4: return $true$
5: **end if**
6: return $false$

6 The Solovay-Strassen algorithm

To understand and implement this algorithm, I based my researches on [2] and [3].

6.1 Number theory for the Solovay-Strassen algorithm

Definition 6.1.1
*Let a and n be two integers. We say that a is a **quadratic residue** modulo n if there exists an integer r such that $a \equiv r^2 \pmod{n}$.*

Definition 6.1.2
*We define the **Legendre symbol** for all integers a and all primes $p > 2$, written $\left(\frac{a}{p}\right)$ $(\in \mathbb{N})$, by*

- $\left(\frac{a}{p}\right) = 0$ *if p divides a;*
- $\left(\frac{a}{p}\right) = 1$ *if p does not divide a and a is a quadratic residue modulo p;*
- $\left(\frac{a}{p}\right) = -1$ *if a is not a quadratic residue modulo p.*

Definition 6.1.3
*The **Jacobi symbol** is the Legendre symbol with extended definition domain. For all integers a and all odd integers $n \geq 3$, we define $\left(\frac{a}{n}\right)$ by*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \cdot \ldots \cdot \left(\frac{a}{p_r}\right)^{\alpha_r},$$

where $n = p_1^{\alpha_1} \cdot \ldots \cdot p_r^{\alpha_r}$ with p_i prime.
We see that $\left(\frac{a}{n}\right)$ lies in $\{0, \pm 1\}$.

Since by construction the Legendre symbol depends only on the value of a modulo p, we have that

$$\left(\frac{a}{p}\right) = \left(\frac{a \pmod{p}}{p}\right).$$

And by the definition of the Jacobi symbol, we have the same property for it. So that

$$\left(\frac{a}{n}\right) = \left(\frac{a \pmod{n}}{n}\right).$$

Properties 6.1
Let m, m' be some integers and $n \geq 3$, $n' \geq 3$ be some odd integers.
The Jacobi symbol has the following properties:

- $\left(\frac{m \cdot m'}{n}\right) = \left(\frac{m}{n}\right) \cdot \left(\frac{m'}{n}\right)$;

- $(\frac{m}{n \cdot n'}) = (\frac{m}{n}) \cdot (\frac{m}{n'})$.

To calculate the Jacobi symbol we need three more functions.

Definition 6.1.4
We define the ***epsilon function*** *for a positive odd integer n as*

$$\epsilon(n) = (-1)^{\frac{n-1}{2}} = \left(\frac{-1}{n}\right).$$

Definition 6.1.5
We define the ***omega function*** *for a positive odd integer n as*

$$\omega(n) = (-1)^{\frac{n^2-1}{8}} = \left(\frac{2}{n}\right).$$

Definition 6.1.6
We define the ***theta function*** *for two positive odd integers m and n which are relatively prime as*

$$\theta(m,n) = (-1)^{\frac{(m-1)(n-1)}{4}} = \left(\frac{\epsilon(n)}{m}\right).$$

These functions are used in the Solovay-Strassen algorithm to calculate the Jacobi symbol.

Theorem 6.2 (Law of Quadratic Reciprocity (LQR))
Let p and q be two odd prime numbers. Then we have

$$\left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)(q-1)}{4}} \cdot \left(\frac{p}{q}\right) = \begin{cases} -\left(\frac{p}{q}\right) & \text{if } p \equiv q \equiv 3 \pmod 4; \\ \left(\frac{p}{q}\right) & \text{otherwise.} \end{cases}$$

Theorem 6.3 (The Euler formula)
Let p be an odd prime number. Then for every integer a, we have

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod p. \tag{7}$$

Proof. Since p is an odd prime number, $p > 2$. Let a be a number in $\mathbb{Z}_p$.
If a is not relatively prime to p, then since p is prime, p divides a. So both sides of equation (7) are 0.
Now suppose that a and p are relatively prime. This implies that a lies in $\mathbb{Z}_p^*$. By Fermat's little theorem, we have that $a^{p-1} \equiv 1 \pmod p$. So $a^{\frac{p-1}{2}} \equiv \pm 1 \pmod p$ (by theorem 4.13). But by definition of a, we have that p does not divide a. Then by the definition of the Legendre symbol, $\left(\frac{a}{p}\right) = \pm 1$. Now let g be a generator of $\mathbb{Z}_p^*$. So there exists j such that $a = g^j$. But we have that a is a quadratic residue modulo p if and only if j

is even. And we have that $a^{\frac{p-1}{2}} \equiv g^{j\frac{p-1}{2}} \pmod{p}$ equal to 1 if and only if $j\frac{p-1}{2}$ is divisible by $p-1$. And this is the case if and only if j is even. So we obtain that $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$ if and only if a is a quadratic residue modulo p, so if and only if $\left(\frac{a}{p}\right) = 1$. Finally, we obtain that $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \pmod{p}$ (since p is odd, $\frac{p-1}{2} \in \mathbb{N}$). □

And conversely we have the following theorem, which is the principal criterion of the Solovay-Strassen algorithm.

Theorem 6.4 (The Solovay-Strassen theorem)
Let $n > 2$ be an odd integer such that for every integer a which is relatively prime to n we have

$$\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}. \tag{8}$$

Then n is prime.

Proof. Since a is relatively prime to n, we have that a lies in $\mathbb{Z}_n^*$.
By contradiction, we suppose that n is not prime. Since $gcd(a, n) = 1$, we have that $\left(\frac{a}{n}\right)^2 = 1$ and by the hypothesis of the theorem, we obtain that $a^{n-1} \pmod{n} = \left(\frac{a}{n}\right)^2 = 1$. Since this holds for every $a \in \mathbb{Z}_n^*$, we have that n is a Carmichael number. We have shown that this implies that n is the product of at least three primes (theorem 4.10).
Let $n = p_1 \cdot \ldots \cdot p_r$ with $r \geq 3$ be the decomposition of n in prime numbers (since n is a Carmichael number, it does not have any square factor, by theorem 4.8). Let a be in $\mathbb{Z}_n^*$. Now, let α_i be the class of a modulo p_i for every $i \in \{1, \ldots, r\}$. So we have

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right) \cdot \ldots \cdot \left(\frac{a}{p_r}\right) = \left(\frac{\alpha_1}{p_1}\right) \cdot \ldots \cdot \left(\frac{\alpha_r}{p_r}\right). \tag{9}$$

But, by assumption, we have that $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}$. We reduce this equation modulo p_i for every $i \in \{1, \ldots, r\}$ to obtain

$$\left(\frac{a}{n}\right) \equiv \alpha_i^{\frac{n-1}{2}} \pmod{p_i} \ \forall i \in \{1, \ldots, r\}. \tag{10}$$

So when we take equations (9) and (10) together, we have

$$\left(\frac{\alpha_1}{p_1}\right) \cdot \ldots \cdot \left(\frac{\alpha_r}{p_r}\right) \equiv \alpha_i^{\frac{n-1}{2}} \pmod{p_i} \ \forall i \in \{1, \ldots, r\}.$$

Now, by the Chinese remainder theorem, we know that there exist $b \in \mathbb{N}$ and $\beta_1, \ldots, \beta_r$ such that $\beta_i \equiv b \pmod{p_i} \ \forall i \in \{1, \ldots, r\}$, with $\beta_i = \alpha_i$ $\forall i \in \{1, \ldots, r-1\}$ and $\beta_r \neq \alpha_r$. So we have that

$$\alpha_i^{\frac{n-1}{2}} \equiv \beta_i^{\frac{n-1}{2}} \pmod{p_i} \ \forall i \in \{1, \ldots, r-1\}.$$

But we have for all $i \in \{1, \ldots, r-1\}$ that

$$\alpha_i^{\frac{n-1}{2}} \pmod{p_i} = \left(\frac{\alpha_1}{p_1}\right) \cdot \ldots \cdot \left(\frac{\alpha_r}{p_r}\right) \neq \left(\frac{\beta_1}{p_1}\right) \cdot \ldots \cdot \left(\frac{\beta_r}{p_r}\right) = \beta_i^{\frac{n-1}{2}} \pmod{p_i}.$$

This is in contradiction with the fact that $r > 1$.
So n is prime. □

Theorem 6.5
Let n be an integer. Then

$$\{a \in \mathbb{Z}_n^* : \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}\} \leq \mathbb{Z}_n^*$$

and if n is composite it is a proper subgroup of $\mathbb{Z}_n^*$.

Proof. First we show that it is a subgroup.
So we have to verify the criteria to be a subgroup:

- $\left(\frac{1}{n}\right) = \left(\frac{1}{p_1^{\alpha_1} \cdot \ldots \cdot p_r^{\alpha_r}}\right) = \left(\frac{1}{p_1^{\alpha_1}}\right) \cdot \ldots \cdot \left(\frac{1}{p_r^{\alpha_r}}\right) = \left(\frac{1}{p_1}\right)^{\alpha_1} \cdot \ldots \cdot \left(\frac{1}{p_r}\right)^{\alpha_r}$.
  But $1 \equiv 1^2 \pmod{p_i}$ and p_i does not divide 1 for every $i \in \{1, \ldots, r\}$. This implies that $\left(\frac{1}{p_i}\right) = 1$ for all $i \in \{1, \ldots, r\}$. So we have that $\left(\frac{1}{n}\right) = 1 \equiv 1^{\frac{n-1}{2}} \pmod{n}$;
- let a and b be in our set. So we have that $\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right) \cdot \left(\frac{b}{n}\right) \equiv a^{\frac{n-1}{2}} \cdot b^{\frac{n-1}{2}} \equiv (ab)^{\frac{n-1}{2}} \pmod{n}$;
- let a be in our set. Let $a^{-1} \in \mathbb{Z}_n^*$ be such that $aa^{-1} \equiv 1 \pmod{n}$. So $\left(\frac{a}{n}\right) \cdot \left(\frac{a^{-1}}{n}\right) = \left(\frac{aa^{-1}}{n}\right) = \left(\frac{1}{n}\right) = 1 \pmod{n}$. So $\left(\frac{a^{-1}}{n}\right)$ is the inverse of $\left(\frac{a}{n}\right)$ in $\mathbb{Z}_n^*$. And we have that $a^{\frac{n-1}{2}} \cdot (a^{-1})^{\frac{n-1}{2}} \equiv (aa^{-1})^{\frac{n-1}{2}} \equiv 1^{\frac{n-1}{2}} \equiv 1 \pmod{n}$ too. So $(a^{-1})^{\frac{n-1}{2}}$ is the inverse of $a^{\frac{n-1}{2}}$ in $\mathbb{Z}_n^*$. But since $\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \pmod{n}$, using the fact that we are in a group (so the inverse is unique) we obtain $\left(\frac{a^{-1}}{n}\right) = (a^{-1})^{\frac{n-1}{2}} \pmod{n}$.

So

$$\{a \in \mathbb{Z}_n^* : \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}\}$$

is a subgroup of $\mathbb{Z}_n^*$.
Now we want to show that if n is composite then our set is a proper subgroup of $\mathbb{Z}_n^*$.
Knowing that there exist integers which are not prime, we can use the contrapositive of the Solovay-Strassen theorem, which implies that for n composite there exists an $a \in \mathbb{Z}_n^*$ such that $\left(\frac{a}{n}\right) \not\equiv a^{\frac{n-1}{2}} \pmod{n}$. So our set is a proper subgroup if n is composite. □

This theorem will help us to calculate the number of witnesses of n (by observations on the cardinality of this proper subgroup). This is done in theorem 6.6.

6.2 General idea

Recall Solovay-Strassen theorem (theorem 6.4). It says that if $n > 2$ is an odd integer such that for every integer a which is relatively prime to n we have $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}$, then n is prime. So the Solovay-Strassen algorithm checks equation (8) for some $a \in \mathbb{Z}_n^*$. If it finds an a such that this equation is not verified, then it declares that n is composite. And if, for a large number of such a (their number, $s \in \mathbb{N}$, is chosen by the user), equation (8) is verified, the Solovay-Strassen algorithm declares that n is prime. These numbers a are randomly chosen in $\{1, \ldots, n\}$.

6.3 More precisely

Let $n \in \mathbb{N}$ be the input of our algorithm. We want to know if n is prime or composite.
The Solovay-Strassen algorithm uses a random function that gives it a random number $a \in \mathbb{Z}_n^*$. This algorithm is composite by three principal functions:

1. the **modular_exponentiation** function;
2. the **jacobi** function;
3. the **solovay_strassen** function.

The **modular_exponentiation** function is the same as in the Miller-Rabin algorithm. It is used to calculate $a^{\frac{n-1}{2}} \pmod{n}$.

The **jacobi** function calculates the Jacobi symbol. We want to calculate $\left(\frac{u}{v}\right)$ for an integer u and an odd integer $v \geq 3$. The principal steps of this calculation are the following:
Let $t \geq 0$ be an integer and r be an odd integer such that $u = 2^t r$. So we have

$$\begin{aligned}\left(\frac{u}{v}\right) &= \left(\frac{2^t r}{v}\right) \\ &= \left(\frac{2^t}{v}\right) \cdot \left(\frac{r}{v}\right).\end{aligned}$$

The second equation is obtained by applying the properties of the Jacobi symbol. But we know that $\left(\frac{2^t}{v}\right) = \pm 1$ because v is odd and 2 (and so 2^t)

does not divide v. And then we can apply the LQR. We have then

$$\begin{aligned}\left(\frac{2^t}{v}\right)\cdot\left(\frac{r}{v}\right) &= \pm\left(\frac{r}{v}\right)\\ &= \pm\left(\frac{v}{r}\right)\\ &= \pm\left(\frac{s}{r}\right),\end{aligned}$$

where s is an integer such that $v = k\cdot r + s$ for some integer k.
We now set $s = \prod p_i^{\alpha_i}$ with α_i integers, p_i primes and $p_i \neq p_j$ for all $i \neq j$. So

$$\begin{aligned}\pm\left(\frac{s}{r}\right) &= \pm\left(\frac{\prod p_i^{\alpha_i}}{r}\right)\\ &= \pm\prod\left(\frac{p_i}{r}\right)^{\alpha_i}.\end{aligned}$$

Again, the second equality is obtained by using the properties of the Jacobi symbol.
If one of the p_i is equal to 2, then we have $\left(\frac{2}{r}\right)^{\alpha_i} = \pm 1$. So we can assume that none of the p_j remaining are equal to 2. We can use again the LQR to have

$$\pm\prod\left(\frac{p_i}{r}\right)^{\alpha_i} = \pm\prod\left(\frac{r}{p_i}\right)^{\alpha_i}.$$

There are now only Legendre symbols and we can calculate them directly (using the definition of Legendre symbol). And the function returns the calcuated value of $(\frac{u}{v})$.

The **solovay_strassen** function tests if the **jacobi** function of a and n is equal to the **modular_exponentiation** function of a and n (with exponent $\frac{n-1}{2}$). If the answer at this question is no, then the **solovay_strassen** function says that n is composite. If not, then it declares that n is prime.

The following pseudocodes are the ones of the **jacobi** function and the **solovay_strassen** function.

Recall that $\epsilon(\cdot)$, $\omega(\cdot)$ and $\theta(\cdot)$ are the functions defined in 6.1.4, 6.1.5 and 6.1.6.

6.4 Running time

The cost of the computation of the jacobi symbol as described above (in the **jacobi** function) is $O(len(n)^2)$. The computation of the right part of the equation of the Solovay-Strassen theorem (which is done by the

Algorithm 6 jacobi(u, v)

Input: $u \in \mathbb{N}$ and $v \geq 3$ odd integer
Output: $\epsilon = \left(\frac{u}{v}\right)$

$\epsilon \leftarrow 0$
if $u \geq 0$ **then**
  $\epsilon \leftarrow 1$
else
  $u \leftarrow -u$
  $\epsilon \leftarrow \epsilon(v)$
end if
while $u \neq 1$ and $u \neq 0$ **do**
  if $u \equiv 0 \pmod 2$ **then**
    $u \leftarrow \frac{u}{2}$
    $\epsilon \leftarrow \epsilon \cdot \omega(v)$
  else
    $u' \leftarrow u$
    $u \leftarrow v \pmod u$
    $v \leftarrow u'$
    $\epsilon \leftarrow \epsilon \cdot \theta(u, v)$
  end if
end while
if $u = 1$ **then**
  return ϵ
else
  return 0
end if

Algorithm 7 solovay_strassen(n, s)

Input: $n, s \in \mathbb{N}$

Output: *true* for n prime, $false$ for n composite

```
1: for j = 1, ..., s do
2:    a = random(1,n-1)
3:    if n = 2 then
4:       return true
5:    else
6:       if n ≡ 0 (mod 2) then
7:          return false
8:       end if
9:    end if
10:   if jacobi(a, n) ≠ modular_exponentiation(a, n) then
11:      return false
12:   end if
13: end for
14: return true
```

above **modular_exponentiation** function) takes $O(len(n)^3)$ bit operations. Since we test the Solovay-Strassen criterion at most s times, we obtain that the running time of the algorithm is $O(s \cdot len(n)^3)$.

6.5 Number of non-witnesses

Theorem 6.6
If n is composite, at least 50% of all $a \in \mathbb{Z}_n^*$ fail to satisfy equation (8).

Proof. We have shown that, for n composite, the set

$$\{a \in \mathbb{Z}_n^* : \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}\}$$

is a proper subgroup of $\mathbb{Z}_n^*$ (theorem 6.5). By the Lagrange theorem, we know that the order of the subgroup divides the order of the group (but is not equal to it because it is proper). So the order of this subgroup is at most the order of the group divided by two. So we can conclude that at least 50% of all $a \in \mathbb{Z}_n^*$ fail to satisfy equation (8).
More, since we know that $|\mathbb{Z}_n^*| = \phi(n)$, we can say that for n composite,

$$|\{a \in \mathbb{Z}_n^* : \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}\}| \leq \frac{\phi(n)}{2},$$

and so

$$|\{a \in \mathbb{Z}_n^* : \left(\frac{a}{n}\right) \not\equiv a^{\frac{n-1}{2}} \pmod{n}\}| \geq \frac{\phi(n)}{2}.$$

□

So if n is composite, then we will have to try equation (8) for a certain number of a before finding a witness of the fact that n is composite. It can be observed that this quantity of non-witnesses can depend on the form of n (by exemple: if n is the product of two primes, if n is the product of two Germain primes, ...).

7 The AKS algorithm

To understand this algorithm (AKS stands for Agrawal, Kayal and Saxena, who founds this algorithm), I based my researches on [4].

Since the last few years, there was no known deterministic polynomial-time algorithm of finding if an integer is prime or not. But the 6 August 2002, Agrawal, Kayal and Saxena, three indian mathematicians, found such an algorithm. This algorithm is called AKS (for the names of the people who found it) and is presented in the following section. The importance of this algorithm is more theoretical than practical, the probabilistic algorithms, such as the Miller-Rabin algorithm seen before, being much more efficient.

7.1 Number theory for the AKS algorithm

Definition 7.1.1
*Let R be a ring. An **R-algebra** is a ring E with a ring homomorphism $\tau : R \to E$.*

In our case, we are interested in this definition for $K[X]$ in the place of E and the homomorphism τ which sends $a \in R$ to the constant polynomial $a \in K[X]$.

Definition 7.1.2
*Let E and E' be some R-algebras with associated maps $\tau : R \to E$ and $\tau' : R \to E'$. We call $\rho : E \to E'$ a **R-algebra homomorphism** if and only if ρ is a ring homomorphism from E to E' and $\rho(\tau(a)) = \tau'(a)$ for all $a \in R$.*

The AKS algorithm bases his criterion on principally one theorem, which is the following.

Theorem 7.1
Let $n > 1$ be an integer.
If n is prime, then

$$(X + a)^n \equiv X^n + a \pmod{n} \tag{11}$$

for every $a \in \mathbb{Z}_n$, where X is an indefinite of $\mathbb{Z}_n$.
If n is composite, then

$$(X + a)^n \not\equiv X^n + a \pmod{n}$$

for every $a \in \mathbb{Z}_n^*$, where X is an indefinite of $\mathbb{Z}_n$.

But this criterion is not efficient because only evaluating the left-hand side of equation (11) already takes time $O(n)$ which is not polynomial in $len(n)$. The AKS algorithm uses another more restrictive criterion, which is to test this equation modulo $X^r - 1$ instead of modulo n for a suitable value of r.

Theorem 7.2
If $(X+a)^n \equiv X^n + a \pmod{X^r - 1}$ *holds in* $\mathbb{Z}_n[X]$ *for a certain* r *and a certain number of* a*, then* n *is prime.*

The r and the number of a necessary to have this result is explained in the algorithm below.

7.2 The algorithm

Let $n \in \mathbb{N}$ be the input of our algorithm. We want to know if n is prime or composite. The AKS algorithm is as follow:

Algorithm 8 AKS(n)

Input: $n \in \mathbb{N}$
Output: *true* for n prime, $false$ for n composite

1: **if** $\exists a, b > 1$ such that $n = a^b$ **then**
2: return $false$
3: **end if**
4: find the smallest integer $r > 1$ such that
$\gcd(n, r) > 1$
or
$\gcd(n, r) = 1$ and $n \pmod{r}$ lies in $\mathbb{Z}_r^*$ and has multiplicative order $> 4len(n)^2$
5: **if** r = n **then**
6: return *true*
7: **end if**
8: **if** $\gcd(n, r) > 1$ **then**
9: return $false$
10: **end if**
11: **for** $j = 1, \ldots, 2len(n)\lfloor r^{\frac{1}{2}} \rfloor + 1$ **do**
12: **if** $(X+j)^n \not\equiv X^n + j \pmod{X^r - 1}$ in $\mathbb{Z}_n[X]$ **then**
13: return $false$
14: **end if**
15: **end for**
16: return *true*

We have to show that the algorithm returns *true* when n is prime and $false$ when n is composite.
We call lines 1-2 step 1, line 4 step 2, lines 5-6 step 3, lines 8-9 step 4 and lines 11-12-13 step 5.

7.3 Running time

The AKS algorithm is not really efficient, but it is really important theoretically because it runs in polynomial time (this means that is polynomial in

$len(n)$).
We show that analyzing each step of the algorithm.

- For step 1, there already exists an algorithm of perfect power testing which runs in $O(len(n)^3 len(len(n)))$, which is polynomial in $len(n)$.
- For step 2, the search of r can be done by brute-force. The computation of $\gcd(n, r)$ can be done using Euclid's algorithm which runs in $O(len(n)^2)$. If r does not divide n and the multiplicative order of $n \pmod r$ in $\mathbb{Z}_r^*$ is greater than $m = 4len(n)^2$, then it can be shown that the least r verifying these hypothesis is $O(m^2 len(n))$. The determination of the multiplicative order of $n \pmod r$ in $\mathbb{Z}_r^*$ can be done by brute-force, using modular exponentiation to compute successive powers of n modulo r, which runs in polynomial time. So r found in step 2 is at most $O(len(n)^5)$.
- For step 3 and 4, which are only tests, we easily see that they are in polynomial time.
- For step 5, we see that we have to perform $O(r^{\frac{1}{2}} len(n))$ exponentiations (one for each iteration). To perform the exponentiation, we use the **modular_exponentiation** function which costs $O(len(n))$ for the $len(n)$ successive squaring steps, $O(r^2)$ for the multiplication of two polynomials of degree at most $r-1$ in each squaring step and $O(len(n)^2)$ for the cost of one operation in $\mathbb{Z}_n$. So the step 5 runs in $O(r^{\frac{5}{2}} len(n)^4)$.

Put together, since the r found in step 2 is in $O(len(n)^5)$, this implies that the AKS algorithm runs in $O(len(n)^{\frac{33}{2}})$.

7.4 Correctness

The proof of the correctness being rather long, this is only the idea of the main steps of the proof of the correctness of the AKS algorithm.

First, we show that if n is prime, then the algorithm outputs *true*.
We clearly see that the test in step 1 will fail. If the algorithm does not return *true* in step 3, then the test in step 4 will fail too (because of the primality of n). Because of theorem 7.1, the test in step 5 will fail for every j. So the output will be *true*.

Now we show that if n is composite, then the algorithm outputs $false$.
If n is a prime power, it will be detected in the first step. So we can assume that n is not a perfect power. Assume that a suitable value has been found in step 2. So the test in step 3 will certainly fail. If the test in step 4 passes, we are done, so we assume that it fails. We can now assume that all prime

factors of n are greater than r (if not we would have taken another value of r in step 2). We have to show that one of the tests in step 5 will pass. We will do this by contradiction. We suppose that every test fails and derive a contradiction.
For the rest of the proof, we fix a prime divisor p of n. Since p divides n, we have a natural ring homomorphism from $\mathbb{Z}_n[X]$ to $\mathbb{Z}_p[X]$. This implies that if the congruence in step 5 holds in $\mathbb{Z}_n[X]$, then it will hold in $\mathbb{Z}_p[X]$ too. So for the rest of the proof we will work in $\mathbb{Z}_p[X]$.
Here is a summary of the assumptions we make:

1. $n > 1$, $r > 1$, and $l \geq 1$ are integers, p is a prime dividing n, and $\gcd(n, r) = 1$;
2. n is not a prime power;
3. $p > r$;
4. the congruence

$$(X + j)^n \equiv X^n + j \pmod{X^r - 1}$$

   holds for $j = 1, \ldots, l$ in $\mathbb{Z}_p[X]$;
5. the multiplicative order of $n \pmod{r}$ in $\mathbb{Z}_r^*$ is greater than $4len(n)^2$;
6. $l > 2len(n)\lfloor r^{\frac{1}{2}} \rfloor$.

From now on, only assumption 1 will always be in force. The other assumptions will be explicitly named when they are necessary.
The goal now is to show that assumptions 1, 2, 3, 4, 5 and 6 cannot all be true simultaneously.
First, we define $E = \mathbb{Z}_p[X]/(X^r - 1)$ and $\xi = X \pmod{X^r - 1} \in E$. So we have that

$$E = \mathbb{Z}_p[\xi].$$

And this implies that every element of E can be uniquely expressed as $g(\xi) = g \pmod{X^r - 1}$ for some $g \in \mathbb{Z}_p[X]$ of degree less than r. These definitions mean that we have $g(\xi) = 0$ for an arbitrary $g \in \mathbb{Z}_p[X]$ if and only if $X^r - 1$ divides g. We can easily see that ξ has multiplicative order r.
We define now the following function for all integers k:

$$\sigma_k : E \to E,\ g(\xi) \mapsto g(\xi^k),$$

with g an arbitrary element of $\mathbb{Z}_p[X]$. Now, for $k \in \mathbb{Z}_r^*$, let's define the following function: $\hat{\sigma}_k : \mathbb{Z}_p[X] \to E$, $g \mapsto g(\xi^k)$, which is the polynomial evaluation map. Note that, by assumption 1, n and p lie in $\mathbb{Z}_r^*$. By showing that the kernel of $\hat{\sigma}_k$ is $(X^r - 1)$ and the image of $\hat{\sigma}_k$ is E, we show that σ_k is a ring automorphism. So for all $k, k' \in \mathbb{Z}_r^*$ we obtain that:

- $\sigma_k = \sigma_{k'}$ if and only if $\xi^k = \xi^{k'}$ if and only if $k = k' \pmod{r}$;
- $\sigma_k \circ \sigma_{k'} = \sigma_{k'} \circ \sigma_k = \sigma_{kk'}$.

Now, since E is of characteristic p and using Fermat's little theorem (theorem 4.6), we have that the p-power map is a $\mathbb{Z}_p$-algebra homomorphism. Let $\alpha \in E$ and its expression $\alpha = g(\xi)$ for some $g \in \mathbb{Z}_p[X]$. We obtain that

$$\alpha^p = g(\xi)^p = g(\xi^p) = \sigma_p(\alpha).$$

So we see that σ_p acts like the p-power map.
We can rewrite assumption 4 as

$$\sigma_n(\xi + j) = (\xi + j)^n \text{ for } j = 1, \ldots, l.$$

This would mean that for every n the map σ_n would act like the n-power map on each element of the form $\xi + j$, which seems really strange and gives the intuition that we will derive a contradiction somewhere looking at elements of this form.
We will now examine the elements and the values of n such that σ_n behaves like the n-power map on these elements. To do so we define two sets:

$$C(\alpha) = \{k \in \mathbb{Z}_r^* : \sigma_k(\alpha) = \alpha^k\}$$

for $\alpha \in E$ and

$$D(k) = \{\alpha \in E : \sigma_k(\alpha) = \alpha^k\}$$

for $k \in \mathbb{Z}_r^*$. So we see that $C(\alpha)$ is the set of values of k such that σ_k acts like the k-power map on α and $D(k)$ is the set of values of α such that σ_k acts like the k-power map on α. We remark that 1 and p are in $C(\alpha)$ for all $\alpha \in E$, α is in $D(p)$ for all $\alpha \in E$ and 1 is in $D(k)$ for all $k \in \mathbb{Z}_r^*$. Using the properties of σ_k, we can easily show that the sets $C(\alpha)$ and $D(k)$ are multiplicative.
Now, let us define

- s as the multiplicative order of $p \pmod{r}$ in $\mathbb{Z}_r^*$, and
- t as the order of the subgroup of $\mathbb{Z}_r^*$ generated by $p \pmod{r}$ and $n \pmod{r}$.

In the following we will work a lot with elements of this subgroup.
Let F be the extension field of degree s over $\mathbb{Z}_p$, which is the field with p^s elements. This implies that F^* is cyclic and has order $p^s - 1$. By definition of s, we have that r divides $p^s - 1$. And this implies that there exists an element $\zeta \in F^*$ of multiplicative order r.
Let us now define the $\mathbb{Z}_p$-algebra homomorphism

$$\tau : E \to F,\ g(\xi) \mapsto g(\zeta)$$

for $g \in \mathbb{Z}_p[X]$ (this is well-defined because $X^r - 1$ is the kernel of the evaluation map that sends $g \in \mathbb{Z}_p[X]$ to $g(\zeta) \in F$).
We define now the set

$$S = \tau(D(n)),$$

which is the set of the images under τ of all elements in E over which σ_n acts like the n-power map. We will observe this set and derive a contradiction from some observations on its cardinality.
First, we have the following lemma.

Lemma 7.3
Under assumption 2, we have

$$|S| \leq n^{2\lfloor t^{\frac{1}{2}} \rfloor}.$$

Proof. To do that we consider the set

$$I = \{n^u p^v : u, v = 0, \ldots, \lfloor t^{\frac{1}{2}} \rfloor\}.$$

Noting that n is not a prime power (by assumption 2), we can easily see that for each distinct pair (u, v) we obtain a distinct value of $n^u p^v$. Since u and v each take $\lfloor t^{\frac{1}{2}} \rfloor + 1$ values, there are strictly more than t distinct values of $n^u p^v$. So $|I| > t$. But recall that $t = |\langle p \pmod r, n \pmod r \rangle|$. So this means that there exist some k and k' in I which are distinct but equal modulo r. Since k and k' lie in I, we have that they both are less than $n^{2\lfloor t^{\frac{1}{2}} \rfloor}$.
Now, let $\alpha \in D(n)$. This implies that $n \in C(\alpha)$. Since $1 \in C(\alpha)$ and $p \in C(\alpha)$, we have that for every u and v non negative integers $n^u p^v \in C(\alpha)$. So k, $k' \in C(\alpha)$. This means that $\sigma_k(\alpha) = \alpha^k$ and $\sigma_{k'}(\alpha) = \alpha^{k'}$. But since $k \equiv k' \pmod r$, we have $\sigma_k = \sigma_{k'}$, which implies that $\alpha^k = \alpha^{k'}$. We apply τ to this and use the fact that it is an homomorphism to obtain $\tau(\alpha)^k = \tau(\alpha)^{k'}$. So $\tau(\alpha)$ is a root of the polynomial $X^k - X^{k'}$ (which is a non-zero polynomial since $k \neq k'$). Since this holds for every $\alpha \in D(n)$, every element of S is a root of $X^k - X^{k'}$. This polynomial being of degree at most $max\{k, k'\} \leq n^{2\lfloor t^{\frac{1}{2}} \rfloor}$, we have that $|S| \leq |\{\text{roots of } X^k - X^{k'}\}| \leq n^{2\lfloor t^{\frac{1}{2}} \rfloor}$. □

Now we limit the other side of $|S|$.

Lemma 7.4
Under assumptions 3 and 4, we have

$$|S| \geq 2^{min(t,l)} - 1.$$

Proof. To simplify the notations, we write $m = min(t, l)$ (remember that $t = |\langle p \pmod r, n \pmod r \rangle|$ and l is the number of values of j we test in step 5 of the algorithm). We see that assumption 4 implies that $\xi + j \in D(n)$

for $j = 1, \ldots, m$. But assumption 3 means that $p > r$ and by definition of t, we have $r > t$ and by definition of m, we have $t \geq m$. So we have $p > r > t \geq m$. This implies that the integers $j = 1, \ldots, m$ are distinct modulo p. So we can see that $D(n)$ is very large if we take t and l large and we will see that nothing "collapses" under τ.
To obtain the $2^m - 1$, we have to work on a set which is the following:

$$P = \left\{ \prod_{j=1}^{m} (X+j)^{e_j} \in \mathbb{Z}_p[X] : e_j \in \{0,1\} \text{ for } j = 1, \ldots, m \text{ and } \sum_{j=1}^{m} e_j < m \right\}.$$

Since all j are distinct modulo p, there exists a bijection between the set P and the different choices of the e_j for $j = 1, \ldots, m$ (with the condition that $\sum_{j=1}^{m} e_j < m$). So we have that $|P| = 2^m - 1$. We define now two sets: $P(\xi) = \{f(\xi) \in E : f \in P\}$ and $P(\zeta) = \{f(\zeta) \in F : f \in P\}$, which is clearly the image of $P(\xi)$ under τ. Since $\xi + j \in D(n)$ for $j = 1, \ldots, m$ and $D(n)$ is multiplicative, we have that $P(\xi) \subseteq D(n)$. So we have that $P(\zeta) = \tau(P(\xi)) \subseteq \tau(D(n)) = S$, hence $|P(\zeta)| \leq |S|$. So it suffices to show that $|P(\zeta)| = 2^m - 1$ (since $|P| = 2^m - 1$, it will not be more). We do this by contradiction.
We suppose that $|P(\zeta)| < 2^m - 1$. This implies that there exist two distinct polynomials $g, h \in P$ such that $g(\zeta) = h(\zeta)$. Since $g, h \in P$, we have that g and h are both of degree at most $t - 1$ (more precisely at most $m - 1$), that $g(\xi), h(\xi) \in D(n)$ and that $\tau(g(\xi)) = \tau(h(\xi))$. So we have that $1, p, n \in C(g(\xi))$ and $1, p, n \in C(h(\xi))$. Hence for all k of the form $n^u p^v$ for u and v non-negative integers, $k \in C(g(\xi))$ and $k \in C(h(\xi))$. So we obtain that, since $\tau(g(\xi)) = \tau(h(\xi))$, $\tau(g(\xi))^k = \tau(h(\xi))^k$. Hence, using that τ is an homomorphism, we have

$$\begin{aligned} 0 &= \tau(g(\xi))^k - \tau(h(\xi))^k \\ &= \tau(g(\xi)^k) - \tau(h(\xi)^k) \\ &= \tau(g(\xi^k)) - \tau(h(\xi^k)) \\ &= g(\zeta^k) - h(\zeta^k). \end{aligned}$$

So we have obtained a non-zero polynomial $f = g - h \in \mathbb{Z}_p[X]$ which has $\zeta^k \in F$ as root for all k of the form $n^u p^v$. Recall that ζ has multiplicative order r. So $\zeta^k = \zeta^{k'}$ if and only if $k \equiv k' \pmod{r}$. But we have shown that there are exactly t distinct values for an integer k of the form $n^u p^v \pmod{r}$. So there are exactly t different values of ζ^k in F, which are all roots of f. But f is of degree at most $t - 1$, so it can only have $t - 1$ roots, which is a contradiction. So $|P(\zeta)|$ has to be equal to $2^m - 1$. □

So we have shown that under assumptions 2, 3 and 4, we have

$$2^{min(t,l)} - 1 \leq |S| \leq n^{2\lfloor t^{\frac{1}{2}} \rfloor}.$$

But we will derive our contradiction by showing the following lemma.

Lemma 7.5
Under assumptions 5 and 6, we have

$$2^{min(t,l)} - 1 > n^{2\lfloor t^{\frac{1}{2}} \rfloor}.$$

Proof. The first thing we see is that it depends on the values of t and l. So we will make considerations over these values to show this inequality.
Since $log_2(n) \leq len(n)$, we only have to show that $2^{min(t,l)} - 1 > 2^{2len(n)\lfloor t^{\frac{1}{2}} \rfloor}$, which we can show by proving that $min(t, l) > 2len(n)\lfloor t^{\frac{1}{2}} \rfloor$ (since for all $a > b \geq 1$, we have $2^a > 2^b + 1$). By assumption 6, we have $l > 2len(n)\lfloor t^{\frac{1}{2}} \rfloor$. Since t is the order of the subgroup generated by $n \pmod r$ and $p \pmod r$, it is at least as large as the multiplicative order of $n \pmod r$ in $\mathbb{Z}_r^*$, which is, by assumption 5, larger than $4len(n)^2$. So $t > 4len(n)^2$, which we can write as $t > 2len(n)t^{\frac{1}{2}} \geq 2len(n)\lfloor t^{\frac{1}{2}} \rfloor$. Hence $2^{min(t,l)} - 1 > n^{2\lfloor t^{\frac{1}{2}} \rfloor}$. □

So it is only in this final step that we understand why the bounds in step 2 and in step 5 of the algorithm were set like this.
We have finally shown that the six assumptions cannot be true all together. So if the input is composite, the algorithm will output $false$.

Put together, this means that the algorithm says that a number n given is prime if and only if it is really prime.

8 Comparison of the algorithms

We can now try these algorithms with all numbers $n \in \mathbb{N}$ (in practice for a large number of $n \in \mathbb{N}$) and check how many errors they make if we compare them with the naive algorithm.
Clearly, the Fermat algorithm makes more errors than the Miller-Rabin algorithm. We have the following theorem (stated without proof) too.

Theorem 8.1
If a lies in

$$\left\{ \begin{array}{ll} a \in \mathbb{Z}_n^* : & a^{n-1} \equiv 1 \pmod{n} \text{ and} \\ & a^{2k} \equiv 1 \pmod{n} \Rightarrow a^k \equiv \pm 1 \pmod{n} \ \forall k \in \{1, \ldots, \frac{n-1}{2}\} \end{array} \right\},$$

then a lies in

$$\{a \in \mathbb{Z}_n^* : \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}\}.$$

We do not prove this result here but we can see that it is correct by running the algorithms, so that the Miller-Rabin algorithm makes less errors than the Solovay-Strassen algorithm.
So if we find a non-witness for the Miller-Rabin criterion, it will be a non-witness for the Solovay-Strassen criterion. By transposition, this implies that if if we find a witness for the Solovay-Strassen criterion, it will be a witness for the Miller-Rabin criterion.

So we have that the Miller-Rabin algorithm makes less errors than the Solovay-Strassen algorithm, which makes less errors than the Fermat algorithm (seen experimentally by running the algorithms). In short:

$$Miller - Rabin \leq Solovay - Strassen \leq Fermat.$$

We know that these algorithms can only make one type of error: if we give them a composite number, they sometimes say that it is prime. We can check experimentally how many errors of this type they make for different values of s by giving them composite numbers and checking how many times they output *prime*. This is a tabular of some values tested experimentally (since they are random algorithms, with the same input we can have different outputs for different times of running the algorithms).

Number of tested n: 2^{20} $1 \leq n \leq 2^{20}$			
s	Fermat	Miller-Rabin	Solovay-Strassen
2	465	347	371
3	399	342	350
5	377	339	341
10	349	339	339
20	340	339	339
50	339	339	339

This seems to be a lot of errors, but since $2^{20} = 1048576$, for $s = 50$, the algorithms make about 0.03% of errors. We see that increasing s increases the performance of the algorithms, but not as much as we would have thought.

Number of tested n: 2^{20} $1 \leq n \leq 2^{100}$			
s	Fermat	Miller-Rabin	Solovay-Strassen
2	0	0	0
3	0	0	0
5	0	0	0
10	0	0	0
20	0	0	0
50	0	0	0

We see that when we increase the size of the input, the algorithms make a lot less errors than in the previous example. This is probably because since the numbers are large they have more possible witnesses and so we have more chance to find one.

Let $n \in \mathbb{N}$ composite number be the input of our algorithm. We can look at how many random values we have to test before finding a witness of n.

Number of tested n: 2^{10} $1 \leq n \leq 2^{20}$				
$s =$	1	2	3	4
Fermat	1022	2	0	0
Miller-Rabin	1024	0	0	0
Solovay-Strassen	1023	1	0	0

We see that we find very quickly a witness of n. This shows that the algorithms are efficient.
But if we compare this result with the ones before, it seems strange to have so many errors in the first example. Since this last example is more in agreement with the theory, there is probably an error in the implementation

of the algorithm for the first example.
To have good examples we have to deal with big numbers and iterate our functions many times. But it is difficult to make that on my own computer. This explain why these examples are so little.

Appendices

A Implementations in C++

Here are the main programs I implemented in C++.
Sometimes there are two versions of the same function. It is because when I included the ntl library (library to work with very large numbers), some of the functions were already implemented in this library. So I replaced my "old" functions (which are explained in the description of the algorithms) by them (which are more efficient). So the commented functions are the functions I implemented myself and the second version of these functions is the one already available in ntl.
The type "NumberLength" here stands for the type "ZZ" (very large integers) of the ntl library.

A.1 Fermat algorithm implementation

```
1  #include <cmath>
2  #include <vector>
3  // for rand:
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  #include "Fermat.h"
9
10 using namespace std;
11
12 /*vector<int> convert_mr(NumberLength m){
13   vector<int> m_binaire;
14   NumberLength r(m %2);
15   while(m!=0){
16     m_binaire.push_back(r);
17     m = (m-r)/2;
18     r = m %2;
19   }
20   return m_binaire;
21 }*/
22
23 /*NumberLength modular_exponentiation_f(NumberLength a,
       NumberLength b, NumberLength n){
24   NumberLength d(1);
25   vector<int> b_binaire;
26   b_binaire = convert_mr(b);
27   NumberLength k(b_binaire.size());
28   for(int i(k-1); i>=0; --i){
29     d = (d*d) %n;
```

```
30      if(b_binaire[i]==1){
31        d = (d*a) %n;
32      }
33    }
34    return d;
35  }*/
36
37  NumberLength modular_exponentiation_f(NumberLength a,
        NumberLength b, NumberLength n){
38    return PowerMod(a, b, n);
39  }
40
41  bool witness_f(NumberLength a, NumberLength n){
42    NumberLength b(n-1);
43    NumberLength x(modular_exponentiation_f(a, b, n));
44    if(x %n !=1){
45      return true;
46    }
47    return false;
48  }
49
50  bool fermat(NumberLength n, NumberLength s){
51    NumberLength a;
52    a = 0;
53    for(int j(1); j<=s; ++j){
54      //a = (rand() %(n-1))+1;
55      a = (RandomBnd(n-1))+1;
56      if(witness_f(a, n)){
57        return false;
58      }
59    }
60    return true;
61  }
```

A.2 Miller-Rabin algorithm implementation

```
1  #include <cmath>
2  #include <vector>
3  // for rand:
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  #include "Miller-Rabin.h"
9
10 using namespace std;
11
12 /*vector<int> convert_mr(NumberLength m){
```

```
13    vector<int> m_binaire;
14    NumberLength r(m %2);
15    while(m!=0){
16      m_binaire.push_back(r);
17      m = (m-r)/2;
18      r = m %2;
19    }
20    return m_binaire;
21  }*/
22
23  /*NumberLength modular_exponentiation_mr(NumberLength a,
        NumberLength b, NumberLength n){
24    NumberLength d(1);
25    vector<int> b_binaire;
26    b_binaire = convert_mr(b);
27    NumberLength k(b_binaire.size());
28    for(int i(k-1); i>=0; --i){
29      d = (d*d) %n;
30      if(b_binaire[i]==1){
31        d = (d*a) %n;
32      }
33    }
34    return d;
35  }*/
36
37  NumberLength modular_exponentiation_mr(NumberLength a,
        NumberLength b, NumberLength n){
38    return PowerMod(a, b, n);
39  }
40
41  bool witness(NumberLength a, NumberLength n){
42    NumberLength u;
43    u = n-1;
44    unsigned long int t;
45    t = 0;
46    while(u %2==0){
47      u = u/2;
48      t = t+1;
49    }
50    vector<NumberLength> x;
51    x.push_back(modular_exponentiation_mr(a, u, n));
52    for(int i(1); i<=t; ++i){
53      x.push_back((x[i-1]*x[i-1]) %n);
54      if(x.at(i) %n==1 && x.at(i-1) %n !=1 &&
            x.at(i-1)!=n-1){
55        return true;
56      }
57    }
58    if(x.at(t) %n !=1){
```

```
59      return true;
60    }
61    return false;
62  }
63
64  bool miller_rabin(NumberLength n, NumberLength s){
65    NumberLength a;
66    a=0;
67    for(int j(1); j<=s; ++j){
68      //a = (rand() %(n-1))+1;
69      a = (RandomBnd(n-1))+1;
70      if(witness(a, n)){
71        return false;
72      }
73    }
74    return true;
75  }
```

A.3 Solovay-Strassen algorithm implementation

```
1   #include <iostream>
2   #include <cmath>
3   #include <vector>
4   // for rand:
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <time.h>
8
9   #include "Solovay-Strassen.h"
10
11  using namespace std;
12
13  /*vector<int> convert(NumberLength m){
14    vector<int> m_binaire;
15    NumberLength r(m %2);
16    while(m!=0){
17      m_binaire.push_back(r);
18      m = (m-r)/2;
19      r = m %2;
20    }
21    return m_binaire;
22  }*/
23
24  /*NumberLength modular_exponentiation(NumberLength a,
        NumberLength n){
25    NumberLength d(1);
26    NumberLength b((n-1)/2);
27    vector<int> b_binaire;
```

```
28    b_binaire = convert(b);
29    NumberLength k(b_binaire.size());
30    for(int i(k-1); i>=0; --i){
31      d = (d*d) %n;
32      if(b_binaire[i]==1){
33        d = (d*a) %n;
34      }
35    }
36    return d;
37  }*/
38
39  NumberLength modular_exponentiation(NumberLength a,
        NumberLength n){
40    return PowerMod(a, (n-1)/2, n);
41  }
42
43  NumberLength jacobi(NumberLength a, NumberLength n){
44    NumberLength n_stock;
45    n_stock = n;
46    NumberLength epsilon;
47    epsilon = 0;
48    if(n %2 ==1 && n>=3){
49      if(a>=0){
50        epsilon = 1;
51      }else{
52        a = -a;
53        epsilon = epsilon * ( ((n-1)/2) %2 == 1 ? -1 : 1 );
54      }
55      while(a!=1 && a!=0){
56        if(a %2 ==0){
57          epsilon = epsilon * ( (((n*n)-1)/8) %2 == 1 ? -1
              : 1 );
58          a = a/2;
59        }else{
60          NumberLength a_stock(a);
61          epsilon = epsilon * ( ((a-1)*(n-1)/4) %2 == 1 ?
              -1 : 1 );
62          a = n %a;
63          n = a_stock;
64        }
65      }
66      if(a==1){
67        if(epsilon < 0){
68          epsilon = n_stock+epsilon;
69        }
70        return epsilon;
71      }else{
72        NumberLength r;
73        r = 0;
```

```
74        return r;
75      }
76    }
77  }
78
79  bool solovay_strassen(NumberLength n, NumberLength s){
80    NumberLength a;
81    a = 0;
82    for(int j(1); j<=s; ++j){
83      while(GCD(a,n)!=1){
84        //a = (rand() %(n-1))+1;
85        a = (RandomBnd(n-1))+1;
86      }
87      if(n == 2){
88        return true;
89      }else if(n != 2 && n %2 == 0){
90        return false;
91      }
92      if(jacobi(a, n)!=modular_exponentiation(a, n)){
93        return false;
94      }
95    }
96    return true;
97  }
```

A.4 Implementation of the comparison of the algorithms

This code is to test how many times, for an input $n \in \mathbb{N}$ composite, the algorithms output *prime*.

```
1  #include <iostream>
2  #include <cmath>
3  #include <vector>
4  // for rand:
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <time.h>
8
9  #include "Miller-Rabin.h"
10 #include "Solovay-Strassen.h"
11 #include "Fermat.h"
12
13 using namespace std;
14
15 int main(){
16
17   SetSeed(to_ZZ(time(NULL)));
18
19   vector<int> valeur_s;
```

```
20  valeur_s.push_back(2);
21  valeur_s.push_back(3);
22  valeur_s.push_back(5);
23  valeur_s.push_back(10);
24  valeur_s.push_back(20);
25  valeur_s.push_back(50);
26
27  NumberLength k;
28  NumberLength max;
29  max.SetSize(50);
30  max = power_ZZ(2,50);
31  NumberLength borne;
32  borne.SetSize(50);
33  borne = power_ZZ(2,20);
34
35  int compteur_f[valeur_s.size()];
36  int compteur_m_r[valeur_s.size()];
37  int compteur_s_s[valeur_s.size()];
38  for(int i(0); i < valeur_s.size(); ++i){
39    compteur_f[i] = 0;
40    compteur_m_r[i] = 0;
41    compteur_s_s[i] = 0;
42  }
43
44  for(k=2; k <= borne; ++k){
45    NumberLength m1;
46    m1 = (RandomBnd(max-1))+1;
47    NumberLength m2;
48    m2 = (RandomBnd(max-1))+1;
49    NumberLength n;
50    n = m1*m2;
51    for(int i(0); i < valeur_s.size(); ++i){
52      NumberLength s;
53      s = valeur_s[i];
54      bool f(fermat(n,s));
55      bool m_r(miller_rabin(n,s));
56      bool s_s(solovay_strassen(n,s));
57      if(f){
58        ++compteur_f[i];
59      }
60      if(m_r){
61        ++compteur_m_r[i];
62      }
63      if(s_s){    }
64        ++compteur_s_s[i];
65      }
66  }
67
68  cout << "Checked␣algorithms␣for␣" << borne << "␣random␣
```

```
      composite␣numbers␣<␣" << max << "." << endl;
  for(int i(0); i < valeur_s.size(); ++i){
    cout << "For␣s␣=␣" << valeur_s[i] << ",␣the␣Fermat␣
       algorithm␣makes␣" << compteur_f[i] << "␣errors."
       << endl;
    cout << "For␣s␣=␣" << valeur_s[i] << ",␣the␣
       Miller-Rabin␣algorithm␣makes␣" <<
       compteur_m_r[i] << "␣errors." << endl;
    cout << "For␣s␣=␣" << valeur_s[i] << ",␣the␣
       Solovay-Strassen␣algorithm␣makes␣" <<
       compteur_s_s[i] << "␣errors." << endl;
  }

  return 0;
}
```

This code is to look at how many random values are needed to declare that an input $n \in \mathbb{N}$ composite is *composite* (so how many random values we have to test before finding a witness of n).

```
#include <iostream>
#include <cmath>
#include <vector>
// for rand:
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "Miller-Rabin.h"
#include "Solovay-Strassen.h"
#include "Fermat.h"

using namespace std;

int main(){

  SetSeed(to_ZZ(time(NULL)));

  NumberLength k;
  NumberLength max;
  max.SetSize(50);
  max = power_ZZ(2, 10);
  NumberLength borne;
  borne.SetSize(50);
  borne = power_ZZ(2, 20);

  NumberLength s;
  s = 1;

  int nb_iterations_s(20);
```

```
31
32  int stock_s_f[nb_iterations_s];
33  int stock_s_mr[nb_iterations_s];
34  int stock_s_ss[nb_iterations_s];
35  for(int i(0); i < nb_iterations_s; ++i){
36    stock_s_f[i] = 0;
37    stock_s_mr[i] = 0;
38    stock_s_ss[i] = 0;
39  }
40
41  for(k=0; k < borne; ++k){
42    int s_f(1);
43    int s_mr(1);
44    int s_ss(1);
45
46    NumberLength m1;
47    m1 = (RandomBnd(max-1))+1;
48    NumberLength m2;
49    m2 = (RandomBnd(max-1))+1;
50    NumberLength n;
51    n = m1*m2;
52
53    bool fct_f(fermat(n,s));
54    bool fct_mr(miller_rabin(n,s));
55    bool fct_ss(solovay_strassen(n,s));
56
57    while(fct_f==1){
58      s_f = s_f + 1;
59      fct_f = fermat(n,s);
60    }
61    while(fct_mr==1){
62      s_mr = s_mr + 1;
63      fct_mr = miller_rabin(n,s);
64    }
65    while(fct_ss==1){
66      s_ss = s_ss + 1;
67      fct_ss = solovay_strassen(n,s);
68    }
69
70    stock_s_f[s_f-1] = stock_s_f[s_f-1]+1;
71    stock_s_mr[s_mr-1] = stock_s_mr[s_mr-1]+1;
72    stock_s_ss[s_ss-1] = stock_s_ss[s_ss-1]+1;
73  }
74
75  cout << "Fermat:␣" << endl;
76  for(int i(0); i<nb_iterations_s; ++i){
77    cout << stock_s_f[i] << "␣|␣";
78  }
79  cout << endl;
```

```
80
81    cout << "Miller-Rabin: " << endl;
82    for(int i(0); i<nb_iterations_s; ++i){
83      cout << stock_s_mr[i] << " | ";
84    }
85    cout << endl;
86
87    cout << "Solovay-Strassen: " << endl;
88    for(int i(0); i<nb_iterations_s; ++i){
89      cout << stock_s_ss[i] << " | ";
90    }
91    cout << endl;
92
93    return 0;
94  }
```

References

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique, 2ème édition.* Dunod, 2002.

[2] Michel Demazure. *Cours d'algèbre. Primalité. Divisibilité. Codes.* Cassini, 1997.

[3] Neal Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, 1988.

[4] Victor Shoup. *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press, 2005.