B.Sc.Eng. Thesis
Bachelor of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Primality Tests

Amalie Due Jensen (s160503)

Kongens Lyngby 2019

DTU

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Prime numbers form the fundamental building blocks for all integers. If an integer is not prime, then it is a product of primes. However, this is not enough knowledge when big integers should be classified as either primes or composite numbers.

In this project, different primality tests are investigated. When people refers to primality tests, they talk exactly about the problem of determining whether an integer is prime or composite.

Two probabilistic primality tests as well as a deterministic primality test will be investigated. The focus throughout the project is to prove the correctness of the tests, and afterwards, as a proof of concept, the primality tests are implemented in Python using SageMath. To underline the thread troughout the thesis, the implementations will be compared to the analytical complexity analysis of the primality tests.

Preface

Kongens Lyngby, September 24, 2019

Amalie Due Jensen (s160503)

Contents

CHAPTER 1

Introduction

Prime numbers form the fundamental building blocks for all integers. If a positive integer is not a prime, then it is a product of primes. Similarly, if a positive integer is a product of primes, then it can not be a prime itself. However, when we want to distinguish composite numbers from primes, this fact is not at all enough, because how do you check whether a big positive integer is a product of primes or not? When people refers to *primality tests*, they talk exactly about the problem of determining whether an integer is prime or not. The most simple primality test is trial division: If an integer n should be tested for primality by trial division, then one simply tests whether each prime $\leq \sqrt{n}$ divides n or not. If none of them do, then n must be prime (as explained, if an integer is not prime, then it has at least two prime factors. If both of them were greater than $\sqrt{n}$, then the product of them $p \cdot q \geq \sqrt{n}\sqrt{n} = n$, contradicting the fact that the product should be equal to n). This primality test is indeed very slow, and fortunately better methods are known.

In these days, many primality tests are known, but the different tests are far from being equally efficient. A central result in algebra is Fermat's little theorem which states that $a^{n-1} \equiv 1 \mod n$ for a prime n and any $a \in \mathbb{Z}$ for which $\gcd(a, n) = 1$. This theorem is used in some primality tests, and in this project two tests using Fermat's little theorem will be examined: The Fermat test and the Miller-Rabin test. These two tests turn out to be probabilistic test in the sense that if they judges n as being a prime, then the answer is always correct while if the tests reply "composite" then the answer is only correct with probability $\geq \frac{1}{2}$. In contrast to these probabilistic tests, there are also so-called *deterministic* tests which always return the correct answer. In this project, a deterministic primality test will be examined as well: the AKS primality test. The AKS primality test was found in 2002 by three Indian computer scientists, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, and the test turns out to be not only a deterministic primality test, but a *polynomial time deterministic primality test* which will be examined in this project.

The AKS primality test was an elusive goal of researchers in the algorithmic number theory world[5, p. 4]. An interesting detail about the test is that it was constructed less than 20 years ago. Since the first announcement of the test in 2002, many ideas and improvements of the test have been coming up, and they still are[5, p. 6].

The reader might wonder: Why do we even care about which numbers are primes and which are not? The answer to this question could be to explain one of todays really important usages of prime numbers - the RSA cryptosystem: In 1982, Ron Rivest, Adi Shamir, and Len Adleman constructed a public key cryptosystem which is an important part of computer security today[5, p. 14]. The idea can be explained in the following way: Bob takes two large primes p and q, $p < q$, computes their product $n = pq$ and then determines two integers d and e such that

$$de \equiv 1 \pmod{(p-1)(q-1)}.$$

The public key which is then used by Alice then consists of n and the encryption key e. Bob keeps

the decryption key d as well as the primes p and q secret. Suppose that the message which should be sent between Alice and Bob is an integer between 1 and $n-1$ (note that an alphabetic message m can be translated into numbers by writing "01" as "A", "02" as "B", and so on). In order to encrypt the message m, Alice computes

$$r := \Phi(m) \equiv m^e \pmod{n},$$

with $\Phi(m) \in \{1, \ldots, n-1\}$. In order to decrypt, Bob computes

$$\Psi(r) := r^d \pmod{n},$$

with $\Psi(r) \in \{1, \ldots, n-1\}$. By using Fermat's little theorem for the primes p and q, one can verify that

$$m^{de} \equiv m \pmod{n},$$

which implies that $\Psi = \Phi^{-1}$. If a third person, say Charlie, knows n and is able to factor n, then he can determine Ψ easily which means the security of the RSA cryptosystem depends on how hard it is to factor n.

While there exists efficient primality tests today, the known primes are getting larger and larger, thus creating the need for more efficient primality tests in order to continue finding ever increasing primes.

CHAPTER 2

Preliminaries

An essential preliminary is the definition of prime numbers, so this is where this section starts:

2.1 Prime numbers

An integer $N \in \mathbb{N}_{\geq 2}$ is a prime number if $N|ab$ implies that $N|a$ or $N|b$, or, equivalently, if $ab = N$ implies that $a \in \{1, -1\}$ or $b \in \{1, -1\}$, for all $a, b \in \mathbb{Z}$. The other integers $N \geq 2$ are called composite. Prime numbers are the foundation of integers as all integers can be represented using primes according to the following theorem:

Theorem 2.1 (Fundamental theorem of arithmetic)**.** *[4, p. 518] Every positive integer can be written as a product of positive prime factors. This is unique up to the order of the factors. In other words,* $\mathbb{Z}$ *is a Unique Factorization Domain.*

2.1.1 Infinitely many primes

Theorem 2.2. *There are infinitely many primes.*

Proof. Assume the opposite - that there are only a finite number of primes. These primes are now denoted $p_1, p_2, \ldots, p_n$. Let q be defined as $q = 1 + \prod_{i=1}^{n} p_i$. It is clear that q is not equal to any of the primes $p_1, \ldots, p_n$. According to the *Fundamental theorem of arithmetic* which is mentioned earlier, an integer is either a prime or a product of some primes. It is clearly seen that none of the primes $p_1, \ldots, p_n$ divide q, and therefore q must be a prime itself. This fact contradicts the assumption that $p_1, \ldots, p_n$ are all the primes. □

2.2 Groups

An abstract structure called a group is now defined. A group involves elements from a set G and a group operation $\cdot$. The full definition of a group is shown in Definition 2.2.1.

Definition 2.2.1. [1, p. 13] A pair $(G, \cdot)$ consisting of a set G and a group operation $\cdot : G \times G \to G$ is called a group if the following three properties (usually called groups axioms) are satisfied:

- for any elements $f, g, h \in G$ we have $f \cdot (g \cdot h) = (f \cdot g) \cdot h$ (one says that the group operation is associative),
- there exists an element $e \in G$, called the identity element of G, such that $e \cdot f = f$ and $f \cdot e = f$ for any $f \in G$,
- for any $f \in G$ there exists an element $g \in G$ such that $f \cdot g = e$ and $g \cdot f = e$ (the element g is called the inverse of f and will be denoted by f^{-1}).

An example of a group is the group $(\mathbb{Z}/N\mathbb{Z})^*$, which will be used a lot throughout the project. It is defined in the following way:

Theorem 2.3. *[4, p. 518] The group*

$$(\mathbb{Z}/N\mathbb{Z})^* = \{a \mod N \in \mathbb{Z}/N\mathbb{Z} : \gcd(a, N) = 1\}$$

is the multiplicative group of units in $\mathbb{Z}/N\mathbb{Z}$.

Proof. Another way of defining $(\mathbb{Z}/N\mathbb{Z})^*$ is the following:

$$(\mathbb{Z}/N\mathbb{Z})^* = \{u \in \mathbb{Z}/N\mathbb{Z} | \exists v \in \mathbb{Z}/N\mathbb{Z} : u \cdot v = 1 \mod N\} \tag{2.1}$$

This definition is more straightforward when proving that $(\mathbb{Z}/N\mathbb{Z})^*$ is a group. It is now shown that $(\mathbb{Z}/N\mathbb{Z})^*$ satisfies the three properties in Definition 2.2.1:

- Take two elements $u_1, u_2 \in (\mathbb{Z}/N\mathbb{Z})^*$. According to the definition in equation (2.1), take the two elements $v_1, v_2 \in (\mathbb{Z}/N\mathbb{Z})^*$ such that they are the multiplicative inverses of u_1 and u_2 respectively. Then the following can be written:

  $$u_1 \cdot u_2 \cdot v_2 \cdot v_1 = u_1 \cdot 1 \cdot v_1 = 1.$$

  Hence, $u_1 u_2 \in (\mathbb{Z}/N\mathbb{Z})^*$, and $v_1 v_2$ is the inverse of $u_1 u_2$. This shows that the first property is satisfied.

- There exists indeed the identity element, 1, in $(\mathbb{Z}/N\mathbb{Z})^*$.

- The v in equation (2.1) is the inverse of u, hence there exists the inverse of any element in $(\mathbb{Z}/N\mathbb{Z})^*$.

The three properties in definition 2.2.1 are satisfied, hence $(\mathbb{Z}/N\mathbb{Z})^*$ is a group. □

In continuation of the definition of the group $(\mathbb{Z}/N\mathbb{Z})^*$ follows the following definition:

Definition 2.2.2. [4, p. 75] Euler's totient function ϕ is defined as

$$\phi(N) = |(\mathbb{Z}/N\mathbb{Z})^*|.$$

2.3 Fermat's little theorem

Fermat's little Theorem plays a big role in the probabilistic Fermat test which is presented in Chapter 3. Therefore, the theorem is now presented, and it is proved why it is correct. The theorem follows directly from Euler's theorem, hence Euler's theorem is first presented and proved. The only thing necessary in order to prove the correctness of Euler's theorem is the following Proposition 2.6. However, in order to show the correctness of this Proposition, the two following results, including Lagrange's theorem, are necessary. Lagrange's theorem will be a central fact of other parts of the thesis as well:

Lagrange's theorem is a fundamental tool in algebra, and it is therefore stated without proof:

Theorem 2.4 (Lagrange's theorem)**.** *[6, p. 63-64] If $H \subseteq G$ is a subgroup of a finite group G, then*

$$|G| = |G/H||H|.$$

The order of a subgroup divides the order of the group.

Since the following Lemma 2.5 is not the main interest in this section, it is stated without proof as well:

Lemma 2.5. *[1, p. 18] Let $(G, \cdot)$ be a group, and let $g \in G$ be a group element. Then the order of the group $\langle g \rangle$ is the same as the order of the element g.*

The Proposition which plays the main role in the proof of Euler's theorem is now presented and proved:

Proposition 2.6. *Let $(G, \cdot)$ be a finite group, and let $g \in G$ be a group element. Then ord(g) divides $|G|$. In particular, $g^{|G|} = e$ for any group element $g \in G$.*

Proof. The first part of the Proposition follows from Lemma 2.5 and from Lagrange's Theorem, Theorem 2.4. To show that $g^{|G|} = e$ for any $g \in G$, note that the first part of the Proposition implies that $|G| = \text{ord}(g)n$ for some natural number n. But then

$$g^{|G|} = g^{\text{ord}(g)n} = (g^{\text{ord}(g)})^n = e^n = e,$$

for any $g \in G$. □

The main theorem which is due to Euler (1707-83) is now stated and proved:

Theorem 2.7 (Euler's Theorem)**.** *[1, p. 28] Let $d, n \in \mathbb{Z}$ be two integers and assume that* $\gcd(d, n) = 1$. *Then $d^{\phi(n)} \equiv 1 \mod n$, where ϕ denotes Euler's totient function defined in Definition 2.2.2.*

Proof. Euler's Theorem follows by applying Proposition 2.6 to the group $((\mathbb{Z}/n\mathbb{Z})^*, \cdot_n)$. □

Fermat's little theorem which, as the name says, is due to Fermat (1601-65), is now stated as a Corollary of Euler's theorem, Theorem 2.7:

Corollary 2.7.1 (Fermat's little theorem)**.** *If $p \in \mathbb{N}$ is prime and $a \in \mathbb{Z}$, then $a^p \equiv a \mod p$ and, if $p \nmid a$, then $a^{p-1} \equiv 1 \mod p$.*

Proof. The Corollary follows directly from Euler's theorem, Theorem 2.7, since $\phi(p) = p - 1$ for a prime p.[6, p. 26] □

2.4 Introduction to complexity analysis

When only an asymptotic upper bound is considered, the $\mathcal{O}$-notation, Definition 2.4.1, is used.

Definition 2.4.1. For a given function $g(n)$, the notation $\mathcal{O}(g(n))$ denotes the set of functions:

$$\mathcal{O}(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0, \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

Using $\mathcal{O}$-notation, the running time of an algorithm is merely described by inspecting the overall structure of the algorithm.[2, p. 47]

In some situations, the $\mathcal{O}$ still includes too much information. In such cases, the $\tilde{O}$-notation can be used: The $\tilde{O}$-notation swallows log-factors, which means for example that $\mathcal{O}(n \log n \log \log n)$, the complexity of multiplying two integers of length n fast, corresponds to $\tilde{O}(n)$.[4, p. 721]

The following Definition will also be used in the sections with complexity analysis:

Definition 2.4.2. [4, p. 244] Let R be a ring (commutative, with 1). A function

$$M : \mathbb{N}_{>0} \to \mathbb{R}_{>0}$$

is called a multiplication time for $R[x]$, if polynomials in $R[x]$ of degree less than n can be multiplied using at most $M(n)$ operations in R. Similarly, a function M as above is called a multiplication time for $\mathbb{Z}$, if two integers of length n can be multiplied using at most $M(n)$ word operations.

In the sections with complexity analysis, let $\lg n := \log_2 n$.

2.5 The square-and-multiply algorithm

As a preparation for the presentation of the Fermat prime number test in Chapter 3, it is interesting to look at the time complexity of computing $x^y \mod z$. The straightforward way of computing this is to multiply x by itself y times and do a modular reduction modulo z every time. Let k denote the number of bits in y. Then this computation has time complexity

$$\mathcal{O}(yk^2) = \mathcal{O}(2^k k^2),$$

which is exponential in k. It is possible to do better than that. Computing the sequence

$$x, x^2, x^3, x^4, x^5, x^6, x^7, x^8$$

in order to compute x^8 requires seven multiplications, whereas the sequence

$$x, x^2, x^4, x^8$$

requires only three squarings. This is the idea of the square-and-multiply algorithm for computing $x^y \mod z$ - to do a faster computation by using squarings instead of multiplications. The first step in the algorithm is to write the exponent y in binary representation

$$y = 2^{k-1}y_{k-1} + \cdots + 2^2 y_2 + 2y_1 + y_0 \qquad y_i \in \{0, 1\}.$$

Then, x^y can be expressed as

$$\begin{aligned} x^y &= x^{2^{k-1}y_{k-1}+\cdots+2^2y_2+2y_1+y_0} \\ &= (((((((x^{y_{k-1}})^2)x^{y_{k-2}})^2)\dots)^2)x^{y_1})^2)x^{y_0} \mod z \end{aligned}$$

If $y_{k-1} = 1$ according to the binary representation of y, $w = x^2$ is computed and then reduced modulo z. If $y_{k-2} = 1$, x is multiplied onto w, and the result is squared and then reduced modulo z. If $y_{k-2} = 0$, w is just squared and then reduced modulo z. This method can be expressed as pseudo-code in the following way:

Algorithm 1: Square-and-multiply algorithm

input : $x \in R$, where R is a ring with 1, and binary representation of $y \in \mathbb{N}_{>0}$, with k bits
output : $x^y \in R$

```
1 set w := 1
2 for i := k − 1 to 0 do
3 |   w := w^2
4 |   if y_i = 1 then
5 |   |   w := w · x
6 |   end
7 end
```

Theorem 2.8. *The running time of the square-and-multiply algorithm is $\mathcal{O}(M(k)k)$.*

Proof. Since k denotes the number of bits in the exponent y, then

$$y \leq 2^k \Leftrightarrow \lg y \leq k.$$

Recall that $\lg(y) = \log_2(y)$. The Algorithm uses $\lfloor \lg(y) \rfloor$ squarings in line 3 and $\leq \lfloor \lg(y) \rfloor$ multiplications in line 5. Hence, the total cost is at most $2\lg(y)$ multiplications. Since there are $\leq k$ bits in y, it can be concluded that the running time of the square-and-multiply algorithm, Algorithm 1 is $\mathcal{O}(M(k)k)$, where $M(k)$ denotes the multiplication time, by Definition 2.4.2.[4, p. 75] □

2.6 Some general remarks

The results in this section are fundamental algebraic results, and they are therefore stated without proof.

The following Theorem 2.9 will be used when proving the correctness of one of the probabilistic tests:

Theorem 2.9. *[4, p. 518] If the prime factorization of an integer n is*

$$n = p_1^{e_1} \cdots p_r^{e_r},$$

where $p_1, \ldots, p_r$ are distinct positive primes and $e_1, \ldots, e_r$ are positive integers, then the Chinese Remainder Theorem says that

$$\mathbb{Z}_N \cong \mathbb{Z}_{p_1^{e_1}} \times \cdots \times \mathbb{Z}_{p_r^{e_r}},$$

and that

$$\mathbb{Z}_N^* \cong \mathbb{Z}_{p_1^{e_1}}^* \times \cdots \times \mathbb{Z}_{p_r^{e_r}}^*.$$

Theorem 2.10. *[4, p. 47, 54] The greatest common divisor of to integers f, g can be calculated by using the Euclidean Algorithm. Assume that the integers f, g both have k bits. If $M(k) = k^2$, where M denotes the multiplication time as described in Section 2.4, then the running time of the Euclidean Algorithm is $\mathcal{O}(k^2)$. If $M(k) = \mathcal{O}(k^2)$ (which means $< k^2$), then the running time of the Euclidean Algorithm is $\mathcal{O}(M(k) \cdot \log k)$.*

Definition 2.6.1. [1, p. 14] Let $g, r \in \mathbb{N}$. Then the smallest natural number n (if it exists) such that $g^n \equiv 1 \mod r$ is called *the order of g modulo r* and will in the following be denoted $\mathrm{ord}_r(g) = n$.

Corollary 2.10.1. *[6, p. 13] Suppose that $a \mid bc$ where $a, b, c \in \mathbb{Z}$ and $\gcd(a, b) = 1$. Then $a \mid c$.*

Definition 2.6.2. [6, p. 145] A non-zero polynomial is called *monic* if its leading coefficient is 1. The leading coefficient of a polynomial is the coefficient a_n where the polynomial $f \in R[x]$ is written as

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

CHAPTER 3

Probabilistic Primality Tests

3.1 Probabilistic tests

A straightforward way to generate a k-bit prime is the following: Choose a random k-bit odd integer, m, and test whether any prime less than m is a divisor of m. If this is not the case, m must be a prime itself according to the Fundamental theorem of arithmetic presented in the introduction. If m is not a prime, then m can be factored into some factors, say a and b:

$$m = a \cdot b.$$

If both a and b are greater than $\sqrt{m}$, then $a \cdot b$ must be greater than m, which means that at least one factor must be less than or equal to $\sqrt{m}$. Therefore, it is actually only necessary to test prime divisors less than or equal to $\sqrt{m}$. If m is not a prime, then another random integer m is chosen, and the test is repeated. This method is called trial division, and the method can be expressed as pseudo-code in the following way:

Algorithm 2: Primality test by trial division

```
   input  : integer m
1  for all primes p ≤ √m do
2  |  if p | m then
3  |  |  print "m is not a prime" and stop
4  |  end
5  end
6  print "m is a prime" and stop
```

One problem is that trial division is very slow for big integers. In the worst case, all primes up to the square-root of m should be tested which is quite slow as the square-root of a k-bit integer is a $k/2$-bit integer. Hence, in the worst case, $\approx 2^{k/2}$ trials are needed. Fortunately, other methods that are faster than trial division exist, and in the rest of this section a probabilistic prime number test will be examined. The fact that it is a probabilistic test means that it returns an integer which is a prime with a certain probability p. To give a brief insight in types of algorithms, they are here classified into three groups of which the first and second group are probabilistic tests[3, p. 3-4]:

- **Monte Carlo algorithms:** Given an input, the output is correct with probability $\geq \frac{1}{2}$. The error probability is decreased by repeating the algorithm on the same input.
- **The category including the Fermat test:** On most input, it will behave as a Monte Carlo algorithm, but on some (small) subset of inputs, it will (almost) always answer incorrectly.
- **Las Vegas algorithms:** Given an input, the algorithm never lies. The output is either a correct answer or an honest report of failure. However, sometimes the algorithm should be run many times, before a correct answer is found. More specifically, a Las Vegas algorithm succeeds with probability $\geq \frac{1}{2}$, so it will usually quickly return the correct answer.

3.2 Fermat witness and liar

The first probabilistic test which is considered is the Fermat test which is based on Fermat's little theorem, Corollary 2.7.1. The algorithm can be seen in Algorithm 3. One step in the algorithm is to choose an $a \in \mathbb{Z}/N\mathbb{Z}\backslash\{0, 1, N-1\}$, uniformly at random, where N is the integer which is being tested for primality. This a is classified into one of two groups: Either a is called a *Fermat witness* to the compositeness of N, or it is called a *Fermat liar* for N. If any Fermat witness is known, then it is completely sure that N is composite[4, p. 519-520]. In order to understand how a is classified as either a Fermat witness or a Fermat liar, the following subset of $(\mathbb{Z}/N\mathbb{Z})^*$ is considered:

$$L_N = \{u \in (\mathbb{Z}/N\mathbb{Z})^* : u^{N-1} = 1\}.$$

By Theorem 2.3, $(\mathbb{Z}/N\mathbb{Z})^*$ is indeed a group, more precisely it is the multiplicative group of units in $\mathbb{Z}/N\mathbb{Z}$.

Lemma 3.1. *L_N is a subgroup of $(\mathbb{Z}/N\mathbb{Z})^*$ which means that L_N is a group itself.*

Proof. The Lemma is proved by showing that L_N satisfies the three properties in Definition 2.2.1:

- Take two elements $u_1, u_2 \in L_N$. Then:

  $$(u_1u_2)^{N-1} = u_1^{N-1}u_2^{N-1} = 1 \cdot 1 = 1.$$

  Hence, $u_1u_2 \in L_N$, and this shows that the first property is satisfied.

- There exists indeed the identity element, 1, in L_N.

- In order to determine whether the inverses exist, take a $u \in L_N$ and a $v \in (\mathbb{Z}/N\mathbb{Z})^*$ such that $u \cdot v = 1$. The question is then whether v is an element in L_N or not, i.e. whether $v^{N-1} = 1$ or not. By what is known so far, the following can be written:

  $$\begin{aligned} & u^{N-1} = 1 \\ \Rightarrow \quad & u^{N-1}v^{N-1} = v^{N-1} \\ \Rightarrow \quad & (uv)^{N-1} = 1 = v^{N-1} \end{aligned}$$

  Hence, $v^{N-1} = 1$, which shows that the inverse of any element in L_N is also contained in L_N.

The three properties are satisfied which justifies that L_N is a group. □

By Fermat's little theorem, Corollary 2.7.1, then $L_N = (\mathbb{Z}/N\mathbb{Z})^*$ if N is a prime. If $L_N \neq (\mathbb{Z}/N\mathbb{Z})^*$, then $|L_N| \leq \frac{1}{2}|(\mathbb{Z}/N\mathbb{Z})^*|$, since the size of a finite group is an integer multiple of the size of any of its subgroups, by Lagrange's theorem, Theorem 2.4. The probability that the a chosen in the algorithm, taken modulo N, is in L_N can be expressed in the following way: It is the cardinality of L_N divided by the cardinality of $(\mathbb{Z}/N\mathbb{Z})^*$ which is then given as

$$\frac{|L_N|}{|(\mathbb{Z}/N\mathbb{Z})^*|} \leq \frac{1}{2},$$

if $L_N \neq (\mathbb{Z}/N\mathbb{Z})^*$. If the chosen a, modulo N, is in $(\mathbb{Z}/N\mathbb{Z})^* \setminus L_N$, then such an a, and also its residue class $a \mod N$, is called a Fermat witness to the compositeness of N. If $a \mod N \in L_N$, then a and also $a \mod N$ is a Fermat liar for N.

3.3 The Fermat test

By definition, 2 is a prime number which implies that any even number > 2 can not be prime. If there existed an even prime number $p \geq 2$, then the prime number 2 would divide p, and hence p would contradict the definition of prime numbers. Therefore, any primality test clearly only focuses on testing odd numbers for primality.

The Fermat primality test is stated as pseudo-code below:

Algorithm 3: Fermat test

input : An odd integer $N \geq 5$
output : Either "composite" or "possibly prime"

```
1 choose a ∈ ℤ/Nℤ\{0, 1, N − 1} uniformly at random
2 call the repeated squaring algorithm 2 to compute b = a^{N−1} mod N
3 if b ≠ 1 then
4 |   return "composite" else
5 |   |   return "possibly prime"
6 |   end
7 end
```

Theorem 3.2. *If Algorithm 3 is given a prime N as input, then the Algorithm returns "possibly prime". If the Algorithm is given a composite number N as input, then the Algorithm recognizes the compositeness of N with probability $\geq \frac{1}{2}$. If it does not recognize the compositeness, then it returns "possibly prime".*

Proof. The test takes as input an odd integer $N \geq 5$ and outputs either "composite" or "possibly prime". If the a chosen at random in step 1 is a Fermat witness which means that $a \mod N \in (\mathbb{Z}/N\mathbb{Z})^* \setminus L_N$ and $b \neq 1$, then the compositeness of N is proved, and the Algorithm correctly returns "composite". If the randomly chosen a is a Fermat liar which means that $a \mod N \in L_N$, then it is neither proved nor rejected that N is a prime. If $L_N = (\mathbb{Z}/N\mathbb{Z})^*$, then N is a prime according to Fermat's little theorem, but by the test it is only known that this single element from $(\mathbb{Z}/N\mathbb{Z})^*$ is in L_N as well. Hence, the test returns "possibly prime". □

3.3.1 Complexity analysis

Theorem 3.3. *The running time of the Fermat test, Algorithm 3, is $\mathcal{O}(M(k)k)$.*

Proof. Step 1 in Algorithm 3 is to choose a random number $a \in \mathbb{Z}/N\mathbb{Z}\backslash\{0, 1, N-1\}$. Here, it is assumed that the complexity of generating a random number is $\mathcal{O}(k)$, but see Remark 5.2. Step 2 is to call the square-and-multiply algorithm, Algorithm 1, to compute $b = a^{N-1} \mod N$. By Theorem 2.8, the running time of this algorithm is $\mathcal{O}(M(k)k)$, where k is the number of bits in the exponent (which means the number of bits in $N - 1$ in this case). The third and last step of Algorithm 3 is to check whether $b = 1$ or not. This operation takes constant time, $\mathcal{O}(1)$. The conclusion is that the running time of the Fermat test is $\mathcal{O}(M(k)k)$. □

3.4 Carmichael numbers

Is the Fermat test sufficient? If N is composite and $L_N = (\mathbb{Z}/N\mathbb{Z})^*$, then the test returns "possibly prime", even though N is composite. This means that the Fermat test can prove compositeness of an integer, but it can not prove primality. Those composite integers N for which $L_N = (\mathbb{Z}/N\mathbb{Z})^*$ are called *the Carmichael numbers*. The Carmichael numbers are in other words the composite numbers without any Fermat witnesses.

Definition 3.4.1. [6, p. 27] Let N be a composite natural number and a an integer. Then N is called a *pseudoprime* relative to the base a, if $a^{N-1} \equiv 1 \pmod{N}$.

By Definition 3.4.1, Carmichael numbers are numbers which are pseudoprime to every relatively prime base. The smallest Carmichael number is $N = 561 = 3 \cdot 11 \cdot 17$.

The following Lemma is stated without proof:

Lemma 3.4. *[4, p. 521] Any Carmichael number is squarefree.*

Because of the existence of Carmichael numbers, there are composite numbers that are not distinguished from prime numbers by Fermat's little theorem, Corollary 2.7.1. However, there is a simple way to improve this situation, and the main ingredient of this improvement is the following Lemma:

Lemma 3.5. *[6, p. 27] Let p be a prime number and $x \in \mathbb{Z}$. If $x^2 \equiv 1 \pmod{p}$, then $x \equiv \pm 1 \pmod{p}$.*

Proof. By definition, if $x^2 \equiv 1 \pmod{p}$, then $p|x^2 - 1 = (x+1)(x-1)$. Hence, $p|x+1$ or $p|x-1$, which completes the proof. □

Consider for example $N = 341$ and $a = 2$. Using the square-and-multiply algorithm, Algorithm 1, one can compute

$$2^{340} \equiv 1 \pmod{341}.$$

From this, it is not possible to say whether 341 is composite or not, but by repeatedly using Lemma 3.5, it can actually be concluded whether 341 is composite: Assume that 341 is a prime number. Then, by Lemma 3.5

$$2^{170} \equiv \pm 1 \pmod{341},$$

since $(2^{170})^2 = 2^{340}$. One can compute that $2^{170} \equiv 1 \pmod{341}$, which means that the Lemma holds so far. Now, since $2^{170} = (2^{85})^2$, Lemma 3.5 implies that

$$2^{85} \equiv \pm 1 \pmod{341}.$$

When computing this, one gets that $2^{85} \equiv 32 \pmod{341}$, which shows that 341 can not be a prime number. This example leads to the following Definition:

Definition 3.4.2. [6, p. 28] An odd composite number N is called a *strong pseudoprime* relative to the base a, if either $a^q \equiv 1 \pmod{N}$, or there exists $i = 0, \ldots, k-1$, such that

$$a^{2^i q} \equiv -1 \pmod{N},$$

where $N - 1 = 2^k q$ and $2 \nmid q$.

The strong pseudoprimes defined in Definition 3.4.2 are exactly the composite numbers, which satisfy both Fermat's little theorem, Corollary 2.7.1, and Lemma 3.5. This leads to the following Proposition, which shows that a number which fails repeated use of Lemma 3.5, as for the example with $N = 341$ and $a = 2$, must be composite:

Proposition 3.6. *Let p be an odd prime, and let*

$$p - 1 = 2^k q,$$

where $2 \nmid q$. If $a \in \mathbb{Z}$ and $\gcd(a, p) = 1$, then either $a^q \equiv 1 \pmod{p}$, or there exists $i = 0, \ldots, k-1$, such that

$$a^{2^i q} \equiv -1 \pmod{p}.$$

Proof. Let $b_i = a^{2^i q}$, $i = 0, \ldots, k$. Clearly, $b_k = a^{2^k q} = a^{p-1} \equiv 1 \pmod{p}$ by Fermat's little theorem, Corollary 2.7.1, and $b_{i+1} = b_i^2$ for $i = 0, \ldots, k-1$. This means that $b_0 \equiv 1 \pmod{p}$, if and only if $b_i \equiv 1 \pmod{p}$ for every $i = 0, \ldots, k$. Hence, if $b_0 \not\equiv 1 \pmod{p}$, then there exists $b_i, i \geq 0$, such that $b_i \not\equiv 1 \pmod{p}$. Now, let j be the largest index with this property ($b_j \not\equiv 1 \pmod{p}$). Since $j < k$ and $b_j^2 \equiv b_{j+1} \equiv 1 \pmod{p}$, then $b_j \equiv -1 \pmod{p}$ by Lemma 3.5 (by the Lemma, $b_j \equiv 1 \pmod{p}$ or $b_j \equiv -1 \pmod{p}$, and since $b_j \not\equiv 1 \pmod{p}$, it can be deduced that $b_j \equiv -1 \pmod{p}$). □

3.5 The Miller-Rabin Primality Test

Algorithm 4: The Miller-Rabin primality test.

input : An odd integer $N \geq 3$.
output : Either "composite" or "probably prime".

choose $a \in \mathbb{Z}/N\mathbb{Z} \backslash \{0, 1, N-1\}$ uniformly at random
$d \leftarrow \gcd(a, N)$
if $d > 1$ **then**
  return "composite"
end
write $N - 1 = 2^v m$ with $v, m \in \mathbb{N}, v \geq 1$, and m odd
Call the square-and-multiply algorithm, Algorithm 1, to compute $b_0 = a^m \mod N$
if $b_0 = 1$ **then**
  return "probably prime"
end
for $i = 1, \ldots, v$ **do**
  $b_i \leftarrow b_{i-1}^2 \mod N$
end
if $b_v \neq 1$ **then**
  return "composite"
end
Let $j \leftarrow \min\{0 \leq i < v : b_{i+1} = 1\}$
if $b_j = N - 1$ **then**
  return "probably prime" **else**
    return "composite"
  end
end

The Miller-Rabin primality test, Algorithm 4, improves the simple Fermat test such that there is no input on which it systematically fails. If a Carmichael number is given as input, then the test recognizes it as a composite number.

Theorem 3.7. *If Algorithm 4 is given a prime N as input, then the Algorithm returns "probably prime". If the Algorithm is given a Carmichael number or another composite number N as input, then the Algorithm recognizes the compositeness of the number and returns "composite" with probability $\geq \frac{1}{2}$.*

Proof. The proof of Theorem 3.7 is divided into three cases: First, the case where N is prime is shown, then the case where N is composite and not a Carmichael number is shown, and finally the case where N is a Carmichael number is shown:

The case where N is prime follows directly from Proposition 3.6. Either, $b_0 = a^m \equiv 1 \mod N$ and the Algorithm returns "probably prime" in line 8, or there exists $i = 0, \ldots, v-1$, such that $b_i = a^{2^i m} \equiv -1 \mod N$ and the Algorithm returns "probably prime" in line 18. By Fermat's little theorem, Corollary 2.7.1, then $b_v = a^{2^v m} \equiv 1 \mod N$, and the Algorithm correctly does not return "composite" in line 14. Hence, if N is prime, then Algorithm 4 always proves the primality by returning "probably prime".

The next case to consider is the case where N is composite and not a Carmichael number. This part is easily proved: If $d = \gcd(a, N) > 1$, then $(\mathbb{Z}/N\mathbb{Z})^* \neq L_N$, and the Algorithm correctly returns "composite" in line 3. If N is composite, then the randomly chosen a is a Fermat witness for N with probability $\geq \frac{1}{2}$. In such a case, then $b_v \neq 1$, and the Algorithm returns "composite" in line 14.

The last case to consider is the case where N is a Carmichael number which is the more complicated part of the proof: First, define

$$I = \{i : 0 \leq i \leq v \quad \text{and} \quad \forall u \in (\mathbb{Z}/N\mathbb{Z})^* u^{2^i m} = 1\}.$$

Since N is a Carmichael number, it is known that $(\mathbb{Z}/N\mathbb{Z})^* = L_N$ by the definition of Carmichael numbers, hence $v \in I$. Since m is odd, one can take $(-1)^{2^0 m} = (-1)^m = -1 \neq 1 \mod N$ which means that $0 \notin I$. By induction, it follows that $i + 1 \in I$ for any $i \in I$ with $i < v$, since

$$u^{2^i m} = 1 \Rightarrow u^{2^{i+1} m} = u^{2^1 2^i m} = (u^{2^i m})^2 = 1^2 = 1.$$

Hence, there exists some $l < v$ such that

$$I = \{l + 1, l + 2, \ldots, v\}.$$

Define the following subset of $(\mathbb{Z}/N\mathbb{Z})^*$:

$$G = \{u \in (\mathbb{Z}/N\mathbb{Z})^* : u^{2^l m} = \pm 1\} \subseteq (\mathbb{Z}/N\mathbb{Z})^*.$$

This subset of $(\mathbb{Z}/N\mathbb{Z})^*$ includes all values of a for which the algorithm returns "probably prime". If it can be shown that G is actually a subgroup of $(\mathbb{Z}/N\mathbb{Z})^*$ and that $G \neq (\mathbb{Z}/N\mathbb{Z})^*$, then the probability that the a chosen in the algorithm, modulo N, is in G is bounded by Lagrange's theorem in Theorem 2.4.

Claim 3.8. *G is a group.*

Proof. The claim is proved by showing that G satisfies the three properties in Definition 2.2.1:

- Take two elements $a_1, a_2 \in G$. Then:

$$(a_1 a_2)^{2^l m} = a_1^{2^l m} a_2^{2^l m} = \pm 1 \cdot \pm 1 = \pm 1.$$

  Hence, $a_1 a_2 \in G$, and this shows that the first property is satisfied.

- There exists indeed the identity element, 1, in G since

$$1^{2^l m} = 1.$$

- If $a \in G$, then

$$(a^{-1})^{2^l m} = (a^{2^l m})^{-1} = \pm 1.$$

  Hence, $a^{-1} \in G$, which shows that the inverse of any element in G is also contained in G.

The three properties are satisfied which proves the claim that G is a group. □

Claim 3.9. *$G \neq (\mathbb{Z}/N\mathbb{Z})^*$.*

Proof. If it is possible to find an element in $(\mathbb{Z}/N\mathbb{Z})^* \setminus G$, then the claim is proved. First, take a $b \in (\mathbb{Z}/N\mathbb{Z})^*$ such that $b^{2^l m} \neq 1$. It might be the case that $b \in G$, but the element b can be used to build an element $c \in (\mathbb{Z}/N\mathbb{Z})^* \setminus G$. Since N is a Carmichael number, N is squarefree by definition. This means that N can be written as

$$N = p_1 \cdot p_2 \cdots p_k,$$

where all the primes are distinct. From the Chinese Remainder Theorem in Theorem 2.9, the following can be written:

$$\mathbb{Z}_N \simeq \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \cdots \times \mathbb{Z}_{p_k} \simeq \mathbb{Z}_{p_i} \times \mathbb{Z}_{(N/p_i)}.$$

Hence, from the Chinese Remainder Theorem, $\exists i$ such that $b^{2^l m} \neq 1 \mod p_i$, since $1 \in (\mathbb{Z}/N\mathbb{Z})^*$ is the only element which is congruent to $1 \mod$ each p_i in the equation above. The Chinese Remainder Theorem implies that there exists a $c \in \mathbb{Z}/N\mathbb{Z}$ such that

$$c \equiv b \mod p_i \quad \text{and} \quad c \equiv 1 \mod N/p_i.$$

It is now claimed that this $c \in (\mathbb{Z}/N\mathbb{Z})^* \setminus G$. If it can be proved that this claim is correct, then the proof is done. First, it is shown that $c \in (\mathbb{Z}/N\mathbb{Z})^*$ by showing that it has a multiplicative inverse. Let $d \in \mathbb{Z}/N\mathbb{Z}$ such that

$$d \equiv b^{-1} \mod p_i \quad \text{and} \quad d \equiv 1 \mod N/p_i.$$

This implies that

$$c \cdot d \equiv 1 \mod p_i \quad \text{and} \quad c \cdot d \equiv 1 \mod N/p_i.$$

Hence, $d = c^{-1}$. To show that $c \notin G$, $c^{2^l m}$ is considered. From what is known so far, the following can be written:

$$c^{2^l m} \equiv b^{2^l m} \neq 1 \mod p_i \quad \text{and} \quad c^{2^l m} \equiv 1^{2^l m} = 1 \mod N/p_i.$$

Hence, by the Chinese Remainder Theorem, $c^{2^l m} \neq 1$, since it is not the case that $c^{n^l m} = 1$, both $\mod p_i$ and $\mod N/p_i$. Similarly, it can be concluded that $c^{2^l m} \neq -1$. The fact that N is odd implies

that $1 \neq -1 \mod N/p_i$. Hence, it is not the case that $c^{n^l m} = -1$, both $\mod p_i$ and $\mod N/p_i$, and therefore it also follows from the Chinese Remainder Theorem that $c^{2^l m} \neq -1$. Hence, $c^{2^l m} \neq 1$ and $c^{2^l m} \neq -1$, which shows that $c \notin G$. To sum up: The group G which includes all values of a for which Algorithm 4 returns "probably prime", is a subgroup of $(\mathbb{Z}/N\mathbb{Z})^*$, and since $G \neq (\mathbb{Z}/N\mathbb{Z})^*$, then the probability that the randomly chosen a is in G is bounded by Lagrange's theorem, Theorem 2.4. This implies that Algorithm 4 recognizes a Carmichael number N as composite with probability $\geq \frac{1}{2}$. □

This completes the proof of Theorem 3.7. □

3.5.1 Complexity analysis

Theorem 3.10. *The running time of the Miller-Rabin test, Algorithm 4, is $\mathcal{O}(M(k)k)$.*

Proof. As for the Fermat test, step 1 in the Miller-Rabin test, Algorithm 4, is to choose a random number $a \in \mathbb{Z}/N\mathbb{Z}\backslash\{0, 1, N-1\}$, and again it is assumed that this step takes $\mathcal{O}(k)$. Step 2 is to check whether $\gcd(a, N) > 1$ or not, and by Theorem 2.10 the complexity of this step is $\mathcal{O}(M(k)\lg(k))$. In line 6, the square-and-multiply algorithm, Algorithm 1, is called to compute $b_0 = a^m \mod N$. By Theorem 2.8, the complexity of this step is $\mathcal{O}(M(k)k)$. To check whether $b_0 = 1$ or not in line 7 takes constant time, $\mathcal{O}(1)$. In line 10-12, $b_i = b_{i-1}^2 \mod N$ is computed for $i = 1, \ldots, v$. This means v multiplications and v modulo reductions, but since these operations are applied to integers modulo N, which means k bits, then the running time of each multiplication or modulo reduction is $M(k)$. This means that the total running time of this step is $\mathcal{O}(v \cdot M(k))$. To make this complexity more readable, note that

$$N - 1 = 2^v m \Leftrightarrow \lg(N-1) = \lg(2^v m) \Leftrightarrow \lg(N-1) = v + \lg(m) \Leftrightarrow v \leq \lg(N-1) \leq \lg(N).$$

This means that the complexity of this step can more precisely be written as $\mathcal{O}(M(k)\lg(N)) = \mathcal{O}(M(k)k)$. The next step in the algorithm is to check whether $b_v \neq 1$ or not, which takes $\mathcal{O}(1)$. In line 16, j is determined by going through at most v numbers. The complexity of this step is $\mathcal{O}(v) = \mathcal{O}(\lg(N))$. The last step in the algorithm is to check whether $b_j = N - 1$ or not, which takes constant time, $\mathcal{O}(1)$. The conclusion is that the running time of the Miller-Rabin test is $\mathcal{O}(M(k)k)$. □

CHAPTER 4

The AKS algorithm

In August 2002, M. Agrawal, N. Kayal, and N. Saxena announced a spectacular new development, a deterministic, polynomial-time primality test. This is now known as the AKS test. The new test is not just sensational because it finally settles the theoretical issue of primality testing after researchers were so close in so many ways, it is remarkable in that the test itself is quite simple.

4.1 Preliminaries

Before the AKS primality test is stated and proved, some preliminaries are introduced. The following Lemmas 4.1 and 4.2 will be useful when proving the correctness of the AKS test.

Lemma 4.1. *If $k|\tau$, then $(x^k-1)|(x^\tau-1)$ in $R[x]$ for some ring R.*

Proof. Let $\tau = k \cdot c$, $c \in \mathbb{Z}[x]$. Consider:

$$\begin{aligned}(x^k-1)(x^{k(c-1)}+\cdots+x^{2k}+x^k+1) =& (x^{kc}+\cdots+x^{3k}+x^{2k}+x^k) \\ & -(x^{k(c-1)}+\cdots+x^{2k}+x^k+1) \\ =& x^{kc}-1 \\ =& x^\tau-1,\end{aligned}$$

which completes the proof. □

Lemma 4.2. *Let $\mathbb{Z}[u,v]$ be a bivariate polynomial ring. For any $g(x) \in \mathbb{Z}[x]$, $u-v$ divides $g(u)-g(v) \in \mathbb{Z}[u,v]$.*

Proof. Let $h(u) = g(u)-g(v) \in \mathbb{Z}[v][u]$, where $\mathbb{Z}[v][u]$ corresponds to $R[u]$, where $R = \mathbb{Z}[v]$ is an integral domain. Then $h(u)$ has a root $v \in R$, which means that $(u-v)|h(u)$, and this completes the proof. □

4.1.1 The Isomorphism Theorem

The Isomorphism Theorem is a fundamental tool in algebra, and therefore it is now stated without proof. In this project, the Isomorphism Theorem for rings will be used, and it is presented in Theorem 4.3.

Theorem 4.3 (The Isomorphism Theorem for rings)**.** *[6, p. 120] Let R and S be rings and $f: \ R \to S$ a ring homomorphism with kernel $K = ker(f)$. Then*

$$\tilde{f}: \ R/K \to f(R)$$

given by $\tilde{f}(r+K) = f(r)$ is a well defined map and a ring isomorphism.

Theorem 4.4. *For any $p \in \mathbb{Z}$ and $h(x) \in \mathbb{Z}[x]$, then $\mathbb{Z}[x]/\langle p, h(x)\rangle$ and $(\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x})\rangle$ are isomorphic.*

Proof. The isomorphism between $\mathbb{Z}[x]/\langle p, h(x)\rangle$ and $(\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x})\rangle$ is proved in the following way: The homomorphism,

$$\phi: \quad \mathbb{Z}[x] \to (\mathbb{Z}/p\mathbb{Z})[\bar{x}]$$

naturally exists, and it simply reduces each coefficient of a polynomial modulo p. Let $\bar{h}(\bar{x}) = \phi(h)$. Then the homomorphism,

$$\begin{aligned} \eta: \quad (\mathbb{Z}/p\mathbb{Z})[\bar{x}] &\to (\mathbb{Z}/p\mathbb{Z}))[\bar{x}]/\langle \bar{h}(\bar{x})\rangle \\ \eta(f(\bar{x})) &= f(\bar{x}) + \langle \bar{h}(\bar{x})\rangle \end{aligned}$$

naturally exists as well, and it simply takes the congruence class of a polynomial modulo $\bar{h}(\bar{x})$. Note that the polynomial f is already reduced modulo p. The composition of the two homomorphisms is then defined as

$$\eta \circ \phi: \quad \mathbb{Z}[x] \to (\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x})\rangle.$$

The composition is a homomorphism as well, and as both ϕ and η are surjective, $(\eta \circ \phi)$ is also surjective. The Isomorphism Theorem, Theorem 4.3, is now used. The fact that $(\eta \circ \phi)$ is surjective means that

$$\mathrm{Im}(\eta \circ \phi) = (\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x})\rangle.$$

If it can be shown that $\ker(\eta \circ \phi) = \langle p, h(x)\rangle$, then it can be concluded that

$$\mathbb{Z}[x]/\langle p, h(x)\rangle \text{ and } (\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x})\rangle$$

are isomorphic, which completes the proof. The kernel of $(\eta \circ \phi)$, $\ker(\eta \circ \phi)$, is the set of polynomials $f(x)$ in $\mathbb{Z}[x]$ such that $f(x)$ reduced modulo p and then reduced modulo $\bar{h}(\bar{x})$ is 0. Clearly, both p and $h(x)$ are in $\ker(\eta \circ \phi)$, which can be expressed as

$$\langle p, h(x)\rangle \subset \ker(\eta \circ \phi).$$

The last step is to show that $\ker(\eta \circ \phi)$ is actually equal to $\langle p, h(x)\rangle$. Let

$$\bar{h}(x) = h(x) + w(x) \cdot p$$

and

$$f(x) = p \cdot u(x) - \bar{h}(x)v(x) = 0,$$

which means that f is first reduced modulo p and then reduced modulo $\bar{h}(\bar{x})$. Then there must exist $u(x), v(x), w(x)$ such that

$$f(x) = p(u(x) + w(x)v(x)) - h(x)w(x) = 0.$$

This means that $f(x) \in \langle p, h(x)\rangle$. Hence, $\ker(\eta \circ \phi) = \langle p, h(x)\rangle$, which completes the proof. □

The following Theorem 4.5 presents a map which will be useful when proving the correctness of the AKS primality test:

Theorem 4.5. *Let n, p be integers such that $p|n$. Then the following map ϕ is a well defined map:*

$$\begin{aligned} \phi: \quad & \mathbb{Z}/n\mathbb{Z} \to \mathbb{Z}/p\mathbb{Z} \\ & a + n\mathbb{Z} \mapsto a + p\mathbb{Z}. \end{aligned}$$

Proof. If $a + n\mathbb{Z} = b + n\mathbb{Z}$, then

$$\phi(a + n\mathbb{Z}) = \phi(b + n\mathbb{Z}) \Rightarrow a + p\mathbb{Z} = b + p\mathbb{Z},$$

since

$$n|(a - b) \Rightarrow p|(a - b) \Rightarrow a + p\mathbb{Z} = b + p\mathbb{Z}.$$

□

4.1.2 Roots of unity and cyclotomic polynomials

Definition 4.1.1. [6, p. 154] A complex number ξ is called an nth root of unity for a positive integer n if

$$\xi^n = 1,$$

which means that the nth roots of unity are exactly the roots of the polynomial $x^n - 1$. The nth roots of unity can also be written as

$$\xi_k = e^{k2\pi i/n},$$

for $k = 1, \ldots, n$.

It may happen that n is not the smallest integer for which $\xi^n = 1$. Now, denote the root as ζ. If $\zeta^n = 1$ and

$$\zeta, \zeta^2, \ldots, \zeta^{n-1} \neq 1,$$

then the complex number ζ is called a primitive nth root of unity. The following Lemma clarifies the concept of primitive nth roots of unity:

Lemma 4.6. *[6, p. 155] A complex number ζ is a primitive nth root of unity if and only if*

$$\zeta = e^{k2\pi i/n},$$

where $1 \leq k \leq n$ and $\gcd(k, n) = 1$. If ζ is a primitive nth root of unity and $\zeta^m = 1$, then $n \mid m$.

Proof. By Definition 4.1.1, the nth roots of unity are

$$e^{k2\pi i/n},$$

where $k = 1, \ldots, n$. Let $\xi = e^{k2\pi i/n}$ be an nth root of unity. If $\xi^m = 1$, then $k2\pi m/n$ is an integer multiple of 2π which implies that $n \mid km$. Assume that $\gcd(k, n) = g > 1$. Then $\xi^{n/g} = 1$ which would imply that ξ is not a primitive nth root of unity. Oppositely, assume that $\gcd(k, n) = 1$. Then $n \mid km$ implies that $n \mid m$, by Corollary 2.10.1. This implies that

$$\xi, \xi^2, \ldots, \xi^{n-1} \neq 1,$$

which shows that ξ is a primitive nth root of unity.
Now, the last part of the Lemma: If ζ is a primitive nth root of unity and $\zeta^m = 1$, then the following can be written

$$m = qn + r,$$

where $0 \leq r < n$. This shows that $\zeta^m = \zeta^r$, and since ζ is an nth root of unity and since $r < n$, then it must be the case that $r = 0$. This implies that

$$m = qn \Rightarrow n \mid m.$$

□

Primitive nth roots of unity are used to define cyclotomic polynomials:

Definition 4.1.2. [6, p. 156] Let $n \in \mathbb{N}$ with $n \geq 1$. Then the nth cyclotomic polynomial is defined as

$$\Phi_n(x) = \prod_{1 \leq k \leq n, \gcd(k,n)=1} (x - e^{k2\pi i/n}),$$

in $\mathbb{C}[x]$. Cyclotomic polynomials have integer coefficients even though they are defined using roots of unity in the complex plane.

The following Proposition 4.7 will be useful when studying the correctness of the AKS primality test:

Proposition 4.7. *[6, p. 157] Let $d, n \in \mathbb{N}$, $n \geq 1$. Then*

$$x^n - 1 = \prod_{d \mid n} \Phi_d(x). \tag{4.1}$$

Proof. By Definition 2.6.2, then the polynomial on the left hand side in Equation (4.1) is monic, and since the right hand side of Equation (4.1) is a product of cyclotomic polynomials, then this side is monic as well. Now, let ζ be an nth root of unity. By definition of nth roots of unity, then the right hand side of Equation (4.1) implies that $\text{ord}(\zeta) = d$ for some $d \mid n$, and this means that ζ is a primitive dth root of unity. By Lemma 4.1, then $d \mid n$ implies that $x^d - 1 \mid x^n - 1$. Therefore, $\zeta^d - 1 \mid \zeta^n - 1$, but since $\zeta^n - 1 = 0$, then also $\zeta^d - 1 = 0 \Leftrightarrow \zeta^d = 1$. Hence, ζ is also a root of the right hand side of Equation (4.1). The conclusion is that the polynomials of each side of the equality in Equation (4.1) are both monic and have the same roots which means that they are equal. □

4.2 From Fermat's little theorem to the AKS primality test

Recall that the binomial theorem gives a formula for expanding $(x+y)^n$ as a sum of multiples of terms $x^i y^{n-i}$. The formula is given as

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i},$$

where $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ and $n!$ denotes the factorial of n ($n! = n \cdot (n-1) \cdots 2 \cdot 1$)[5, p. 8]. However, often the expression $(x+y)^n$ is written incorrect, namely as

$$(x+y)^n = x^n + y^n \tag{4.2}$$

The formula in Equation (4.2) is by some called the *Child's Binomial Theorem*, and it turns out that there are some circumstances for which the expression is actually correct. This claim will be explained by stating and proving the following Theorem:

Theorem 4.8. *[5, p. 12] An integer n is prime if and only if*

$$(x+a)^n \equiv x^n + a \pmod{n}, \tag{4.3}$$

in $\mathbb{Z}[x]$ for all $a \in \mathbb{Z}$.

Proof. The proof is divided into two cases: First, it is shown that if n is prime, then Equation (4.3) holds. Afterwards, it is oppositely shown that if n is composite, then Equation (4.3) does not hold.
So now, assume that n is prime. Recall that the left hand side of the congruence in Equation (4.3) can be written as

$$(x+a)^n = \sum_{i=0}^{n} \binom{n}{i} x^i a^{n-i}.$$

When only looking at $i = 0$, the binomial theorem gives

$$\binom{n}{0} x^0 a^{n-0} = \frac{n!}{0!(n-0)!} a^n = \frac{n!}{n!} a^n = a^n, \tag{4.4}$$

where it has been used that $0! = 1$. Similarly, when only looking at $i = n$, the binomial theorem gives

$$\binom{n}{n} x^n a^{n-n} = \frac{n!}{n!(n-n)!} x^n = \frac{n!}{n!} x^n = x^n. \tag{4.5}$$

Equation (4.4) and (4.5) implies that

$$(x+a)^n = \sum_{i=0}^{n} \binom{n}{i} x^i a^{n-i} = x^n + a^n + \sum_{i=1}^{n-1} \binom{n}{i} x^i a^{n-i}. \tag{4.6}$$

It is now claimed that since n is prime, then n divides $\binom{n}{i}$ for all i such that $1 \leq i < n$. By definition, it can be seen that

$$\binom{n}{i} i! = \frac{n!}{(n-i!)} = \frac{n \cdot (n-1) \cdots 2 \cdot 1}{(n-i) \cdot (n-i-1) \cdots 2 \cdot 1} = n \cdot (n-1) \cdots (n-i+2) \cdot (n-i+1). \tag{4.7}$$

Clearly, n divides the right hand side of Equation (4.7), since n divides $m \cdot n$ for some integer m. This means that n must also divide the other parts of the Equation. Since all terms in $i!$ are less than n, since $1 \leq i < n$, and since n is prime, then indeed $n \nmid i!$. Hence, the left hand side of Equation (4.7) shows that n divides $\binom{n}{i}$ for all i such that $1 \leq i < n$, as claimed. Hence, Equation (4.6) can be reduced to

$$(x+a)^n \equiv x^n + a^n \pmod{n}, \tag{4.8}$$

since all the terms $\binom{n}{i} x^i a^{n-i}$ are cancelled out when reduced modulo n. Equation (4.8) does not look completely like Equation (4.3) yet. Recall that by Fermat's little theorem, Theorem 2.7.1,

$$a^n \equiv a \mod n.$$

Applying Fermat's little theorem to Equation (4.8), it can be seen that

$$(x+a)^n \equiv x^n + a \mod n,$$

which completes this part of the proof.
Now, assume that n is composite, and let p be a prime dividing n. Consider

$$\binom{n}{p} = \frac{n!}{p!(n-p)!} = \frac{n \cdot (n-1) \cdots 2 \cdot 1}{p \cdot (p-1) \cdots 2 \cdot 1 \cdot (n-p) \cdot (n-p-1) \cdots 2 \cdot 1} = \frac{n \cdot (n-1) \cdots (n-p+2) \cdot (n-p+1)}{p \cdot (p-1) \cdots 2 \cdot 1}.$$

The only terms that p divides in the expression above is the n in the numerator and the p in the denominator. If p^k is the largest power of p dividing n, then $\frac{p^k}{p} = p^{k-1}$ is the largest power of p dividing $\binom{n}{p}$. Therefore, n does not divide $\binom{n}{p}$, and therefore

$$(x+a)^n \not\equiv x^n + a \mod n,$$

when n is composite. This completes second part of the proof. □

Theorem 4.8 is the basis of the AKS primality test. An obvious question arising from this fact is the following: Why does the AKS test not simply contain computing $(x+a)^n - (x^n+a) \mod n$ for integers a and determine whether n divides each coefficient or not? This is indeed a correct primality test. The problem is that $(x+a)^n$ has in its expansion n coefficients, so computing $(x+a)^n$ would involve storing all these n coefficients which would take too much time. One solution to this problem is to reduce $(x+a)^n$ modulo the polynomial $x^r - 1$ for a small choice of r as well as the reduction modulo n. Hence, the primality test is now to check whether

$$(x+a)^n \equiv x^n + a \mod (n, x^r - 1) \tag{4.9}$$

is satisfied or not[5, p. 13]. In this way, the number of evaluations when computing $(x+a)^n$ are reduced, which means that the computation is faster. However, recall what happened if the Fermat test presented in Chapter 3 was given a Carmichael number as input. Similarly, it can happen that Equation (4.9) is satisfied for composite n for some values of a and r. In order to avoid this problem, the value of r and the number of values of a are chosen appropriately, and it turns out that in cases where n is composite and Equation (4.9) is satisfied, then n will be a prime power. This explains why the check of the congruence in Equation (4.9) does not account for the whole AKS primality test (see Theorem 4.9).

Now, it is finally time to present the AKS primality test:

Theorem 4.9 (The AKS primality test)**.** *[5, p. 4] For given integer $n \geq 2$, let r be a positive integer $< n$, for which $ord_r(n) > (\log n)^2$. Then n is prime if and only if*

- *n is not a perfect power,*
- *n does not have any prime factor $\leq r$,*
- *$(x+a)^n \equiv x^n + a \pmod{n, x^r - 1}$ for each integer a, $1 \leq a \leq \sqrt{r}\log n$.*

The proof of Theorem 4.9 is the subject of the following section.

4.3 Correctness of the AKS primality test

Below, Theorem 4.9 is translated into pseudocode:

Algorithm 5: Agrawal, Kayal, and Saxena

input : An odd integer $n \geq 2$, a positive integer $r < n$, for which n has order $> (\log n)^2$
output : Either "composite" or "prime"

```
1  if ∃a, b ∈ ℕ, a, b > 1, such that n = a^b then
2  |  return "composite"
3  end
4  if ∃z ≤ r, such that 1 < gcd(z, n) < n then
5  |  return "composite"
6  end
7  for a = 1 to ⌊√r log n⌋ do
8  |  if (x + a)^n ≢ x^n + a (mod n, x^r − 1) then
9  |  |  return "composite"
10 |  end
11 end
12 return "prime"
```

Even though the AKS primality test, Algorithm 4.9, is actually very simple, the proof of the correctness of the algorithm is quite complicated. Therefore, it is divided into some Lemmas:

Lemma 4.10. *If n is prime, then Algorithm 5 returns "prime".*

Proof. If n is prime, then n is not a perfect power, by the Fundamental Theorem of Arithmetic, Theorem 2.1. The integer n does not have any prime factor $\leq r$, again by Theorem 2.1. Finally, by Section 4.2, the congruences $(x+a)^n \equiv x^n + a \mod (n, x^r - 1)$ are satisfied for each integer a, $1 \leq a \leq \sqrt{r} \log n$. Hence, if Algorithm 5 is given an integer n which is prime as input, then the Algorithm returns "prime" in line 12. □

The next Lemma should logically describe what happens when n is composite, but since it now gets more complicated, it makes more sense to dig into the theory, before the Lemma is finally stated and proved:

In the following, it is assumed that all the criteria stated in Theorem 4.9 can hold for a composite n. It will later turn out that this can not be true. In particular, the group G which will be defined in the following will turn out to be fictive, because it can actually not exist:

Now, take an odd integer $n > 1$ and assume that n is not a perfect power, n does not have any prime factor $\leq r$, for which $\mathrm{ord}_r(n) = d > (\log n)^2$, and such that the congruences

$$(x+a)^n \equiv x^n + a \mod (n, x^r - 1), \tag{4.10}$$

are satisfied for each integer a, $1 \leq a \leq A$, where $A = \sqrt{r} \log n$. Furthermore, assume that n is composite. If it can be shown that these assumptions can not all be true, then it is proved that the criteria in Theorem 4.9 can only hold when n is prime. Since n is composite and not a perfect power, then n must have at least two prime factors, by the Fundamental Theorem of Arithmetic, Theorem 2.1. Choose a prime p such that p divides n. Equation (4.10) can be seen as elements in the quotient ring $\mathbb{Z}[x]/\langle n, x^r - 1\rangle$. By Theorem 4.5, it can be seen that $\langle n, x^r - 1\rangle \subseteq \langle p, x^r - 1\rangle$, since $p \mid n$. Therefore, the congruences in Equation (4.10) can also be expressed as elements in the quotient ring $\mathbb{Z}[x]/\langle p, x^r - 1\rangle$ which means that

$$(x+a)^n \equiv x^n + a \mod (p, x^r - 1). \tag{4.11}$$

By Proposition 4.7, the polynomial $x^r - 1$ can be factored into irreducibles in $\mathbb{Q}[x]$ as $\prod_{d|r} \Phi_d(x)$, where $\Phi_d(x)$ is the dth cyclotomic polynomial whose roots are the primitive dth roots of unity. Each $\Phi_r(x)$ is indeed irreducible in $\mathbb{Q}[x]$, but may not be irreducible in $(\mathbb{Z}/p\mathbb{Z})[x]$. However, this problem is solved by the following Lemma:

Lemma 4.11. *Let $h(x)$ be an irreducible polynomial dividing $x^r - 1$ in $(\mathbb{Z}/p\mathbb{Z})[x]$. Then*

$$(x + a)^n \equiv x^n + a \mod \langle p, h(x)\rangle, \tag{4.12}$$

for each integer a, $1 \leq a \leq A$.

Proof. By Theorem 4.5, then $\langle p, h(x)\rangle \subseteq \langle p, x^r - 1\rangle$, since $h(x) \mid x^r - 1$. □

By Theorem 4.4, the Isomorphism Theorem, Theorem 4.3, shows that the quotient ring $\mathbb{F} := \mathbb{Z}[x]/\langle p, h(x)\rangle$ is isomorphic to the field $(\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x})\rangle \simeq \mathbb{F}_{p^m}$, where m denotes the degree of h. The field is finite, since p is a prime, and since $h(\bar{x})$ is irreducible modulo p.

An algebraic remark to the reader:

Remark. The difference between x and $\bar{x}$ is explained in the following way: A polynomial with $\bar{x}$ means that each coefficient has been reduced modulo p,

$$f(\bar{x}) \in (\mathbb{Z}/p\mathbb{Z})[\bar{x}],$$

while the notation x is simply

$$f(x) \in \mathbb{Z}[x].$$

When $a \equiv b \mod \langle p, x^r - 1\rangle$ is written, it is assumed that

$$\mathbb{Z}[x]/\langle p, x^r - 1\rangle \simeq (\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle \bar{x}^r - 1\rangle,$$

and when $a \equiv b \mod \langle p, h(x)\rangle$ is written, it is assumed that

$$\mathbb{Z}[x]/\langle p, h(x)\rangle \simeq (\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x})\rangle. \tag{4.13}$$

Note that Equation (4.13) corresponds to the isomorphism in Theorem 4.4.

Back to the story towards proving the correctness of the AKS primality test: The non-zero elements of $\mathbb{F}$ form a cyclic group of order $p^m - 1$:

$$\mathbb{F}_q^* = (\mathbb{F}_q \backslash \{0\}, *).$$

It is a multiplicative group, and the elements are defined as

$$\mathbb{F}_q^* = \{1, \alpha, \alpha^2, \ldots, \alpha^{q-2}\} = \langle \alpha \rangle,$$

where α is a primitive element.

Lemma 4.12. *The cyclic group $\mathbb{F}_q^*$ has a cyclic subgroup,*

$$\langle x \rangle = \{1, x, x^2, \ldots, x^{r-1}\}, \tag{4.14}$$

which has the element x as generator, and where x has order r.

Proof. The fact that $h(x)$ is a divisor of $\Phi_r(x)$ and not just another divisor of $x^r - 1$ is now used: In the finite field $(\mathbb{Z}/p\mathbb{Z})[x]/\langle h(x)\rangle$, x is a root of $h(x)$. Any root of $h(x)$ is also a root of $\Phi_r(x)$, and the only roots of $\Phi_r(x)$ are the elements which have multiplicative order r. In this way, the cyclic subgroup of $\mathbb{F}_q^*$ in Equation (4.14) arises. □

By Lagrange's theorem, Theorem 2.4, the order of the cyclic subgroup generated by $\langle x\rangle$ divides the order of the cyclic group generated by $\langle \alpha\rangle$ which means that $r \mid p^m - 1$. It also implies that

$$\langle x\rangle \subseteq \langle \alpha\rangle.$$

Define the multiplicative group

$$H = \{ \prod_{0\leq a\leq A} (x+a)^{e_a} | e_a \in \mathbb{Z}\} \subseteq (\mathbb{Z}/p\mathbb{Z})[x]/\langle x^r - 1\rangle. \tag{4.15}$$

The notation e_a in Equation (4.15) means that different values of a can have different exponents, hence e_a denotes the exponent of that corresponding value of a. Furthermore, let

$$G = \{g \mod h(x) \mid g \in H\} \subseteq \mathbb{F}.$$

Claim 4.13. *All elements of G are non-zero.*

Proof. Since $G \subseteq \mathbb{F}$, then an element of G is zero, if one of the terms $(x+a)$ is zero. Consider a term $(x+a)$ and assume that it is equal to zero,

$$x + a = 0 \in \mathbb{Z}[x]/\langle p, h(x)\rangle. \tag{4.16}$$

By Equation (4.12), this means that

$$(x+a)^n \equiv x^n + a \equiv 0 \mod \langle p, h(x)\rangle. \tag{4.17}$$

Combining Equation (4.16) and (4.17) would then imply

$$x^n \equiv -a \equiv x \mod \langle p, h(x)\rangle. \tag{4.18}$$

It follows from Equation (4.18) that

$$x^n - x = x(x^{n-1} - 1) \equiv 0 \mod \langle p, h(x)\rangle. \tag{4.19}$$

Equation (4.19) implies that the multiplicative order of x which is r must divide $n - 1$:

$$r \mid n - 1 \Rightarrow n \equiv 1 \mod r. \tag{4.20}$$

Equation (4.20) then implies that

$$\mathrm{ord}_r(n) = 1.$$

This contradicts the assumption that $\mathrm{ord}_r(n) > (\log n)^2$ which proves that all elements of G must be non-zero. □

It follows from Claim 4.13 that G is a multiplicative subgroup of $\mathbb{F}^*$.

Now, take an element $g(x) = \prod_{0 \leq a \leq A}(x+a)^{e_a} \in H$ and consider

$$\begin{aligned} g(x)^n &= \prod_a ((x+a)^n)^{e_a} \\ &\equiv \prod_a (x^n + a)^{e_a} \qquad \text{by Equation (4.11)} \\ &= g(x^n) \mod (p, x^r - 1). \end{aligned} \tag{4.21}$$

From this consideration, it is natural to define the following set:

$$S = \{k \in \mathbb{Z}_{>0} \mid g(x^k) \equiv g(x)^k \mod (p, x^r - 1) \text{ for all } g \in H\}.$$

Since $\langle p, x^r - 1\rangle \subseteq \langle p, h(x)\rangle$ as already described, then

$$g(x)^k \equiv g(x^k) \mod (p, h(x)),$$

for each $k \in S$ and any $g(x) \in \mathbb{F}$ which means that an analogy of Child's Binomial Theorem, Theorem (4.2) holds for elements of G for elements of S as exponents. Recall that the Child's Binomial Theorem,

$$(x+y)^n = x^n + y^n, \tag{4.22}$$

for integers x, y and a prime n, is a "wrong" way to rewrite the binomial theorem,

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i},$$

but that Equation (4.22) turns out to be correct when n is prime.

Note the following Lemma:

Lemma 4.14. *It holds that $n, p \in S$.*

Proof. Since p is prime, it follows from the Child's Binomial Theorem that

$$g(x^p) \equiv g(x)^p \mod (p, x^r - 1),$$

and hence $p \in S$. It follows from Equation (4.21) that $n \in S$. □

As mentioned earlier, it will turn out that the group G can not exist. If this can be proved, then it is proved that all the criteria in Theorem 4.9 can not hold if n is composite. In order to make this proof more clear and readable, some Lemmas are first introduced:

Lemma 4.15. *The set S is closed under multiplication. This means that if $a, b \in S$, then $ab \in S$.*

Proof. Take $g(x) \in H$ and $a, b \in S$. By definition,

$$g(x)^b \equiv g(x^b) \mod (p, x^r - 1), \tag{4.23}$$

and similarly for b. Replacing x by x^a in Equation (4.23) gives

$$g(x^a)^b \equiv g((x^a)^b) \mod (p, x^{ar} - 1). \tag{4.24}$$

Clearly, $r \mid ar$, and then Lemma 4.1 implies that

$$x^r - 1 \mid x^{ar} - 1. \tag{4.25}$$

Equation (4.25) implies that Equation (4.24) is also satisfied modulo p and $x^r - 1$:

$$g(x^a)^b \equiv g(x^a)^b \mod (p, x^r - 1). \tag{4.26}$$

Then

$$\begin{aligned} g(x)^{ab} &= (g(x)^a)^b \\ &\equiv g(x^a)^b && \text{since } a \in S \\ &\equiv g((x^a)^b) && \text{by Equation (4.26)} \\ &= g(x^{ab}) \mod (p, x^r - 1). \end{aligned}$$

Hence, $ab \in S$. □

Lemma 4.16. *If $a, b \in S$ and $a \equiv b \mod r$, then $a \equiv b \mod |G|$.*

Proof. The congruence $a \equiv b \mod r$ implies that

$$r | a - b.$$

By Lemma 4.1, $r|a - b$ implies that

$$x^r - 1 | x^{a-b} - 1 \in \mathbb{Z}[x].$$

Clearly, $x^{a-b} - 1 | c \cdot (x^{a-b} - 1)$ for any $c(x) \in \mathbb{Z}[x]$. If c is chosen as x^b, then $x^b(x^{a-b} - 1) = x^a - x^b$. This means that

$$x^{a-b} - 1 | x^a - x^b.$$

By Lemma 4.2,

$$x^a - x^b | g(x^a) - g(x^b),$$

for any $g(x) \in \mathbb{Z}[x]$. Chaining these divisibility results implies that

$$x^r - 1 \mid g(x^a) - g(x^b), \tag{4.27}$$

for any $g(x) \in \mathbb{Z}[x]$. Equation (4.27) implies that

$$g(x^a) \equiv g(x^b) \mod x^r - 1 \Rightarrow g(x^a) \equiv g(x^b) \mod (p, x^r - 1). \tag{4.28}$$

Hence, if $g(x) \in H$, then

$$\begin{aligned} g(x)^a &\equiv g(x^a) && \text{since } a \in S \\ &\equiv g(x^b) && \text{by Equation (4.28)} \\ &\equiv g(x)^b \mod (p, x^r - 1) && \text{since } b \in S. \end{aligned} \tag{4.29}$$

Equation (4.29) implies that

$$g(x)^a \equiv g(x)^b \mod \langle p, h(x) \rangle \tag{4.30}$$

$$\Rightarrow g(x)^a - g(x)^b \equiv 0 \mod \langle p, h(x) \rangle \tag{4.31}$$

$$\Rightarrow g(x)^b(g(x)^{a-b} - 1) \equiv 0 \mod \langle p, h(x) \rangle, \tag{4.32}$$

since $h(x) \mid x^r - 1$. Equation (4.32) then implies that

$$g(x)^{a-b} \equiv 1 \mod \langle p, h(x) \rangle, \tag{4.33}$$

for any $g(x) \in H$. Since any $g(x) \in G$ can be lifted to $g(\bar{x}) \in H$, since $h(x) \mid x^r - 1$, then Equation (4.33) must hold for any $g(x) \in G$ as well. Recall that G is a cyclic subgroup of $\mathbb{F}^*$, and hence there exists a generator $t(x)$ of G such that

$$G = \{1, t, t^2, \dots\}.$$

By Equation (4.32), it is then concluded that

$$t(x)^{a-b} = 1,$$

which implies that

$$|G| \mid a - b \Leftrightarrow a \equiv b \mod |G|.$$

□

Lemma 4.17. *If $a \in S$ and $b \equiv a \mod n^d - 1$, then $b \in S$.*

Proof. By definition, $\mathrm{ord}_r(n) = d$, and what this actually means is that $n^d \equiv 1 \mod r$. This congruence implies that

$$r | n^d - 1. \tag{4.34}$$

By Lemma 4.1, Equation (4.34) implies that

$$x^r - 1 | x^{n^d - 1} - 1 \in \mathbb{Z}[x].$$

Then, $x^r - 1$ clearly divides $c \cdot (x^{n^d-1} - 1)$ for any $c(x) \in \mathbb{Z}[x]$. If c is chosen as x such that $x(x^{n^d-1} - 1) = x^{n^d} - x$, then

$$x^r - 1 | x^{n^d} - x. \tag{4.35}$$

The congruence $b \equiv a \mod n^d - 1$ implies that $n^d - 1 | b - a$. Now, consider

$$x^{n^d} - x = x(x^{n^d-1} - 1)$$
$$x^b - x^a = x^a(x^{b-a} - 1).$$

Clearly, $x | x^a$, and by Lemma 4.1, $n^d - 1 | b - a$ implies that $x^{n^d-1} - 1 | x^{b-a} - 1$. Now, using the fact that if $j | k$ and $l | m$ for $j, k, l, m \in \mathbb{Z}[x]_{>0}$, then $j | km$ and $l | mk$, and hence $jl | km$, it can be seen that

$$x(x^{n^d-1} - 1) | x^a(x^{b-a} - 1) \Rightarrow x^{n^d} - x | x^b - x^a.$$

By Lemma 4.2,

$$x^b - x^a | g(x^b) - g(x^a),$$

for any $g(x) \in \mathbb{Z}[x]$, and hence,

$$g(x^b) \equiv g(x^a) \mod (p, x^r - 1), \tag{4.36}$$

since the divisibility results implies that

$$x^r - 1 \mid g(x^b) - g(x^a).$$

If $g(x) \in H$, then $g(x)^{n^d} \equiv g(x^{n^d}) \mod (p, x^r - 1)$ by Lemma 4.15, since $n \in S$. By Lemma 4.2, $x^{n^d} - x | g(x^{n^d}) - g(x)$, and chaining this with Equation (4.35), it can be seen that $x^r - 1 \mid g(x^{n^d}) - g(x)$. This implies that $g(x^{n^d}) \equiv g(x) \mod x^r - 1$, and hence $g(x^{n^d}) \equiv g(x) \mod (p, x^r - 1)$. This implies that

$$g(x)^{n^d} \equiv g(x^{n^d}) \equiv g(x) \mod (p, x^r - 1). \tag{4.37}$$

Clearly,

$$g(x)^a = g(x)^b (g(x)^{a-b}). \tag{4.38}$$

Since $n^d - 1 \mid a - b$, then there exists some $c \in \mathbb{Z}[x]$ such that $a - b = c(n^d - 1)$ which implies that $g(x)^{a-b} = (g(x)^{n^d-1})^c$. By Equation (4.37), it is known that

$$\begin{aligned} &g(x)^{n^d} \equiv g(x) \mod (p, x^r - 1) \\ \Rightarrow &g(x)^{n^d} - g(x) = g(x)(g(x)^{n^d-1} - 1) \equiv 0 \mod (p, x^r - 1), \end{aligned}$$

and hence

$$g(x)^{n^d-1} \equiv 1 \mod (p, x^r - 1). \tag{4.39}$$

Since $g(x)^{n^d-1} \mid g(x)^{a-b}$, Equation (4.39) implies that

$$g(x)^{a-b} \equiv 1 \mod (p, x^r - 1).$$

Combining this with Equation (4.38), it follows that

$$g(x)^b \equiv g(x)^a \mod (p, x^r - 1). \tag{4.40}$$

Hence,

$$\begin{aligned} g(x^b) &\equiv g(x^a) && \text{by Equation (4.36)} \\ &\equiv g(x)^a && \text{since } a \in S \\ &\equiv g(x)^b && \text{by Equation (4.40)}, \end{aligned}$$

which implies that $b \in S$. □

Lemma 4.18. *The following congruence holds:*

$$n/p \equiv np^{\phi(n^d-1)-1} \mod n^d - 1.$$

Proof. By definition, $p|n$, and therefore also $p|n^d$. This means that

$$\gcd(p, n^d - 1) = 1,$$

and then, by Euler's Theorem, Theorem 2.7,

$$p^{\phi(n^d-1)} \equiv 1 \mod (n^d - 1).$$

Hence,

$$p^{\phi(n^d-1)-1} \equiv \frac{1}{p} \mod (n^d-1),$$

where $\frac{1}{p}$ is the multiplicative inverse of p in the multiplicative group $(\mathbb{Z}/(n^d-1)\mathbb{Z})^*$. Now, multiplying by n on both sides gives

$$np^{\phi(n^d-1)-1} \equiv \frac{n}{p} \mod (n^d-1),$$

which completes the proof. □

Lemma 4.19. *It holds that $n/p \in S$.*

Proof. Let $a = np^{\phi(n^d-1)-1}$ and $b = n/p$. If it can be shown that $b \in S$, then the claim is proved. First, note that $a \in S$ since $n, p \in S$, by Lemma 4.15. By Lemma 4.18, it can be seen that

$$n/p \equiv np^{\phi(n^d-1)-1} \mod n^d - 1 \Rightarrow b \equiv a \mod n^d - 1. \tag{4.41}$$

Finally, combining Equation (4.41) and the fact that $a \in S$, it follows from Lemma 4.17 that

$$b = n/p \in S,$$

which completes the proof. □

Before the next Lemma is stated, the following definition is introduced: Let R be the subgroup of $(\mathbb{Z}/r\mathbb{Z})^*$ generated by n and p,

$$R = \langle n, p \rangle \subseteq (\mathbb{Z}/r\mathbb{Z})^*. \tag{4.42}$$

Lemma 4.20. *Suppose that $f(x), g(x) \in \mathbb{Z}[x]$, with $\deg f, \deg g \leq |R|$, and that $f(x), g(x) \mod (p, h(x))$ both are in G. If $f(x) \not\equiv g(x) \mod p$, then $f(x) \not\equiv g(x) \mod (p, h(x))$.*

Proof. The Lemma is proved by showing the contrapositive, i.e. if $f(x) \equiv g(x) \mod (p, h(x))$, then $f(x) \equiv g(x) \mod p$. As this proof gets a bit complicated, an algebraic substitution used in the proof is first exemplified:

Example 4.21. *Let $p = 3$ and $h(x) = x^2 + x + 2$, an irreducible polynomial modulo p. Let $f(x) = x^2 + 11x + 1$. Now, $f(x)$ is reduced modulo p:*

$$\bar{f}(\bar{x}) = \bar{x}^2 + 2\bar{x} + 1 \in (\mathbb{Z}/p\mathbb{Z})[\bar{x}].$$

Now, either $\bar{f}(\bar{x})$ is reduced modulo $h(\bar{x})$, which gives

$$\hat{f}(\bar{x}) = \bar{x} + 2 \in (\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x}) \rangle,$$

or a new variable y is put into $\bar{f}$ and then considered over the polynomial ring $(\mathbb{Z}/p\mathbb{Z})[y]$ over the big field $\mathbb{F}_{p^m}[y]$:

$$\bar{f}(y) = y^2 + 2y + 1 \in (\mathbb{Z}/p\mathbb{Z})[y] \subseteq \mathbb{F}_{p^m}[y].$$

In this way, the field $(\mathbb{Z}/p\mathbb{Z})$ is now considered as a subset of the bigger field $\mathbb{F}_{p^m}$, where m denotes the degree of h.

Now, the proof of Lemma 4.20: Assume that $f(x) \equiv g(x) \mod (p, h(x))$. Let $\bar{f}(\bar{x}), \bar{g}(\bar{x}) \in (\mathbb{Z}/p\mathbb{Z})[\bar{x}]$ and consider $\bar{f}(y), \bar{g}(y) \in (\mathbb{Z}/p\mathbb{Z})[y] \subseteq \mathbb{F}_{p^m}[y]$ as described in the Example. Furthermore, let

$$\Delta(y) = \bar{f}(y) - \bar{g}(y) \in \mathbb{F}_{p^m}[y].$$

For an element $\bar{x} \in \mathbb{F}_{p^m} = (\mathbb{Z}/p\mathbb{Z})[\bar{x}]/\langle h(\bar{x})\rangle$, if $k \in S$, then

$$\begin{aligned}\Delta(\bar{x}^k) &= \bar{f}(\bar{x}^k) - \bar{g}(\bar{x}^k) \\ &= \bar{f}(\bar{x})^k - \bar{g}(\bar{x})^k \\ &= (\bar{f}(\bar{x}) - \bar{g}(\bar{x}))(\bar{f}(\bar{x})^{k-1} + \bar{f}(\bar{x})^{k-2}\bar{g}(\bar{x}) + \cdots + \bar{f}(\bar{x})\bar{g}(\bar{x})^{k-2} + \bar{g}(\bar{x})^{k-1}) \\ &= 0 \in \mathbb{F}_{p^m},\end{aligned}$$

because of the assumption that $f(x) \equiv g(x) \mod (p, h(x))$. This means that $\Delta(y) \in \mathbb{F}_{p^m}[x]$ has a root $\bar{x}^k$ for any $k \in S$. Recall that R is defined as $R = \langle n, p\rangle \subseteq (\mathbb{Z}/r\mathbb{Z})^*$. Let $k = t + u \cdot r$, $t \in R$, $k \in S$, and some integer u. Then

$$\bar{x}^k = \bar{x}^{t+u\cdot r} = \bar{x}^t + (\bar{x}^r)^u = \bar{x}^t,$$

since $\mathrm{ord}_{(p,h(x))}(\bar{x}) = r$, which means that $\bar{x}^r = 1 \mod (p, h(x))$. Then $\Delta(\bar{x}^k) = 0$ implies that $\Delta(\bar{x}^t) = 0$. This means that each $t \in R$ leads to a different root in $\Delta(y)$, and hence $\Delta(y)$ has $\geq |R|$ distinct roots. By assumption, $\deg\Delta < |R|$, which means that it must be the case that $\Delta(y) = 0$. Hence,

$$\bar{f}(y) - \bar{g}(y) = 0 \in (\mathbb{Z}/p\mathbb{Z})[y] \Rightarrow f(x) \equiv g(x) \mod p,$$

which completes the proof. □

The following Lemma 4.22 shows the opposite of Lemma 4.10: Lemma 4.10 showed that the criteria in Theorem 4.9 holds when n is prime, while Lemma 4.22 shows that the criteria in Theorem 4.9 does not hold when n is composite. Hence, the following Lemma 4.22 is the main part of the proof of Theorem 4.9.

Lemma 4.22. *If n is composite, then all the criteria in Theorem 4.9 for n being a prime can not hold. Hence, if n is composite, then Algorithm 5 returns "composite".*

Proof. Lemma 4.22 is proved by showing that the group G can not exist. This is done by showing an upper and lower bound on the size of G:

Claim 4.23. *The size of G has the following lower bound:*

$$|G| > n^{\sqrt{|R|}} - 1.$$

Proof. Recall that R is defined as $R = \langle n, p\rangle \subseteq (\mathbb{Z}/r\mathbb{Z})^*$. Since there are at most $r - 1$ elements of $(\mathbb{Z}/r\mathbb{Z})^*$, then $|R| < r$. The set of elements generated by n modulo r is a subset of R, $\langle n\rangle \subseteq R$. Since

$$\mathrm{ord}_r(n) = d > (\log n)^2,$$

then

$$d < r \Rightarrow d \leq |R| \Rightarrow (\log n)^2 < |R| \Rightarrow \log n < \sqrt{|R|}.$$

Recall that $A = \sqrt{r}\log n$, defined in Theorem 4.9. It is now claimed that $A < p$, which is shown in the following way: From above, it is known that

$$|R| < r \wedge \log n < \sqrt{|R|} \Rightarrow \log n < \sqrt{r},$$

which means that

$$A = \sqrt{r}\log n < \sqrt{r}\sqrt{r} = r.$$

Step 2 in the AKS primality test, Theorem 4.9, is to check whether n has any prime factor $\leq r$, which means that indeed

$$p > r \Rightarrow p > A.$$

The fact that $A < p$ will be useful later. Now, define

$$B := \sqrt{|R|}\log n.$$

Clearly,

$$B = \sqrt{|R|}\log n < \sqrt{r}\log n = A, \tag{4.43}$$

and

$$B < \sqrt{|R|}\sqrt{|R|} = |R|, \tag{4.44}$$

since $\log n < \sqrt{|R|}$. Consider $f(x) = \prod_{a\in T_f}(x+a)$ and $g(x) = \prod_{b\in T_g}(x+b)$, $f, g \in \mathbb{Z}[x]$, where $T_f, T_g \subseteq \{1, 2, \ldots, B\}$, $T_f \neq T_g$. If it can be shown that

- $f, g \mod (p, h(x))$ both are in the group G,
- $f(x) \not\equiv g(x) \mod p$,
- $\deg f, \deg g < |R|$,

then it can be concluded that $f(x) \not\equiv g(x) \mod (p, h(x))$, by Lemma 4.20. The three steps above are proved in the following way:

- By the definition of G, then $f, g \mod (p, h(x))$ both are in G, if $B < A$. By Equation (4.43), it is the case that $B < A$.
- Let $\bar{f}(x), \bar{g}(x) \in (\mathbb{Z}/p\mathbb{Z})[x]$. Now, the fact that for a field k, then $k[x]$ is always a unique factorization domain, is used: Since $(\mathbb{Z}/p\mathbb{Z})$ is a field, then there is unique factorization in $(\mathbb{Z}/p\mathbb{Z})[x]$. This means that if $\bar{f}(x)$ and $\bar{g}(x)$ have distinct factorizations, then $\bar{f}(x)$ and $\bar{g}(x)$ are distinct. By the definition of $\bar{f}$ and $\bar{g}(x)$,

$$\begin{aligned}\bar{f}(x) &= \prod_{a\in T_f}(x+a) \mod p \\ &= \prod_{a\in T_f}(x+[a]_p) \in \mathbb{F}_p[x],\end{aligned}$$

  and similarly,

$$\bar{g}(x) = \prod_{b\in T_g}(x+[b]_p) \in \mathbb{F}_p[x],$$

  where $[a]_p$ denotes the reduction of a modulo p. The following Example shows a situation where $\bar{f}$ and $\bar{g}$ do not have unique factorization:

Example 4.24. *Let $p = 11$ and $B = 20$. Furthermore, let $T_f = \{1, 2\}$ and $T_g = \{12, 13\}$. Then*

$$\bar{f}(x) = (x+1)(x+2)$$
$$\bar{g}(x) = (x+12)(x+13) = (x+1)(x+2) = \bar{f},$$

which shows that $\bar{f}$ and $\bar{g}$ have the same factorization.

However, it is already known that $B < A < p$, which means that $\bar{f} \neq \bar{g}$, when $T_f \neq T_g$. By definition, it is the case that $T_f \neq T_g$, and therefore it can be concluded that $f(x) \not\equiv g(x) \mod p$.

- By the definition of f and g, then $\deg f, \deg g \leq B$. By Equation (4.44), it is known that $B < |R|$. Hence, $\deg f, \deg g < |R|$.

Since all three conditions are satisfied, it follows from Lemma 4.20 that $f(x) \not\equiv g(x) \mod (p, h(x))$. This shows that the products $\prod_{a \in T}(x + a)$ give distinct elements of G for every proper subset T of $\{0, 1, 2, \ldots, B\}$. The set $\{0, 1, 2, \ldots, B\}$ clearly consists of $B + 1$ distinct elements, and when creating a subset T of this set, there are two possible choices for each element: To include it or not to include it. Only the empty set where none of the elements are included should not be a possible subset T. Hence, it can be concluded that

$$|G| \geq 2^{B+1} - 1 > n^{\sqrt{|R|}} - 1.$$

□

Claim 4.25. *The size of G has the following upper bound:*

$$|G| \leq n^{\sqrt{|R|}} - 1.$$

Proof. Since it is assumed that n is not a perfect power, then n is not equal to p^l for some integer l. This means that all the integers $n^i p^j$ with $i, j \geq 0$ are distinct. If $0 \leq i, j \leq \sqrt{|R|}$, then there are

$$(\sqrt{|R|} + 1)^2 > |R|$$

distinct integers of the form $n^i p^j$. The same holds when considering integers of the form $b^i p^j$ where $b = \frac{n}{p}$, since $\frac{n}{p} \in S$, by Lemma 4.19. When considering these elements in $(\mathbb{Z}/r\mathbb{Z})$, it follows that some of them must be congruent modulo r. Let $b^{i_1} p^{j_1}$ and $b^{i_2} p^{j_2}$ be congruent modulo r:

$$b^{i_1} p^{j_1} \equiv b^{i_2} p^{j_2} \mod r. \tag{4.45}$$

By Lemma 4.15, it can be seen that

$$b^{i_1} p^{j_1}, b^{i_2} p^{j_2} \in S, \tag{4.46}$$

since $\frac{n}{p}, p \in S$. Combining Equation (4.45) and (4.46), it follows from Lemma 4.16 that

$$b^{i_1} p^{j_1} \equiv b^{i_2} p^{j_2} \mod |G|.$$

Hence,

$$\begin{aligned} |G| &\leq |b^{i_1} p^{j_1} - b^{i_2} p^{j_2}| && \text{by Lemma 4.16} \\ &\leq (\frac{n}{p} \cdot p)^{\sqrt{|R|}} - 1 && \text{since } 0 \leq i, j \leq \sqrt{|R|} \\ &= n^{\sqrt{|R|}} - 1, \end{aligned}$$

which proves that $|G|$ has the upper bound which was claimed. □

To sum up: The size of G has the following lower bound

$$|G| > n^{\sqrt{|R|}} - 1,$$

but the size of G also has the following upper bound

$$|G| \leq n^{\sqrt{|R|}} - 1,$$

which is a contradiction. The conclusion is that all the criteria in Theorem 4.9 can not hold if n is composite. Hence, if Algorithm 5 is given a composite integer n as input, then the Algorithm returns "composite". □

4.3.1 Complexity analysis

The complexity analysis of the AKS primality test is a bit more comprehensive than the complexity analysis of the other algorithms. Therefore, some other results are presented, before the running time of the AKS test is stated and proved.

Note that in the following, some of the running times which are presented, including the total running time of the AKS test, is output sensitive which means that the running time depends on the size of r (which is still unknown). Afterwards, the size of r will be discussed.

Theorem 4.26. *It can be checked whether n is a perfect power or not by using p-adic Newton iteration, see Algorithm (?). The running time of this step therefore becomes $\mathcal{O}(\lg n M(\lg n))$.*

Proof. Algorithm (?) is structured in the following way: Given $n, t \in \mathbb{N}$, $n > 1$, it is checked whether there exists $m \in \mathbb{Z}$ such that $m^t = n$. The running time of this is $\mathcal{O}(M(\lg n))$[4, p. 272]. Note that if $t > \lg n$, then $2^t > n$ which means that such an integer m does clearly not exist. Hence, $t \leq \lg n$. For the purpose of the AKS primality test, it should be checked whether there exist any value of t such that there exists a value of m satisfying $m^t = n$. However, since $2 \leq t \leq \lg n$, then the algorithm should only be run $\lg n$ times. Hence, the total running time is $\mathcal{O}(\lg n M(\lg n))$. However, there is one detail about this algorithm and its running time: The algorithm described here only works for odd values of t, but by modifying the algorithm such that it uses a 3-adic Newton iteration, it works for even values of t. The running time of the modified algorithm is also $\mathcal{O}(\lg n M(\lg n))$[4, p. 292], hence the final running time of checking whether n is a perfect power or not is $\mathcal{O}(\lg n M(\lg n))$. □

The following Theorem describes the complexity of finding an appropriate r. Note that this complexity is output sensitive, which means that the complexity actually depends on the size of r. The size of r is discussed in Section 4.3.2:

Theorem 4.27. *The running time of finding an integer r for which $ord_r(n) > (\log n)^2$ is $\mathcal{O}(r \cdot (\log n)^2 \cdot M(\log r))$.*

Proof. The integer r is determined as described in the implementation Section. The outer loop in the Algorithm for determining r runs $r - (\log n)^2 < \mathcal{O}(r)$ times. The inner loop runs at most $(\log n)^2$ times for each iteration of the outer loop. For each iteration of the inner loop, one multiplication modulo q is used. The running time of this operation is $M(\log q) \leq M(\log r)$. Hence, the total running time of finding r is

$$\mathcal{O}(r \cdot (\log n)^2 \cdot M(\log r)).$$

□

Theorem 4.28. *The running time of checking whether the congruence*

$$(x+a)^n \equiv x^n + a \mod (n, x^r - 1), \tag{4.47}$$

for each integer a, $1 \leq a \leq \sqrt{r}\log n$ *is* $\tilde{O}(r^{3/2}(\log n)^3)$.

Proof. Recall the $\tilde{O}$-notation introduced in Section 2.4 which is used when the $\mathcal{O}$-notation inludes too much information. The $\tilde{O}$-notation is less informative, since it swallows log-factors. The $\tilde{O}$-notation will be used to show this running time, since it would get too complicated otherwise:
A way to compute $(x+a)^n \mod (n, x^r-1)$ is to use the square-and-multiply algorithm, Algorithm 1, in the ring $R = \mathbb{Z}[x]/\langle n, x^r - 1\rangle$. By Section 2.5, it is known that the complexity of this is $\mathcal{O}(\log n)$ operations in the ring R. The next question is what the complexity of one operation in R is. This is explained in the following way: An element of R can be represented as

$$f(x) + \langle x^r - 1\rangle,$$

where $f(x) \in (\mathbb{Z}/n\mathbb{Z})[x]$ and has degree at most $r-1$. Take two elements of R, $f(x) + \langle x^r - 1\rangle$ and $g(x) + \langle x^r - 1\rangle$. In order to multiply $f(x)$ and $g(x)$, they are first multiplied in $(\mathbb{Z}/n\mathbb{Z})[x]$, and this result is then reduced modulo $x^r - 1$. The complexity of multiplying $f(x)$ and $g(x)$, both of degree less than r, is $M(r)$ operations in $(\mathbb{Z}/n\mathbb{Z})[x]$, and the complexity of each operation in $(\mathbb{Z}/n\mathbb{Z})$ is $M(\log n)$ bit operations. Hence, the total running time of multiplying $f(x)$ and $g(x)$ in $(\mathbb{Z}/n\mathbb{Z})[x]$ is $\tilde{O}(r\log n)$. The next step is to reduce the result modulo $x^r - 1$. This is done as a division in $(\mathbb{Z}/n\mathbb{Z})[x]$, and the complexity of this step is the same as for a multiplication which means $\tilde{O}(r\log n)$[4]. Hence, the complexity of the reduction modulo $x^r - 1$ is $\tilde{O}(r\log n)$. To sum up: The complexity of computing $(x+a)^n \mod (n, x^r-1)$ is $\tilde{O}(r\log n)$ operations in R, and the complexity of each of these operations is $\tilde{O}(r\log n)$. Hence, the complexity of checking the congruence in Equation (4.47) for one value of a is $\tilde{O}(r\log n^2)$. Since the congruence should be checked for $\sqrt{r}\log n$ values of a, the total complexity of this step becomes

$$\tilde{O}(\sqrt{r}\log n \cdot r(\log n)^2) = \tilde{O}(r\sqrt{r}(\log n)^3) = \tilde{O}(r \cdot r^{1/2}(\log n)^3) = \tilde{O}(r^{3/2}(\log n)^3).$$

□

Theorem 4.29. *The running time of the AKS primality test, Algorithm 5, is* $\tilde{O}(r^{3/2}(\log n)^3)$.

Proof. Step 1 in Algorithm 5 is to determine whether n is a perfect power. According to Theorem 4.26, this can be determined by using p-adic Newton iteration which has running time $\mathcal{O}(\lg n M(\lg n))$. The next step is to find an integer r for which $\mathrm{ord}_r(n) > (\log n)^2$. By Theorem 4.27, this step takes $\mathcal{O}(r(\log n)^2 M(\log r))$. The next step in Algorithm 5 is to determine whether $\gcd(z, n) > 1$ for some $z \leq r$. By Theorem 2.10, the running time of calculating gcd is $\mathcal{O}((\log n)^2)$, and since gcd should be calculated for $\leq r$ integers, the running time of this step is $\mathcal{O}(r(\log n)^2)$. The last step in Algorithm 5 is to determine whether $(x+a)^n \equiv x^n + a \pmod{n, x^r - 1}$ for $a = 1, 2, \ldots, [\sqrt{r}\log n]$. By Theorem 4.28 is

$$\tilde{O}(r^{3/2}(\log n)^3).$$

The conclusion is that the running time of the AKS primality test is $\tilde{O}(r^{3/2}(\log n)^3)$. □

4.3.2 Specifying the size of r

By definition, r is indeed greater than $\text{ord}_r(n) > (\log n)^2$ which means that $r > (\log n)^2$. Assume that $r \in \mathcal{O}((\log n)^2)$, then

$$\tilde{O}(r^{\frac{3}{2}}(\log n)^3) = \tilde{O}((\log n)^{\frac{3}{2}\cdot 3}(\log n)^3) = \tilde{O}((\log n)^{3+3}) = \tilde{O}((\log n)^6),$$

which means that the running time of the AKS test can not be smaller than $\tilde{O}((\log n)^6)$.

The following Lemma is stated without proof:

Lemma 4.30. *[5, p. 20] If $n \geq 6$, then there is a prime $r \in [(\log n)^5, 2(\log n)^5]$ for which $ord_r(n) > (\log n)^2$.*

In continuation of Lemma 4.30, assume that $r \in \mathcal{O}((\log n)^5)$, then

$$\tilde{O}(r^{\frac{3}{2}}(\log n)^3) = \tilde{O}((\log n)^{\frac{3}{2}\cdot 5}(\log n)^3) = \tilde{O}((\log n)^{\frac{15}{2}}(\log n)^3) = \tilde{O}((\log n)^{\frac{15}{2}+3}) = \tilde{O}((\log n)^{10\frac{1}{2}}).$$

The size of r can be discussed much further. However, this part is for now left as future work.

CHAPTER 5

Implementations

The primality tests have been implemented in Python using SageMath. In this chapter, the implementations are shown, and some parts of the implementations are explained. Furthermore, it is explained how the implementations have been tested for correctness.

5.1 Implementation of trial division

First of all, the primality test by trial division, Algorithm **??**, has been implemented. The implementation is shown below:

```
def trial_division(n): #Primality test by trial division. Input: an odd integer n
↪  which is being tested for primality
    a = []
    for f in [1..floor(sqrt(n))]:
        if is_prime(f):
            a.append(f) #Makes a list of all primes <= sqrt(n)
    for prime in a:
        if n % prime == 0: #Checks whether any of the primes divide n. If n has a
        ↪  prime divisor, then n is composite and the test returns "False",
        ↪  otherwise "True" is returned
            return false
    return true
```

Since this primality test is not the one of biggest interest in this project, and since the test turns out to be very slow for big integers, the proving of correctness of this implementation has been kept quite simple:

```
    [trial_division(n) for n in [3..10]]

Out: [True, False, True, False, True, False, False, False]
```

This shows that the test correctly recognizes the integers 3, 5, and 7 as prime numbers and the rest as composite numbers.

5.2 Implementation of the probabilistic tests

The implementations of the Fermat test, Algorithm 3, and the Miller-Rabin test, Algorithm 4, are now presented. As a part of the implementation, a couple of auxiliary functions have been made. First of

all, in both the Fermat test and the Miller-Rabin test, some random numbers $a \in \mathbb{Z}/N\mathbb{Z}\backslash\{0, 1, N-1\}$ should be chosen. Therefore, a small function called `RandomNumber(n)` has been made:

```
import random

def RandomNumber(n):
    a = random.randint(2,n-1) #Generates a random number between 2 and n-1
    R = ZZ.quotient(n) #Defines the ring ZZ/nZZ
    return R(a)
```

Remark. So-called *random number generators* written in computer software are actually not random, since they are generated on a computer where everything is designed to be determined. However, a sequence of ”random” numbers appear to be random when they are used for randomness tests, since the tester of the code would not be able to know how the sequence of random numbers was generated[5, p. 12]. In other situations, the way to generate random numbers as in this project would not be ”random enough”, but it is assumed that the `random.randint()`-function in Python is good enough for this purpose.

The square-and-multiply algorithm, Algorithm 1 is used in both the Fermat test and the Miller Rabin test as well, therefore this algorithm has been implemented as an auxiliary function as well:

```
def convertToBinary(x): #Convertes x into binary representation
    y = bin(x)[2:]
    return [int(k) for k in y]

def SquareAndMultiply(x,n): #Computes x^n
    y = convertToBinary(n) #The convertToBinary function is called
    w = 1
    for dig in y:
        w = w * w
        if dig:
            w = w * x
    return w
```

The code sequence below shows how the implementation has been tested:

```
    # Testing SquareAndMultiply
    n = 13
    R = ZZ.quotient(n) #Defines a quotient ring
    x = R(5)
    SquareAndMultiply(x,n-1) #Testing the SquareAndMultiply function

Out: 1

    # Testing SquareAndMultiply
    n = 15
    R = ZZ.quotient(n) #Defines a quotient ring
    x = R(5)
```

```
    SquareAndMultiply(x,n-1) #Testing the SquareAndMultiply function

Out: 10
```

In the first example above, the code computes $5^{12} \mod 13$. Since 13 is prime, it is correct that the output is 1. In the second example, the code computes $5^{14} \mod 15$. One can check that this is indeed congruent to $10 \mod 15$.

In the Miller-Rabin test, Algorithm 4, $N - 1 = 2^v m$ with $v, m \in \mathbb{N}$, $v \geq 1$, and m odd should be computed. This is done in the following auxiliary function called `v_and_m(n_minus1)`:

```
    def v_and_m(n_minus1): #Computing n-1=2^v*m
    v = 0
    m = n_minus1
    while (m%2 == 0):
        m = m/2
        v = v+1
    return (v,m)
```

This function has been tested in the following way:

```
    # Testing v_and_m
    n = 69
    print("{0}=2^({1})*{2}".format(n-1,step5(n-1)[0],step5(n-1)[1]))

Out: 68=2^(2)*17
```

The output is indeed correct.

The final implementations of the Fermat test and the Miller Rabin test are shown below:

```
    def Fermat_Test(n,rounds): #Input: An odd integer n>=5 which should be tested
    ↪  for primality, and the number of rounds the test should run (i.e. how many
    ↪  random numbers "a" should be picked and tested)
    if (n%2 == 0):   #If an even n is given as input, then n is clearly not prime,
    ↪  and the function returns "False"
        return False

    for i in [1..rounds]:
        a = RandomNumber(n) #Generates a random number in ZZ/nZZ
        b = SquareAndMultiply(a,n-1) #Computes a^(n-1) mod n
        if (b != 1): #If b != 1, then a is a Fermat witness for the compositeness of
        ↪  n. The function returns "false"
            return False
    return True #If none of the a's is a Fermat witness, then the function returns
    ↪  "true"
```

```
    def Miller_Rabin(n,rounds): #Input: An odd integer n>=3 which should be tested
    ↪  for primality, and the number of rounds the test should run (i.e. how many
    ↪  random number "a" should be picked and tested)
    (v,m) = v_and_m(n-1) #Computes v and m such that n-1=2^v*m
    for i in [1..rounds]:
        a = RandomNumber(n) #Generates a random number in ZZ/nZZ
        if (gcd(a,n) > 1):
            return False
        b = SquareAndMultiply(a,m) #Computes a^m mod n
        if (b == 1):
            return True
        c = []
        c.append(b)
        for i in [1..v]:
            b = b^2 % n
            c.append(b)
            i = i+1
        if (b != 1): #This b corresponds to b_v=a^(2^v*m). Hence, if b != 1, then
        ↪  Fermat's little theorem is not satisfied, and the compositeness of n is
        ↪  proved
            return False
        count = 0
        for i in c: #Finds the smallest b_i such that b_(i+1) = 1. If this b_i !=
        ↪  n-1, then compositeness is proved
            if (i == 1):
                j = c[count-1]
                break
            count += 1
        if (j == n-1):
            return True
        else:
            return False
```

The correctness of the implementations have been tested in the following way. The outputs given by the tests are compared to the SageMath-function `is_prime()`. Furthermore, the SageMath-function `random_prime` has been used in order to generate a big random prime:

```
    c = [] #Generating some big numbers
    for n in range(10,150,30):
    c.append(2^n - 1)

    print("The Fermat test: {0}".format([Fermat_Test(n,1000) == is_prime(n) for n in
    ↪  c]))
    print("The Miller Rabin test: {0}".format([Miller_Rabin(n,1000) == is_prime(n)
    ↪  for n in c]))

Out: The Fermat test: [True, True, True, True, True]
```

```
     The Miller Rabin test: [True, True, True, True, True]
```

```
    RandPrime = random_prime(10^13-1,False,10^12) #Generates a big random prime
    print("The random prime is: {0}".format(RandPrime))
    print("The Fermat test replies: {0}".format(Fermat_Test(RandPrime,1000)))
    print("The Miller Rabin test replies: {0}".format(Miller_Rabin(RandPrime,1000)))

Out: The random prime is: 5707935571043
     The Fermat test replies: True
     The Miller Rabin test replies: True
```

5.3 Implementation of the AKS primality test

The implementation of the AKS primality test, Algorithm 5, is now presented. As for the other implementations, some auxiliary functions have been made in order to make the implementation more readable.

The first step in the AKS primality test, Algorithm 5, is to check whether n is a perfect power or not. The implementation of this step has been handled by using p-adic Newton iteration as described in Algorithm 9.22 and Theorem 9.28 in [4]. One important remark to this Algorithm is that it can only check for nth powers where n is an odd number. This problem has been handled by also making a modification of the Algorithm which uses 3-adic Newton iteration - this modification of the Algorithm can check for the nth powers where n is an even number. A function called `Perfect_Power_odd` which checks for odd powers has been implemented as well as a function `Perfect_Power_even` which logically checks for even powers:

```
def p_adic_Newton(phi,p,l,g0, s0):
    phi_der = phi.derivative()
    r = ceil(log(l,2))
    for i in range(1,r):
        g1 = (g0-phi(g0)*s0) % p^(2^i)
        g0 = g1
        s1 = (2*s0-phi_der(g1)*s0^2) % p^(2^i)
        s0 = s1
    return (g0-phi(g0)*s0) % p^l

def Perfect_Power_odd(a,n): #Checks whether n is an odd perfect power
    R = ZZ
    Ry.<y> = R[] #Defines the ring
    phi = y^n - a #Defines the phi-function
    phi_der = phi.derivative() #Calculates the derivative of the phi-function
    g0 = R(1) #Starting guess
    k = ceil(log(a,2)/n)
    s0 = inverse_mod(phi_der(g0),2) #Determines s0 as the inverse of the derivative
    ↪  of the phi-function, evaluated in the starting guess
```

```
    g = p_adic_Newton(phi,2,k,g0,s0) #Calls the p_adic_Newton function
    return g^n == a #Returns "True" if a^n exists with an odd n, else it returns
    ↪ "False"

def Perfect_Power_even(a,n): #Checks whether a^n exists. Note that the inputs to
↪ this function are: An integer a which corresponds to the n which is being
↪ checked for primality in the AKS test (!), and an integer n which is here the
↪ power being checked for
    if (a % 3 == 0): #In case a is congruent to 0 modulo 3, a should be reduced such
    ↪ that it is congruent to 1 or 2 modulo 3
        if (a % 3^(n) != 0):
            return False
        while (a % 3 == 0):
                a = a/(3^(n))

    a = ZZ(a) #Specifies that a is still an integer

    if(a == 1): #If a is 1, then a^n is clearly a perfect power
        return True

    if (a % 3 == 2): #If a is congruent to 2 modulo 3, then a^n does not exist
         return False

    if (a % 3 == 1): #If a is congruent to 1 modulo 3, then a^n possibly exists
        R = ZZ
        Ry.<y> = R[] #Defines the ring

        phi = y^n - a #Defines the phi-function
        phi_der = phi.derivative() #Calculates the derivative of the phi-function
        g00 = R(1) #First starting guess
        g01 = R(2) #Second starting guess
        k = ceil(log(a,3)/n)

        s00 = inverse_mod(phi_der(g00),3) #Determines s0 as the inverse of the
        ↪ derivative of the phi-function, evaluated in the first starting guess
        s01 = inverse_mod(phi_der(g01),3) #Determines s0 as the inverse of the
        ↪ derivative of the phi-function, evaluated in the second starting guess
        g11 = p_adic_Newton(phi,3,k,g00,s00) #Calls the p_adic_Newton function
        g12 = p_adic_Newton(phi,3,k,g01,s01) #Calls the p_adic_Newton function
        return (g11^n == a) or (g12^n == a) #Returns "True" if a^n exists with an
        ↪ even n, else it returns "False"
```

The next auxiliary function is called `Find_r(n)`, and it finds a value of r for which $\mathrm{ord}_r(n) > (\log n)^2$:

```
def Find_r(n): #Finds a value of r such that ord_r(n)>(log(n))^2
    m = ceil(log(n,2)^2)
    q = m #Starting from (log(n))^2, since r>(log(n))^2
```

```
    while True:
        success = True
        for j in range(1,m):
            if (n^j)%q == 1:
                success = False
                break
        if success:
            return q
        q += 1
```

The last auxiliary function which checks whether $(x+a)^n \equiv x^n + a \pmod{n, x^r - 1}$ for each integer a, $1 \leq a \leq \sqrt{r}\log n$ is called `CheckEquations(n)`, and it is written as:

```
def CheckCongruences(n):
    r = Find_r(n) #Finds a value of r such that ord_r(n)>(log(n))^2
    l = floor(sqrt(r)*log(n,2)) #The congruence should be checked for each integer
    ↪  a, 1 <= a <= sqrt(r)*log(n)
    R = ZZ.quotient(ZZ(n))
    Rx.<x> = R[] #Defines the ring
    Rquot = Rx.quotient(x^r-1)
    X = Rquot.gen()
    for a in [1..l]: #It is checked whether the congruence is satisfied or not
        if (X+a)^n != X^n + a:
            return False #Returns "False", if the congruences are not satisfied,
            ↪  else it returns "True"
    return True
```

The final implementation of the AKS primality test is then written as:

```
def AKS(n): #The AKS primality test
    for i in [3..20]:
        if (i % 2 != 0):
            if (Perfect_Power_odd(n,i) == True): #Checks the odd powers
                return False
    for i in [2,4,8,10,14,16,20]: #Unfortunately, the code fails on powers divisible
    ↪  by 6 (which is why the 6th, 12th, and 18th power are not checked)
        if (Perfect_Power_even(n,i) == True): #Checks the even powers
            return False
    r = Find_r(n)
    for a in [1..r]:
        if (1 < gcd(a,n) < n): #Checks whether n has a prime power less than r
            return False
    if (CheckCongruences(n) != True): #Checks whether the congruences are satisfied
    ↪  for each integer a, 1 <= a <= sqrt(r)*log(n)
        return False
    return True #If n passed all tests, then n is prime and the function returns
    ↪  "True"
```

The correctness of the implementation is shown by generating some big integers, using these integers in the AKS function, and then comparing with the Sagemath-function `is_prime`:

```
    c = [] #Generating some big numbers
for n in range(10,150,30):
    c.append(2^n - 1)

    print("The AKS test: {0}".format([AKS(n) == is_prime(n) for n in c]))

Out: The AKS test: [True, True, True, True, True]
```

```
    RandPrime = random_prime(10^13-1,False,10^12) #Generates a big random prime
    print("The random prime is: {0}".format(RandPrime))
    print("The AKS test replies: {0}".format(AKS(RandPrime)))

Out: The random prime is: 3023372979947
     The AKS test replies: True
```

5.4 Comparing the tests

Figure 51 shows a comparison of the four tests which have been implemented. Note that the blue graph for the Fermat test is hidden behind the red Miller-Rabin test which show that both of them are really fast. The two probabilistic tests are given the input `rounds=20`, and the following values of n has been tested in each of the four implementations:

```
    c = [1023,43251,3571385,357138533,35713853347,1099511627775]
```

Even though only 6 different values of n have been tested and plotted in this Figure, the Figure might give a give about the robustness of the tests. It seems very clear that the Fermat Test, the Miller-Rabin test, and the AKS test work much better for big values of n that trial division which was as expected. However, when considering the choices of values of n being tested for, it might be the case that all 6 values are composite numbers. Take for example the AKS test, it is indeed faster for composite numbers than for primes, since it only continues through all steps of the test when n is prime. Take for example $n = 66453917$ which is prime:

```
    timeit('AKS(66453917)',seconds=True)

Out: 13.161669588088989
```

Hence, if this value of n had been chosen as one of the values being tested in the comparison, then the plot would have looked quite different from Figure 51.

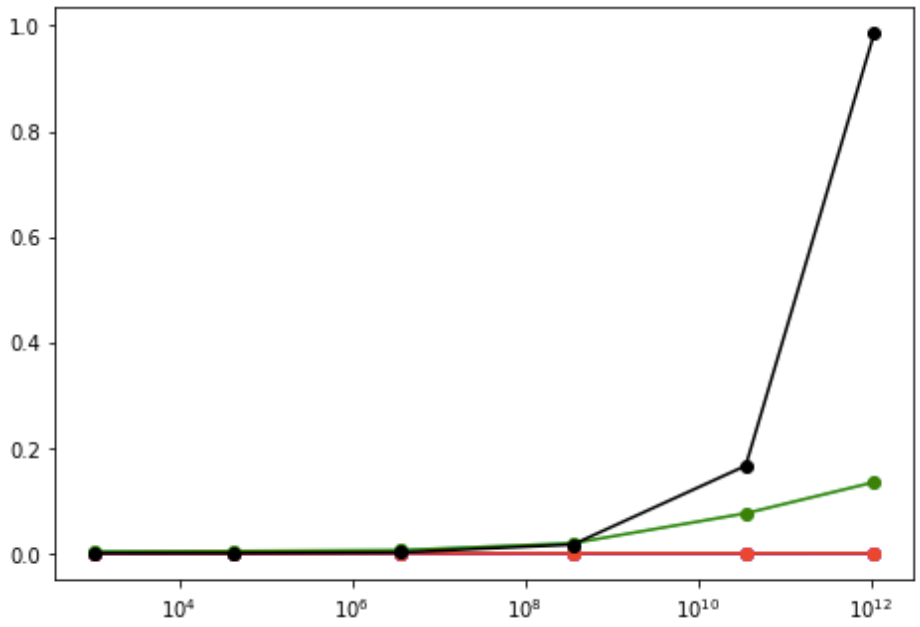

Figure 51: Comparison of trial division, the Fermat test, the Miller Rabin test, and the AKS test. Note that the x-axis is the size of the numbers being tested for primality, and the y-axis is the time in seconds it takes for the tests to determine whether the number is prime or composite. Note that blue is the Fermat test, red is the Miller-Rabin test, green is the AKS test, and black is trial division..

Since the time in seconds it takes for the two probabilistic tests to verify whether n is prime or not seem to grow very slowly, Figure 52 shows a comparison of only these two tests. In this comparison, both tests are given the input `rounds=20` and the following values for n:

```
    # Define scale of numbers, i.e. big N
    c = []
    for n in range(10,200,10):
        c.append(2^n - 1)
    print(c)

Out: [1023, 1048575, 1073741823, 1099511627775, 1125899906842623,
↪ 1152921504606846975, 1180591620717411303423, 1208925819614629174706175,
↪ 1237940039285380274899124223, 1267650600228229401496703205375,
↪ 1298074214633706907132624082305023, 1329227995784915872903807060280344575,
↪ 1361129467683753853853498429727072845823,
↪ 1393796574908163946345982392040522594123775,
↪ 1427247692705959881058285969449495136382746623,
↪ 1461501637330902918203684832716283019655932542975,
↪ 1496577676626844588240573268701473812127674924007423,
↪ 1532495540865888858358347027150309183618739122183602175,
↪ 1569275433846670190958947355801916604025588861116008628223]
```

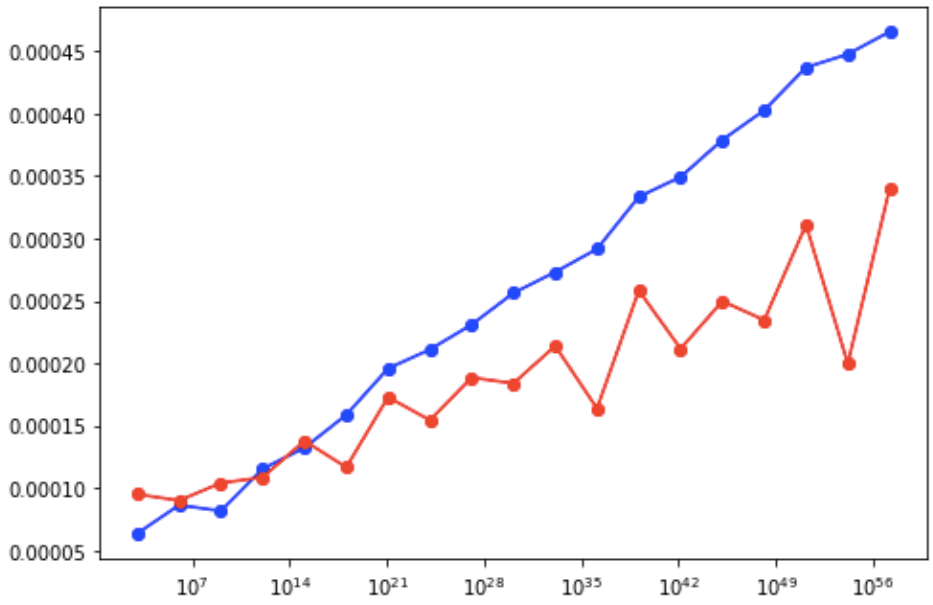

Figure 52: Comparison of the Fermat test and the Miller-Rabin tests as these two tests seemed to behave very similar in Figure 51. As for Figure 51, the x-axis is the size of the number being tested for primality, while the y-axis is the time in seconds it takes for the tests to determine whether the number is prime or composite. Note that blue is the Fermat test and red is the Miller-Rabin test..

In the last comparison with only the two probabilistic tests, each value of n have been tested in the two algorithms 10 times, and the final result is then the mean of the 10 times in seconds. This is another factor indicating that Figure 52 shows a more precise result than Figure 51.

CHAPTER 6

Conclusion

Different primality tests have been considered and compared in this project. First of all, the implementations showed that it is indeed much easier and faster to determine whether a small integer is prime or not compared to a very big integer.

The probabilistic Fermat test which have been considered performed very well in the tests of the implementations. However, from a theoretical point of view this test is not at all sufficient in the sense that it is a probabilistic test. Throughout the theoretical discussion as well as the implementation, it can be concluded that the Fermat test indeed fits in the middle category described in Section 3.1. A disadvantage of the Fermat test is that it does not recognize the so-called Carmichael numbers - numbers for which all values of a satisfying $a^{n-1} \equiv 1 \mod n$ and $\gcd(a, n) = 1$ are Fermat liars - but often incorrectly classifies these as primes.

Contrary to the Fermat test, the probabilistic Miller-Rabin test is able to recognize the Carmichael numbers as composite numbers. As such, this test is a great improvement of the Fermat test. When the implementations were tested, these two tests seemed to behave very equally which follows the analytical complexity analysis. The conclusion from the analysis was that both tests have running time $\mathcal{O}(M(k)k)$.

Even though the AKS primalty test, Theorem 4.9, at first sight looks very simple, the part with proving that the test is actually correct turned out to be very complicated. However, in the end it was proved that the criteria for the number n in Theorem 4.9 can indeed only hold when n is prime. The implementation showed that this test gets slower than the Fermat and the Miller-Rabin test when the size of n is increased - again, as expected when comparing to the analytical complexity analysis. When studying the complexity of the AKS test, first an output sensitive running time was given: $\tilde{O}(r^{3/2}(\log n)^3)$. The logical question following from this running time is what the size of r is which have been discussed a bit in the end of Chapter **??**.

Throughout the project, the focus has been on the theoretical point of view which is why the proofs of the primality tests have been discussed much more into details than the implementations of the primality tests. The implementations have been a proof of concept work, and indeed some improvements of the implementations can be mentioned: One detail is in the `Perfect_Power_even(n)`-function as a part of the implementation of the AKS test. The function can only test for nth powers where n is even and n is not divisible by 6. However, it might be very rarely that the only detail distinguishing an integer n between being prime and being composite is the fact that n is a perfect 6th power of some number.

As already mentioned, when comparing the implementations with the analytical complexity analysis, there seems to be an overall agreement. Clearly, the Fermat and the Miller-Rabin tests were the fastest, but this detail is indeed due to the simplicity of these two probabilistic tests and comes at the cost of uncertainty.

Bibliography

[1] Peter Beelen. *Notes from the course, 01018 Discrete Mathematics 2: Algebra E17*. DTU Compute, 2017.

[2] Cormen et al. *Introduction to Algorithms, third edition*. The MIT Press, 2009.

[3] Larry Finkelstein and William M. Kantor. *Groups and Computation II*. American Mathematical Society, 1995.

[4] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge, 2013.

[5] Andrew Granville. "It is easy to determine whether a given integer is prime". In: *Bulletin of the American Mathematical Society* 42.1 (September 30, 2004), pages 3–38. DOI: https://doi.org/10.1090/S0273-0979-04-01037-7.

[6] Niels Lauritzen. *Concrete Abstract Algebra*. Cambridge, 2003.

$\sqrt{17}$
$\Omega \int \delta e^{i\pi} = -1$
{2.7182818284}
ϑφερτυθιοπσδφγηξκλ
χ^2
$\sum$