# PRIMALITY TESTING


Professor : Mr. Mohammad Amin Shokrollahi
Assistant : Mahdi Cheraghchi

By TAHIRI JOUTI Kamal

# TABLE OF CONTENTS

**I-** FUNDAMENTALS FROM NOMBER THEORY FOR RANDOMIZED ALGORITHMS**:** ……page 4

1) Divisibility : ……...page 4

2) Modular Arithmetic : ……...page 5

3) Properties of prime numbers and Algorithm of The Sieve of Eratosthenes : ……page 6

**II-** RANDOMIZED ALGORITHMS **:** ……...page 10

1) The Miller-Rabin algorithm : ……..page 10

2) The Solovay-Strassen algorithm : ……...page 12

**III-** FUNDAMENTALS FROM POLYNOMIALS AND FIELDS FOR DETERMINISTIC ALGORITHM**:** ……page 14

1) Groups and Subgroups : ……..page 14

2) Rings and Fields :……page 14

3) Polynomials over Rings : ……page 15

4) Division with Remainder and Divisibility for Polynomials : …..page 15

5) Irreducible Polynomials and Factorization : ……page 16

**IV- DETERMINISTIC ALGORITHM :** ……..page 16

1) The basic idea : ……....page 16

2) Agrawal, Kayal and Saxena algorithm : ……....page 16

3) Running time : ……..……..page 17

# INTRODUCTION :

Since the discovery of the utility of the numbers, the human being tried to differentiate them. We decide between them according to whether they are even or odd. Or, according to the fact that they are prime or composite. A natural number n >1 is called a prime number if it has no positive divisors other than 1 and n. Therefore, other numbers that are not prime have other divisors. That is why we call them composite numbers because we can write them : n = p*q with $\{p,q\} \neq \{1,n\}$. The problem has always been to decide whether a number is prime or not. To answer this problem, many algorithms have been created like the Trial Division. It uses the property which says that the biggest divisor of n is smaller or equal to the square root of n. But for numbers that exceed 30 digits , it will take more than 10^13 years to know the answer. So, the interest would be to create an algorithm using mathematical bases which would answer to this question as fast as possible. This is what we will see in this project.

The study of prime numbers became really important to code texts. Cryptography is one of the most important application of prime numbers theory. At the beginning, it was only used to code texts during the wars and more recently it was used for other applications, like the security of an account.

Fist of all, I will focus on randomized algorithms for primality testing. Then, I will focus on a deterministic algorithm that I have implemented.

# I- FUNDAMENTALS FROM NOMBER THEORY FOR RANDOMIZED ALGORITHMS:

## 1) Divisibility :

Definition 1.1: m|n means that m divides n, or m is a divisor of n. We can write it : m*x = n for some x. It rises from this, if n |m and m|k then n|k.

Proof : n|m n*x = m for some x ; m|k m*y = k for some y

n*x*y = k n|k

Definition 1.2 : For any integer a and b ( b<a), we can write uniquely a = b*q + r with r < b. q is the quotient and r the remainder. We write a div b = q and a mod b = r. For example, 9 div 4 = 2 and

9 mod 4 = 1. a div b = [n/d], that is why (-9) div 4 = -3 and (-9) mod 4 = 3.

Another important definition is the one of the greatest common divisor denoted by gcd(n,m). It represents the largest integer that divides n and m. For example, gcd(42,12) = 6. If gcd(n,m) = 1, we say that n and m are relatively prime since 1 is the largest integer that divides n and m. A quite easy way to compute the gcd is the following. For this, we use another interesting property which says that any number can be written as a product of prime numbers. With this factorisation, we easily compute the gcd.

For example : 42 = 2*3*7 , 12 = 2*2*3

The factors in common are 2 and 3. Therefore, gcd(42,12) = 6.

49 = 7*7 , 4 = 2*2

There are no common factors excepting 1 so gcd(49,4) = 1.

Proposition 1 : n and m are relatively prime if and only if there are integers x and y so that

1 = n*x + m*y.

There exists an algorithm that computes the gcd, it is the Euclidean algorithm.

Euclidean algorithm :

Input : Two integers n,m.

METHOD :

a,b : integer ;

if |n| >|m|

```
        then a = |n| , b = |m|
        else b = |m| , a =|n|
while b>0 repeat
        (a,b) = (b, a mod b)
return a ;
```

The variables n and m are placed into the variables a and b such that b is not larger than a. In each loop, the remainder a mod b is computed and placed into b, and b into a. The loop in this algorithm is carried out at most 2*min {||n||, ||m||}. This algorithm uses a mathematic property which says that gcd(n,m) = gcd(n mod m, m) for all n.

Proof : n mod m = n - q*m. Therefore, gcd(n mod m, m) = gcd(n - q*m, m)
= gcd(n, m).
n + m*x = d*u and m = d*v then n = d(u - v*x).

## 2) Modular Arithmetic :

Definition 2.1 : For arbitrary integers a and b we say that a is congruent to b modulo m and write a b mod m if a mod m = b mod m. Therefore, we have a =% b (mod m) if and only if m divides (b - a). a = m*q + r and b = m*q' + r', then b - a = m(q' - q) + (r' - r). But, r'- r = 0 so m divides (b - a).

Definition 2.2 : Let the Zm be the set {0,1,…,m-1}. We define the operations +m and *m as follows: a +m b = (a+b) mod m and a *m b = (a*b) mod m .

Definition 2.3 : Zm* = {a | 1 a < m, gcd(a, m) = 1 } and (m) = |Zm* |
is called Euler's totient function. If n = a*b, for a and b relatively prime, then (n) = (a)*(b).

Application : **The Chinese Remainder Theorem**.
Let study an example before.

| a | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a mod 3 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| a mod 8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 |

| a | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a mod 3 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| a mod 8 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

We have 24 = 3*8 with gcd(3, 8) = 1. We set up a table of the remainders a mod 3 and a mod 8. Let add the pair (1, 4 ) and the pair (1, 6) which correpond to the a equal to 4 and 22. Adding the two pairs, we obtain (2, 2) which corresponds to a equal to 2. We note that (4 + 22) mod 22 = 2 .
The mapping a -> (a mod 3, a mod 8) is a bijection between $Z_{24}$ and $Z_3 * Z_8$.
We deduce the following theorem called the Chinese Remainder theorem :

**Let n = p*q for p, q relatively prime. Then the mapping : $Z_n$ -> $Z_a$*$Z_b$, a -> (a mod p, a mod q) is a bijection. Moreover, if (a) = ($a_1$, $a_2$) and (b) = ($b_1$, $b_2$) then**

**(i) (a +n b) = ($a_1$ +p $b_1$, $a_2$ +q $b_2$) ;**
**(ii) (a *n b) = ($a_1$ *p $b_1$, $a_2$ *q $b_2$) ;**
**(iii) (a ^ m mod n) = ( ($a_1$)^m mod p, ($a_2$)^m mod q ) for m 0.**

## 3) Properties of prime numbers and Algorithm of The Sieve of Eratosthenes :

Fundamental Theorem of Arithmetics :

**Every integer n 1 can be written as a product of prime numbers in exactly one way (if the order of the factors is disregarded).**

Proposition 1:

Let n and m have prime decompositions n = $p_1$*$p_2$*….*$p_r$ and m = $q_1$*$q_2$*…*$q_s$. Then a and b are relatively prime if and only if {$p_1$, $p_2$, …, $p_r$} inter {$q_1$, $q_2$, …, $q_s$} = 0.

Proposition 2 :

If n 1 has the prime decomposition n = ($p_1$^$k_1$)*($p_2$^$k_2$)*…*($p_r$^$k_r$) for distinct prime numbers $p_1$, $p_2$, …, $p_r$ then (n) = Mult (1- 1/$p_i$) (1 i r ).

Algorithm of the sieve of Eratosthenes :

The sieve of Eratosthenes marks each composite number [2, n] with its smallest prime divisor. Each composite number is assigned a non zero value. We create a table with (n-1) elements such that if a number i is composite, we assign m [i] with the smallest prime divisor of n.

ALGORITHM :

INPUT : Integer n 2

```
METHOD :
        m [2...n] : array of integer ;
        for j from 2 to n do m [j] = 0 ;
        j = 2 ;
        while j*j  n do
                if  m [j] = 0
                        then i = j*j ;
                        while i  n do
                                if  m [i] = 0 then m [i] = j ;
                                i = i + j ;
                j = j + 1 ;
        return m [2...n]  ;
```

The number of iterations of the j-loop is bounded by (n) If m [j] turns out to be non zero, then in the i-loop marks all the multiples of j. For each prime p (n) , there are no more than n/p many multiples sp n. So $\sum_{p \le sqrt(n)} (n/p) \le n * \sum_{p \le sqrt(n)} (1/p) \le n * (1 + (\ln n)/2)$. Therefore, the number of marking steps is bounded by O(n* log n).

Example for n = 200 :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | **7** | **11** | **13** | **17** | **19** | **23** | **29** |
| **31** | **37** | **41** | **43** | **47** | 7\|49 | **53** | **59** |
| **61** | **67** | **71** | **73** | 7\|77 | **79** | **83** | **89** |
| 7\|91 | **97** | **101** | **103** | **107** | **109** | **113** | 7\|119 |
| 11\|121 | **127** | **131** | 7\|133 | **137** | **139** | 11\|143 | **149** |
| **151** | **157** | 7\|161 | **163** | **167** | **169** | **173** | **179** |
| **181** | 11\|187 | **191** | **193** | **197** | **199** | | |

Chebychev's Theorem on the density of Prime numbers :

*Definition :* For x > 1, let (x) denote the number of primes p x.

*Prime Number Theorem :* $\lim_{x \to} (x)/(x/\ln x) = 1.$

So, we can estimate that up to 50 digits, there is a percentage of 1 / (50* ln 10 - 1) = 0.88 percent prime numbers.

*Theorem :* for all integers n 2 we have **n/log n -2 (n) 3*n/log n.**

The Fermat Test :

*Theorem :* **If p is a prime number and 1 a < p, then a^(p-1) mod p = 1.**

*Definition 1 :* A number a, 1 a < n, is called an **F-witness** for n if a^(n-1) mod n ≠ 1.

*Definition 2 :* For an odd composite number n we call an element a, 1  a < n, an **F-liar** if  a^(n-1) mod n = 1.

ALGORITHM :

INPUT : Odd integer n  3
METHOD :
    Let a be randomly chosen from {2, … , n}
    if  a^(n-1) mod n ≠ 1
        then return 1 ;
    else return 0 ;

If the algorithm outputs 1, then it has detected an F-witness a for n, hence n is guaranteed to be composite. For n = 91 and a = 27, the algorithm returns 0, therefore 27 is a F-liar. There are 34 numbers in the interval [1; 90] that are F-liars. Therefore, there is a probability of 34/88 = 39 percent to have a wrong answer.

*Theorem :* **If n  3 is an odd composite number such that there is at least one F-witness a $Z^*_n$, then the Fermat test applied to n gives answer 1 with probability more than ½.**

The iterated Fermat test uses the Fermat test explained before in a loop in such way that the algorithm is tested m times. This way, the probability that in all attempts an F-liar a is chosen is smaller than (1/2)^ m.
There exists some exceptions : the **Carmichael** numbers. n is a Carmichael number if a^(n-1) mod n = 1 for all a $Z^*_n$.
Now, I will focus on a really important definition because the Miller-Rabin deterministic algorithm is based on it : the A-witness.

*Definition 3 :* Let n  3 be odd, and write n-1 = u*2^k, u odd, k  1. A number a,  1  a < n, is called an **A-witness** for n if a ^u mod n ≠ 1 and a^ (u*2^i) mod n ≠ n-1 for all i, 0  i < k. If n is composite and a is not an A-witness for n, then a is called an A-liar for n.

*Lemma :* **If a is an A-witness for n, then n is composite.**

Quadratic Residues :

*Definition 1 :* For m  2 and a Z with gcd(a, m) = 1 we say that a is a **quadratic residue** modulo m if a  x^2 mod m for some x Z. If a satisfies gcd(a, m) = 1 and is not a quadratic residue modulo m, it is called a (**quadratic**) **nonresidue**.

*Example :* For m = 13, the quadratic residues modulo 13 of 1, 2, …, 12 are 1, 3, 4, 9, 10, 12. There are 6 residues and 6 nonresidues. This is a property of m being a prime number.

*Lemma (Euler's criterion) :* If p is an odd prime number, then the set of quadratic residues is a subgroup of $Z^*_p$ of size (p-1) / 2. Moreover, for a $Z^*_p$, we have
a ^ ( (p-1)/2 )  = 1 if a is a quadratic residue modulo p
and
a ^( (p-1)/2 ) = -1 if a is a nonresidue modulo p.

*Definition 2 :* For a prime number p  3 and an integer a we let
(a/p) = 1, if a is a quadratic residue modulo p

(a/p) = -1, if a is a nonresidue modulo p
(a/p) = 0, if a is a multiple of p.
**(a/p) is called the Legendre symbol of a and p.**

The Jacobi Symbol :

*Definition 1 :* Let n  3 be an odd integer with prime decomposition $n = p_1 \ldots p_r$. For integers a we let $(a/n) = (a/p_1) \ldots (a/p_r)$. (a/n) is called the **Jacobi symbol** of a and n.

There is an extremely efficient procedure for evaluating the Jacobi Symbol (a/n) for any odd integer n  3 and any integer a. Let us formulate it recursively :

(1) If a is not in the interval {1, …, n-1}, the result is ( (a mod n ) / n ).
(2) If a = 0, the result is 0.
(3) If  a = 1, the result is 1.
(4) If 4 | a, the result is ( (a/4) / n ).
(5) If 2 | a, the result is ( (a/2) / n ) if n mod 8 = {1, 7} and  -( (a/2) /n ) if n mod 8 = {3, 5}.
(6) If  a > 1 and a  1 or n  1 mod 4, the result is ( n mod a / a).
(7) If a  3 and n  3 mod 4, the result is -(n mod a /a ).

*Example :* (773/1373) (6)= (600/773) (4)= (150/773) (5)= -(75/773) (6)= -(23/75)

(7)= (6/23) (5)= (3/23) (7)= -(2/5) (5)= (1/3) (3)= 1.

I have implemented the Jacobi Symbol algorithm for the Solovay-Strassen. The number of iterations of the first while loop is O(log n). Here how the algorithm looks like :

```
ALGORITHM :
INPUT  :  Integer a , odd integer n  3
METHOD :
        b, c, s : integer ;
        b = a mod n;
        c = n ;
        s = 1 ;
        while b  2 repeat
                while 4 | b repeat b = b / 4 ;
                if 2 | b then
                        if c mod 8 = {3, 5} then s = -s ;
                        b = b / 2 ;
                if b = 1 then break ;
```

```
            if b mod 4 = c mod 4 = 3 then s = -s ;
            (b, c) = (c mod b, b) ;
    return s*b ;
```

In fact, this algorithm is just the application of the 7 properties that I have written before. This algorithm is really efficient to compute the Jacobi Symbol and it is really easy.

*Lemma 1 :* If p is an odd prime number, then a^( (p-1) / 2) *(a/p) mod p = 1, for all a {1, …, p-1}.

*Definition 2 :* Let n be an odd composite number. A number a, 1  a < n, is called an **E-witness** for n if  a^( (p-1) / 2) *(a/p) mod p ≠ 1. It is called an **E-liar** otherwise.

*Lemma 2 :* Let n  3 be an odd composite number. Then every E-liar for is also an F-liar for n.

# II- RANDOMIZED ALGORITHMS :

## 1) The Miller-Rabin algorithm :

The Miller-Rabin algorithm is called a randomized algorithm because it is based on a randomly choice of a value to make tests. This algorithm is also based on the definition of the A-witness just explained before. Now, I will explain my MillerRabin code method by method.

The method NumKey( ) takes a value from the keyboard. This value is the number n whose we want to know if it is prime or composite. I use for this an object of type BufferedReader and the methods readLine( ), parseInt ( ). ReadLine( ) returns a String with the value of the number. ParseInt( ) converts the String into an integer. The loop “do while” checks if the number is really odd and asks the user to enter another number if the first one is not odd. Finally, the method NumKey( ) returns the value of n.

The following method is exposantOfTwo( ). It takes a long value as a parameter and returns a long value k in such a way that n-1 = u* 2^k. We use for this a loop “for” which divides (n-1) by 2 until the result is an odd number. In every step, we add one to k. The result is the value of k.

The method findOddInt( ) takes as parameters the long values n and k just computed before and returns a long value u. u = (n-1) / 2^k, and I use an explicit cast to convert a double into a long.

The method randInt( ) takes as parameter the value n and returns a random value between 2 and (n-2). Math.random( ) returns a double randomly chosen between 0 and 1. Multiplying this number by (n-2) gives us a number randomly chosen smaller than (n-2). The “do while” loop chooses another number if the product of the randomly chosen value by (n-2) is smaller than 2.

The method fastExponent( ) takes as parameters three long values. It computes the value of b = (a ^ u) mod n. For this, I use the Fast Modular Exponentiation Algorithm which I will explain in details because it is the most important algorithm in the Miller-Rabin algorithm :

```
INPUT : Integers a,n and m  1
METHOD :
        u, s, c : integer
        u = n;
        c = 1 ;
        s = a mod m;
        while u  1 repeat
                if u is odd then c = (c*s) mod m ;
                s = s*s mod m ;
                u = u div 2 ;
        return c;
```

This algorithm speeds up the calculation by computing a “repeated squaring”, $s_0 = a \bmod n$ and $s_i = (s_{i-1})^2 \bmod m$. If $b_k b_{k-1} \ldots b_1 b_0$ is the binary representation of n,

$n = \sum_{p \le sqrt(n)} 2^i$, then $(a^n) \bmod m = \prod_{p \le sqrt(n)} s_i \bmod m$. Therefore,

we need k multiplications and divisions to calculate (a ^ n) mod m. Let us illustrate that with an example :

How to calculate 2 ^ 297 mod 105.

| i | $s_i$ | $b_i$ | $c_i$ |
|---|---|---|---|
| 0 | 2^1 = 2 | 1 | 2 |
| 1 | 2^2 = 4 | 0 | 2 |
| 2 | 4^2 = 16 | 0 | 2 |
| 3 | 16^2 mod 105 = 46 | 1 | 2*46 mod 105 = 92 |
| 4 | 46^2 mod 105 = 16 | 0 | 92 |
| 5 | 16^2 mod 105 = 46 | 1 | 92*46 mod 105 = 32 |
| 6 | 46^2 mod 105 = 16 | 0 | 32 |
| 7 | 16^2 mod 105 = 46 | 0 | 32 |
| 8 | 46^2 mod 105 = 16 | 1 | 32*16 mod 105 = 92 |

Therefore, 2^297 mod 105 = 92.
**Calculating a^n mod m takes O (log n) multiplications and divisions of numbers from {0, …, m^2 - 1}, and O~( (log n) (log m) ) bit operations.**

The method test( ) takes as parameters three long numbers and returns a Boolean. It uses the definition of the A-witness and the lemma which says that if a is an A-witness for n, then n is composite. So, the method checks if (a ^ u) mod n ≠ 1 and a ^ ( u* (2^i) ) mod n ≠ n - 1 for all i between 0 and k.

But the Miller-Rabin algorithm has some particularities. When, the test yields output "composite", we are sure that n is composite, whereas the contrary is not always true. This means that if the test yields output "prime", we are not sure at 100 percent that the number is prime. This is due to the fact that the randomly chosen number can be an **A-liar** for n. But we know that that the algorithm gives the erroneous output "prime" with a probability smaller than 1/4. *So, I have decided to execute five times the method test( ), if in the first time, this method returns true. If among the five executions, one returns false, the number is composite. If all the executions return true, the number is prime with a possibility of error smaller than 1/1024.*

Finally, I will finish concerning the Miller-Rabin algorithm, explaining the code MillerRabin2 that I have implemented. The only difference with the first code MillerRabin is the number of decimal digits of the number n. For the first algorithm, n can have at most 10 digits. But in the second, the number can exceed much more than 10 digits. Instead of working with long numbers, I worked with BigInteger, a class already defined in the API Java. The known operators like +, -, *, /, % can not be used for his class. Instead, I have just used the methods add( ), subtract( ), multiply( ),divide( ) and mod( ) which are implemented in the API Java. For the variables, I have created objects of type BigInteger. The second implementation of the Miller-Rabin algorithm is more interesting than the first one because the user is not limited with the number of decimal digits of n.

In conclusion for this algorithm, I will focus on the fact that my code is really efficient and the probability that it could be erroneous is smaller than 1/1024 if it returns that n is prime. **The algorithm uses O(5 * log n) arithmetic operations and O( 5 * (log n)^3 ) bit operations**, 5 being the number of iterations to be (almost) sure of the result.

## 2) The Solovay-Strassen algorithm :

The Solovay-Strassen algorithm is also a randomized algorithm based on the randomly choice of a value. It is based on the Jacobi Symbol and the definition of the E-liar. Now, I will explain my SolovayStrassen algorithm method by method.

The method NumKey( ) takes a value from the keyboard. It is exactly the same as for the MillerRabin code. An object BufferedReader is created and the methods readLine( ) and parseInt( ) are called. Finally, the "do while" loop checks if the number is really odd.

The method randInt( ) is exactly the same as the one in the MillerRabin code. It returns a long

number randomly chosen between 2 and (n-2). The explicit cast converts the double into a long.

To compute the value of a^( (n-1) / 2)*(a / n) mod n, I first compute the value of a^( (n-1) / 2) mod n by the Fast Modular Exponentiation Algorithm just explained before with the MillerRabin code. Here, I have just created this method taking the 2 parameters a and n instead of 3.

The following method is the one which computes the Jacobi. It takes as parameters long values a and n. It returns the Jacobi of (a / n) which is equal to -1,0, 1.

The method test( ) takes as a parameter a long value. It returns a Boolean : true if the parameter is different from 1 and false otherwise.

The method main is really easy to understand. The variable res is equal to the product of the Jacobi of a and n and a^( (n-1) / 2) mod n. We know that the variable res is between -n and n, because the Jacobi is equal to -1, 0, 1. Therefore, I wanted to compute res % n but I have found a problem. The problem is that in Java we have (-2) mod 3 = (-2) instead of 1. So, I had to add n to res if res is negative. Finally, if the method test( ) is true, it writes that n is composite.

Like the Miller-Rabin algorithm, the Solovay-Strassen has some particularities. If n is a prime number, the output is 0, if n is composite, the probability that output 0 is given is smaller than ½. This is due to the fact that the randomly chosen value can be an E-liar for n. For example, for n = 49 and a = 18, ( ( 18^24)*(18 / 49) ) mod 49 = 1 so the output is "prime" whether 49 = 7*7. *So, I have decided to execute eight times the method test( ), if in the first time, this method returns false. If among the five executions, one returns true, the number is composite. If all the executions return false, the number is prime with a possibility of error smaller than 1/256.*

Finally, I will finish concerning the Solovan-Strassey algorithm, explaining the code SolovanStrassey2 that I have implemented. The only difference with the first code Solovan-Strassey is the number of decimal digits of the number n. For the first algorithm, n can have at most 10 digits. But in the second, the number can exceed much more than 10 digits. Instead of working with long numbers, I worked with BigInteger, a class already defined in the API Java. The known operators like +, -, *, /, % can not be used for his class. Instead, I have just used the methods add( ), subtract( ), multiply( ),divide( ) and mod( ) which are implemented in the API Java. For the variables, I have created objects of type BigInteger. The second implementation of the Solovan-Strassey algorithm is more interesting than the first one because the user is not limited with the number of decimal digits of n.

In conclusion for this algorithm, I will focus on the fact that my code is really efficient and the probability that it could be erroneous is smaller than 1/256 if it returns that n is prime. **The algorithm uses O(8 * log n) arithmetic operations and O( 8 * (log n)^3 ) bit operations**, 8 being the number of iterations to be (almost) sure of the result.

Let's finish with a little compareason between my MillerRabin code and the SolovayStrassen one. The number of arithmetic operations is bigger for the SolovayStrassen than for the MillerRabin one, O(8*log n) comparing to O(5*log n). Moreover, the probability of error is bigger for the SolovanStrassey code : 1/256 comparing to 1/1024. The SolovanStrassey needs more mathematics bases like the definition of the Jacobi Symbol and quadratic residues. Therefore, I think the MillerRabin is more interesting.

# III- FUNDAMENTALS FROM POLYNOMIALS AND FIELDS FOR DETERMINISTIC ALGORITHM:

## 1) Groups and Subgroups :

*Definition 1 :* A group is a set G together with a binary operation ° on G with the following properties :

(i) **(Associativity)** (a ° b) ° c = a ° (b ° c), for all a, b, c G.

(ii) **(Neutral element)** There is an element e G that satisfies a ° e = e ° a = a for each a G.

(iii) **(Inverse element)** For each a G there is some b G such that a ° b = b ° a = e.

We write (G,°,e) for a group with these components.

*Definition 2 :* We say a group (G,°,e) is commutative or **abelian** if a ° b = b ° a for all a,b G.

*Lemma 1 :* If (G,°,e) is a finite group, and H is a subset of G with

(i) e H, and

(ii) H s closed under the group operation °,

then H is a subgroup pf G.

*Proposition 1 :* If H is a subgroup of the finite group G, then |H| divides |G|.

*Proposition 2 :* Let (G, °, e) be a group. For a G define

< a > = { a^i | i Z } = {e, a, a^(-1), a^2, a^(-2), … }

Then < a > is a (commutative) subgroup of G and it contains a. In fact, it is the smallest subgroup of G with this property. It is called the **subgroup generated by** a.

*Definition 3 :* We say a group (G, °, e) is **cyclic** if there is an a G such that G = < a >. An element a G with this property is called a **generating element** of G. The order $\mathrm{ord}_G$ (a) of an element a is defined as |< a >| , if < a > is finite.

## 2) Rings and Fields :

*Definition 1:* A **monoid** is a set M together with a binary operation ° on M with the following properties :

(i) **(Associativity)** (a ° b) ° c = a ° (b ° c), for all a, b, c M.

(ii) **(Neutral element)** There is an element e M that satisfies a°e = e°a = a for each m M.

A monoid is commutative if a ° b = b ° a

*Definition 2 :* A **ring** is a set R together with 2 binary operations and on R and 2 distinct elements 0 and 1 of R with these properties :

(i) (R, , 0) is an abelian group

(ii) (R, , 1) is a monoid

(iii) **(Distributive Law)** For all a, b, c R : (a b) c = (a c) (b c).

We write (R, , , 0, 1) for such a ring.

*Example :* For m  2 the structure $Z_m = \{0, 1, \ldots, m-1\}$ with the binary operations
a  b = (a+b) mod m and a  b = (a*b) mod m
for which the numbers 0 and 1 are neutral numbers, is a ring.

*Definition 3 :* A field is a set of F together with 2 binary operations  and  on F and 2 distinct elements 0 and 1 of F with the following properties :
(i) (F, , , 0, 1) is a ring ;
(ii) (F - {0}, , 1) is an abelian group, denoted by F*.
We write (F, , , 0, 1) for such a field.

*Proposition 1:* for m  2, the ring $Z_m = \{0, 1, \ldots, m-1\}$ is a field  m is a prime number.

## 3) Polynomials over Rings :

The structure ( $Z_m$, $+_m$, $*_m$, 0, 1) is a finite ring for each m  2, and it is a field if and only if m is a prime number. Let (R, +, *, 0, 1) be a ring, the elements of R[X] are called all the polynomials over R.

*Definition 1 :* Let (R, +, *, 0, 1) be a ring. For $f = (a_0, a_1, \ldots)$ and $g = (b_0, b_1, \ldots)$ R[X] let
(a)  $f + g = (a_0 + b_0, a_1+b_1, \ldots)$ and
(b) $f * g = (c_0, c_1, \ldots)$, where $c_i = (a_0*b_i) + (a_1*b_{i-1}) + \ldots + (a_i*b_0)$ for i = 0, 1, …

*Definition 2 :* For $f = (a_0, a_1, \ldots)$ R[X] we let $\deg(f) = \max\{ i \mid a_i \neq 0 \}$

*Proposition 1 :* For f,g R[X] we have $\deg(f + g)$  max $\{ \deg(f), \deg(g) \}$ and $\deg(f * g)$  $\deg(f) + \deg(g)$.

*Proposition 2 :* Let p be a prime number. Then,
$(f + g)^p = f^p + g^p$ and  $(f * g)^p = f^p * g^p$

## 4) Division with Remainder and Divisibility for Polynomials :

*Proposition 1 :* Let R be a ring, and let h R[X] be a monic polynomial (its leading coefficient is 1). Then for each f R[X] there are unique polynomials q, r R[X] with f = h * q + r and deg (r) < deg (h).

*Definition 1 :* For f, h R[X], we say that **h divides f** if  f = h * q for some q R[X]. If  0 < deg(h) < deg (f), then h is called a **proper divisor** of  f.

*Definition 2 :* Let h R[X] be a polynomial whose leading coefficient is a unit. For f, g R[X] we say that f and g are **congruent modulo** h, in symbols f  g mod h, if (f - g) is divisible by h.

### 5) Irreducible Polynomials and Factorization :

*Definition 1 :* A polynomial f F[X] - {0} is called **irreducible** if f does not have a proper divisor.

**Theorem (Unique Factorization for Polynomials) :** Let F be a field. Then every nonzero polynomial f F[X] can be written as a product $a.h_1 \ldots h_s$, s 0, where a F* and $h_1, \ldots, h_s$ are monic irreducible polynomials in F[X] of degree > 0. This product representation is unique up to the order of the factors.

## IV- DETERMINISTIC ALGORITHM :

### 1) The basic idea :

*Lemma :* Let n 2 be arbitrary, and let a < n be an integer that is relatively prime to n. Then,

n is a prime number in Zn[X] we have (X + a) ^ n = X ^ n +a

The lemma suggests a simple method to test whether an odd number n is prime. We choose a < n that is relatively prime to n. By fast exponentiation in the ring $Z_n[X]$, we compute the coefficients of the polynomial (X+a) ^ n in $Z_n[X]$. If the result is X ^ n + a, then n is a prime number, otherwise it is not.

This algorithm is the Agrawal, Kayal and Saxena algorithm which I will explain method by method and class by class.

### 2) Agrawal, Kayal and Saxena algorithm :

The first method is the method test( ) which returns a Boolean and takes as parameters the number n. The first if checks if the number n is a perfect power . If it is the case, the method isPerfectPower returns true and n is composite. The “while” loop checks with n is dividible by some values r from 2 until a value r which is prime and n^i mod r ≠ 1 for all i between one and 4*Math.ceil( log n )*Math.ceil( log n ). Finally, a “for” loop checks if in $Z_n[X]$, for a value a between 1 and 2*sort( r )*log n, (X+a) ^ n mod (X ^ r -1) is different from X ^ (n mod r) +a. In this case, the algorithm returns composite, otherwise it is prime.

The folowing method is NumKey( ) which is exactly the same as for the MillerRabin algorithn and the SolovayStrassen one.

The method isPerfectPower( ) works like this. The median m = (n+1)/2 is calculated and so is the power m ^ b. However, as soon as numbers larger than n appear in the calculation, we break off and report the answer (n+1). In this way, numbers larger than n never appear as factors in a multiplication. If and when m = (a+c) / 2 satisfies m ^ b = n, the algorithm stops and reports success. Otherwise, either a or c is updated to the new value m, so that the invariant is maintained.

Testing whether n is a perfect power is not more expensive than O( ( (log n)^2 )*log log n).

The method isPrime( ) takes a parameters the numbers r and returns if r is prime or not. I have used for this the sieve of Eratosthenes just explaines in the first part of my report.

The method exponent( ) checks if n ^ i mod r is different from 1for all I between 1 and 4*log n*log n. This method uses the method fastExponent( ) which calculates n ^ i mod r by the Fast Modular Exponentiation explained before for the MillerRabin code and the SolovayStrassen one.

Finally , I have done a Polynomial class such that I can create objects Polynomials, add them and multiply them using the Fast Modular Exponentiation.

### 3) Running time :

**For perfect power test :** The algorithm needs no more than O( ( ( log n )^2 ) *log log n ) **For testing r :** p(n) is the maximal r for which the loop is executed on input n. p(n) = O( (log n)^c ). The test requires one division for each r so the cost is O(p(n) ).

**For the sieve of Eratosthenes :** total number of arithmetic operations for maintaining and updating this table is O( p(n) *log p(n) ).

**For the Polynomial Operations :** Number of arithmetic operations bounded by O( p(n) ^(5/2) *(log n)^2 ).

**Total time :** The algorithm carries out O( (log n) ^14.5 ) arithmetic operations on numbers smaller than n^2.

As a conclusion, I just want to add that this deterministic algorithm is efficient at 100 %.

## CONCLUSION :

As a conclusion, I want to say that I have found this project very interesting and I regret that I did not have more time to work on other algorithms. I have learned so many things about mathematics, programming : how deterministic and randomized algorithms work. I hope you will enjoy it and I would like to thank Mr. SHOKROLLAHI for giving me this project and Mahdi CHERAGHCHI for helping me and supporting during this semester.