Université Paris 5 Descartes
UFR Mathématiques et Informatique
Licence 1, 2$^e$ semestre 2017-2018


Rapport de Projet – Groupe Étoile

# Cryptage RSA

*Auteurs :*
Damien LEGROS
Danil BERRAH

*Encadrant :*
Nasser Tebbache

Soutenu le 15 mai 2018

# Table des matières

## Première partie

# Définition du Chiffrement RSA

# Chapitre 1

# Fonctionnement général

## 1.1 Principe de base

**Le chiffrement RSA est asymétrique : il utilise une paire de clés composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer des données confidentielles.**

**Les deux clés sont créées par une même personne, nommée par convention Alice. Alice souhaite que les données qui lui sont transmises soient confidentielles. Alice rend la clé publique accessible. Cette clé est utilisée par ses correspondants pour chiffrer les données qui lui sont envoyées. La clé privée, quant à elle, est réservée à Alice, et lui permet de déchiffrer ces données.**

**Tous les calculs se font modulo un nombre entier $n$ qui est le produit de deux nombres premiers. Le petit théorème de Fermat joue un rôle important dans la conception du chiffrement. Les messages clairs et chiffrés sont des entiers inférieurs à l'entier $n$ (tout message peut être codé par un entier). Les opérations de chiffrement et de déchiffrement consistent à élever le message à une certaine puissance modulo $n$ : c'est l'opération d'exponentiation modulaire.**

**Une condition indispensable est qu'il soit "calculatoirement impossible" de déchiffrer le message à l'aide de la seule clé publique, ou de reconstituer la clé privée à partir de cette clé publique. Autrement dit, les moyens de calcul disponibles et les méthodes connues au moment de l'échange ne doivent pas permettre de déchiffrer le message ou de retrouver la clé privée pendant le temps de secret souhaité.**

## 1.2 Exemple d'utilisation

**Le chiffrement RSA est souvent utilisé pour communiquer une clé de chiffrement symétrique, qui permet alors de poursuivre l'échange de façon confidentielle : Bob envoie à Alice une clé de chiffrement symétrique qui peut ensuite être utilisée par Alice et Bob pour échanger des données.**

**La clé privée peut aussi être utilisée par Alice pour signer une donnée qu'elle envoie, la clé publique permettant à n'importe lequel de ses correspondants de vérifier la signature.**

**Le chiffrement RSA est d'ailleurs utilisé par les banques pour chiffrer les transactions.**

# Chapitre 2

# Création des clés

## 2.1 Les étapes de la création des clés

**L'Étape de création des clés n'intervient pas à chaque chiffrement car les clés de chiffrement peuvent être réutilisées. Leur renouvellement n'est nécessaire que si la clé privée est compromise ou au bout de quelques années par souci de sécurité.**

**On choisit tout d'abord deux nombres premiers $p$ et $q$, plus ils sont grands plus le chiffrement est difficile à craquer.**

**On calcule ensuite leur produit $n = pq$ , on l'appel module de chiffrement.**

**On calcule l'indicatrice d'Euler en $n$ :**

$$\Phi(n) = (p-1)(q-1)$$

**Il faut ensuite choisir l'exposant de chiffrement : un entier naturel $e$ premier avec $\Phi(n)$ et strictement inférieur à $\Phi(n)$.**

**L'exposant de déchiffrement $d$ est l'inverse de $e$ modulo $\Phi(n)$, il peut se calculer efficacement par l'algorithme d'Euclide étendu.**

**Comme $e$ est premier avec $\Phi(n)$, d'après le théorème de Bachet-Bézout il existe deux entiers $d$ et $k$ tels que $ed + k\Phi(n) = 1$. On a donc $ed \equiv 1(\Phi(n))$**

**Le couple ($n$,$e$) est la clé publique qui sert au chiffrement, le couple ($n$,$d$) est quant à lui la clé privée qui sert au déchiffrement.**

**Algorithme de generation des clés en C**

```
void rsa() {
    int p, q, pgcd1 = 0, e = 0, space = 0;
    printf("Entrez un nombre premier p : ");
    scanf("%d",&p);
    printf("Entrez un nombre premier q : ");
    scanf("%d",&q);
    int n = p*q;
    printf("\nn = %d", n);
    int phiden = (p-1)*(q-1);
    printf("\n\nphi de n = %d", phiden);
    while(pgcd1 != 1) {
        if((p<e)&&(q<e)&&(e<phiden)) {
            pgcd1 = pgcd(e,phiden);
            if(pgcd1 != 1) {
```

```
                e++;
            }
        }
        else {
            e++;
        }
    }
    printf("\n\nCle publique (%d,%d)", e, n);
    int d = aee(e,phiden);
    printf("\n\nCle privee (%d,%d)", (d+phiden)%phiden, n);
}
```

## 2.2 Outils Arithmétiques

### 2.2.1 Indicatrice d'Euler

**L'indicatrice d'Euler est la fonction $\Phi$ qui s'obtient à partir de la décomposition en facteurs premiers de $n$ :**

**Si**

$$n = \prod_{i=1}^{r} p_i^{k_i}$$

**Alors**

$$\Phi(n) = \prod_{i=1}^{r} (p_i - 1) p_i^{k_i - 1} = n \prod_{i=1}^{r} (1 - \frac{1}{p_i})$$

**où chaque $p_i$ désigne un nombre premier et $k_i$ un entier strictement positif.**

### 2.2.2 Théorème de Bachet-Bézout

**Soient $a$ et $b$ deux entiers relatifs. Si $d$ est le PGCD de $a$ et $b$, alors il existe deux entiers relatifs $x$ et $y$ tels que $ax + by = d$.**

**Deux entiers relatifs $a$ et $b$ sont premiers entre eux si et seulement s'il existe deux entiers relatifs $x$ et $y$ tels que $ax + by = 1$.**

### 2.2.3 Algorithme d'Euclide étendu

**L'algorithme d'Euclide étendu est une variante de l'algorithme d'Euclide qui permet, à partir de deux entiers a et b, de calculer leur PGCD et un de leurs couples de coefficients de Bézout (deux entiers $u$ et $v$ tels que $au + bv = \mathbf{PGCD}(a, b)$).**

**L'algorithme d'Euclide étendu fournit également une méthode efficace non seulement pour déterminer quand une équation diophantienne $ax + by = c$ possède une solution mais également pour en calculer dans ce cas une solution particulière, dont on déduit facilement la solution générale.**

**Comme l'algorithme d'Euclide, l'algorithme étendu se généralise aux anneaux euclidiens, tels celui des polynômes à une variable sur un corps commutatif. De même que pour les entiers, il permet alors de calculer l'inverse d'un polynôme modulo 1, polynôme avec lequel il est premier et donc des calculs d'inverse dans les anneaux ou corps construits par quotient sur l'anneau des polynômes.**

**Algorithme d'Euclide étendu en C**

```
int aee(int a, int b) {
    int r1=a, r2=b, u1=1, u2=0, v1=0, v2=1, q, rs ,us, vs;
```

```
    while(r2!=0) {
        q=r1/r2;
        rs=r1;
        us=u1;
        vs=v1;
        r1=r2;
        u1=u2;
        v1=v2;
        r2=rs-(q*r2);
        u2=us-(q*u2);
        v2=vs-(q*v2);
    }
    return u1;
}
```

**Algorithme du PGCD en C**

```
int pgcd(int a, int b) {
    while(a!=b) {
        if(a>b) {
            a=a-b;
        }
        else {
            b=b-a;
        }
    }
    return a;
}
```

## 2.3 Chiffrement et déchiffrement du message

### 2.3.1 Méthode de chiffrement

**Si $M$ est un entier naturel strictement inférieur à $n$ représentant un message, alors le message chiffré sera représenté par**

$$C \equiv M^e (\textbf{mod } n)$$

**Algorithme de chiffrement en C**

```
void chiffre() {
    int e, n, i, y, ascii, crypt;
    char msg[100];
    printf("Entrez votre cle publique (e, n)\n");
    printf("e = ");
    scanf("%d", &e);
    printf("n = ");
    scanf("%d", &n);
    printf("\nEntrez un message a chiffrer\n");
    printf("message a chiffrer = ");
    scanf("%s", msg);
    int taille = strlen(msg);
    printf("\nmot chiffre = |");
    for(i=0;i<taille;i++) {
        ascii =(int) msg[i];
        crypt = 1;
        for(y=0;y<e;y++) {
            crypt = (crypt*ascii)%n;
```

```
        }
        printf("%d|", crypt);
    }
}
```

### 2.3.2 Méthode de déchiffrement

**Pour déchiffrer $C$, on utilise $d$, l'inverse de $e$ modulo $(p-1)(q-1)$, et l'on retrouve le message clair $M$ par**

$$M \equiv C^d (\mathbf{mod}\ n)$$

**Algorithme de déchiffrement en C**

```
void dechiffre() {
    int d, n, y, ascii, taille, cpt, crypt;
    int msg;
    printf("Entrez votre cle privee (d, n)\n");
    printf("d = ");
    scanf("%d", &d);
    printf("n = ");
    scanf("%d", &n);
    printf("Entre la taille du message a dechiffre (nombre de lettres) ");
    scanf("%d", &taille);
    printf("\nEntrez un message a dechiffrer (lettre chiffre par lettre chiffre)\n");
    for(cpt=0;cpt<taille;cpt++) {
        printf("lettre a dechiffrer = ");
        scanf("%d", &msg);
        ascii = msg;
        crypt = 1;
        for(y=0;y<d;y++) {
            crypt = (crypt*ascii)%n;
        }
        char decrypt =(char) crypt;
        printf("lettre dechiffre = %c\n", decrypt);
    }
}
```

# Deuxième partie

# Nombres Premiers Particuliers

# Chapitre 3

# Nombres Premiers de Pythagore

## 3.1 Forme des nombres premiers pythagoricien

**Un nombre premier de Pythagore est un nombre premier $p$ qui est l'hypothénuse d'un triangle rectangle ($p^2 = a^2 + b^2 \ (a, b \in \mathbb{N}^*)$) à côtés entiers. Ils sont donc de la forme :**

$$p = x^2 + y^2 \ (x, y \in \mathbb{N})$$

**Ces nombres premiers sont donc des nombres premiers impaires sommes de deux carrés, or d'après le théorème des deux carrés de Fermat les nombres premiers de Pythagore peuvent s'écrire de la forme :**

$$p = 4k + 1 \ (k \in \mathbb{N})$$

**Les dix premiers nombres premiers de Pythagore sont donc : 5, 13, 17, 29, 37, 41, 53, 61, 73 et 89**

## 3.2 Démonstration

### 3.2.1 Théorème des deux carrés de Fermat

**Un nombre premier impair $p$ est somme de deux carrés parfaits si et seulement si $p$ est un nombre premier de Pythagore, c'est-à-dire congru à 1 modulo 4 :**

$$(\exists (x, y) \in \mathbb{N}^2 \ p = x^2 + y^2) \iff p \equiv 1 \ (\textbf{mod } 4)$$

**Cette décomposition est unique et il y a une infinité de nombre premiers impairs de cette forme.**

### 3.2.2 Adaptation du théorème d'Euclide aux nombres premiers

**Soient $p$ un nombre premier et $q = (2^2 \times 3 \times 5 \times 7 \times ... \times p) - 1$. Alors, tous les facteurs premiers de $q$ sont strictement supérieurs à $p$, et puisque $q \equiv -1$ mod $4$ au moins l'un d'eux n'est pas congru à 1 mod 4.**

**Il y a donc une infinité de nombres premiers congru à $-1$ modulo 4.**

**Soient $p$ un nombre premier et $q = (3^2 \times 5^2 \times 7^2 \times ... \times p^2) + 2^2$. Alors, $q \equiv 5$ mod $8$ et $q$ est une somme de deux carrés premiers entre eux, donc tous ses facteurs premiers sont de la forme $4k+1$, c'est-à-dire congrus à 1 ou 5 modulo 8, et au moins l'un d'eux n'est pas congru à 1 modulo 8. De plus, ce facteur premier est strictement supérieur à $p$.**

**Il y a donc une infinité de nombres premiers congrus à 5 modulo 4 et 5 modulo 8.**

### 3.2.3 Algorithme des nombres de Pythagore en C

```
void pythagore() {
    int k = 0;
    printf("Entrez un nombre k tel que 4k+1 : ");
    scanf("%d",&k);
    printf("4*%d+1 = %d\n", k, 4*k+1);
    printf("4k+1 est un nombre de Pythagore possiblement Premier.\n");
}
```

# Chapitre 4

# Nombres Premiers de Mersenne et de Fermat

## 4.1 Les nombres de Mersenne

**Les nombres de Mersenne sont les nombres de la forme :**

$$M_n = 2^n - 1,\ n \geq 1$$

**Un nombre premier de Mersenne est un nombre qui est à la fois de Mersenne et premier. Pour que le $n$-ième nombre de Mersenne $M_n$ soit premier, il est nécessaire (mais non suffisant, contrairement à ce qu'affirmait Mersenne) que $n$ soit premier**

**Les plus petits nombres premiers de Mersenne connus sont :**

$$M_2 = 2^2 - 1 = 3$$
$$M_3 = 2^3 - 1 = 7$$
$$M_5 = 2^5 - 1 = 31$$
$$M_7 = 2^7 - 1 = 127$$
$$M_{13} = 2^{13} - 1 = 8191$$

### 4.1.1 Algorithme des nombres de Mersenne en C

```
void mersenne() {
    int n = 0;
    printf("Entrez un nombre n tel que (2^n)+1 : ");
    scanf("%d",&n);
    printf("(2^%d)+1 = %d\n", n, (int) pow(2,n)+1);
    printf("(2^n)+1 est un nombre de Mersenne possiblement Premier.\n");
}
```

## 4.2 Les nombres de Fermat

### 4.2.1 La forme des nombres de fermat

**Un nombre de Fermat est un nombre qui peut s'écrire sous la forme :**

$$2^{2^n} + 1,\ (n \in \mathbb{N})$$

**Le $n$-ième nombre de Fermat est souvent noté $F_n$. Ces nombres doivent leur nom à Pierre de Fermat qui avait émis la conjecture que tous ces nombres étaient premiers, or cette conjecture est fausse dès $F_5$.**

### 4.2.2 Démonstration

**Il existe deux entiers $a$ impair et $b$ tels que $k = a2^b$. En posant $c = 2^{(2^b)}$, on dispose alors des égalités suivantes :**

$$1 + 2^k = 1 + c^a = (1 + c) \sum_{i=0}^{a-1} (-1)^i c^i$$

**qui montrent que $1+c$ est un diviseur du nombre premier $1+2^k$ et donc lui est égal, on en déduit que $k = 2^b$**

### 4.2.3 Algorithme des nombres de Fermat en C

```
void fermat() {
    int n = 0;
    printf("Entrez un nombre n tel que (2^(2^n))+1 : ");
    scanf("%d",&n);
    printf("(2^(2^%d))+1 = %d\n", n, (int) pow(2,pow(2,n))+1);
    printf("(2^(2^n))+1 est un nombre de Fermat possiblement Premier.\n");
}
```

## 4.3 Primalité

**Les nombres de Mersenne et les nombres de Fermat sont deux ensembles de nombres qui ont beaucoup de propriétés, cependant ils ne sont pas tous premiers.**

**On suppose que l'ensemble des nombres premiers de Mersenne est infini et on continue à les calculer et à tester leur primalité.**

**Au contraire, les nombres de Fermat ne sont plus premiers à partir de $F_5$ jusqu'à $F_{33}$. Pour l'instant, et on ne sait pas s'il existe d'autres nombres de Fermat premiers.**

**Pour savoir si un nombre de Fermat ou un nombre de Mersenne est premier, on se doit de le calculer puis de tester sa primalité à l'aide de différent tests.**

## Troisième partie

# Tests de Primalité

# Chapitre 5

# Test de Primalité de Fermat

## 5.1 Base arithmétique

### 5.1.1 Petit théorème de Fermat

**Le petit théorème de Fermat dit que si $p$ est premier et que $a$ est premier avec $p$ alors on a, $a^{p-1}-1$ divisible par $p$.**

$$a^{p-1} \equiv 1(p)$$

### 5.1.2 Démonstration

**On suppose que $a$ n'est pas divisible par $p$.**

**On a,**

$N = a * 2a * 3a * 4a * 5a * ... * (p-1)a$

$r_k$ **tel que** $ka = pq + r_k$

**On sait que,** $N = 1 * 2 * 3 * 4 * 5 * ... * (p-1) * a^{p-1} = (p-1)! * a^{p-1}$

**En remplacant dans $N$, un par un, les $ka$ par $r_k$ on obtient**

$$N \equiv r_1 * r_2 * r_3 * r_4 * r_5 * ... * r_{p-1}(p)$$

**Car si l'on remplace dans un produit un facteur par un entier qui lui est congru modulo $p$ alors le nouveau produit est congru modulo $p$ a l'ancien.**

**Or comme $(r_1, r_2, ..., r_{p-1})$ est une permutation de $(1, 2, ..., p-1)$**

$$r_1 * r_2 * r_3 * r_4 * r_5 * ... * r_{p-1} = (p-1)!$$

**Car on sait que $0 \leq r_k \leq p-1$ et que d'apres le lemme d'Euclide $ka$ n'est pas divisible par $p$ donc aucun $r_k$ n'est nul.**

**On sait aussi que si $r_i = r_j$ alors $(i-j) * a$ est divisble par $p$, donc d'apres le lemme d'Euclide $i-j$ est aussi divisible par p, comme $-p < i-j < p$ alors $i = j$ et les $r_k$ sont distincts deux a deux.**

**On en deduit ainsi que,**

$$(p-1)! * a^{p-1} \equiv (p-1)!(p)$$

**Ainsi $(p-1)! * (a^{p-1}-1)$ est divisible par $p$.**

**En utilisant le lemme d'Eucilde à plusieurs reprises comme on a,**

$$r_1 * r_2 * r_3 * r_4 * r_5 * ... * r_{p-1} * (a^{p-1} - 1) \equiv 1(p)$$

**Alors on en conclut que $a^{p-1} - 1$ divisible par $p$.**

$$a^{p-1} \equiv 1(p)$$

## 5.2 Algorithmie

### 5.2.1 Test PGP

**Le test de primalité de Fermat repose sur le fait que si $p$ n'est pas premier alors $a^{p-1}$ n'est probablement pas divisible par $p$ pour une valeur arbitraire de $a$ première avec $p$.**

**"Pretty Good Privacy" plus connu sous le nom de PGP est un logiciel de chiffrement cryptographique utilisant la logique du test de primalité de Fermat.**

**Pour verifier si un nombre $p$ est premier le logiciel teste en prenant 4 valeurs de $a$ différentes $(2, 3, 5, 7)$ si $a^{p-1} \equiv 1(p)$.**

**Ainsi si,**

$$2^{p-1} \equiv 3^{p-1} \equiv 5^{p-1} \equiv 7^{p-1} \equiv 1(p)$$

**Alors $p$ est probablement un nombre premier.**

**Si un de ces nombres congru vers un nombre différent de $1$ alors $p$ n'est pas premier.**

**Il est possible de prendre plus de valeurs différentes pour $a$ pour augmenter la probabilité que les nombres premiers soient vrais.**

### 5.2.2 Algorithme en C

```
void fermat2() {
    int x = 0, a = 1, b = 1, c = 1, d = 1, i = 0;
    printf("Entrez un nombre x a tester : ");
    scanf("%d",&x);
    for(i=0;i<x-1;i++) {
        a = (a*2)%x;
        b = (b*3)%x;
        c = (c*5)%x;
        d = (d*7)%x;
    }
    if ((1==a)&&(1==b)&&(1==c)&&(1==d)) {
        printf("1=2^x-1=3^x-1=5^x-1=7^x-1 [mod x] est vrai \n\n");
        printf("Donc x = %d est un nombre probablement premier.\n\n",x);
    }
    else {
        printf("1=2^x-1=3^x-1=5^x-1=7^x-1 [mod x] est faux \n\n");
        printf("Donc x = %d n'est pas un nombre premier.\n\n",x);
    }
}
```

# Chapitre 6

# Test de Primalité de Solovay-Strassen

## 6.1 Base arithmétique

### 6.1.1 Symbole de Legendre

**Définition**

**Le symbole de Legendre $(\frac{a}{p})$ est une fonction de deux variables entieres retournant une valeur entiere comprise dans $[-1, 1]$.**

**Soit $p$ un nombre premier et $a$ un entier, alors le symbole de Legendre $(\frac{a}{p})$ vaut,**
- $-1$ si $a$ n'est pas un residu quadratique modulo $p$.
- 0 si $a$ est divisble par $p$.
- 1 si $a$ est un residu quadratique modulo $p$ mais n'est pas divisible par $p$.

**Ou un residu quadratique $a$ modulo $p$ signifie qu'il existe un entier $k$ tel que,**
$a \equiv k^2(p)$**.**

**Algorithme en C**

```
int symbole(int n, int p) {
        if(n%p==0) {
                return 0;
        }
        if((n%p)!=0) {
                if (residu(p,n)==1) {
                        return 1;
                }
        }
        return -1;
}

int residu(int p, int n) {
        int i;
        for(i=0;i<(p*p);i++) {
                if(((i*i)%p)==n) {
                        return 1;
                }
        }
        return 0;
```

}

### 6.1.2 Symbole de Jacobi

**Definition**

**Le Symbole de Jacobi $(\frac{a}{n})$ est une généralisation du symbole de Legendre.**

**Pour tout entier relatif $a$ et tout entier naturel impair $n$, le Symbole de Jacobi $(\frac{a}{n})$ est un produit de symbole de Legendre $(\frac{a}{p_k})$ pour tout $k \in \mathbb{N}$ tel que,**

$$p_1 * p_2 * p_3 * p_4 * p_5 * ... * p_k = n$$

**Selon la décomposition en facteur premier de n.**

**On a donc**

$$(\frac{a}{\prod_{1 \leq i \leq k} p_i}) = (\frac{a}{p_1}) * (\frac{a}{p_2}) * (\frac{a}{p_3}) * (\frac{a}{p_4}) * (\frac{a}{p_5}) * ... * (\frac{a}{p_k}) = \prod_{1 \leq i \leq k} (\frac{a}{p_i}) = (\frac{a}{n})$$

**Algorithme de décomposition en facteur premier en C**

```
void decompose(int n, int *tableau) {
    int k = 2, y = 0;
    while (n!=1) {
        k = 2;
        while (n%k!=0) {
            k++;
        }
        n = n/k;
        tableau[y] = k;
        y++;
    }
}
```

### 6.1.3 Critère d'Euler

**Le Critère d'Euler permet de déterminer si un entier est un residu quadratique modulo un nombre premier ou non.**

**Soient $p$ un nombre premier différent de 2 et $a$ un entier premier avec $p$,**

- Si $a$ est un residu quadratique modulo $p$ alors $a^{\frac{p-1}{2}} \equiv 1(p)$.
- Si $a$ n'est pas un residu quadratique modulo $p$ alors $a^{\frac{p-1}{2}} \equiv -1(p)$.

**En utilisant le symbole de Legendre on a donc,**

$$a^{\frac{p-1}{2}} \equiv (\frac{a}{p})(p)$$

## 6.2 Algorithmie

### 6.2.1 Test probabiliste

**Le test de primalité de Solovay-Strassen est un test probabiliste permettant de déterminer si un entier impair $n$ est premier ou non.**

**Pour cela, on test pour un nombre $a$ la congruence de,**

$$a^{\frac{p-1}{2}} \equiv (\frac{a}{p})(p)$$

**Si elle est satisfaite alors $a$ est probablement premier, sinon $a$ n'est pas premier.**

**Ce test etant probabiliste pour augmenter la probabilite que si la congruence est satisfaite $a$ est premier, alors il faut tester la congruence pour un grand nombre de valeurs de $a$.**

### 6.2.2 Algorithme en C

```
void solovaystrassen() {
    int i = 0, y = 0, a = 0, an = 1, anmod = 1;
    int n = 0, k = 0, tableau[50], e = 0, x = 0;
    printf("Entrez un nombre n a tester : ");
    scanf("%d",&n);
    printf("Entrez un nombre k corresepondant ");
    printf("au nombre de fois que le test va etre lance : ");
    scanf("%d",&k);
    srand(time(NULL));
    for (i=0;i<50;i++) {
        tableau[i] = 0;
    }
    decompose(n, tableau);
    for(y=0;y<k;y++) {
        a = 0;
        an = 1;
        anmod = 1;
        while ((a==0)||(a==1)) {
            a =(rand())%n;
        }
        for(i=0;i<50;i++) {
            if (tableau[i]!=0) {
                an = an*symbole(a,tableau[i]);
            }
        }
        for(i=0;i<((n-1)/2);i++) {
            anmod = (anmod*a)%n;
        }
        x = an+n;
        if ((x==0)||(x%n!=anmod)) {
            e = 1;
        }
    }
    if (e == 1) {
        printf("\nn = %d n'est pas premier.\n", n);
    }
    if (e == 0) {
        printf("\nn = %d est probablement premier si k est assez grand.\n", n);
    }
}
```

# Chapitre 7

# Test de Primalité de Miller-Rabin

## 7.1 Base arithmétique

### 7.1.1 Petit théorème de Fermat

**Rappel**

**Le petit théorème de Fermat dit que si $p$ est premier et que $a$ est premier avec $p$ alors on a, $a^{p-1}-1$ divisible par $p$.**

$$a^{p-1} \equiv 1(p)$$

**Propriete**

**Dans un anneau $\mathbb{Z}/p\mathbb{Z}$ si $p$ est premier alors l'équation $x^2 = 1$ n'a pour solutions que $1$ et $-1$.**
**Donc on a $1^2 = 1(p)$ et $(-1)^2 = 1(p)$**

### 7.1.2 Théorème de Rabin

**Définition**

**Pour un entier impair composé $n$ superieur a $9$ et $d$ impair tel que $n-1 = 2^s * d$**
**Alors il existe $\varphi(n)/4$ menteurs forts $a$ pour $1 < a < n$**
**Ou $a$ est un menteur fort si il verifie soit $a^d \equiv 1(p)$, soit $a^{2^r d} \equiv -1(p)$ pour $r$ tel que $0 \leq r < s$**

**Avec $\varphi(n)$ fonction indicatrice d'Euler.**

**Corollaire**

**Pour un nombre impair composé $n$, 3/4 au moins des entiers $a$ tel que $1 < a < n$ sont des témoins de Miller pour $n$.**

**Si $n$ est compose alors $a$ est un témoin de Miller si $a^d \not\equiv 1(p)$**
**et pour quelque soit $r \in [0, s-1]$ $a^{2^r d} \not\equiv -1(p)$**

## 7.2 Algorithmie

### 7.2.1 Test probabiliste

**Le Test de Miller-Rabin est un test probabiliste de type Monte Carlo donnant une réponse oui ou non permettant de savoir si un nombre est de facon certaine composé ou si il est probablement premier.**

**On sait d'apres le petit théorème de Fermat et le théorème de Rabin que,**

$$a^{p-1} = (a^d)^{2^s} \equiv 1(p)$$

**Pour $p$ premier, un entier $a$ non divisble par $p$, $s$ non nul et $d$ impair.**

**Ainsi pour verifier si $n$ est premier on prend une valeur de $a$ aléatoire non divisble par $n$ est on verifie $a^d \equiv 1(n)$ et $r \in [0, s-1]$ $a^{2^r d} \equiv -1(n)$**

- Si les congruences sont valides alors $a$ n'est pas un témoin de Miller et $n$ est probablement premier.
- Sinon $a$ est un témoin de Miller et $n$ est de facon certaine composé.

**On relance ensuite le test pour augmenter la probabilité que $n$ puisse etre premier si aucun témoin de Miller n'a ete trouvé.**

### 7.2.2 Algorithme en C

```
void millerrabin() {
    int a = 0, s = 0, d = 0, n = 1, k = 0, e = 0, i = 0;
    printf("n est un nombre a tester tel que n = (2^s)*d+1 \n");
    printf("Entrez un nombre s : ");
    scanf("%d",&s);
    printf("Entrez un nombre d : ");
    scanf("%d",&d);
    for(i=0;i<s;i++) {
        n = n*2;
    }
    n = n*d+1;
    printf("(2^s)*d+1 = %d\n", n);
    printf("Entrez un nombre k corresepondant ");
    printf("au nombre de fois que le test va etre lance : ");
    scanf("%d",&k);
    srand(time(NULL));
    for(i=0;i<k;i++) {
        a = 0;
        while ((a==0)||(a==1)) {
            a = (rand())%(n-1);
        }
        if (temoinmiller(a,s,d)) {
            e = 1;
        }
    }
    if (e == 1) {
        printf("\nn = %d n'est pas premier.\n", n);
    }
    if (e == 0) {
        printf("\nn = %d est probablement premier si k est assez grand.\n", n);
    }
}
```

```
int temoinmiller(int a, int s, int d) {
    int n = 1, i = 0, x = 1;
    for(i=0;i<s;i++) {
        n = n*2;
    }
    n = n*d+1;
    for(i=0;i<d;i++) {
        x = x*a%n;
    }
    x = x+n%n;
    if ((x==1)||(x==(n-1))) {
        return(0);
    }
    while (s>1) {
        x = x*x%n;
        x = x+n%n;
        if (x==(n-1)) {
            return(0);
        }
        s--;
    }
    return(1);
}
```