

# TIPE cryptographie

Samuel MIMRAM

Joseph PEREZ

2000-2001

Dernière révision : 17 mai 2006.

## Table des matières

<b>1</b>	<b>Différents cryptosystèmes</b>	<b>1</b>
1.1	Quelques procédés de cryptographie	1
1.2	RSA	2
1.2.1	RSA en pratique	2
1.2.2	Génération d'un couple de clés	2
1.2.3	Démonstration du fonctionnement du système RSA	2
1.2.4	Sécurité du système RSA	3
<b>2</b>	<b>Tests de primalité</b>	<b>3</b>
2.1	Tests simples	3
2.2	Test de Fermat	4
2.3	Test de Miller-Rabin	4
2.4	Test de Lehmer	4
2.5	Test de Lucas	4
2.6	Test de Pépin	4
<b>3</b>	<b>Multiplication rapide</b>	<b>5</b>
3.1	Principes de multiplication rapide	5
3.1.1	Coût de la méthode de Hörner	5
3.1.2	L'identité de Karatsuba	5
3.2	La transformée de Fourier	5
3.2.1	Représentation polynômiale d'un nombre	5
3.2.2	Évaluation, interpolation	5
3.2.3	Théorèmes préliminaires sur les racines complexes de l'unité	6
3.2.4	La transformée discrète de Fourier (DFT)	6
3.2.5	La transformée rapide de Fourier (FFT)	6
3.2.6	Implémentation algorithmique de la transformée rapide de Fourier	7
3.2.7	La transformée inverse de Fourier	8
3.2.8	Déroulement d'une multiplication	8
<b>4</b>	<b>Factorisation</b>	<b>9</b>
4.1	Par la différence de deux carrés	9

## 1 Différents cryptosystèmes

### 1.1 Quelques procédés de cryptographie

**La méthode par substitution** consiste à remplacer chaque lettre par une autre, par exemple en décalant de  $n$  rangs les lettres dans l'alphabet (*méthode César*). On peut aussi remplacer chaque lettre par un nombre donné. Ce cryptage est très facilement cassable en se basant sur la fréquence d'apparition des lettres dans la langue dans laquelle a été écrit le message et n'est donc plus utilisé depuis plusieurs siècles.

**Le ou-exclusif (xor)** est la seule opération booléenne à être bijective. Sa table de vérité est :

	0	1
0	0	1
1	1	0

On

remarquera que  $(A \text{ xor } B) \Leftrightarrow ((A \text{ OR } B) \text{ AND NOT } (A \text{ AND } B))$ . La méthode d'encryption par XOR nécessite une clé  $C$  (de préférence assez longue). Pour coder un message  $M$  on fait alors un XOR bit à bit entre  $M$  et  $C$  :  $M' = M \text{ XOR } C$ . A l'inverse, pour décrypter le message on effectue l'opération booléenne suivante, bit à bit :  $M = M' \text{ XOR } C$ .

Cette méthode a l'avantage d'être peu coûteuse en temps de calcul, d'être facile à implémenter et d'être *relativement*

sécurisée (il est néanmoins envisageable, avec les puissances actuelles de calcul, de décrypter un message codé par cette méthode). Son inconvénient majeur est que la clé qui sert à décrypter le message est la même que celle qui sert à l'encrypter.

**L'algorithme de Diffie-Hellman** permet à deux personnes A et B de se mettre d'accord sur une clé privée commune sans qu'elle risque d'être interceptée. L'opération se déroule de la façon suivante :

- A et B choisissent  $(g, n) \in \mathbb{N}^2, 1 < g < n$  (pour que la communication soit réellement sécurisée, il faut que  $n$  soit au moins de l'ordre de 1024 bits).
  - Chacun choisit ensuite de son côté un nombre aléatoire (A choisit  $x$  et B choisit  $y$ ).
  - A calcule  $X \equiv g^x [n]$  et l'envoie à B. Réciproquement, B calcule  $Y \equiv g^y [n]$  et l'envoie à A. L'échange  $(X, Y)$  n'a pas forcément besoin d'être sécurisé.
  - A calcule la clé privée  $k \equiv Y^x [n]$  et B calcule  $k' \equiv X^y [n]$ . Et on a :  $k \equiv k' \equiv g^{xy} [n]$ .
- Il est impossible, avec les puissances de calcul actuelles, de déduire  $x$  et  $y$  (et donc  $k$ ) à partir de  $(g, n, X, Y)$ .

## 1.2 RSA

### 1.2.1 RSA en pratique

RSA est un cryptosystème à clé publique : les messages sont encodés avec une clé publique mais seule la clé privée permet de décoder le message. Si  $M$  est le message,  $E$  désigne la fonction d'encodage et  $D$  celle de décodage, on a :  $E$  et  $D$  sont des fonctions inverses c'est à dire  $M = D(E(M)) = E(D(M))$ .

Si A veut envoyer un message  $M$  à B, la procédure de transmission d'un message est donc la suivante :

- A a la clé publique de B.
- A encode le message :  $E(M)$ .
- A transmet le message codé à B.
- B décode le message à l'aide de la clé privée qu'il est le seul à connaître :  $M = D(E(M))$ .

Réciproquement les messages peuvent être signés.

### 1.2.2 Génération d'un couple de clés

La procédure de génération d'une clé privée et de la clé publique associée est la suivante :

- Choisir aléatoirement deux nombres premiers  $p$  et  $q$  de 100 à 200 chiffres chacun (en dessous le cryptage devient "cassable" en un temps accessible et au delà, l'encodage et le décodage des messages deviennent trop longs).
- Calculer  $n = p \cdot q$  (dans les logiciels actuels de cryptage (e.g. PGP), la taille de  $n$  est comprise entre 1024 et 8192 bits).
- Choisir un petit entier impair  $e$  premier avec  $(p-1)(q-1)$ .
- Calculer  $d$ , inverse de  $e$  modulo  $(p-1)(q-1)$  :  $ed \equiv 1 [(p-1)(q-1)]$ . Cet inverse existe car  $e \wedge (p-1)(q-1) = 1$ , d'où, d'après le théorème de Bezout :  $\exists(d, u) \in \mathbb{Z}^2, ed + u(p-1)(q-1) = 1$  et on a :  $ed \equiv 1 [(p-1)(q-1)]$ .
- La clé publique est le couple  $(e, n)$  et la clé privée est le couple  $(d, n)$ .

Les procédures d'encodage et de décodage sont alors :  $E(M) \equiv M^e [n]$  et  $D(N) \equiv N^d [n]$  (restes des divisions euclidiennes par  $n$ ).

Remarque :  $(p-1)(q-1) = \phi(n)$  où  $\phi(n)$  est l'indicateur d'Euler qui à un entier  $n$  associe le nombre d'entiers compris entre 1 et  $n$  et premiers avec  $n$

### 1.2.3 Démonstration du fonctionnement du système RSA

Soit  $M \in \mathbb{Z}/n\mathbb{Z}$  ( $M$  doit être assez petit pour qu'on soit sûr de toujours avoir :  $M < n$ ). On a alors :  $D(E(M)) \equiv E(D(M)) \equiv M^{ed} [n]$ .

Soit  $M' \equiv M^e [n]$ . Alors  $\exists k \in \mathbb{Z}, M' = M^e + kn$  i.e.  $n|(M' - M^e)$ . De plus, comme  $ed \equiv 1 [(p-1)(q-1)]$ , alors :  $\exists k' \in \mathbb{Z}, ed = 1 + k'(p-1)(q-1)$ .

On cherche à montrer que  $M'^d \equiv M [n]$  i.e.  $n|(M - M'^d)$ .

Développons  $M'^d = (M^e + kn)^d$  avec la formule du binôme de Newton :

$$M'^d = \sum_{j=0}^d C_d^j (M^e)^j (kn)^{d-j}$$

De façon triviale, on a :  $\forall j \in [0, d-1], n|(C_d^j (M^e)^j (kn)^{d-j})$  donc

$$n | \sum_{j=0}^{d-1} C_d^j (M^e)^j (kn)^{d-j}$$

Montrons que on a aussi :  $n|(M - M^{ed})$ , soit que  $n|(M - M^{1+k'(p-1)(q-1)})$ .

D'après le petit théorème de Fermat,  $p$  étant premier, on a :  $\forall a \in \mathbb{Z}, a^p \equiv a [p]$  donc

$$M(M^{k'q})^p M^{-k'q} (M^{-k'})^p M^{k'} \equiv M^{1+k'q-k'q-k'+k'} \equiv M [p]$$

donc

$$M^{1+k'(p-1)(q-1)} \equiv M [p]$$

soit  $p|(M - M^{1+k'(p-1)(q-1)})$ . De même,  $q|(M - M^{1+k'(p-1)(q-1)})$  donc

$$pq = n|(M - M^{1+k'(p-1)(q-1)})$$

soit enfin  $n|(M - M^{ed})$ . Finalement,  $n|(M - \sum_{j=0}^d C_d^j (M^e)^j (kn)^{d-j})$  donc  $n|(M - M'^d)$  d'où :

$$M'^d \equiv (M^e)^d \equiv (M^d)^e \equiv M [n]$$

### 1.2.4 Sécurité du système RSA

Le couple  $(e, n)$  est public il est donc supposé connu de tous. Pour pouvoir décrypter un message, il *suffirait* donc de retrouver  $d$ , ce qui suppose qu'on soit capable de factoriser  $n$ . Or l'algorithme de factorisation le plus rapide connu à ce jour est celui de Richard Schroepel qui peut factoriser un entier  $n$  en  $O(e^{\sqrt{\ln(n) \cdot \ln(\ln(n))}})$ . A titre d'exemple, en considérant que chaque opération s'effectue en une microseconde voici le temps que prendrait approximativement la fonction pour factoriser un entier  $n$  (la longueur  $l$  est le nombre de chiffres de  $n$  écrit en base 10 ( $l = E(\log n) + 1$ )).

Longueur	Nb. d'opérations	Durée
50	$1.4 \cdot 10^{10}$	3.9 heures
75	$9.0 \cdot 10^{12}$	104 jours
100	$2.3 \cdot 10^{15}$	74 années
200	$1.2 \cdot 10^{23}$	$3.8 \cdot 10^9$ années
300	$1.5 \cdot 10^{29}$	$4.9 \cdot 10^{15}$ années
500	$1.3 \cdot 10^{39}$	$4.2 \cdot 10^{25}$ années

Ainsi, mis à part une évolution majeure dans la technologie informatique ou dans l'arithmétique, ce système restera sûr à quasiment 100 % pendant encore plusieurs siècles. En effet la longueur des clés d'encryptage utilisables augmentera toujours plus rapidement que la rapidité de factorisation. La seule perspective à ce jour est constituée par les ordinateurs quantiques qui, du fait de leur structure, seraient à même de factoriser les clés qui sont utilisées et rendraient ainsi obsolète ce système de cryptographie. Les connaissances actuelles rendent cependant inenvisageable la réalisation de tels ordinateurs dans les prochaines années.

**Les procédés qui doivent être mis en œuvre par le système RSA** sont donc :

- **Les tests de primalité**, qu'il soient déterministes ou probabilistes, sont indispensables pour déterminer les deux grands nombres premiers  $p$  et  $q$ .
- **La multiplication de grands nombres** est nécessaires pour pouvoir encoder et décoder des informations en un temps raisonnable et permettre une utilisation courante du système (aujourd'hui ces opérations requièrent encore trop de temps et les implémentations du système RSA (e.g. le logiciel PGP) sont couplées avec d'autres procédés d'encryption (DES pour PGP)).
- **La factorisation** qui, tant qu'elle ne sera pas réalisable en un temps envisageable pour les grands nombres (une centaine de chiffres), garantit la sécurité du cryptage des données.

Nous allons étudier dans la suite ces différents points cruciaux du système RSA et de nombreux autres.

## 2 Tests de primalité

### 2.1 Tests simples

**Les critères de divisibilité** sont un moyen peu coûteux de prouver qu'un nombre n'est pas premier. Ils sont pour la plupart basés sur des calculs de congruences modulo la base dans laquelle on travaille (souvent 10 ou 16).

Soit  $(c_n)_{0 \leq n < m} \in ([0, 9])^m$  l'ensemble des  $m$  chiffres d'un nombre  $n$  écrit en base décimale ( $n = \sum_{k=0}^{m-1} 10^k c_k$ )

Ainsi  $n$  est divisible par neuf si et seulement si la somme de ses chiffres l'est (preuve par 9). En effet :  $n = \sum_{k=0}^{m-1} 10^k c_k \equiv \sum_{k=0}^{m-1} 1^k c_k [9]$ .

De même si alors  $n$  est divisible par 7 si et seulement si  $\sum_{k=0}^{m-1} 3^k c_k$  est divisible par 7 (cette somme pourra se faire modulo 7 pour simplifier le calcul). En effet :  $n = \sum_{k=0}^{m-1} 10^k c_k \equiv \sum_{k=0}^{m-1} 3^k c_k [7]$ .

**Diviser  $n$  par tous les entiers inférieurs à  $\sqrt{n}$**  est aussi une méthode pour savoir si un nombre  $n$  est premier. On sait en effet que si  $n$  n'est pas premier alors il admet au moins un diviseur premier inférieur à  $\sqrt{n}$ . Bien que semblant à première vue un peu naïf, cet algorithme est néanmoins encore utilisé pour tester si un nombre a un diviseur premier inférieur à environ 1000. Il est en effet plus rapide, pour les petits nombres premiers, de les tester un par un pour voir s'ils divisent le nombre dont on souhaite connaître la primalité que de mettre en œuvre d'autres tests plus coûteux. Ce test est donc systématiquement réalisé avant tout autre pour éviter de perdre du temps de calcul.

## 2.2 Test de Fermat

Ce test est basé sur la “réciproque” du petit théorème de Fermat. On sait en effet que :  $\forall p \in \mathbb{P}, \forall a \in \mathbb{Z}, a \wedge p = 1 \Rightarrow a^{p-1} \equiv 1 [p]$ . La réciproque de cette proposition n’est pas toujours vraie (e.g.  $2^{340} \equiv 1 [341]$ ). Un nombre  $p$  qui vérifie la conclusion de Fermat pour un entier  $a$  sans être premier est dit *pseudo-premier de base a*. Ces nombres sont très rares bien qu’il y en ait une infinité. Si pour un nombre  $n$  d’entiers  $a_i$  premiers avec  $p$ , on a :  $a^{p-1} \equiv 1 [p]$  alors  $p$  a d’autant plus de chances d’être premier que  $n$  est grand.

## 2.3 Test de Miller-Rabin

On peut augmenter la fiabilité du précédent test en lui adjoignant la réciproque du théorème d’Euler qui stipule que :  $\forall p \in \mathbb{P}, \forall a \in \mathbb{Z}, a \wedge p = 1 \Rightarrow a^{\frac{p-1}{2}} \equiv 1 [p]$ .

Voici l’algorithme ( $n$  est le nombre dont on veut tester la primalité et  $t$  est un paramètre dont dépendront le temps d’exécution et la fiabilité du test) :

```
1  MILLER-RABIN( $n, t$ )
2  SOIT ( $r, s$ ) TEL QUE  $n - 1 = 2^s r$  AVEC  $r$  IMPAIR
3  FOR  $i$  FROM 1 TO  $t$ 
4     $a \leftarrow \text{RANDOM}([2, n - 2])$ 
5     $y \equiv a^r [n]$ 
6    IF ( $y <> 1$ ) AND ( $y <> n - 1$ ) THEN
7       $j \leftarrow 1$ 
8      WHILE ( $j < s$ ) AND ( $y <> n - 1$ )
9         $y \equiv y^2 [n]$ 
10     IF  $y = 1$  THEN RETURN(“PREMIER”)
11      $j \leftarrow j + 1$ 
12     IF ( $y <> n - 1$ ) THEN RETURN(“NON PREMIER”)
13  RETURN(“PREMIER”)
```

## 2.4 Test de Lehmer

Voici les trois propositions de Lehmer qui permettent de déterminer si un nombre est premier :

- Si  $a^x \equiv 1 [N]$  pour  $x = N - 1$  mais non pour tout  $x$  diviseur propre de  $N - 1$ , alors  $N$  est premier.
- Si  $a^x \equiv 1 [N]$  pour  $x = N - 1$  mais non pour  $x$  quotient de  $N - 1$  par division d’un de ses facteurs premiers, alors  $N$  est premier.
- Si  $a^x \equiv 1 [N]$  pour  $x = N - 1$  et si  $a^{(N-1)/p} \equiv r > 1 [N]$  et si  $r - 1$  est premier avec  $N$ , alors tous les facteurs premiers de  $N$  sont de la forme  $np^\alpha + 1$ , où  $\alpha$  est la plus grande puissance du nombre premier  $p$  qui divise  $N - 1$ .
- Si  $\forall p \in \mathbb{P}, p|(N - 1) \Rightarrow (\exists a, a^{(N-1)/q} \not\equiv 1 [N] \text{ et } a^{N-1} \equiv 1 [N])$  alors  $N$  est premier.

## 2.5 Test de Lucas

Soit  $(P, Q) \in \mathbb{Z}$  tels que  $P \wedge Q = 1$ . On note  $a$  et  $b$  les racines de l’équation  $x^2 - Px + Q = 0$ . Les suites  $(U_n)_{n \in \mathbb{N}}$  et  $(V_n)_{n \in \mathbb{N}}$  définies par :

$$\forall n \in \mathbb{N}, \begin{cases} U_n = \frac{a^n - b^n}{a - b} \\ V_n = a^n + b^n = U_{n+1} - Q \cdot U_{n-1} \end{cases}$$

sont alors appelées suites de Lucas (on remarquera que la suite de Fibonacci est la suite de Lucas obtenue en prenant  $P = 1$  et  $Q = -1$ ). Et le théorème fondamental de Lucas est :

Si dans l’une des suites récurrentes  $(U_n)$  (de Lucas), le terme  $U_{p-1}$  est divisible par  $p$ , sans qu’aucun des termes de la suite dont le rang est un diviseur de  $p - 1$  le soit (dans la pratique, le calcul des termes de la suite se fera modulo  $p$ ), le nombre  $p$  est premier ; de même, si  $U_{p+1}$  est divisible par  $p$  sans qu’aucun des termes de la suite dont le rang est un diviseur de  $p + 1$  le soit,  $p$  est premier.

## 2.6 Test de Pépin

Ce test permet de déterminer la primalité de nombres de la forme  $2^{2^n} + 1$  ( $n \in \mathbb{N} \setminus \{0, 1\}$ ). En effet, si  $n \in \mathbb{N} \setminus \{0, 1\}$ , le théorème de Pépin stipule que : le nombre  $a_n = 2^{2^n} + 1$  est premier si et seulement si  $5^{(a_n-1)/2} + 1$  est divisible par  $a_n$ .

## 3 Multiplication rapide

### 3.1 Principes de multiplication rapide

#### 3.1.1 Coût de la méthode de Hörner

Soit  $A \in \mathbb{R}[X]$  et soit  $n = d^o(A)$ . Alors :  $\exists (a_0, \dots, a_n) \in \mathbb{R}^n, A = \sum_{k=0}^n a_k X^k$ .  
Soit  $x_0 \in \mathbb{R}$ . Calculons  $A(x_0)$  par la méthode de Hörner :

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots + x_0(a_n + x_0 a_n)) \dots)$$

Il nous faut  $n + 1$  multiplications. Le coût en temps de cette méthode sera donc en  $O(n)$ .

#### 3.1.2 L'identité de Karatsuba

Soit  $(X, Y) \in \mathbb{Z}^2$  deux grands nombres considérés en base  $B$ . L'identité de Karatsuba permet de multiplier  $X$  par  $Y$  plus rapidement qu'en  $O(n^2)$ .

Soit  $n$  le nombre de chiffres de du plus grand des deux nombres ( $n = E(\max(\log_B X, \log_B Y)) + 1$ ). Soit  $m = E(\frac{n}{2})$ . On "coupe" alors  $X$  et  $Y$  en deux :

$$\exists (X_0, X_1, Y_0, Y_1) \in \mathbb{Z}^4, \begin{cases} X = X_0 + B^m X_1 \\ Y = Y_0 + B^m Y_1 \end{cases}$$

et on a :

$$XY = X_0 Y_0 + B^m (X_0 Y_1 + X_1 Y_0) + B^{2m} X_1 Y_1$$

Mais au lieu de calculer les quatre produits  $(X_0 Y_0, X_0 Y_1, X_1 Y_0, X_1 Y_1)$ , l'idée de la méthode de Karatsuba est de ne calculer que trois produits en écrivant :

$$XY = P_0 + B^m (P_1 - P_2 - P_0) + B^{2m} P_2$$

avec

$$P_0 = X_0 Y_0, P_1 = (X_0 + X_1)(Y_0 + Y_1), P_2 = X_1 Y_1$$

L'opération peut être faite récursivement et on montre qu'avec cette méthode, le coût d'une multiplication est en  $O(n^{\frac{\log 3}{\log 2}})$  (avec  $\frac{\log 3}{\log 2} \approx 1.585$ ).

### 3.2 La transformée de Fourier

#### 3.2.1 Représentation polynômiale d'un nombre

Soit  $B \in \mathbb{N}$  ( $B$  est la base dans laquelle on va représenter les nombres). Soit  $X = (x_k)_{k \in \mathbb{N}} \in \mathbb{Z}^{(\mathbb{N})}$  et  $Y = (y_k)_{k \in \mathbb{N}} \in \mathbb{Z}^{(\mathbb{N})}$ . On définit la relation  $\mathcal{R}$  par :  $X \mathcal{R} Y \Leftrightarrow \sum_{k=0}^{\infty} x_k B^k = \sum_{k=0}^{\infty} y_k B^k$  ( $X$  et  $Y$  sont des familles presque nulles donc ces sommes sont en réalité finies).

L'ensemble  $\mathbb{Z}$  des entiers relatifs peut être considéré comme  $\mathbb{Z}^{(\mathbb{N})}/\mathcal{R}$  i.e. l'ensemble quotient de l'ensemble des familles presque nulles d'entiers relatifs par la relation d'équivalence  $\mathcal{R}$ . Si  $X = (x_k)_{k \in \mathbb{N}} \in \mathbb{Z}^{(\mathbb{N})}$ , le représentant principal de  $X$  est celui où tous les coefficients  $x_k$  ( $k \in \mathbb{N}$ ) sont tels que  $0 \leq x_k < B$ .

Par exemple, le nombre 216 a pour représentant principal  $(0, \dots, 0, 2, 1, 6)$  mais les représentations  $(0, \dots, 0, 11, 106)$  et  $(0, \dots, 0, 4, -19, 6)$  sont d'autres représentations possibles qui appartiennent à la classe d'équivalence de 216.

En pratique, on prend souvent  $B = 2^{33}$  ou  $B = 2^{65}$  car, du fait de la représentation binaire qu'ils ont des nombres, les ordinateurs sont efficaces pour traiter des nombres de 32 ou 64 bits.

**Existence et unicité du représentant principal** Soit  $N \in \mathbb{Z}^{(\mathbb{N})}/\mathcal{R}$  alors :  $\exists ! (a_k) \in ([0, B - 1])^{(\mathbb{N})}, N = \sum a_k X^k$  et  $A = \sum a_k X^k$  est alors appelé *représentant principal* de  $N$ .

Multiplier deux nombres revient donc à multiplier deux polynômes.

#### 3.2.2 Évaluation, interpolation

Un polynôme de degré  $n$  est déterminé de façon unique par la donnée de  $n + 1$  de ses évaluation en  $n + 1$  points distincts.

En effet soit  $n \in \mathbb{N}$ , soit  $(P, Q) \in \mathbb{Z}_n[X]$  tel que :  $\exists (a_0, \dots, a_n) \in \mathbb{C}^{n+1}, \forall k \in [0, n], P(a_k) = Q(a_k)$ . On a alors :  $\forall k \in [0, n], (P - Q)(a_k) = 0$ . Le polynôme  $(P - Q)$  est un polynôme de degré au plus  $n$  admettant  $n + 1$  racines c'est donc le polynôme nul. Donc  $(P - Q) = 0$ , soit  $P = Q$ .

Ainsi, si  $A$  et  $B$  sont deux polynômes, pour connaître le polynôme  $A \cdot B$ , il nous suffit de calculer  $(A \cdot B)(x) = A(x) \times B(x)$  pour  $n + 1$  valeurs de  $x$  différentes. Le calcul de  $A$  à partir de  $n + 1$  de ses évaluations s'appelle l'*interpolation*.

Soit  $A$  un polynôme de degré  $n$ . Nous avons vu que l'évaluation de  $A$  en un point par la méthode de Hörner se faisait en  $O(n)$ . L'évaluation de  $A$  en  $n + 1$  points se fera donc en  $O(n^2)$ .

La transformée de Fourier permet, elle, de réaliser ces évaluations et de calculer les polynômes d'interpolation en  $O(n \ln n)$ .

### 3.2.3 Théorèmes préliminaires sur les racines complexes de l'unité

On notera dans la suite  $\omega_n$  la racine complexe  $n$ -ième de l'unité ( $\omega_n = e^{\frac{2i\pi}{n}}$ ).

**Lemme de l'annulation** Soit  $k \in \mathbb{N}$ . Alors on a :

$$\omega_{\lambda n}^{\lambda k} = \omega_n^k$$

En effet :  $\omega_{\lambda n}^{\lambda k} = e^{\frac{\lambda k i \pi}{\lambda n}} = e^{\frac{k i \pi}{n}} = \omega_n^k$ .

**Lemme de la bipartition** Si  $n$  est pair alors les carrés des racines  $n$ -ièmes de l'unité sont les racines  $\frac{n}{2}$ -ièmes de l'unité.

$$(\omega_n^k)^2 = e^{\frac{2k i \pi}{n}} = e^{\frac{k i \pi}{\frac{n}{2}}} = \omega_{\frac{n}{2}}^k$$

Soit  $k \in [0, \frac{n}{2}]$  alors  $\exists k' \in [0, \frac{n}{2}], k = k' + \frac{n}{2}$  et on a :  $\omega_{\frac{n}{2}}^k = e^{\frac{2k' i \pi}{n}} \times e^{\frac{2 \frac{n}{2} i \pi}{n}} = \omega_{\frac{n}{2}}^{k'}$ .

**Corollaire**

$$\omega_{\frac{n}{2}}^{\frac{n}{2}} = \omega_2 = -1$$

**Lemme de la sommation** Soit  $k \in \mathbb{N}$ . Alors on a :

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

En effet :  $\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{1 - (\omega_n^k)^n}{1 - \omega_n^k} = \frac{1 - (\omega_n^n)^k}{1 - \omega_n^k} = \frac{1 - 1}{1 - \omega_n^k} = 0$

### 3.2.4 La transformée discrète de Fourier (DFT)

Soit  $A \in \mathbb{Z}[X]$ . Soit  $n = d^o(A) + 1$ . Alors :  $\exists a = (a_0, \dots, a_{n-1}) \in \mathbb{Z}^n, A = \sum_{j=0}^{n-1} a_j X^j$ . On définit  $(y_k)_{k \in [0, n-1]}$  par :

$$\forall k \in [0, n-1], y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

$y = (y_0, \dots, y_{n-1})$  est appelé transformée discrète de Fourier et est noté :  $y = DFT_n(a)$ .

### 3.2.5 La transformée rapide de Fourier (FFT)

Soit  $A \in \mathbb{Z}[X]$ . Soit  $n \in \mathbb{N}$  tel que  $n$  soit une puissance de 2 et  $n > d^o(A)$  (on complète avec des zéros). Alors :

$\exists a = (a_0, \dots, a_{n-1}) \in \mathbb{Z}^n, A = \sum_{j=0}^{n-1} a_j X^j$ .

On définit ensuite  $A_0$  et  $A_1$  par :

$$A_0(X) = a_0 + a_2 x + a_4 x^2 + \dots + a_{(n/2-2)} x^{n/2-1}$$

$$A_1(X) = a_1 + a_3 x + a_5 x^2 + \dots + a_{(n/2-1)} x^{n/2-1}$$

( $A_0$  est le polynôme des coefficients d'indices pairs et  $A_1$  celui des indices impairs).

On a alors de façon triviale :

$$A = A_0(X^2) + X A_1(X^2) \tag{1}$$

Le principe pour avoir une évaluation de  $A$  aux  $n$  racines  $n$ -ièmes complexes de l'unité est le suivant :

– On évalue  $A_0(x)$  et  $A_1(x)$  aux points  $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$  i.e. aux  $\frac{n}{2}$  racines  $\frac{n}{2}$ -ièmes complexes de l'unité d'après le *lemme de la bipartition*.

– On combine les résultats avec (1) pour obtenir  $y = (y_0, \dots, y_{n-1}) = (A(\omega_n^0), \dots, A(\omega_n^{n-1}))$ .

Cette opération est notée  $y = FFT_n(a)$ . Son écriture matricielle est :  $y = TDF_n(a) = V_n \cdot a$  où  $(V_{n,(j,k)})_{0 \leq j, k \leq n-1}$  est la matrice de Vandermonde définie par  $\forall (j, k) \in [0, n-1]^2, V_{n,(j,k)} = \omega_n^{jk}$ . Ainsi on a :

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Les évaluations de  $A_0$  et  $A_1$  devant être faites au  $\frac{n}{2}$  racines complexes  $\frac{n}{2}$ -ièmes de l'unité, on va utiliser récursivement la transformée : on est ramené au calcul de  $FFT_{\frac{n}{2}}(A_0)$  et  $FFT_{\frac{n}{2}}(A_1)$ .

### 3.2.6 Implémentation algorithmique de la transformée rapide de Fourier

L'algorithme est le suivant :

```

1  FFT-RÉCURSIVE( $a$ )
2   $n \leftarrow \text{LONGUEUR}[a]$ 
3  IF  $n=1$  THEN RETURN  $a$ 
4   $\omega_n \leftarrow e^{\frac{2i\pi}{n}}$ 
5   $\omega \leftarrow 1$ 
6   $a' \leftarrow (a_0, a_2, \dots, a_{n-2})$ 
7   $a'' \leftarrow (a_1, a_3, \dots, a_{n-1})$ 
8   $y' \leftarrow \text{FFT-RÉCURSIVE}(a')$ 
9   $y'' \leftarrow \text{FFT-RÉCURSIVE}(a'')$ 
10 FOR  $k$  FROM 0 TO  $(\frac{n}{2} - 1)$ 
11   DO
12      $y_k \leftarrow y'_k + \omega y''_k$ 
13      $y_{(k+\frac{n}{2})} \leftarrow y'_k - \omega y''_k$ 
14      $\omega \leftarrow \omega \omega_n$ 
15   LOOP
16  $y \leftarrow (y_0, y_1, \dots, y_{n-1})$ 
17 RETURN  $y$ 

```

Voici l'analyse du code :

- La ligne 3 est la condition d'arrêt des appels récursifs de la fonction : l'évaluation d'un polynôme de degré 0 (i.e. une constante) est la constante elle-même et ce, en tout point.
- Les lignes 6 et 7 séparent le polynôme  $A$  (de coefficients  $a$ ) en deux polynômes ; l'un des coefficients pairs ( $A_1$  de coefficients  $a'$ ) et l'autre des coefficients impairs ( $A_2$  de coefficients  $a''$ ).
- Les lignes 8 et 9 font des appels à la fonction elle-même pour traiter récursivement les polynômes  $A_1$  et  $A_2$ .
- Pour les lignes 10 à 15, cf. ci-dessous.
- La ligne 14 sert à ce que l'on ait en permanence  $\omega = \omega_n^k$ .
- Les lignes 16 et 17 recombinaient les résultats dans le vecteur  $y$  et la fonction retourne  $y$ .

Montrons le fonctionnement de l'algorithme par récurrence sur  $n$  tel que la longueur de  $a$  soit  $2^n$  (cette méthode est valable pour les polynômes de tous degrés, il suffit de compléter avec des zéros).

Au rang 1 l'algorithme renvoie bien l'évaluation de  $a$  de façon triviale (cf. ligne 3).

Soit  $n \in \mathbb{N}$ . On suppose que l'algorithme a bien fonctionné jusqu'au rang  $n$ . Alors  $y'$  et  $y''$  contiennent les évaluations de respectivement  $A_0$  et  $A_1$  aux  $\frac{n}{2}$  racines complexes  $\frac{n}{2}$ -ièmes de l'unité :

$$y'_0 = A_0(1), y'_1 = A_0(\omega_{\frac{n}{2}}), y'_2 = A_0(\omega_{\frac{n}{2}}^2), \dots, y'_{\frac{n}{2}-1} = A_0(\omega_{\frac{n}{2}}^{\frac{n}{2}-1})$$

et

$$y''_0 = A_1(1), y''_1 = A_1(\omega_{\frac{n}{2}}), y''_2 = A_1(\omega_{\frac{n}{2}}^2), \dots, y''_{\frac{n}{2}-1} = A_1(\omega_{\frac{n}{2}}^{\frac{n}{2}-1})$$

On va alors avoir (ligne 12) :  $\forall k \in [0, \frac{n}{2} - 1], y_k = y'_k + \omega_n^k y''_k$ , soit :

$$\forall k \in [0, \frac{n}{2} - 1], y_k = A_0(\omega_{\frac{n}{2}}^k) + \omega_n^k A_1(\omega_{\frac{n}{2}}^k)$$

De plus, on a vu que  $\omega_n^{\frac{n}{2}} = -1$ , donc :  $\forall k \in [0, n - 1], -\omega_n^{k-\frac{n}{2}} = -\omega_n^k \cdot \omega_n^{\frac{n}{2}} = \omega_n^k$ . D'où (ligne 13) :

$$\forall k \in [\frac{n}{2}, n - 1], y_k = A_0(\omega_{\frac{n}{2}}^k) - \omega_n^{k-\frac{n}{2}} A_1(\omega_{\frac{n}{2}}^k) = A_0(\omega_{\frac{n}{2}}^k) + \omega_n^k A_1(\omega_{\frac{n}{2}}^k)$$

On a donc :

$$\forall k \in [0, n - 1], y_k = A_0(\omega_{\frac{n}{2}}^k) + \omega_n^k A_1(\omega_{\frac{n}{2}}^k)$$

d'où, d'après le lemme de la bipartition :

$$\forall k \in [0, n - 1], y_k = A_0((\omega_n^k)^2) + \omega_n^k A_1((\omega_n^k)^2)$$

Soit, enfin, d'après (1) :

$$\forall k \in [0, n - 1], y_k = A(\omega_n^k)$$

Ainsi  $y = (y_0, \dots, y_{n-1})$  contient bien les  $n$  évaluations de  $A$  aux  $n$  racines complexes  $n$ -ièmes de l'unité. L'algorithme fonctionne donc bien au rang  $n + 1$ .

Nous avons ainsi montré par récurrence la validité de l'algorithme.

Estimons le temps  $T(n)$  que va demander l'algorithme pour traiter un polynôme  $A$  de longueur  $n$  (on considèrera le temps que demandent les additions et les soustractions négligeables devant celui que demandent les multiplications).

Celui-ci fait récursivement appel à lui-même pour traiter deux sous polynômes de longueur  $\frac{n}{2}$  ( $A_0$  et  $A_1$  : lignes 8 et 9). De plus, l'algorithme demande  $n$  multiplications (lignes 12 et 13). On négligera le temps demandé pour calculer les  $\omega_n^k$  (ligne 14).

On a donc :  $T(n) = 2T(\frac{n}{2}) + O(n)$ . Montrons que  $T(n) = O(n \ln n)$ .

Soit la suite  $(u_n)_{n \in \mathbb{N}}$  telle que  $\forall n \in \mathbb{N}, u_n = T(2^n)$ . On a alors :  $u_n = T(2^n) = 2T(2^{n-1}) + O(2^n) = 2u_{n-1} + O(2^n)$ .  $(u_n)$  est croissante donc  $\exists M > 0, u_n \leq 2u_{n-1} + 2^n M$ . Soit  $(v_n)_{n \in \mathbb{N}}$  telle que  $v_0 = u_0$  et  $\forall n \in \mathbb{N}^*, v_n = 2v_{n-1} + 2^n M$ . On a :  $\forall n \in \mathbb{N}, u_n \leq v_n$ .

De plus,  $(\frac{v_n}{2^n}) = (\frac{v_{n-1}}{2^{n-1}}) + M$ . La suite  $(\frac{v_n}{2^n})_{n \in \mathbb{N}}$  est donc arithmétique d'où :  $\forall n \in \mathbb{N}, \frac{v_n}{2^n} = v_0 + nM$ , soit :  $\forall n \in \mathbb{N}, v_n = 2^n(nM + v_0)$ .

Ainsi :  $\forall n \in \mathbb{N}, T(2^n) = u_n \leq v_n = 2^n(nM + v_0)$ . D'où, en posant  $n = \log_2 n' : T(n') \leq n'(\log_2 n' M + v_0)$ . Or  $n'v_0 = o(n' \log_2 n' M)$  donc, finalement,

$$T(n) = O(n \ln n)$$

### 3.2.7 La transformée inverse de Fourier

Elle permet d'interpoler aux racines complexes de l'unité en  $O(n \log n)$ .

La transformée inverse est notée  $a = TDF_n^{-1}(y) = V_n^{-1}y$  où la matrice inverse de Vandermonde est telle que :  $\forall (j, k) \in [0, n-1]^2, V_{n,(j,k)}^{-1} = \frac{\omega_n^{-jk}}{n}$ .

Soit  $(j, j') \in [0, n-1]^2$  alors on a :

$$[V_n \cdot V_n^{-1}]_{(j,j')} = \sum_{k=0}^{n-1} \frac{\omega_n^{-jk}}{n} \times \omega_n^{j'k} = \sum_{k=0}^{n-1} \frac{\omega_n^{k(j'-j)}}{n} = \delta_{jj'}$$

En effet, si  $j = j'$  alors  $[V_n \cdot V_n^{-1}]_{(j,j')} = \sum_{k=0}^{n-1} \frac{\omega_n^0}{n} = \frac{n}{n} = 1$  et si  $j \neq j'$  alors, d'après le lemme de la sommation,  $[V_n \cdot V_n^{-1}]_{(j,j')} = 0$ . D'où :  $V_n \cdot V_n^{-1} = I_n$ .

Donc, si  $a = TDF_n^{-1} = (a_0, \dots, a_{n-1})$  alors  $\forall j \in [0, n-1], a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}$ .

La formule étant similaire à celle de la transformée normale, il est facile de comprendre que, en modifiant un peu l'algorithme ci-dessus, on peut aussi effectuer la transformée inverse en  $O(n \log n)$ . Le temps total d'exécution de l'algorithme de multiplication de grands nombres par la transformée de Fourier sera donc lui aussi en  $O(n \ln n)$  :

```

1  FFTINV-RÉCURSIVE(y)
2  n ← LONGUEUR[y]
3  IF n=1 THEN RETURN y
4  ω'_n ← e-2iπ/n
5  ω ← 1
6  y' ← (y0, y2, ..., yn-2)
7  y'' ← (y1, y3, ..., yn-1)
8  a' ← FFTINV-RÉCURSIVE(y')
9  a'' ← FFTINV-RÉCURSIVE(y'')
10 FOR k FROM 0 TO (n/2 - 1)
11   DO
12     ak ← a'_k + ω a''_k
13     a(k+n/2) ← a'_k - ω a''_k
14     ω ← ω ω'_n
15   LOOP
16 a ← (a0/n, a1/n, ..., an-1/n)
17 RETURN a

```

**Remarque**  $\{\omega_n^k, k \in [0, n-1]\}$  étant isomorphe à  $\mathbb{Z}/n\mathbb{Z}$ , au lieu d'évaluer les polynômes aux racines complexes  $n$ -ièmes de l'unité, on peut les évaluer en  $1, 2, 3, \dots, n-1$  modulo  $n$ . Il s'est cependant avéré que l'utilisation des racines complexes  $n$ -ièmes de l'unité est plus rapide sur les ordinateurs actuels.

### 3.2.8 Déroulement d'une multiplication

Soit  $(A, B) \in (\mathbb{Z}[X])^2$ . On veut calculer  $C = A \cdot B$  à l'aide de la transformé de Fourier rapide.

$(A, B) \in (\mathbb{Z}[X])^2$  donc :  $\exists n \in \mathbb{N}, \exists a = (a_0, a_1, \dots, a_{n-1}) \in \mathbb{Z}^n, \exists b = (b_0, b_1, \dots, b_{n-1}) \in \mathbb{Z}^n, A = \sum_{k=0}^{n-1} a_k X^k$  et  $B = \sum_{k=0}^{n-1} b_k X^k$ . L'opération se déroule ainsi :

- On calcule  $y' = (y'_0, y'_1, \dots, y'_{n-1}) = TDF(a)$  et  $y'' = (y''_0, y''_1, \dots, y''_{n-1}) = TDF(b)$ . Cette opération se déroule en  $O(n \ln n)$ .
- On calcule  $y = (y_0, y_1, \dots, y_{n-1})$  tel que :  $\forall k \in [0, n-1], y_k = y'_k \cdot y''_k$ . Cette opération se déroule en  $O(n)$ .
- On calcule  $c = TDF^{-1}(y)$ . Cette opération se déroule en  $O(n \ln n)$ .

Et on a :  $c = \sum_{k=0}^n c_k X^k$  (pour la multiplication de nombres, il faut ensuite se ramener au représentant principal).  
 Le coût total de la multiplication est :  $2O(n \ln n) + O(n) = O(n \ln n)$  (car  $n = o(n \ln n)$ ), ce qui est bien plus rapide que la méthode de Hörner pour les grands nombres (il ne faut pas oublier que le temps est en  $(n \ln n)$  à une constante multiplicative près).

## 4 Factorisation

### 4.1 Par la différence de deux carrés

Soit  $N \in \mathbb{N}$ . Pour factoriser  $N$  par cette méthode, l'idée de départ est d'écrire  $N$  sous la forme d'une différence de deux carrés :  $N = x^2 - y^2 = (x + y)(x - y)$ . L'algorithme est le suivant :

- La première étape consiste à écrire  $N$  sous la forme  $N = x^2 - z$  avec la plus petite valeur possible pour  $x$ , à savoir  $x = E(\sqrt{N}) + 1$ . Posons  $n = E(\sqrt{N})$  et  $r = N - n^2$ . Les plus petites valeurs de  $z$  et  $x$  à tester sont donc :  $z_1 = x_1^2 - N = (n + 1)^2 - n^2 - r = 2n + 1 - r$  avec  $x_1 = n + 1$ . Si  $z_1$  est un carré, on a terminé :  $N = x_1^2 - z_1 = (x_1 + \sqrt{z_1})(x_1 - \sqrt{z_1})$  et on factorise récursivement les deux facteurs trouvés. Pour tester si  $z_1$  est un carré on pourra s'aider de calcul de congruences : on sait par exemple que, pour que  $z_1$  soit un carré, il faut que son dernier chiffre soit 1, 4, 5, 6 ou 9 ; il faut aussi que le nombre formé par ses deux derniers chiffres soit un carré ; etc.
- Soit  $m \in \mathbb{N}^*$ . On suppose qu'on a réalisé le test jusqu'à  $x_m = n + m$  et que pour tout  $i \in [1, m]$ ,  $z_i$  n'est pas un carré.
- On pose alors  $z_{m+1} = 2x_m + 1 + z_m$  et  $x_{m+1} = x_m + 1 = n + m + 1$ . Si  $z_{m+1}$  est un carré alors  $N = x_{m+1}^2 - z_{m+1} = (x_{m+1} + \sqrt{z_{m+1}})(x_{m+1} - \sqrt{z_{m+1}})$  on a terminé et on factorise récursivement les deux facteurs trouvés. Sinon, on continue.

Lucas a montré qu'un nombre différent de 2 est premier si et seulement si il s'écrit de façon unique comme différence de deux carrés. Donc dans le cas où  $a$  est premier, on aura donc forcément :  $x + y = N$ ,  $x - y = 1$  et  $x = \frac{N+1}{2}$ . Si  $N = ab$  (avec  $a > b$ ) est composé alors :  $x + y = a$ ,  $x - y = b$  et  $x = \frac{a+b}{2} = \frac{a+N/a}{2} < \frac{N+1}{2}$ . Donc si on a effectué les calculs jusqu'au rang  $m$  tel que  $x_m = \frac{N+1}{2}$  (donc  $m = \frac{N+1}{2} - n$ ) sans que  $z_i$  ( $i \in [1, m]$ ) soit un carré alors  $N$  est premier.

Le temps d'exécution de l'algorithme est en général très long. En effet, si  $N = x^2 - y^2$ , on pose  $a = x + y$  et  $b = x - y$  (alors  $N = ab$ ). Le nombre d'étapes de l'algorithme sera alors :  $x - \sqrt{N} = \frac{1}{2}(a + b) - \sqrt{N} = \frac{1}{2}(a + \frac{N}{a}) - \sqrt{N} = \frac{(a - \sqrt{N})^2}{2a}$ .

D'autres méthodes de factorisation existent notamment par les résidus quadratiques et les fractions continues.

## Références

- [1] *Introduction à l'algorithmique*, Thomas Cormen, Charles Leiserson, Ronald Rivest (1994) - Dunod
- [2] *Histoires d'algorithmes*, J.L. Chabert, ... (1995) - Belin
- [3] *TIPE - La Cryptographie*, Benoit Kloeckner, Guillaume Ménard, Jérôme Truffot - <http://www.multimania.com/menardg/crypto>
- [4] *TIPE - La Cryptographie*, Régis Décamps, Thomas Juès - <http://faq.maths.free.fr/tipe/crypto/cryptographie2.htm>
- [5] *Algorithme RSA*, Jean Berstel (1999)
- [6] *Théoreme des restes chinois* - <http://ybonnet.free.fr/tipe>
- [7] *Breaking the public key cryptosystems*, Maverick (1995) - <http://www.inu.net/maverick/contents.html>
- [8] *Arbitrary precision computation et FFT based multiplication of large numbers* - <http://xavier.gourdon.free.fr/Constants>