Fondements des protocoles cryptographiques

Cours de Fondement des protocoles de cryptographie,
dispensé par M^r Stéphane BALLET <stephane.ballet@univmed.fr>,
et M^r Alexis BONNECAZE <alexis.bonnecaze@univmed.fr>,
ESIL, département IRM option SICA

Table des matières

Table des codes sources

Première partie

Complexité et temps de calcul

La théorie de la complexité a pour but de classifier le nombre de calcul en assignant d'éventuel ressources nécessaires pour les résoudre. Cette classification essai de mesurer la difficulté intrinsèque (essentielle) d'un problème. Les ressources que l'on évalues sont essentiellement :

- Le temps de calcul ;
- L'espace de stockage nécessaire.

1 Représentation des entiers

Notation de la entière (*floor* et *ceiling*) :

$$\lfloor 3,14 \rfloor = 3 \qquad \lfloor -\pi \rfloor = 4 \qquad \lceil \pi \rceil = 4 \qquad \lceil -\pi \rceil = -4$$

Plus généralement, les entiers peuvent être représentés en base B où B est un entier ≥ 2.

théorème 1.1 Soit $B \geq 2$. À tout entier $a > 0$ correspond un entier positif k et une unique suite $(a_k, a_{k-1}, \ldots, a_1, a_0)^{k+1}$ dans $0, 1, \ldots, B-1$ où $a = \sum_{j=0}^{k} a_j.B_j$

En outre :

$$k = L\log_B(a) \qquad a_k = \lfloor \tfrac{a}{B^k} \rfloor \qquad a_j = \lfloor \tfrac{(a - \sum_{i=j+1}^{k} a_i B^i)}{B^j} \rfloor$$

Pour $0 \leq j \leq k-1$

La suite $(a_k, \ldots, a_1, a_0)$ est le *développement B-antique* de a ou développement en base B. La longueur de a en base B est :

$$L_B(a) = k + 1 = \lfloor \log_B(a) \rfloor + 1 = \lceil \log_B(a) \rceil$$

Si $B = 2$, on obtient le développement binaire de a.
Si $B = 16$, on obtient le développement hexadécimal de a.
Si $B = 2$, la longueur de a est aussi appelé longueur binaire ou taille de a, et on pose :

$$L(a) = L_2(a) = \lfloor \log_2(a) \rfloor + r$$

Lorsqu'on a calculé le développement B-antique de l'entier a, on écrit :

$$a = (a_k, a_{k-1}, \ldots, a_1, a_0)$$

Exemple :
En base 16, $DADA = 14 \times 16^2 + 14 \times 16 + 10$

La transformation de la base 2 à la base 16 est particulièrement simple :

$$(10111000)_2 = (B8)_{16}$$

2 Complexité de l'arithmétique

2.1 la notation O

Déf. 1.1 Soient $f : \mathbb{N} \to \mathbb{R}^+$ et $g :\to \mathbb{R}^+$ deux fonctions positives, on écrit $f = O(y)$ s'il existe un entier $M > 0$ et une constante $C > 0$ tel que tout entier $n \geq M$, on ait :

$$f(n) \leq C.g(n)$$

Lorsque g est constante, on écrit $O(1)$
Plus généralement si $f : \mathbb{N}^2 \to \mathbb{R}^+$, $g : \mathbb{N}^2 \to \mathbb{R}^+$, on écrit $f = O(g)$ s'il existe un entier $M > 0$ et une constante $C > 0$ tel que pour tout n-upplet $(n_1, \ldots, n_L) \in \mathbb{N}^2$ vérifiant $n_j \geq M$ pour $j = 1, \ldots, L$; on ait :

$$f(n_1, \ldots, n_L) \leq C.g(n_1, \ldots, n_L)$$

Cette notation est utilisé lors de l'évaluation de la complexité (temps de calcul, nombres d'instructions élémentaires d'un algo).

Exemple :
Si L(n) désigne la longueur B-adique de l'entier n, on a :

$$L(n) = O(\log n)$$

En effet :

$$L(n) = \lfloor \log_B(n) \rfloor + r \leq \log_B(n) + r = \frac{\log(n)}{\log(B)} + 1$$

si $n \geq 3$, $\log(n) > 1$ et donc $\forall n \geq 3 = M$:

$$L(n) \leq (\frac{1}{\log B} + r) \log(B) \text{ avec } C = \frac{1}{\log B} + r$$

2.2 Coût de l'addition, de la multiplication et de la division avec reste des entiers

Soient a et b, deux entiers dont les longueurs binaires sont $m = L(a)$ et $n = L(b)$. On cherche à évaluer le nombre d'opérations sur les bits nécessaires pour les quatre opérations.

On commence par des exemples :

2.2.1 Addition

On suppose que l'addition de deux bits prend un temps $O(1)$. Pour calculer $a + b$, il faut un temps $O(max(m, n))$.

Exemple : $a = (100101)_2 = (37)_{10}$
$b = (1101)_2 = (13)_{10}$

Pour calculer $a + b = (110010)_2 = (50)_{10}$:

	1	0	0	1	0	1
+			1	1	0	1
Retenue		1	1		1	
$a + b$	1	1	0	0	1	0

Pour calculer $a - b$,
même résultat $O(\log n)$.

2.2.2 Multiplication

Pour calculer $a \times b$, il faut un temps $O((m \times n))$.

Exemple : $a \times b = (111100001)_2 = (481)_{10}$

					1	0	0	1	0	1
$\times$							1	1	0	1
					1	0	0	1	0	1
				0	0	0	0	0	0	0
			1	0	0	1	0	1	0	0
		1	0	0	1	0	1	0	0	0
Retenue				1	1	1	1			
		1	1	1	1	0	0	0	0	1

Il existe des algorithmes plus rapides, comme *Schönhage - Strassen*. Ils permettent de multiplier deux nombres de n bits en un temps :

$$O(n \log n \log(\log n))$$

Les algorithmes sont totalement inefficaces en pratique, ils deviennent performants pour des nombres de plus de 10 000 bits.

2.2.3 Division

Soit $a = bq + r$ où $0 \leq r < b$, la division euclidienne de a par b. Soit $L(q)$ la longueur binaire de quotient q et $n = L(b)$. Pour calculer $a \div b$ il faut un temps $O(nL(q)) = \Theta(L(b).L(q))$

2.2.4 Résumé

Soit a et b, deux entiers et $L(a)$ et $L(b)$ leurs longueurs binaires.
– L'addition $a + b$et la soustraction prennent un temps $O(max(L(a), L(b)))$
– La division $a = bq + r$ prend un temps en $O(L(b).L(q))$ où q est le quotient.
Tous ces algorithmes utilisent une place de taille $O(L(a) + L(b))$.

2.3 Complexité des opérations

Après cette 1reapproche empirique, on va pouvoir *chiffrer le nombre d'opérations nécessaires* pour accomplir les quatres opérations de l'arithmétique. Les opérations élémentaires que l'on vient d'effectuer sont l'addition, la soustraction, la multiplication de deux nombres de 1 bit et la division d'un nombre de 2 bits par un nombre de 1 bit. Les opérations se font *bit à bit*.

Théorème de la complexité de l'arithmétique On peut additionner ou soustraire deux entiers de L bits en $O(L)$ opérations bit à bit. On peut multiplier deux entiers de L bits par un entier de L bits et avoir un quotient de L bits et un reste de L bits en $O(L^2)$ opérations bit à bit.

Preuve On écrit les opérations élémentaires en base B. On se donne deux entiers a et b, supposons déjà écrits en base B.

$$a = \sum_{j=0}^{k-1} a_j.B^j \text{ et } b = \sum_{j=0}^{k-1} b_j.B^j (0 \leq a_j, b_j \leq B - 1)$$

On écrit en pseudo-code les algorithmes permettant d'effectuer les opérations élémentaires de l'arithmétique.

2.3.1 Addition

Listing 1: $s = a + b$ **en base** B.

```
//Input: Le dev. de a et b en base B.
//Output: Le dev. en base B de s = a + b.

retenue = 0;

for (i=0 to k-1)
{
  Si = ai + bi + retenue;
  if (Si < B)
  {
    retenue = 0;
  }
  else
  {
    retenue = 1;
    Si = Si - B;
  }
}
Sk = retenue;
```

2.3.2 Soustraction

On suppose $a > b$. Posons $t = a - b = \sum_{j=0}^{k-1} t_j.B^j$

L'algorithme s'écrit :

Listing 2: $t = a - b$ **en base** B $(a \geq b)$

```
//Input: Le dev. de a et b en base B (a >= b).
//Output: Le dev. en base B de t = a - b.

retenue = 0;

for (i=0 to k-1)
{
  Ti = ai - bi - retenue;
  if (Ti < 0)
  {
    retenue = 1;
    Ti - Ti + B;
  }
  else
  {
    retenue = 0;
  }
}
```

2.3.3 Multiplication

$$a = \sum_{j=0}^{k-1} a_j.B^j \text{ et } b = \sum_{j=0}^{m-1} b_j.B^j$$

$$p = a \times b = \sum_{i=0}^{k \times m-1} p_i.B^i$$

Listing 3: $p = a \times b$

```
//Input: Dev. de a et b en base B
//Output: Dev. de p = a * b en base B

retenue = 0;

for (i = 0 to k + m - 1)
{
  pi = 0;
}
for (i = 0 to k - 1)
{
  retenue = 0;
  for (j = 0 to m - 1)
  {
    y = ai * bj + P(i + j) + retenue;
    P(i + j) = y mod B;
    retenue = partieEntiere(Y/B);
    P(i + m + 1) = retenue;
  }
}
```

La boucle intérieure est exécutée $m \times k$ fois.
La variable temporaire y vérifie $0 \leq y \leq B^2$, ceci se prouve par récurrence.

En effet, $0 \leq a_i \leq B-1, 0 \leq b_i \leq B-1$ et $\begin{cases} 0 \leq & \text{retenue} & \leq B-1 \\ 0 \leq & P_{i+j} & \leq B-1 \end{cases}$

Il resterait à expliquer comment dans cet algorithme, on effectue la division de la variable y par B_i. Mais dans le cas qui nous intéresse, $B = 2$ et comme la variable $y < 4$, cette division est évidente à écrire.

3 Complexité d'un algorithme

3.1 Temps de calcul

En cryptologie, lorsqu'on étudie un algorithme, on a souvent besoin d'évaluer :
- le nombre d'opérations nécessaires à l'exécution d'un algorithme, c'est-a-dire le temps de calcul nécessaire pour obtenir un résultat ;
- la place à allouer en mémoire pour pouvoir exécuter l'algorithme.

Ici, cet algorithme peut être soit un algorithme permettant d'exécuter une cryptographique (chiffrement, déchiffrement, signature, vérification de signature, hachage, génération de clé aléatoire, etc.), soit un algorithme permettant d'attaquer une fonction cryptographique préalablement cryptanalysée.

3.1.1 Définition du temps de calcul

Le temps de calcul d'un algorithme pour un entrée particulière est le nombre d'opérations primitives nécessaires pour obtenir la sortie.
Le plus souvent une opération primitive sera une opération bit à bit. Pour un certain nombre d'algorithme il sera plus commode d'utiliser comme opération primitive une opération plus complexe comme une multiplication modulaire, une opération de groupe ou une instruction machine.

3.1.2 Définition du temps de calcul le pire

Le temps de calcul le pire d'un algorithme est la borne supérieure sur toutes les entrées possibles ayant une taille donnée des temps de calcul.

Le temps de calcul moyen d'un algorithme est la moyenne des temps de calcul sur toute les données ayant une taille donnée.

Ces deux temps de calcul s'exprimeront comme un fonction de la taille de l'entrée. Pour manipuler facilement les estimations de temps de calcul, la notation O est commode.

3.2 Temps polynomial

Lors de l'analyse d'un algorithme cryptographique, on doit montrer qu'il est efficient, c'est-a-dire qu'il s'exécute facilement. Parfois c'est le contraire, on veut prouver qu'un algorithme est difficile à exécuter (comme par exemple, fonction *trap*, facile à exécuter dans un sens mais difficile à exécuter en sens inverse). On précise ici la notion d'algorithme efficient. Le cas le plus fréquent sera celui où l'algorithme prend en entrée une seule donnée.

3.2.1 Algorithme polynomial

On dit qu'un algorithme est polynomial si son temps de calcul le pire $T = O(L^a)$ où L est la taille de l'entrée et a une constante. Autrement dit, un algorithme est polynomial s'il existe $a > 0$, $C > 0$, tel que pour toute entrée de L bits, l'algorithme termine son exécution en moins de $C \times L^a$ opérations bit à bit. Ceci se généralise dans le cas d'un algorithme prenant en entrée plusieurs variables.

Définition Supposons que les algorithmes reçoivent en entrée les entiers A, B et C, de longueur binaire $L(A)$, $L(B)$ et $L(C)$. On dirait que cet algorithme s'exécute en un temps polynomial s'il existe des réels non négatifs a, b et c, tels que le temps de calcul le pire $T = T(a, b, c)$ de l'algorithme, c'est-a-dire le nombre d'opérations bit à bit, pour exécuter l'algorithme soit de la forme $T = O(L(A)^a, L(B)^b, L(C)^c)$.

Un algorithme est donc polynomial ou efficient s'il s'exécute en temps polynomial. Dans la pratique pour qu'un algorithme soit efficient, il faudrait qu'il soit polynomial et que les exposants a, b, c et la constante C du O soient petits.

On a vu dans la section 2.3 (*Complexité des opérations*) que les quatre opérations de l'arithmétique sont des algorithmes polynomiaux. Quelques exemples ci-dessous.

Algorithme carré et multiplication (*Square and Multiply*)

Soit G un groupe, $x \in G$, $a \geq 1$ et m entier. Cherchons à évaluer x^a et à calculer le coût T de cette exponentiation.

L'opération *élévation à la puissance* est très fréquente en cryptographie à clé publique et dans des groupes G très variés. Dans des applications concrètes, implémentées dans des systèmes existants où le groupe $G = (\frac{\mathbb{Z}}{n\mathbb{Z}})^*$, l'exposant a est énorme. Il peut avoir une longueur binaire allant jusqu'à 4096 bits (aux alentours de l'année 2003). Donc a est équivalent à 2^{4096}, l'entier m ayant aussi cette taille. Dans des cartes à puces, la longueure binaire de a peut être supérieure à 1024. Il est donc nécessaire d'avoir un algorithme permettant d'effectuer le calcul de x^a en temps raisonnable.

On suppose que les éléments de G sont codés par des nombres binaires de longueur L_G. La mutliplication dans G est supposée polynomiale :
$\exists k \geq 0$ tel que le nombre σ d'opérations bit à bit nécessaires pour calculer le produit de deux éléments de G vérifie $\sigma = O(L_G^k)$

Square and Multiply ou méthode binaire (cf. algorithme de *Horner*) permet de faire le calcul de x^a simplement. Pour effectuer ce calcul on va se ramener à deux opérations qui sont des multiplications dans G :

– élévation à la puissance : $y \leftarrow y^2$;

– multiplication par $x : y \leftarrow xy$.

L'ordre dans lequel on effectue ces deux opérations dépend très simplement de la décomposition binaire de a.

Exemple On commence par les bits de gauche de l'exposant et on ignore le premier. Chaque fois qu'il y à un 0 on élève au carré et chaque fois qu'il y a un 1 on élève au carré et on multiplie par x.

x	x^2	x^4	$x \times (x^4)^2 = x^9$	x^{19}	x^{38}	x^{77}
1	0	0	1	1	0	1

Avec $x^{99} \rightarrow 99 = (1100011)_2$:

x	x^3	x^6	x^{12}	x^{24}	x^{49}	x^{99}
1	1	0	0	0	1	1

Il faut prouver que dans le cas général on obtient x^a. Écrivons la décomposition binaire de a : $a = \sum_{j=0}^{N-1} b_j 2^j$ où les b_j sont des bits.

Raisonnement par récurrence sur le nombre de bits N de a

Si $N = 1$, c'est vrai.

Supposons prouvé, qu'en utilisant les deux opérations ci-dessus dans l'ordre précisé par la règle, on ait obtenu x^a. Soit b un nombre de $N+1$ bits : $b = (b_N \times 2^{N-1} + b_{N-1} \times 2^{N-2} + \cdots + b_1) \times 2 + b_0$. $b = 2a + b_0$

L'application de l'algorithme sur les N bits de gauche de b donnera x^a.
En applicant encore une fois la règle, on obtiendra si $b_0 = 0$, $((x^a)^2$ et si $b_0 = 0$, $x \times (x^a)^2 = x^b$.
Avec ces notations, le nombre $Mult(a)$ de multiplications pour calculer x^a par l'algorithme *Square and Multiply* est $Mult(a) = \sum_{i=0}^{N-2}(b_i + 1)$.

– Si $b_i = 0$, $Mult(a) = 1$;
– Si $b_i = 1$, $Mult(a) = 2$.

Notons $L(a) = N$ la longueur binaire de a. On voit que pour tout $a > 1$, on a $Mult(a) \leq 2L(a)$. On a donc prouvé que $Mult(a) = 0(L(a))$.
L'opération pour mesurer la complexité de l'algorithme est l'opération de groupe dans G.

Remarques

- On remarquera que la place mémoire nécessitée par l'algorithme est négligeable car on stocke seulement la valeur initiale de x et la variable de la constante y.
- – En revenant à la taille des éléments de G, on voit que la complexité de x^a dans G s'écrit : $T = O(L(a) \times (L_G)^2)$. Dans le cas où $G = (\frac{\mathbb{Z}}{n\mathbb{Z}})^*$ avec $k = 2$ et en supposant, ce qui est vrai en cryptographie, que la taille de a est de l'ordre de celle de n, on voit que le coût de l'exponentiation modulaire est $T = O(L(n)^3)$.

Opérations bit à bit Les opérations que l'on a rencontrées précédemment étaient l'addition, la soustraction, la multiplication de deux nombres de un bit ou la division d'un nombre de deux bit par un nombre de un bit. C'est ce que l'on a nommé *opération bit à bit*.

Coût de la multiplication modulo n Le théorème de complexité de l'arithmétique permet d'estimer le nombre d'opérations bit à bit pour effectuer une multiplication modulaire c'est-a-dire un produit dans $\frac{\mathbb{Z}}{n\mathbb{Z}}$.

Corollaire Soit n un entier positif de longueur binaire $L(n)$ et a, b des éléments de $0, \ldots, n-1$. On peut calculer le nombre $c = (a \times b) \mod (n)$ où $c \in 0, 1, \ldots, n-1$ en $O(L(n)^2)$.

Opération bit à bit (preuve du corollaire) En effet, on commence par calculer le produit $z = a \times b$, ce qui nécessitera $O(L(n)^2)$ opérations. Le nombre z a $2 \times L(n)$ bits. Ensuite on divise z par n pour obtenir le reste c. La division nécessite aussi $O(L(n)^2)$ opérations.

- L'algorithme *Square and Multiply* n'est pas toujours le meilleur algorithme en ce qui concerne le nombre de multiplication. Par exemple : $a = 85 = (1010101)_2$
  L'algorithme donne :
  $x \to x^2 \to x^5 \to x^{10} \to x^{21} \to x^{42} \to x^{85}$. Le nombre de multiplications est 9.
  Si l'on suppose que l'on conserve la valeur x^5 après trois multiplications on pourrait effectuer le calcul de la manière suivante en effectuant seulement huit multiplications :
  $x \to x^2 \to x^5 \to x^{10} \to x^{20} \to x^{40} \to x^{80} \to x^{80} \times x^5 = x^{85}$
  On remarquera qu'ici, comme dans le cas de l'algorithme *Square and Multiply*, on a utilisé que deux registres.
- Si $G = (\frac{\mathbb{Z}}{n\mathbb{Z}})^*$, avec l'algorithme *Square and Multiply* le calcul de $x^a \mod (n)$ s'écrira :

Listing 4: Pseudo code de l'algorithme *Square and Multiply*.

```
y = x;

for (i = N-1 to 0)
{
  y = y * y mod(n);
  if (bi = 1) then y = y x mod(n);
}
```

Exemple d'algorithme non polynomial

Soit n un entier impaire non premier. L'entier n est le produit de deux nombres impairs u et v et on peut écrire : $n = u \times v = (\frac{1}{2} \times (u+v))^2 - (\frac{1}{2} \times (u-v))^2 = a^2 - b^2$. La *méthode de factorisation de Fermat* consiste donc étant donné un entier n impair, à chercher à mettre n sous la forme de différence de deux carrés.
Pour formaliser cette méthode et la transformer en algorithme, on effectue des essais pour trouver le nombre a sur la suite $\lceil \sqrt{n} \rceil, \lceil \sqrt{n} \rceil + 1, \ldots$ et on examine si $a^2 - n$ est un carré. Si c'est le cas, soit b^2 ce carré. Alors on a $n = a^2 - b^2 = (a-b)(a+b)$. Si n est un entier composé impair, cette procédure se termine avec une factorisation non triviale avant d'avoir atteint $a = \lfloor \frac{n+9}{6} \rfloor$.
En effet le cas pire (factorisation contenant le plus gros facteur) apparait lorsque $n = 3p$ où p est un nombre premier impair. Dans ce cas, la factorisation non triviale de n est obtenue avec comme entier : $a = \frac{3+p}{2}$ et $p = 3$ donc $a = \frac{3+\frac{n}{3}}{2} = \frac{n+9}{6}$
Et $b = \frac{n-9}{6}$

Méthode de Fermat Etant donné un entier impaire $n > 1$. Cet algorithme trouve un facteur non trivial de n ou démontre que n est premier.

Listing 5: Pseudo code de l'algorithme de Fermat

```
for(floor(sqrt(n))<=a<=(n+9)/6) do{
  if(b=sqrt(a*a-n) est entier) alors retourne a-b;
}
```

```
retourne "n est premier";
```

Si on suppose que l'on est dans le cas le pire, le nombre d'itérations N_{It} dans la boucle sera $N_{It} = \frac{n+9}{6} - \lceil \sqrt{n} + 1 \rceil \leq \frac{n}{6}$.
Dans chaque boucle on effectue une multiplication $a \rightarrow a^2$ et on examine si $a^2 - n$ est le carré d'un entier en calculant la partie entière de la racine par l'algorithme de Newton.
L'algorithme de Newton se termine au bout de $\ln(n)$ itérations.
À chacune de ces itérations on effectue une division, une addition et une division par 2. Si on note $F(n)$ le nombre d'opérations nécessaires à l'exécution de l'algorithme, on voit que l'on pourrait écrire $F(n) = O(\ln(\ln(n)) \times n)$.
Si $L(n) = \lceil \frac{\ln(n)}{\ln(2)} \rceil$ est la longueur binaire de n, on obtient : $F(n) = O(\ln(L(n)) \times e^{\ln(2) \times L(n)})$.
La méthode de Fermat est donc exponentielle en le nombre de bits du nombre impaire à factoriser. Cette méthode n'est performante que dans le cas où l'on sait que les deux facteurs u, v sont voisins de $\sqrt{n}$.

Factorisation d'un entier

Théorème fondamental de l'arithmétique Tout entier n s'écrit de manière unique, à une permutation près, de l'ordre des facteurs sous forme d'un produit de nombres premiers.
Si $n = p_1, p_2, \ldots, p_r = q_1, q_2, \ldots, q_s$ où les nombres p_j et q_k sont des nombres premiers, alors $r = s$ et tout indice j compris entre 1 et r, il existe k tel que $p_j = q_k$.

L'entier n s'écrira : $n \prod_{j=1}^{k} p_j m_j$ où les p_j sont des nombres premiers deux à deux distincts et les entiers $m_j > 0$.
Le problème de la factorisation s'énonce :
Étant donné un entier $m > 1$, trouver les nombres premiers p_j et les entiers m_j. Le problème de la factorisation est à l'heure actuelle un problème central en cryptographie à clé publique. Tous les algorithmes de factorisation connus ont une complexité prouvé ou heuristique non polynomiale en la longueur binaire de l'entier n à factoriser.

En 2003 on sait factoriser en un temps raisonnable des nombres de 512 bits, c'est-a-dire des nombres ayant 154 chiffres dans le système décimal. Mais un tel calcul nécessite de grosses ressources informatiques. Le 11 octobre 1988, *Mark Manasse* et *Arjen Lenstra*, deux factoriseurs de renom, avaient réussi à factoriser le nombre $11^{104} + 1$ qui à 360 bits en utilisant des centaines d'ordinateurs pendant 31 semaines. En 30 ans, 150 bits ont été gagnés.

4 Plus grand commun diviseur

4.1 Rappel

Théorème Soient a et b deux entiers non simultanément nuls. Alors parmi les diviseurs communs de a et b, il existe exactement un plus grand élément appelé le plus grand commun diviseur de a et b et noté $\text{pgcd}(a, b)$.

Soit $d = \text{pgcd}(a, b)$. Il existe des entiers u et v tels que $d = ua + vb$ (Égalité de Bezout).

Définition Deux nombres a et b sont *premiers entre eux* lorsque $\text{pgcd}(a, b) = 1$.

Soient a et b deux nombres positifs premiers entre eux. Il existe alors un couple unique d'entiers positifs ou nuls u et v tels que $ua - vb = 1, u < b$ et $v < a$.

Les autres solutions de l'équation $ax - by = 1$ sont de la forme $x = u + kb, y = v + ka$ où k est un entier relatif.

4.2 Complexité de l'algorithme d'Euclide

L'algorithme d'Euclide permet d'obtenir le plus grand commun diviseur de deux entiers très rapidement. Sa complexité polynomiale a été prouvée par Lamé au milieu du 19^{e}siècle. Cet algorithme a une importance fondamentale en cryptographie à clef publique.

Soient a et b deux entiers positifs tels que $b < a$. Posons $r_{-1} = a$ et $r_0 = b$. L'algorithme d'Euclide consiste à effectuer la suite de divisions "euclidiennes" ci-dessous :

$$\begin{array}{rcll} r_{-1} & = & q_1 r_0 + r_1 & 0 < r_1 < r_0 \\ r_0 & = & q_2 r_1 + r_2 & 0 < r_2 < r_1 \\ r_1 & = & q_3 r_2 + r_3 & 0 < r_3 < r_2 \\ \vdots & & & \\ r_{n-2} & = & q_n r_{n-1} + r_n & 0 < r_n < r_{n-1} \\ r_{n-1} & = & q_{n+1} r_n & \end{array}$$

Le dernier reste non nul r_n est alors le pgcd cherché :

$$\text{pgcd}(a, b) = r_n$$

Exemple Calcul du $\text{pgcd}(16983, 7667) = 17$:

$$\begin{array}{rcrcr} 16983 & = & 2.7667 & + & 1649 \\ 7667 & = & 4.1649 & + & 1071 \\ 16983 & = & 1.1071 & + & 578 \\ 16983 & = & 1.578 & + & 493 \\ 16983 & = & 1.493 & + & 85 \\ 16983 & = & 5.85 & + & 68 \\ 16983 & = & 1.68 & + & 17 \\ 16983 & = & 4.17 & + & 0 \end{array}$$

Dénombrement du nombre d'opérations élémentaires de l'algorithme d'Euclide On se propose ici d'évaluer le nombre maximal de divisions nécessaires pour calculer le pgcd de deux entiers par l'algorithme d'Euclide. Ce résultat a été obtenu par le mathématicien français *G. Lamé* au 19^{e}siècle.

Notation $\lfloor x \rfloor$ désigne la partie entière du réel x et R le nombre d'or : $R = \frac{(1+\sqrt{5})}{2}$.

Théorème de Lamé Soient a et b deux entiers tels que $2 \leq b \leq a$. Le nombre de divisions nécessaires pour calculer le $\text{pgcd}(a, b)$ est au plus égal à $\lfloor \frac{\ln(a)}{\ln(R)} \rfloor + 1$.

Démonstration L'algorithme d'Euclide consiste à poser $r_{-1} = a$ et $r_0 = b$ et à effectuer ensuite une suite de divisions entre deux restes consécutifs :

$$r_{i-2} = q_i r_{i-1} + r_1$$

pour $i = 1, 2, \ldots, n, n+1$ avec $r_i < r_{i-1}$.
Le *dernier reste non nul* $r_n = \text{pgcd}(a, b)$. On a donc effectué $n + 1$ divisions et on cherche à évaluer ce nombre de divisions dans "le cas le pire" en fonction de la taille de a. Cette démonstration fait intervenir la suite de *Fibonacci* (F_n) qui est définie par récurrence par :
$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ si $n \geq 2$
Le polynôme caractèristique de cette suite est $X^2 - X - 1$, dont une des racine est R. Ce qui implique que $R^2 = R + 1$.

Les étapes de la démonstration du théorème de Lamé sont les suivantes :

a) Pour tout $n > 1$, $F_n \geq R^{n-2}$. On raisonne par récurrence sur n. Si $n = 2$ c'est vrai. En effet, $F_2 = 1 \geq R_0$.
Supposons $F_n \geq R^2$, alors : $F_{n+1} = F_n + F_{n-1} \geq R^{n-2} + R^{n-3} = R^{n-3} \times (R+1) = R^{n-3}R^2 = R^{n-1}$.

b) On vérifie par récurrence que pour $i = n, n-1, \ldots, 0, -1$ on a $r_i \geq F_{n+1-i}$. Si $i = n$, c'est vrai puisque $F_1 = 1$ et $r_n > 0$.
Supposons $r_i \geq F_{n+1+i}$, on a : $r_{i-1} = q_{i+1}r_i + r_{i+1} \geq r_i + r_{i+1} \geq F_{n+1+i} + F_{n-i} = F_{n+2-i} = F_{n+1-(i-1)}$.
On peut alors conclure : $a = r_{-1} \geq F_{n+2} \geq R^n$. Soit $n \leq \lfloor \frac{\ln(a)}{\ln(R)} \rfloor$ et comme il y avait $n+1$ divisions :

$nbrDivisionPGCD(a,b) \leq \lfloor \frac{\ln(a)}{\ln(R)} \rfloor + 1$.

Note Le cas le pire sera atteint lorsque $a = F_n$ et $b = F_{n-1}$.

Corollaire Le nombre d'opérations bit à bit pour obtenir le plus grand commun diviseur de deux nombres par l'algorithme d'Euclide est $O(L^3)$ où L est le nombre de bits du plus grand de ces nombres.
En effet, soient a et b ces nombres ($a \geq b$) et $L = L(a) = \lceil \log_2(a) \rceil$ la longueur binaire de a. On devra effectuer au plus $C' \times L$ divisions (théorème de Lamé), chacune nécessitant au plus $C''L^2$ opérations bit à bit.
On peut prouver un mieux.

Théorème 4.2.1 Soient a et b deux entiers de longueur binaire $L(a)$ et $L(b)$. L'algorithme d'Euclide permet d'obtenir le plus grand commun diviseur de ces deux nombres en $O(L(a) \times L(b))$ opérations bit à bit.

Démonstration Nous effectuons l'algorithme d'Euclide sur a et b avec $a \geq b$. Commençons par prouver que le produit des quotients de l'algorithme d'Euclide est majoré par :
$q_1, q_2, \ldots, q_{n+1} \leq a$.

À chaque étape de l'algorithme on efectue une division de la forme $r_{i-1} = q_{i+1}r_i + r_{i+1}$ où l'entier $i = 0, \ldots, n$ et donc $q_{i+1}r_i \leq r_{i-1}$. En multipliant ces $n+1$ inégalités, on obtient :
$q_1.r_0.q_2.r_1 \ldots q_{n+1}.r_n \leq r_{-1}.r_0.r_1 \ldots r_{n-1}$.

En divisant par $r_0.r_1 \ldots r_{n-1}$, il vient $q_1.q_2 \ldots q_{n+1}.r_n \leq r_{-1} = a$, ce qui implique l'inégalité cherchée.
La longueur binaire de l'entier x est noté $L(x)$. Lors de l'exécution de l'algorithme, pour $i = 0, \ldots, n$, chaque division $r_{i-1} = q_{i+1}r_i + r_{i+1}$ nécessite $O(L(q_{i+1})L(r_i))$ opérations bit à bit. La complexité totale sera :

$$T = C\sum_{i=0}^{n} L(q_{i+1})L(r_i) \leq CL(b).\sum_{i=0}^{n} L(q_{i+1}) \leq CL(b).\sum_{i=0}^{n}(\log_2(q_{i+1}+1))$$

$$T \leq CL(b).[n+1+\log_2(q_1.q_2\ldots.q_{n+1})] \leq CL(b).[n+1+\log_2(a)]$$

Le théorème de Lamé nous dit que $n = O(L(a))$ et $\log_2(a) \leq L(a)$, on peut conclure.
Complexité moyenne de l'algorithme d'Euclide. On démontre le théorème suivant.

Théorème Soient $a > b$ des entiers dans l'intervalle $[1, N]$. Alors le nombre moyen d'itérations de l'algorithme d'Euclide (sur tous les choix de a et b) est asymptotiquement équivalent à $\frac{12 \times \ln(2)}{\Pi^2} \times \ln(N)$.

En notant $a \mod b$ le reste de la division de a par b, l'algorithme d'Euclide s'écrit :

Listing 6: Pseudo code de l'algorithme d'Euclide

```
//Input: Deux entiers a >= b >= 0.
//Output: d = pgcd(a, b).

while (b != 0)
{
  r = a mod b;
  a = b;
  b = r;
}

return a;
```

La version récursive d'Euclide s'écrit :

Listing 7: Pseudo code de l'algorithme d'Euclide récursif

```
// Euclide(a, b)

if (b = 0)
{
  return a;
}
else
{
  return Euclide(b, a mod b);
}
```

4.3 Algorithme d'Euclide étendu

Dans les applications, on a souvent besoin d'avoir les entiers u et v introduits ci-dessous :

$$d = \text{pgcd}(a, b) = u \times a + v \times b$$

La méthode naïve consiste à exécuter l'algorithme d'Euclide en sauvant toutes les étapes et à "remonter" ensuite jusqu'au début en éliminant les restes successifs.

Exemple $a = 949, b = 767$

$$\begin{array}{lllll} 949 = 767 + 182 & & 21 \times 949 = 26 \times 767 - 13 \\ 767 = 4 \times 182 + 39 & & 5 \times 767 = 21 \times 182 + 13 \\ 182 = 4 \times 39 + 26 & & 182 = 5 \times 39 - 13 \\ 39 = 26 + 13 & & 39 = 26 + 13 \\ 26 = 2 \times 13 \end{array}$$

On a obtenue $\text{pgcd}(949, 767) = 13$ et la relation de Bezout :
$(-21) \times 949 + 26\times = 13 : u = -21, v = 26$

4.3.1 Première version de l'algorithme d'Euclide étendu

Nous allons exhiber un algorithme permettant de calculer les entiers u et v en même temps que le pgcd de a et de b. Nous allons évaluer ensuite la complexité de cet algorithme.

Dans l'algorithme d'Euclide la suite des quotients et des restes est calculée à partir de $r_{-1} = a$ et $r_0 = b$ par division des restes successifs : $r_{i-2} = q_i r_{i-1} + r_i$ pour $i = 1, 2, \ldots, n+1$.

Écrivons ces relations sous la forme matricielle :

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}, \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} q_2 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}, \ldots, \begin{pmatrix} r_{i-2} \\ r_{i-1} \end{pmatrix} = \begin{pmatrix} q_i & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix}, \ldots,$$
$$\begin{pmatrix} r_{n-2} \\ r_{n-1} \end{pmatrix} = \begin{pmatrix} q_n & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_{n-1} \\ r_n \end{pmatrix}, \begin{pmatrix} r_{n-1} \\ r_n \end{pmatrix} = \begin{pmatrix} q_{n+1} & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_n \\ r_{n+1} \end{pmatrix}$$

Ici $r_{n+1} = 0$ et $d = \text{pgcd}(a, b) = r_n$, donc :

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_2 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_{n+1} & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} d \\ 0 \end{pmatrix}$$

Posons pour $i = 1, \ldots, n+1$:

$$Q_i = \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_i & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} \alpha_i & \beta_i \\ \gamma_i & \sigma_i \end{pmatrix}, R_i = \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix}$$

Pour $i = 1, \ldots, n+1$, nous avons $\begin{pmatrix} a \\ b \end{pmatrix} = Q_i R_i$.
Nous obtenons en remarquant que $det(Q_i) = (-1)^i$:

$$\begin{pmatrix} d \\ 0 \end{pmatrix} = (Q_{n+1})^{-1} \times \begin{pmatrix} a \\ b \end{pmatrix} = (-1)^{n+1} \begin{pmatrix} \sigma_{n+1} & -\beta_{n+1} \\ -\gamma_{n+1} & \alpha_{n+1} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

Ceci fournit la relation cherchée et les entiers u et v :

$d = ua + vb = (-1)^{n+1}\sigma_{n+1}a + (-1)^n\beta_{n+1}b.$
$u = (-1)^{n+1}\sigma_{n+1}, v = (-1)^n\beta_{n+1}.$

Cette démonstration de la relation de Bezout, fournit un algorithme efficient permettant de calculer u et v.

Listing 8: Pseudo code de l'algorithme d'Euclide étendu.

```
// EuclideEtendu(a, b)
// Calcul d = pgcd(a, b) et u, v tels que ua + vb = d ou a >= b.

Q[0][0] = 1;
Q[0][1] = 0;
Q[1][0] = 0;
Q[1][1] = 1;

n = 0;

while (b != 0)
{
  q = partieEntiere(a / b);
  matriceTemp[0][0] = q1;
  matriceTemp[0][1] = 1;
  matriceTemp[1][0] = 1;
  matriceTemp[1][1] = 0;
  M = M*matriceTemp;
  a = b;
  b = a - qb;
  n = n + 1;
}

return(a, (-1)^(n+1)$M22, (-1)^{n}*M12);
```

Théorème 4.3.1 Étant donné deux entiers positifs a et b de longueur binaire $L(a)$ et $L(b)$, on peut calculer des entiers u et v tels que :
$ua + vb = d$, en $O(L(a)L(b))$ opérations bit à bit.

Démonstration En reprenant les notations ci-dessus, les égalités :

$$Q_{i+1} = Q_i \times \begin{pmatrix} q_{i+1} & 1 \\ 1 & 0 \end{pmatrix} \text{ et } \begin{pmatrix} a \\ b \end{pmatrix} = Q_i \times R_i = \begin{pmatrix} \alpha_i & \beta_i \\ \gamma_i & \sigma_i \end{pmatrix} \times \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix}$$

prouvent que les coefficients des matrices Q_i sont positifs et majorés par a $(a \geq b)$:

$$a = \alpha_i r_{i-1} + \beta_i r_i$$

$$b = \gamma_i r_{i-1} + \sigma_i r_i$$

Par ailleur, $\alpha_{i+1} = \alpha_i q_{i+1} + \beta_i$. Le nombre d'opérations bit à bit pour calculer tous les α_i est :

$$T_{11} = C \sum_{i=1}^{n} [L(\alpha_i)L(q_{i+1}) + L(\alpha_i) + L(q_{i+1}) + L(\beta_i)] \leq CL(a) \sum_{i=1}^{n} [L(q_{i+1}) + 3]$$

$$T_{11} \leq CL(a) \sum_{i=1}^{n} [\log_2(q_{i+1}) + 4] = CL(a)[\log_2(q_2 \dots q_{n+1}) + 4n]$$

En imaginant que l'on fasse débuter l'algorithme d'Euclide avec b et r_1 (au lieu de a et b), on voit que $n = O(L(b))$ et comme dans la démonstration de la complexité de l'algorithme d'Euclide, on obtient $q2 \dots q_{n+1} \leq b$. Finalement, il existe une constante $K > 0$ tel que :

$$T_{11} \leq KL(a)L(b)$$

Le temps de calcul des quatre coefficients de la matrice Q_{n+1} sera en $O(L(a)L(b))$. Pour obtenir le coût total de l'algorithme étendu, il faudra rajouter la somme des temps de calcul des divisions. Cette somme est en fait le temps de calcul de l'algorithme d'Euclide lui même, elle est également en $O(L(a)L(b))$. On a prévu que le temps de calcul de l'algorithme d'Euclide étendu est $O(L(a)L(b))$.

4.3.2 Seconde version de l'algorithme d'Euclide étendu

L'algorithme suivant permet d'obtenir directement le résultat en effectuant un seul "passage" dans l'algorithme sans introduire les matrices.

Listing 9: Pseudo code alternatif de l'algorithme d'Euclide étendu

```
//Input: Deux entier a >= b > 0.
//Output: d = pgcd(a, b), u et v tels que ua + ub = pgcd(a, b).

u = 1;
v = 0;
d = a;
r = 0;
s = 1;
t = b;

while (t > 0)
{
  q = partieEntiere(d/t);
  x = u - qr;
  y = v - qs;
  w = d - qt;
  u = r;
```

```
  v = s;
  d = t;
  r = x;
  s = y;
  t = w;
}

return (d, u, v);
```

Montrons que cet algorithme fournit bien l'*output* annoncé. Examinons les variables d, t et w. À l'intérieur de la boucle *while*, ces variables prennent bien les valeurs des restes successifs de l'algorithme d'Euclide. Comme d et t sont initialisées par a et b, à la fin de la boucle on a bien $d = \text{pgcd}(a, b)$. Prouvons ensuite que pendant toute la boucle $ua + vb = d$. Pour cela démontrons par récurrence que pendant toute la boucle, les deux relations $ua + vb = d$ et $ra + sb = t$ sont satifaites. On effectue une récurrence sur le compteur de la boucle, qui n'est pas mis dans l'algorithme car il ne sert à rien ici.

À l'initialisation $ua + vb = a = d$ et $ra + sb = b = t$. Les relations sont vérifiées.
Supposons $ua + vb = d$ et $ra + sb = t$. Effectuons un nouveau passage dans la boucle.
Nous devons remplacer r par $r' = x = u - qr$, s par $s' = y = v - qs$ et t par $t' = w = d - qt$. La nouvelle valeur de $ra + sb$ sera :
$r'a + s'b = (u - qr)a + (v - qs)b = ua + vb - q(ra + sb) = d - qt = t'$.
Et t' est la nouvelle valeur de t. De même les nouvelles valeurs des variables u, v et d sont respectivement $u' = r$ et $v' = s$ et $d' = t$. Donc $u'a + v'b = ra + sb = t = d'$.

On a prouvé que les deux relations $ua + vb = d$ et $ra + sb = t$ restent vraies dans toute la boucle. À la sortie, on a bien $ua + vb = d = \text{pgcd}(a, b)$.

Exemple $a = 949$, $b = 767$

Variables	U	V	D	R	S	T	Q
Initialisation	1	0	A	0	1	B	
Boucle	1	0	949	0	1	767	1
	0	1	767	1	-1	182	4
	1	-1	182	-4	5	39	4
	-4	5	39	17	-21	26	1
	17	-21	26	-21	26	13	2
	-21	26	13	59	73	0	

On retrouve $(-21) \times 949 + 26 \times 767 = 13$.

4.3.3 Version récursive de l'algorithme d'Euclide étendu

Listing 10: Pseudo code de l'algorithme d'Euclide étendu récursif

```
// EuclideEtendu(a, b)

if (b = 0)
{
  return (a, 1, 0);
}

(d', x', y') = EuclideEtendu(b, a mod b);
(d, x, y) = (d', y', x' - partieEntiere(a/b) * y');

return(d, x, y);
```

Exactitude de l'algorithme

Elle est évidente :

$d = \text{pgcd}(a, b) = \text{pgcd}(b, a \mod b)$

$x'b + y'(a \mod b) = y'a + (x' \lfloor \frac{a}{b} \rfloor y')b.$

Exemple d'exécution

A	B	$\lfloor \frac{a}{b} \rfloor$	D	X	Y
200	41	4	1	8	-39
41	36	1	1	-7	8
36	5	7	1	1	-7
5	1	5	1	0	1
1	0	$-$	1	1	0

$\text{pgcd}(200, 41) = 1$

$8 \times 200 + (-39) \times 41 = 1$

4.3.4 L'algorithme binaire du calcul du pgcd

Il existe une méthode "non euclidienne" pour calculer le pgcd de deux entiers qui est apparu dans les années 1960. Cet algorithme utilise les faits évidents suivants. Soient a et b deux entiers positifs :

- si a et b sont tous les deux pairs $\text{pgcd}(a, b) = 2\text{pgcd}(\frac{a}{2}, \frac{b}{2})$,
- si a est pair et b impair $\text{pgcd}(a, b) = \text{pgcd}(\frac{a}{2}, b)$,
- si a et b sont tous les deux impairs $\text{pgcd}(a, b) = \text{pgcd}(\frac{|a-b|}{2}, b)$.

Ces constations conduisent à l'algorithme suivant :

Listing 11: Pseudo code de l'algorithme binaire du calcul du PGCD

```
//Input : Deux entiers a, b > 0.
//Output : d = pgcd(a, b).

d = 1;
while (a = 0 mod 2) and (b = mod 2)
{
  a = a/2;
  b = b/2;
  d = 2d;
}
while (a != 0)
{
  if(a = 0 mod 2) then a = a/2;
  else if (b = 0 mod 2) then b = b/2;
  else
  {
    x = |a - b| / 2;
    if (a >= b) then a = x;
    else b = x;
  }
  d = d * b;
}

return d;
```

Théorème L'algorithme pgcd binaire calcul le plus grand commun diviseur de deux nombres en $O((L(ab))^2)$ opérations bit à bit où a est le plus grand de ces nombres.

Démonstration Lors de chaque passage dans la première ou la seconde boucle, le produit $a \times b$ est divisé au moins par 2. Donc au bout d'au plus $\log_2(ab) + 1$ passage dans les boucles, on obtient $a \times b = 0$. Lorsque $a \times b = 0$, b n'est jamais nul, car sinon on aurait obtenu avant $a = b$ et l'algorithme aurait assigné a à la valeur 0. Donc si $a \times b = 0$, l'entier $a = 0$ et alors la seconde boucle s'arrête.
Compte tenu des faits énoncés au début, la valeur final de $d = \text{pgcd}(a, b)$.
L'algorithme n'effectue que des divisions par 2 et des soustractions. Il y a donc au plus $\log_2(ab) + 1$ passages dans les boucles. Le nombre N d'opérations bit à bit est égal à $N = O((log_2(ab))^2)$ et si $a \geq b$, on pourra conclure que $N = O((L(ab))^2)$.

Exemple d'exécution de l'algorithme binaire de calcul du pgcd

$a = 1207$, $b = 4046$

A	B	D	X
2414	4046	1	
1207	2023	2	
Sortie de la première boucle			
	1207	2023	2
	1207	408	2
	1207	204	2
	1207	102	2
	1207	51	2
	578	51	2
	289	51	2
	119	51	2
	34	51	2
	17	51	2
	17	17	2
	0	17	2
Sortie de la seconde boucle			
		34	

$\text{pgcd}(2414, 4046) = 34$

Deuxième partie

Cryptographie sûre et problèmes difficiles

Un système cryptographique est constitué :
- d'un ensemble $\mathcal{M}$ de messages (*clairs*) ;
- d'un ensemble $\mathcal{C}$ de *chiffrés* ;
- d'un ensemble $\mathcal{K}$ de *clés*.

Pour chaque clé $k \in \mathcal{K}$, une fonction de *chiffrement* $e_k : \mathcal{M} \to \mathcal{C}$ et une fonction $d_k : \mathcal{C} \to \mathcal{M}$ de telle sorte que $d_k \circ \mathcal{C}_k = \text{Id}$
e_k et d_k sont *injectives*

Notation Si $x \in \mathcal{M}$, on note x l'évènement $\{x\} \times \mathcal{K}$ et $P(x) = P_{\mathcal{M}}(\{x\})$
Si $k \in \mathcal{K}$, on note k l'évènement $\mathcal{M} \times \{k\}$ et $P(k) = P_{\mathcal{K}}(\{k\})$

Si $y \in \mathcal{C}$, on note y l'évènement $\{(x,k)|e_k(x) = y\}$

Remarque

$$\begin{aligned} y &= \{(x,k)|e_k(x) = y\} \\ &= \bigcup_{\substack{k \in \mathcal{K} \\ y \in e_k(\mathcal{M})}} \{(d_k(y),k)\} \end{aligned}$$

Donc $P(y) = \sum_{\substack{k \in \mathcal{K} \\ y \in e_k(\mathcal{M})}} P(d_k(y)).P(k)$

1 Systèmes cryptographiquement sûrs

Définition Un système est parfaitement sûr si :

$$\forall x \in \mathcal{M}, \forall y \in \mathcal{C} P(x|y) = P(x)$$

La probabilité d'un clair x connaissant le chiffré y est la même que la probabilité de x le chiffré n'apporte aucune information sur la connaissance du clair.

Théorème Si on suppose que $\forall y \in \mathcal{C}$, on a $P(y) > 0$ et que le système est parfaitement sûr, alors $|\mathcal{K}| \geq |\mathcal{C}| \geq |\mathcal{M}|$.

Remarque La condition $P(y) > 0$ est naturelle car un chiffré qui a une proba nulle d'être atteint peut être supprimé de $\mathcal{C}$.

Preuve Fixons $x \in \mathcal{M}$ tel que $P(x) > 0$ pour chaque $y \in \mathcal{C}$ on a $P(x|y) = P(x)$.

D'après Bayes $P(y)P(x|y) = P(x|y)P(x)$
$P(y)P(x) = P(x|y)P(x)$
donc $P(y|x) = P(y) > 0$.
Cela signifie qu'il existe au moins $k \in \mathcal{K}$ tel que $e_k(x) = y$, donc $|\mathcal{K}| \geq |\mathcal{C}|$.
De plus e_k est injective, donc $|\mathcal{C}| \geq |\mathcal{M}|$.

théorème Soit un système vérifiant $|\mathcal{K}| = |\mathcal{C}| = |\mathcal{M}|, P(y) > 0$
Il est à sécurité parfaite si et seulement si on a deux conditions :

- toutes les clés sont équiprobables ;
- pour chaque $x \in \mathcal{M}$ et chaque $y \in \mathcal{C}$ il existe une clé k unique vérifiant $e_k(x) = y$.

Preuve Si les 2 conditions sont vérifiés pour tout $y \in \mathcal{C}$, on a :

$$\begin{aligned} P(y) &= \sum_{\substack{k \\ y}} P(d_k(y))P(k) \\ &= \frac{1}{|\mathcal{K}|} \sum P(d_k(y)) \\ &= \frac{1}{|\mathcal{K}|} \sum_{x \in \mathcal{M}} (P(x)) \\ &= \frac{1}{|\mathcal{K}|} \end{aligned}$$

D'autre part pour $x \in \mathcal{M}, y \in \mathcal{C}$ puisqu'il n'existe qu'une clé vérifiant $e_k(x) = y$, on a :

$$P(y|x) = \sum_{k|e_k(x)=y} P(k) = \frac{1}{|\mathcal{K}|}$$

par Bayes $P(x|y) = \frac{P(x)P(y|x)}{P(y)}$
$P(x|y) = \frac{P(x)/|\mathcal{K}|}{1/|\mathcal{K}|} = P(x)$
Sécurité parfaite !

Réciproquement Montrons que k est unique.
Pour chaque couple $(x, y) \in \mathcal{M} \times \mathcal{C}, \exists k | e_k(x) = y$.
Pour x fixé, on a $\{e_k(x) | k \in \mathcal{K}\} = \mathcal{C}$
$|\mathcal{C}| = |\mathcal{K}|$
Les $e_k(x)$ sont donc distincts 2 à 2 et pour chaque $y \in \mathcal{C}$, la clé k vérifiant $e_k(x) = y$ est unique.
$(x_{\text{fixé}}, k) \leftrightarrow y$

Équiprobabilité Pour 2 clés k_1 et k_2, comparons $P(k_1)$ et $P(k_2)$.
Fixons $y \in \mathcal{C}$ et soit x_1, $x_2 \in \mathcal{M}$ tel que $e_{k_1}(x_1) = y$ et $e_{k_2}(x_2) = y$.
e_k étant injective, e_k est *bijective* et la sécurité parfaite.

Par Bayes

$$P(y) = P(y|x_1) = \sum_{k|e_k(x_1)=y} P(k) = P(k_1)P(y) = P(y|x_2) = \sum_{k|e_k(x_2)=y} P(k) = P(k_2)$$

Exemple de crypto système parfaitement sûr :

One time pad $m \oplus k = c$
$|k| = |m|$

1.1 Cas de la cryptographie à clé publique

X une variable aléatoire discrète sur un espace probabiliste Ω.

$$H(X) = -\sum_{\omega \in \Omega} P(X = \omega) \log_2(P(X = \omega))$$

"A mathematical theory of communication", Shannon 1948.

L'entropie est la mesure *moyenne* de l'incertitude sur X.

Entropie grande = incertitude grande.

Entropie conditionnelle $H(x|y)$ incertitude moyenne sur X connaissant y.

Soit $\mathcal{M}, \mathcal{C}, \mathcal{K}$ les variables aléatoires discrètes associées aux choix d'un message m d'un chiffré c, d'une clé k.

Définition On dit qu'un système de chiffrement à clé secrète est à confidentialité parfaite si $H(\mathcal{M}|\mathcal{C}) = H(\mathcal{M})$.

En cryptographie à clé publique, l'attaquant dispose de $\mathcal{C}$ et k. Que vaut $H(\mathcal{M}|\mathcal{C}, k)$? Puisque m est entièrement déterminé par c et k, on a $H(\mathcal{M}|\mathcal{C}, k) = 0$.

- Il n'existe pas de cryptographie système à clé publique à confidentialité parfaite.
- Il existe suffisament d'information dans $\mathcal{C}$ et k pour retrouver m.

Comment vérifier qu'un crypto système à clé publique est sûr ?

En pratique L'adversaire n'a pas une puissance de calcul illimitée.

Le challenge est en général un *chiffré* et l'adversaire doit retrouver le clair (ou la clé). Il faut définir la puissance de l'adversaire et définir ce que l'adversaire ne peut pas faire en un temps raisonnable.

Calcul en temps raisonnable Cela signifie que le calcul se fait par un algorithme polynomial. Un algorithme polynomial est sensé être un calcul en temps réaliste pour une taille raisonnable de l'entrée, cependant, ce n'est pas toujours le cas.

Remarques

- Un problème réputé difficile dans le plus mauvais cas peut avoir des instances faciles.
- Un problème réputé difficile peut donner lieu (dans le futur) à un algorithme polynomial.

Pour montrer qu'un problème est sûr :

1. on suppose qu'il existe des problèmes difficiles ;
2. on montre que si on suppose l'existence d'un algorithme polynomial qui casse le système, alors on sait résoudre par un algorithme polynomial un problème difficile.

C'est un raisonnement par *réduction*. Mais à quel coût ?

Exemple Supposons A la factorisation de n, $n = p.q$.

On veut se servir de A pour prouver la difficulté de B (on réduit A par B). Dans la réduction, a une instance de taille t pour la factorisation correspond une instance de taille $100t$ de B.

Si on veut faire reposer la scurité de B sur le fait que la factorisation est un problème difficile et qu'on ne sait pas en pratique factoriser 2048 bits, il faut imposer aux entrées de B d'avoir au moins 2048×100 bits. C'est le *Coût de la réduction*.

Actuellement il existe beaucoup de problèmes (supposés) difficiles. Mais les réductions ont un coût trop élevé. On est donc obligé d'ajouter des hypothèses.

Random Oracle Model

Rappel sur les fonctions de hachage
$n \rightarrow h(n)$
$\{0,1\}^* \rightarrow \{0,1\}^k$
C'est une fonction *one way*.

Les réponses de l'*Oracle* sont parfaitement *aléatoires* donc les fonctions de hachage sont supposées parfaitement aléatoires.

Types de sécurité Sécurité sémantique, indistinguabilité.
- *Sécurité sémantique* : impossibilité d'obtenir en temps de calcul raisonnable une information sur le clair connaissant le chiffré (à part sa longueur).
- *L'indistinguabilité* : (prouvée, contrairement à la sécurité sémantique)
  Étant donnés 2 clairs et le chiffré de l'un d'eux, on ne peut décider en temps raisonnable avec une probabilité supèrieure à $\frac{1}{2}$ de quel clair provient le chiffré.
- *Non maléabilité* (1991) : impossibilité de construire à partir d'un chiffré y un chiffré y' dont le clair soit relié
- *Chiffrement probabiliste* : deux chiffrés d'un même message sont distincts.
- *Chiffrement déterministe* : un même clair donne le même chiffré (Par exemple : DES-ECB, RSA).

Types d'attaques
- attaque à texte clair choisi (CPA)
  l'attaquant peut chiffrer les textes clairs de son choix ;
- attaque non adaptative à textes chiffrés choisis (CCA1, 1990)
  l'attaquant dispose en plus d'un oracle de déchiffrement qu'il peut utiliser *avant* de recevoir le challenge ;
- attaque adaptative à textes chiffrés choisis (CCA2, 1992)
  en plus l'attaquant dispose de l'oracle de déchiffrement après avoir reçu le challenge (sans pouvoir déchiffrer le challenge lui-même).

Rappel sur RSA
$n = p.q$
m : message à chiffrer
e : élément premier avec n
$X^a \mod n = X^{\varphi(a)} \mod n$
et $\varphi(n) = (p-1)(q-1)$
Publique n et e : $e.d = 1 \mod \varphi(n)$
Privé d (d est un inversible de e)

Rappel sur El Gamal
X : clé privée
g : générateur de $\mathbb{Z}_q^*$
y : g^X clé publique

DLP : Connaissant y, il est difficile de calculer X dans un temps raisonnable.
$C = (g^k, y^k.m) = (C_1, C_2)$: k est aléatoire.
$m = \frac{C_2}{C_1^X}$

Listing 12: Code Ruby de l'algorithme ρ de Pollard

```
# Source: http://sobe-session.blogspot.com/2008/07/algorithme-rho-de-pollard-en-ruby.
    html

def pgcd(m, n)
  (n == 0) ? m : pgcd(n, m % n)
end

def f(x, n)
  (x**2 + 1) % n
end

def x(i, n)
  (i == 0) ? 2 : f(x(i - 1, n), n)
end

class Integer
  def prime?
    res = true
    if self < 2
      res = false
    else
      for i in 2..Math.sqrt(self).to_i
        if i.prime? and (self % i == 0)
          res = false
        end
      end
    end
    res
  end
end

def pollard(n)
  r = []
  if n.prime?
    r.push n
  elsif n != 1
    i, p = 0, 1
    while p == 1
      i += 1
      p = pgcd((x(i, n) - x(2 * i, n)).abs, n)
    end
    raise "Pollard Algorithm: Cycle FAIL" if p == n
    if p.prime?
      r.push p
    else
      r.push pollard(p)
    end
    r.push pollard(n / p)
  end
  r.flatten
end

input = ARGV[0].to_i
if input == 0
  puts "Usage: ruby #{$0} <num>"
else
  puts pollard(input).join(" * ")
end

exit
```