Arithmétique et algorithmique

Master Informatique — Semestre 2 — UE optionnelle de 3 crédits

Année 2019-2020

Version du 15 janvier 2020

Premier rappel de l'école primaire

Rappelons tout d'abord que pour tout couple d'entiers $a, b \in \mathbb{N}$ avec b *non nul*, il existe un et un seul couple d'entiers (q, r) tels que $a = b \times q + r$ et $0 \leqslant r < |b|$. L'entier q est appelé le **quotient** et r le **reste** de la division euclidienne de a par b.

Le quotient et le reste peuvent être calculés en suivant l'algorithme ci-dessous :

```
DIVISION EUCLIDIENNE(a, b, Q, R)
1  R ← a
2  Q ← 0
3  while R ⩾ b
4  do R ← R − b
5       Q ← Q + 1
6  return
```

Principe : À chaque tour, on incrémente Q et diminue R jusqu'à obtenir les valeurs recherchées.

Remarques

Cet algorithme, quelque peu naïf, possède deux intérêts :

1. Il est simple et correct.
2. Il est facile à prouver, à l'aide de la technique usuelle des **invariants de boucle**.

Mais ce n'est pas l'algorithme le plus efficace !

Un algorithme beaucoup plus rapide (linéaire en la **taille** de a)

Dans cet algorithme, on calcule avant toute chose un entier $n \geqslant 0$ tel que $2^n \times b > a$. Ainsi, le quotient Q s'écrit sur n bits : $Q \leqslant 2^n - 1$.

L'idée consiste à calculer les bits de Q un par un, en commençant par le bit de poids fort.

DIVISION EUCLIDIENNE RAPIDE(a, b, Q, R, n)

```
1  R ← a; Q ← 0
2  N ← n;                          # C'est le nombre de bits restant à calculer
3  Aux ← 2^n × b
4  while N > 0                     # Tant qu'il reste un bit à calculer
5  do Aux ← Aux/2
6     if R < Aux
7       then Q ← 2 × Q             # Ajout d'un 0 au résultat
8       else Q ← 2 × Q + 1         # Ajout d'un 1 au résultat
9            R ← R − Aux
10    N ← N − 1                    # Il y a un bit en moins restant à calculer
11 return
```

Première remarque pratique

Avec des entiers qui s'écrivent sur **1000 bits** :

— le premier algorithme nécessitera environ 2^{1000} tours de boucle ;

C'est hors d'atteinte !

— le second algorithme nécessitera seulement 1000 tours de boucle.

Ce sera instantanné !

En pratique, le standard RSA manipule des entiers dont la longueur est en effet de l'ordre d'**un millier de bits**. Il nécessite donc de faire appel à des bibliothèques permettant d'utiliser ce type de données efficacement.

- `BigInteger` est la classe Java permettant de manipuler efficacement des entiers très longs ;
- la bibliothèque **GMP** permet également de manipuler ce type de données en C.

Seconde remarque pratique

L'**addition** et la **division euclidienne** s'effectuent en temps linéaire (en la *taille* des données).

La **multiplication** s'effectue en revanche en temps quadratique.

Par définition : $x^0 = 1$; $x^1 = x$; $x^2 = x \times x$;... $x^k = x^{k-1} \times x$.

Combien de multiplications faut-il pour calculer x^d *?*

✔ *Calculs dans* $\mathbb{Z}$

☛ *Calculs modulo n, c'est-à-dire dans* $\mathbb{Z}/n.\mathbb{Z}$

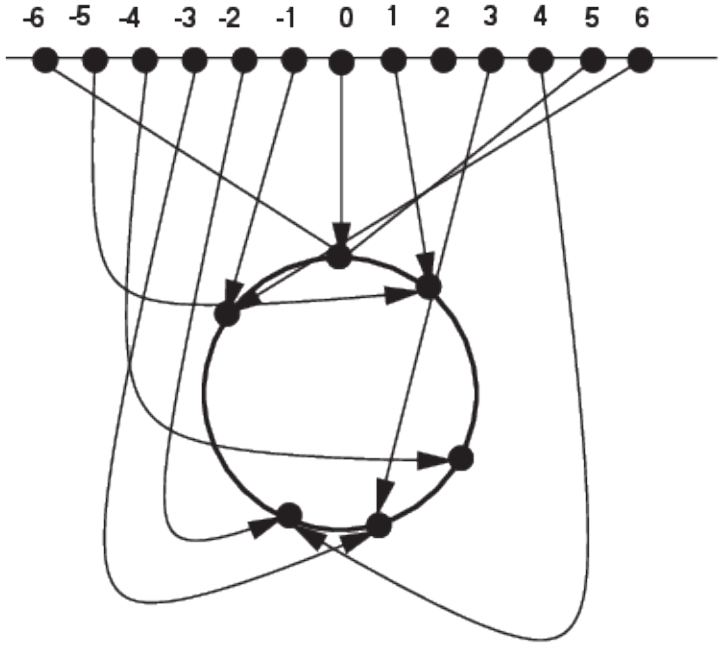
Notations et intuitions

Soit n un entier > 0. On note $\mathbb{Z}/n.\mathbb{Z}$ (ou parfois $\mathbb{Z}_n$) l'ensemble $\{0, 1, ..., n-1\}$.

Ce sont tous les n restes possibles quand on divise un entier a par n.

On notera $\overline{a}$ le **reste** de la division de a par n (ou « **modulo** n »).

Intuitivement, les restes modulo n correspondent à l'enroulement de tous les nombres entiers sur un **cercle** dont le périmètre vaudrait n : ainsi, 0, n et $-n$ sont des représentants équivalents.



Exemple

Si $n = 5$ alors $\overline{17} = 2$.

Autre notation

$x = y \pmod{n}$ si $\overline{x} = \overline{y}$,

c'est-à-dire si n divise $x - y$.

Rmq. Dans le système RSA en pratique, la valeur de n s'écrira sur 1000 bits, au moins ; le nombre d'éléments dans $\mathbb{Z}/n.\mathbb{Z}$ est donc de l'ordre de 2^{1000} : c'est astronomique !

Calculs sur le cercle des restes

Il est possible (et facile) de calculer, c'est-à-dire additionner, soustraire ou même multiplier, les éléments de $\mathbb{Z}/n.\mathbb{Z}$.

Comment fait-on ?

Par définition, pour tous $x, y \in \mathbb{Z}/n.\mathbb{Z}$,

$$x \boldsymbol{+} y = \overline{x + y}$$

$$x \boldsymbol{\times} y = \overline{x \times y}$$

Il suffit donc de faire le calcul dans $\mathbb{Z}$ puis de conserver le reste modulo n.

Exemple : Avec $n = 5$, nous avons donc $3 \boldsymbol{\times} 2 = \overline{3 \times 2} = \overline{6} = 1$.

Soustraire x, c'est simplement additionner l'*opposé de* x : $x + (n - x) = 0 \pmod{n}$.

Diviser par x, c'est simplement multiplier par l'*inverse de* x.

Mais parfois, x n'a pas d'inverse !

Calcul d'une puissance modulo n

Pour calculer $x^d \pmod n$, où d s'écrit sur un millier de bits, on calcule d'abord les mille **carrés successifs** de x : x, x^2, x^4, x^8, x^{16}, x^{32}, x^{64}, ... $x^{2^{999}}$ *en prenant soin de ne conserver que le reste modulo n après chaque multiplication.*

Puis on utilise l'**écriture de d en binaire** pour calculer

$$x^d = x^{\sum_{k=0}^{999} 2^k . d_k} = \prod_{k=0}^{k=999} x^{2^k . d_k}$$

en prenant soin à nouveau de ne conserver que le reste modulo n après chaque nouvelle multiplication.

Pour calculer $x^d \pmod n$ sans vous poser de questions, et sans risque, vous utiliserez :

- la méthode **`modPow()`** avec les BigIntegers en Java
- ou la fonction **`mpz_powm()`** avec GMP en C.

- ✔ *Calculs dans $\mathbb{Z}$*
- ✔ *Calculs modulo n, c'est-à-dire dans $\mathbb{Z}/n.\mathbb{Z}$*
- ☛ *Les éléments inversibles*

Diviseur et plus grand diviseur

Un entier positif d **divise** un entier a s'il existe un entier q tel que $a = q \times d$

$$\textit{c'est-à-dire :}\ a = 0 \mod d.$$

Soient a et b deux entiers. Un entier positif d est appelé **diviseur commun** de a et de b s'il divise a et b.

Définition

Un entier d est appelé le **plus grand diviseur commun (PGCD)** de a et de b si c'est le plus grand des diviseurs communs de a et b.

On note alors $d = \mathrm{PGCD}(a, b)$.

Par exemple le PGCD de 42 et 56 est 14. En effet, $\frac{42}{14} = 3$ et $\frac{56}{14} = 4$.

Deux cas particuliers

Les diviseurs d'un nombre ne dépendent pas de son signe. On peut donc supposer que $a \geqslant 0$ et $b \geqslant 0$.

Si $a = 0$ **et** $b = 0$ alors il n'y a pas de plus grand diviseur de a et b, car tout nombre entier $x > 0$ divise 0.

Si $a = 0$ **et** $b > 0$ alors le PGCD de a et b, c'est b. Car b est le plus grand diviseur de b.

Dans la suite, nous supposerons $a > 0$ et $b > 0$.

Calcul du PGCD à l'aide de l'algorithme d'Euclide (pour $a > 0$ et $b > 0$).

L'algorithme d'Euclide permet de calculer le PGCD de deux entiers strictement positifs.

ALGORITHME D'EUCLIDE(a, b)

1 $R_0 \leftarrow a$
2 $R_1 \leftarrow b$
3 **while** $R_1 > 0$
4 **do** DIVISION EUCLIDIENNE(R_0, R_1, Q, R)
5 $\quad R_0 \leftarrow R_1$
6 $\quad R_1 \leftarrow R$
7 **return** (R_0)

C'est un algorithme étudié au collège...

Algorithme d'Euclide étendu

L'algorithme d'Euclide étendu permet de calculer *en outre* deux entiers u et v tels que

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

ALGORITHME D'EUCLIDE ÉTENDU(a, b)

1 $R_0 \leftarrow a;$
2 $R_1 \leftarrow b$
3 $U_0 \leftarrow 1; U_1 \leftarrow 0; V_0 \leftarrow 0; V_1 \leftarrow 1;$ *
4 **while** $R_1 > 0$
5 **do** DIVISION EUCLIDIENNE(R_0, R_1, Q, R)
6 $R_0 \leftarrow R_1;$
7 $R_1 \leftarrow R$
8 $U \leftarrow U_0 - Q \times U_1; V \leftarrow V_0 - Q \times V_1$ *
9 $U_0 \leftarrow U_1; U_1 \leftarrow U; V_0 \leftarrow V_1; V_1 \leftarrow V$ *
10 **return** (R_0, U_0, V_0)

Application : calcul de l'inverse d'un nombre modulo n

Supposons que $a \in \mathbb{Z}/n.\mathbb{Z}$ soit **non nul** et « *premier avec* n », c'est-à-dire PGCD$(a, n) = 1$: le seul diviseur commun à a et n est 1.

Alors l'algorithme d'Euclide étendu permet de calculer deux entiers u et v tels que :

$$a \times u + n \times v = 1$$

et donc

$$a \times u = 1 \pmod{n}$$

Par conséquent, a est inversible !

Réciproque : les éléments qui ne sont pas non inversibles

Supposons que $a \in \mathbb{Z}/n.\mathbb{Z}$ soit **non nul** mais qu'il **ne soit pas premier** avec n, c'est-à-dire qu'il existe un entier $d > 1$ qui divise a et n. Alors a n'est pas inversible modulo n.

Car sinon il existerait un entier u tel que

$$a \times u = 1 \pmod{n}$$

ce qui signifie qu'il existerait également un entier v tel que

$$a \times u + n \times v = 1$$

et alors d diviserait 1, puisqu'il divise a et n...

Au total : les seuls éléments inversibles de $\mathbb{Z}/n.\mathbb{Z}$ sont ceux qui sont non nuls et qui sont premiers avec n.

Calcul en C et en Java (1/2)

Nous avons déjà rappelé qu'un entier a est inversible modulo n si, et seulement si, le PGCD de a et n vaut 1.

Pour déterminer si un nombre a est inversible modulo n, vous utiliserez :

- en Java : la méthode **`a.gcd(n)`**, car elle renvoie le PGCD de a et n ;
- en C : la fonction **`mpz_gcd(g,a,n)`** qui place dans g le PGCD de a et n.

Calcul en C et en Java (2/2)

Pour calculer l'inverse d'un nombre a modulo n, vous utiliserez :

- la méthode **`modInverse()`** des BigIntegers en Java.
- la fonction **`void mpz_gcdext(g,u,v,a,n)`** en C avec GMP ;

N.B. Contrairement à **`modInverse()`** en Java, la fonction **`mpz_gcdext`** de GMP renverra un entier u qui n'est pas nécessaire inférieur strictement à n ni même positif : il suffira néanmoins de considérer u modulo n.

En effet, si $u' = u + k.n$ alors

$$a \times (u + k.n) + n \times (v - k.a) = a \times u + n \times v = 1$$

et donc $a \times u' = 1 \pmod{n}$.

- ✔ *Calculs dans* $\mathbb{Z}$
- ✔ *Calculs modulo n, c'est-à-dire dans* $\mathbb{Z}/n.\mathbb{Z}$
- ✔ *Les éléments inversibles*
- ☛ *Les nombres premiers*

Les nombres premiers

Un nombre **premier** est un entier (positif) admettant exactement deux diviseurs distincts : 1 et lui-même. Voici la liste de *tous les nombres premiers inférieurs à 100* :

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

On sait, depuis le collège (et Euclide), qu'il existe une infinité de nombres premiers. D'autre part, ils servent à décomposer chaque nombre entier sous la forme d'un produit de facteurs premiers (unique à l'ordre près des facteurs).

N.B. Si p est un nombre premier, alors chaque $x \in \mathbb{Z}/p.\mathbb{Z}$ *non nul* est premier avec p et donc il admet un **inverse**, c'est-à-dire qu'il existe un $y \in \mathbb{Z}/p.\mathbb{Z}$ tel que $x \times y = 1$.

On peut donc « toujours » diviser dans ce cas !

La recherche de grands nombres premiers

À ce jour, depuis décembre 2018, le plus grand nombre premier trouvé vaut

$$p = 2^{82\,589\,933} - 1$$

Il s'écrit avec plus de *82 millions de bits* et précisément 24 862 048 chiffres en décimal.

C'est un nombre de Mersenne dont la primalité a été établie à l'aide d'un algorithme spécifique (le test de primalité de Lucas-Lehmer), particulièrement rapide, et la mise en oeuvre de calculs distribués sur Internet (cf. Great Internet Mersenne Prime Search).

En cryptographie moderne, il est utile de savoir trouver rapidement des nombres premiers de *quelques milliers de bits*, afin notamment de fabriquer des clefs RSA.

Comment trouver un grand nombre premier ?

En pratique, la construction d'un nombre premier p de taille T *bits* fixée s'appuie simplement sur le tirage aléatoire d'un nombre X, *impair* et de taille T bits.

On teste alors si X est un nombre premier. Tant qu'il ne l'est pas, on lui ajoute 2. Finalement, lorsque X est premier, on renvoie $p = X$.

Il suffit donc de savoir déterminer si X est premier !

Rmq. 1. À vrai dire, il vaudrait mieux retirer au sort un nouveau nombre plutôt que d'ajouter 2 à chaque fois, car cette méthode ne respecte pas l'équiprobabilité des tirages.

Rmq. 2. Pour s'assurer que p est assez grand, il suffit de s'assurer, lors du tirage aléatoire, que le T-ième bit du nombre X soit égal à 1. Pour que X soit impair (et gagner du temps), le premier bit doit également être forcé à 1.

Rmq. 3. Il faudra grosso modo de l'ordre de T tentatives, en moyenne, avant de « tomber » sur un nombre premier, comme nous le verrons plus loin.

- ✔ *Calculs dans $\mathbb{Z}$*
- ✔ *Calculs modulo n, c'est-à-dire dans $\mathbb{Z}/n.\mathbb{Z}$*
- ✔ *Les éléments inversibles*
- ✔ *Les nombres premiers*
- ☛ *Tests de primalité*

Comment s'assurer qu'un nombre est premier ?

Le problème calculatoire **PRIMES** consiste à déterminer si un nombre donné est premier.

C'est bien ce qu'on veut pouvoir faire !

On sait (depuis quelques années) que ce problème est **polynomial** :

MANINDRA AGRAWAL, NITIN SAXENA, NEERAJ KAYAL : *PRIMES is in P*, 2002.

Cependant l'algorithme polynomial qui permet d'établir ce résultat, bien qu'il soit relativement simple, ne donne pas un temps d'exécution acceptable *pour les tailles d'entiers utilisés en cryptographie*.

Dans la pratique, on utilise en général un *test probabiliste* : le résultat obtenu est juste avec une certitude aussi grande que l'on souhaite, mais pas absolue. L'un des plus efficaces et des plus utilisés est le test de MILLER-RABIN.

Primalité vs. factorisation

Le problème qui consiste à déterminer si un nombre n donné est premier *ne nécessite pas* d'exhiber une factorisation de n dans le cas où n n'est pas premier.

En particulier, l'algorithme polynomial développé par Manindra Agrawal, Nitin Saxena et Neeraj Kayal permet de détecter qu'un nombre n'est pas premier sans pour autant produire une factorisation.

C'est aussi le cas pour le test de primalité de Miller-Rabin.

Le problème de la factorisation d'un nombre entier reste donc à ce jour un problème qui semble plus difficile que celui de la primalité.

La notion de témoin

L'idée sous-jacente de tous ces algorithmes de primalité est celle de « *témoin* » : si l'on trouve un témoin pour le nombre n, on a la certitude que n n'est pas premier.

Il existe plusieurs sorte de témoins. Les plus simples sont les *témoins de Fermat*.

Le *petit théorème de Fermat* affirme que si n est un nombre premier alors :

$$a^{n-1} = 1 \pmod{n}$$

pour chaque nombre a compris entre 1 et $n-1$.

Par conséquent, si $a^{n-1} \neq 1 \pmod{n}$ avec $0 < a < n$, on a la certitude que n n'est pas premier : le nombre a est alors le témoin du fait que le nombre n n'est pas premier.

Exemple : $24^{220} = 81 \pmod{221}$ donc 221 n'est pas un nombre premier et 24 en est témoin.

Pour autant, on ne dispose pas d'un *diviseur* de n.

Primalité en Java et en C

En Java

La méthode **`boolean isProbablyPrime(int c)`** indique si nombre est premier avec une probabilité d'erreur inférieure à $\frac{1}{2^c}$.

En C

La fonction **`int mpz_probab_prime_p (mpz_t n, int r)`** renvoie :

- 0 si le nombre n n'est pas premier
- 1 si le nombre n est premier avec une probabilité d'erreur inférieure à $\frac{1}{4^r}$
- 2 si le nombre n est premier

Exercice : Quelle valeur faut-il adopter pour **`c`** (ou **`r`**) afin d'obtenir un nombre qui est premier avec une probabilité d'erreur inférieure à $\frac{1}{10^{15}}$?

Nouvelles questions de chance

Master Informatique — Semestre 2 — UE optionnelle de 3 crédits

Année 2019-2020

Version du 15 janvier 2020

Rappel (déjà vu et déjà prouvé !)

Combien de fois en moyenne faut-il lancer un dé à 6 faces avant d'obtenir un 3 ?

6 fois, en moyenne !

Car la probabilité de gagner *en un coup* est $\frac{1}{6}$.

La proportion de nombres inversibles modulo n

L'indicatrice d'Euler $\varphi(n)$.

On appelle *indicatrice d'Euler* la fonction qui à un entier n fait correspondre le nombre $\varphi(n)$ d'entiers m premiers à n et inférieurs à n.

Autrement dit, $\varphi(n)$ est le nombre d'éléments de $\mathbb{Z}/n.\mathbb{Z}$ qui sont inversibles.

Il est clair que $1 \leqslant \varphi(x) \leqslant x - 1$ puisque seuls $1, 2, ..., x - 1$ sont candidats.

On a d'ailleurs $\varphi(x) = x - 1$ si, et seulement si, x est un nombre premier.

La proportion de nombres inversibles est donc $\frac{\varphi(x)}{x-1}$ si on met l'élément 0 de côté, comme il se doit.

De manière certaine : $0 < \frac{\varphi(x)}{x-1} \leqslant 1$.

Mais on ne peut guère dire plus a priori : les mathématiciens nous disent en effet que 0 et 1 sont les limites inférieure et supérieure de $\varphi(x)/x$.

La proportion d'inversibles si x s'écrit sur 1000 bits

Ne baissons pas les bras ! Les mathématiciens nous disent aussi que

$$\varphi(x) > \frac{x}{e^{\gamma}\ \ln\ln x + \frac{3}{\ln\ln x}} \text{ dès que } x > 2$$

où γ est la constante d'Euler : $\gamma = 0,577215665...$

Si x s'écrit sur 1000 bits, alors $2^{999} < x < 2^{1000}$.

Donc

- $\ln(x) > \ln(2^{999}) = 999 \times \ln(2) > 692$ et $\ln\ln(x) > 6$.
- D'autre part, $x < 2^{1000}$ donc $\ln(x) < 1000 \times \ln(2) < 693$ et $\ln\ln(x) < 7$.

D'où

$$\frac{\varphi(x)}{x-1} > \frac{\varphi(x)}{x} > \frac{1}{e^{\gamma} \times 7 + \frac{3}{6}} > \frac{1}{13}$$

En choisissant un nombre a au hasard, il y a plus d'une chance sur 13 de tomber sur un nombre inversible.

✔ *La proportion de nombres inversibles modulo n*

☛ *La proportion de nombres premiers*

Construction de grands nombres premiers

En pratique, la construction d'un nombre premier p inférieur à x s'appuie simplement sur le tirage aléatoire d'un nombre impair n inférieur à x.

On teste alors si n est un nombre premier (à l'aide de l'algorithme de Miller-Rabin ou un autre) : tant que n n'est pas premier, on lui ajoute 2. Finalement, on renvoie $p = n$.

Quelle est la probabilité de gagner en un coup ?

Quelles sont nos chances de réussite ?

On note traditionnellement $\pi(x)$ le nombre de nombres premiers inférieurs à x.

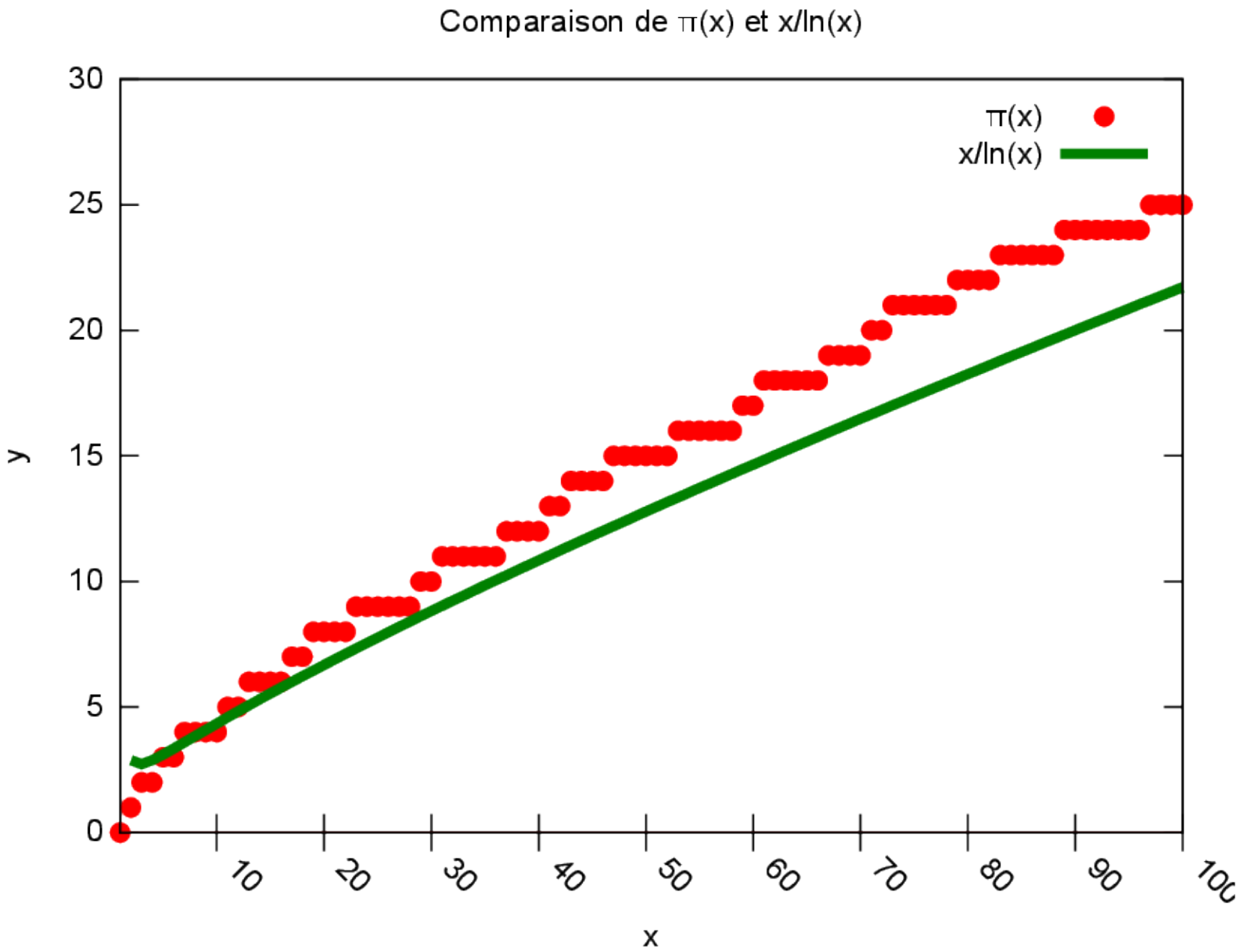
Par exemple : $\pi(10) = 4$.

On sait, grâce aux mathématiciens, que

$$\pi(x) \geqslant x/\ln(x) \quad \text{si } x \geqslant 17$$

selon J. Barkley Rosser et Lowell Schoenfeld : « *Approximate formulas for some functions of prime numbers.* » Illinois J. Math. 6 : p. 64-94 (1962).

Valeurs exactes de $\pi(x)$ avec $x \leqslant 100$



Quelles sont nos chances de réussite ?

La probabilité qu'un nombre pris aléatoirement inférieur à x soit un nombre premier (c'est-à-dire de gagner « en un coup ») vaut $\pi(x)/x \geqslant 1/\ln x$.

Il faudra donc en moyenne tirer moins de $\ln(x)$ nombres aléatoires inférieurs à x avant de tomber sur un nombre premier.

Autrement dit, pour trouver un nombre premier *de* T *bits*, c'est-à-dire inférieur à $x = 2^T - 1$, le nombre de tirages à réaliser en moyenne sera inférieur à $\ln(2^T - 1) \cong 0.7 \times T$.

- ✔ *La proportion de nombres inversibles modulo n*
- ✔ *La proportion de nombres premiers*
- ☛ *Le test de non-primalité de Miller-Rabin*

Le test de non-primalité de Miller-Rabin (1/3)

Dans ce qui suit, n est un entier *impair* et on veut déterminer si n est un nombre premier.

Tout d'abord : $n - 1 = 2^s \times t$ avec t **impair** et $s \geqslant 1$.

Comment calculer s et t ?

Définition

Soit $a \in [1, n-1]$. Si $a^t - 1 \neq 0 \pmod{n}$ et $a^{2^i \times t} + 1 \neq 0 \pmod{n}$ pour chaque $i = 0, ..., s-1$ alors a est appelé *témoin de Miller* pour n.

Exemple : Étudions le cas de $n = 81$ et choisissons $a = 8$.

- $80 = 2^4 \times 5$ donc $s = 4$ et $t = 5$.
- $8^5 - 1 = 43 \pmod{81}$ est différent de 0.
- $8^5 + 1 = 45 \pmod{81}$ est différent de 0.
- $8^{2\times5} + 1 = 74 \pmod{81}$ est différent de 0.
- $8^{2^2\times5} + 1 = 65 \pmod{81}$ est différent de 0.
- $8^{2^3\times5} + 1 = 47 \pmod{81}$ est différent de 0.

Le test de non-primalité de Miller-Rabin (1/3)

Lemme (pas trop compliqué)

S'il existe un témoin de Miller pour n, alors n n'est pas premier.

Autrement dit, *les témoins de Miller sont vraiment des témoins du fait que n n'est pas premier.*

Ainsi, si l'on trouve (par hasard) un témoin, on a la preuve absolue que n n'est pas premier.

Exemple (suite) : Poursuivons l'étude du cas de $n = 81$. Le nombre 8 est un témoin de Miller pour 81. Donc 81 n'est pas premier.

Le test de non-primalité de Miller-Rabin (2/2)

Le **théorème de Rabin** affirme que si n n'est pas premier et $n > 9$, alors parmi tous les entiers $a \in [1, n-1]$, il y a *au moins* $3/4$ de témoins de Miller.

Algorithme probabiliste

Soit $n > 9$ et n impair. On tire au hasard k entiers dans $[1, n-1]$. Et on vérifie si l'un d'entre eux est un témoin.

— Si un témoin est trouvé, l'algorithme retourne que « n n'est pas premier ».

— Si au contraire aucun de ces entiers n'est un témoin, l'algorithme retourne que « n est très probablement premier ».

Si n n'est pas premier, la probabilité qu'aucun de des entiers étudiés et choisis au hasard ne soit un témoin est $\leqslant (1/4)^k$. La probabilité d'erreur est donc $\leqslant (1/4)^k$. Pour $k = 25$, cette probabilité d'erreur est inférieure à 10^{-15}. Le nombre n est donc *très probablement* premier ; on parle alors de « **nombre premiers industriels**. »

Justification pragmatique de cette approche

Pour les spécialistes, il est inutile de lancer plus de 25 tests.

Selon Donald Knuth : Si un nombre aléatoire ≪ passe ≫ 25 tests de Miller-Rabin indépendants alors la probabilité que ce nombre ne soit pas premier est inférieure à la probabilité d'un dysfonctionnement du processeur dû à un rayon cosmique ou à un défaut du processeur.

≪ For the probability less than $(\frac{1}{4})^{25}$, that such a 25-times-in-row procedures gives the wrong information about n. It's much more likely that our computer has dropped a bit in its calculations, due to hardware malfunctions or cosmic radiations, than that algorithm P has repeatedly guessed wrong. ≫

D.E. KNUTH : *The Art of Computer Programming*, Vol. 2 : Semi-Numerical Algorithms

Codage en C pour de petits nombres (1/3)

```
int est_probablement_premier_selon_Miller_Rabin (long n) {
  int probablement_premier = 1;
  long a;
  int nb_tentatives = 0;
  while(nb_tentatives < 25 && probablement_premier == 1) {
    a = (rand() % (n-1)) + 1;          // 0 < a < n-1 au hasard
    if (est_un_temoin_de_Miller(a, n)) probablement_premier = 0;
    nb_tentatives++;
  }
  return probablement_premier;
}
```

Codage en C pour de petits nombres (2/3)

```
int est_un_temoin_de_Miller(long a, long n) {
  int s = 0; long t = n-1;
  while( (t%2) == 0 ) {
    t = t/2;
    s = s+1;
  }
  long r = mod_pow(a, t, n);          // Calcul de pow(a,t) % n
  if (r == 1) return 0;
  else {
    for (int i=0; i<s; i++) {
      if (r == (n-1)) return 0;
      r = (r*r) % n;                  // Carrés successifs de r
    }
    return 1;
  }
}
```

Codage en C pour de petits nombres (3/3)

```
long mod_pow(long m, long k, long n) {
  /* retourne pow(m,k) % n avec la "méthode indienne" */
  if (k == 0) return 1;
  long solution = 1;          // solution = pow(m,0)
  long carres = m;            // carres = pow(m,pow(2,0))
  for(int i=0; i<32; i++) { // carres = pow(m,pow(2,i))
      if(((k>>i)&1) == 1) { // Si le i-ème bit de k vaut 1
          solution = (solution * carres) % n;
        }
      carres = (carres*carres) % n; // carres = pow(m,pow(2,i+1))
    }                 // On suppose que k s'écrit sur 32 bits
  return solution;
}
```

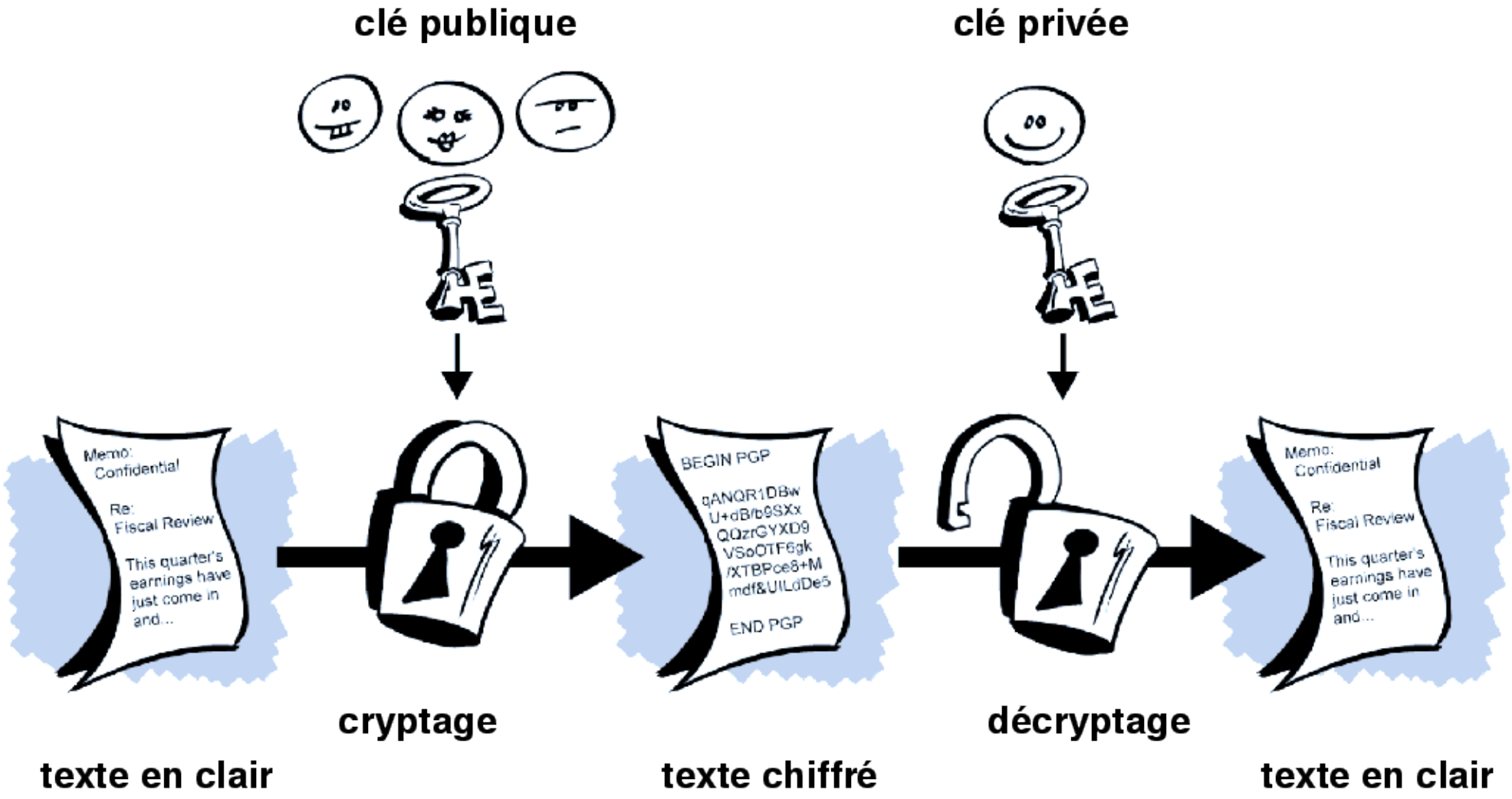
Le protocole cryptographique RSA

Master Informatique — Semestre 2 — UE optionnelle de 3 crédits

Année 2019-2020

Version du 15 janvier 2020

Cryptographie asymétrique (ou à clef publique)



La clef publique fonctionne intuitivement comme un cadenas, mis à la disposition de tout un chacun, et dont l'ouverture ne peut être réalisée qu'avec la clef privée.

L'algorithme RSA

Algorithme RSA

Nous appellerons *Alice* la destinatrice du message chiffré et *Bernard* l'émetteur.

L'algorithme RSA se décompose en trois phases :

1. **Fabrication de deux clefs par Alice**, l'une publique, l'autre secrète (ou privée).
   Cette première phase se divise elle-même en trois parties :
   (a) Alice choisit deux grands nombres premiers p et q *distincts*, et calcule les valeurs de $n = p \times q$ et $w = (p-1) \times (q-1)$.
   (b) Alice trouve un entier d inversible modulo w et calcule son inverse e : on a alors $d \times e = 1 \pmod{w}$.
   (c) Alice diffuse sa clef publique (n, e) mais garde d pour elle : c'est sa clef privée ! Elle peut aussi **oublier** w, devenu inutile.

Algorithme RSA

2. **Envoi par Bernard d'un message chiffré à Alice.**

   Bernard chiffre un message M, compris entre 0 et $n-1$, à l'aide de la clef publique (n, e) connue de tous, en calculant le message chiffré $C = M^e \pmod{n}$, qu'il envoie à Alice.

3. **Réception et déchiffrement du message chiffré par Alice grâce à sa clef privée.**

   Alice déchiffre C reçu en calculant $M = C^d \pmod{n}$ à l'aide de la clef privée (n, d) connue par elle seule, et récupère ainsi le message M que Bernard voulait transmettre.

Notez que la fonction de chiffrement utilisée par Bernard est la **même** que la fonction de déchiffrement utilisée par Alice. La seule différence est que Bernard et Alice n'utilisent pas la même clef.
Nous utiliserons naturellement à nouveau la méthode **`modPow()`** avec les BigIntegers ou la fonction **`mpz_powm()`** avec GMP pour effectuer ce calcul sans difficulté.

✔ *L'algorithme RSA*

☛ *Un exemple pour comprendre*

Exemple de fabrique de clefs

Voyons ce qui se passe si l'on prend pour les deux nombres p et q les valeurs 11 et 17.

On a alors $n = 187$ et $w = (11 - 1) \times (17 - 1) = 160$.

Comme $7 \times 23 = 161$, on peut prendre $e = 7$ et $d = 23$, car $7 \times 23 = 1 \pmod{w}$.

Alice rend donc publique la clef $(187, 7)$.

Les messages possibles sont les nombres de 0 à 186. Mais 0 et 1 sont clairement des messages non recommandés !

Chiffrement d'un message

Supposons que Bernard veuille envoyer le message « 10 » à Alice de manière codée grâce à la clef publique $(187, 7)$. Quelle valeur chiffrée doit-il en fait lui envoyer ?

Bernard va calculer

$$10^7 = 187 \times 53475 + 175.$$

Donc

$$10^e = 175 \pmod{n}$$

Donc Bernard va envoyer le message « 175 » à Alice.

Déchiffrement

Quels calculs effectuera Alice pour décoder correctement ce message chiffré ?

Alice doit calculer « simplement » le reste de la division euclidienne de 175^{23} par 187. Tout d'abord elle observe que $23 = 16 + 4 + 2 + 1$ donc 23 s'écrit 10111 en base 2.
Alice va d'abord calculer 175^2, 175^4, 175^8, 175^{16} (modulo 187) par des élévations au carré successives. Elle pourra alors facilement calculer

$$175^{23} = 175^{16} \times 175^4 \times 175^2 \times 175$$

- $175^2 = 30625 = 163 \times 187 + 144$, donc $175^2 = 144 \pmod{187}$.
- $144^2 = 20736 = 110 \times 187 + 166$, donc $175^4 = 166 \pmod{187}$.
- $166^2 = 27556 = 147 \times 187 + 67$, donc $175^8 = 67 \pmod{187}$.
- $67^2 = 4489 = 24 \times 187 + 1$, donc $175^{16} = 1 \pmod{187}$.
- $175^{23} = 175^{16} \times 175^4 \times 175^2 \times 175$ et $166 \times 144 \times 175 = 4183200 = 22370 \times 187 + 10$ donc $175^{23} = 10 \pmod{187}$.

Alice retrouve donc bien le message envoyé par Bob, à savoir 10.

- ✔ *L'algorithme RSA*
- ✔ *Un exemple pour comprendre*
- ☛ *Validité et sécurité*

Sécurité du chiffrement

La **confidentialité** de la transmission d'un message repose sur *la difficulté à trouver les valeurs secrètes des facteurs* p *et* q *à partir de la valeur publique du module* n (c'est-à-dire à factoriser n), dès lors que les valeurs de p et q sont suffisamment grandes.

De plus,

① Connaître p et q équivaut à connaître la valeur de $w = (\mathrm{p} - 1) \times (\mathrm{q} - 1)$;

② Connaître la valeur de w permet de calculer facilement la clef privée d à partir de la clef publique (e, n) ;

③ Réciproquement, connaître la clef privée d permet de calculer p et q.

Conséquence pratique : *Deux clefs RSA ne doivent jamais utiliser le même module* n. Car la connaissance d'une des deux clefs privées permet de trouver l'autre.

Les records de factorisation

Actuellement, le plus grand module RSA dont une factorisation ait été trouvée contient 768 bits.

Cf. T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, P. Zimmermann : ≪ Factorization of a 768-bit RSA modulus ≫, 2009.

Certains experts estiment que d'ici une dizaine d'années, un module de taille 1024 bits ne sera plus d'une sûreté absolue ; il faut donc, selon eux, adopter dès à présent des modules RSA dont la taille dépasse 1024 bits.

Validité du procédé (sans entrer dans les détails)

Le fait que ce procédé fonctionne bien, c'est-à-dire qu'Alice récupère bien le message clair M, s'appuie sur le **théorème d'Euler** qui affirme que

$$\mathrm{M}^{w} = 1 \pmod{\mathrm{n}} \qquad \text{(si le nombre } \mathrm{M} \text{ est premier avec } \mathrm{n}\text{)}$$

Il faut d'abord observer que $\mathrm{d} \times e = 1 \pmod{w}$ signifie que $\mathrm{d} \times e + w \times \mathrm{l} = 1$ pour un certain entier l ; de plus, les valeurs de d, e et w sont positives, donc $\mathrm{l} < 0$.

$$(\mathrm{M}^{e})^{\mathrm{d}} = \mathrm{M} \times \mathrm{M}^{e \times \mathrm{d} - 1} = \mathrm{M} \times \mathrm{M}^{-\mathrm{l} \times w} = \mathrm{M} \times (\mathrm{M}^{w})^{-\mathrm{l}} = \mathrm{M} \pmod{\mathrm{n}}$$

Dans le cas très rare où le nombre M n'est pas premier avec n, le procédé fonctionne aussi, mais pour d'autres raisons.

- ✔ *L'algorithme RSA*
- ✔ *Un exemple pour comprendre*
- ✔ *Validité et sécurité*
- ☛ *La technique du padding*

La technique du padding (1/2)

Pour chiffrer proprement avec RSA, il ne suffit pas de prendre le message et de lui appliquer la fonction de chiffrement RSA.

D'une part, ceci conduirait à un **chiffrement déterministe**, c'est-à-dire que si on chiffre deux fois le même message avec la même clef on obtient deux fois le même chiffré, ce qui peut avoir des inconvénients.

En particulier, un intrus à l'écoute des communications chiffrées pourra détecter qu'un même message est transmis plusieurs fois (ou dans certaines circonstances particulières) et en tirer profit.

La technique du padding (2/2)

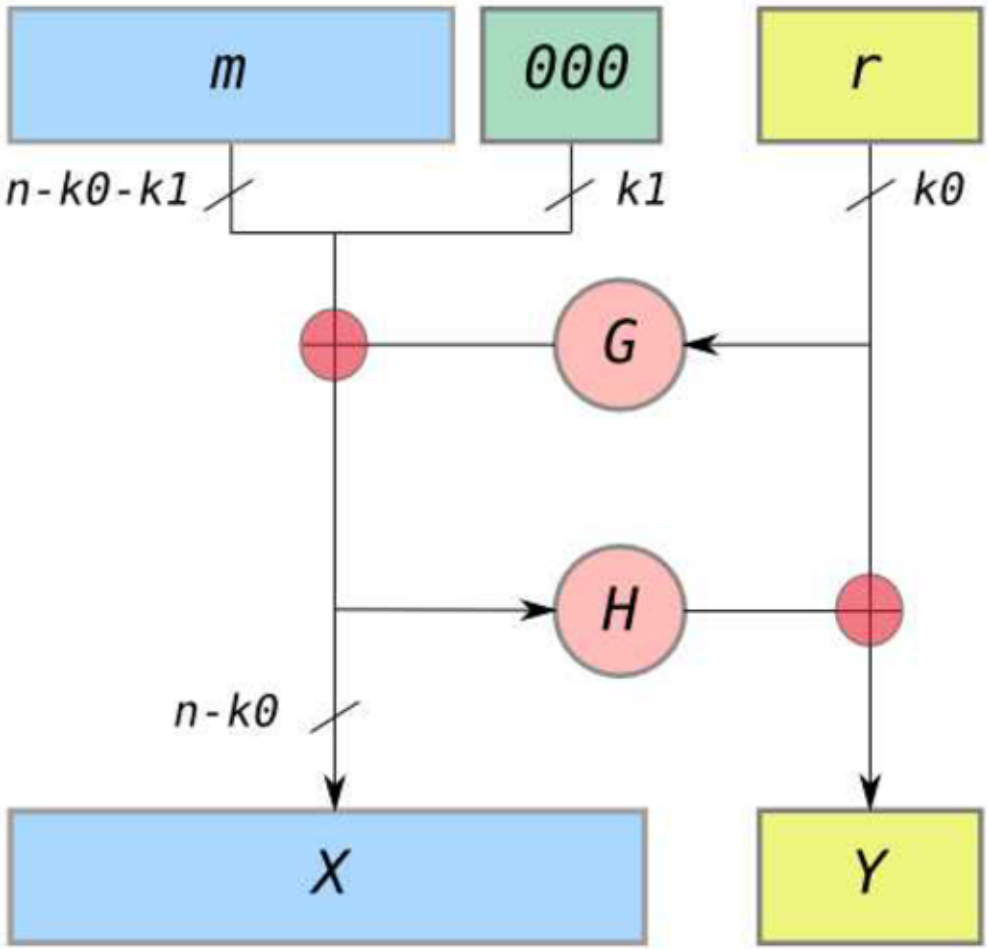
D'autre part, dans le cas de **messages très courts** on risque des attaques spécifiques, puisqu'on ne bénéficie plus de la totale diversité fournie par la taille de l'espace des messages potentiels.

Un adversaire capable d'écouter les messages en transit sera capable de déchiffrer les messages s'ils sont trop courts. Pour cela, il lui suffit de chiffrer *à l'aide de la clef publique* tous les messages courts potentiels, et de stocker les messages chiffrés obtenus dans une table de correspondance.
Lorsque qu'un message chiffré est intercepté, il suffit de chercher dans la table obtenue à quel message clair il correspond.

Il convient donc de préparer le message à chiffrer par une phase dite **phase de padding** (bourrage en français) qui introduit du *non-déterminisme* et proscrit les messages *trop courts*.

Optimal Asymetric Encryption Padding : schéma simplifié

En 1994, M. Bellare et P. Rogaway ont introduit le système **OAEP** (Optimal Asymetric Encryption Padding) qui implémente cette phase de padding.



- ✔ *L'algorithme RSA*
- ✔ *Un exemple pour comprendre*
- ✔ *Validité et sécurité*
- ✔ *La technique du padding*
- ☛ *La fabrique de clefs RSA*

Fabrication de deux clefs par Alice

La fabrique de deux clefs se divise en trois parties :

1. Alice génère deux gros nombres premiers p et q distincts

   Nous savons déjà comment faire en pratique !

   Elle calcule ensuite $n = p \times q$ et $w = (p-1) \times (q-1)$.

2. Alice trouve ensuite deux entiers d et e (inférieurs strictement à w) tels que $d \times e = 1 \pmod{w}$, c'est-à-dire tels que

   $$d \times e + w \times l = 1 \text{ pour un certain } l \in \mathbb{Z}$$

   Pour cela, **elle tire au sort** d et vérifie si un tel e existe (ou inversement).

   L'algorithme d'Euclide permet de déterminer si un tel e existe, i.e. pgcd(d,w)=1.

   L'algorithme d'Euclide étendu permet de calculer un tel e lorsqu'il en existe.

3. Alice diffuse sa clef publique (n, e) mais garde d pour elle : c'est sa clef privée !

Calcul de d à partir de w

Dans la seconde phase de la fabrique des clefs, Alice dispose de w et tire au sort un entier positif d (non nul et inférieur strictement à w).

Elle doit s'assurer que :

- d est inversible modulo w ;
- c'est-à-dire que d est premier avec w ;
- c'est-à-dire que $\mathrm{PGCD}(\mathrm{d}, w) = 1$.
- c'est-à-dire qu'il existe deux entiers e et l tels que $\mathrm{d} \times e + w \times \mathrm{l} = 1$.

Il lui suffit de faire appel à l'algorithme d'Euclide.

Si le test est concluant, Alice dispose de la clef privée d : la fabrique des clefs est presque terminée. **Sinon, elle recommence et tire un nouveau d au hasard**.

Calcul de e à partir de d et w

L'algorithme d'Euclide *étendu* permet alors de calculer deux entiers e et l tels que

$$d \times e + w \times l = 1$$

L'algorithme d'Euclide étendu produisant une solution $(e, l) \in \mathbb{Z}^2$, on peut trouver facilement une autre solution $(e', l') \in \mathbb{Z}^2$ pour laquelle $0 \leqslant e' < w$.

En effet, il suffit pour cela de considérer le reste de la division euclidienne de e par w :

$$e = q \times w + e' \text{ avec } 0 \leqslant e' < w$$

En effet :

$$d \times (e - q \times w) + w \times (l + q \times d) = d \times e - d \times q \times w + w \times l + w \times q \times d = 1$$

Ainsi, **les deux clefs d et e sont plus petites strictement que w (et non nulles)**.

Travaux pratiques

Lors du prochain TP, vous compléterez le squelette de programme `rsa.c` qui construira une paire de clefs RSA, chiffrera le texte ASCII : « Alfred », et enfin déchiffrera le message chiffré.

```
$ gcc rsa.c -lgmp
$ ./a.out
Clef publique (n) : 196520034100071057065009920573
Clef publique (e) : 7
Clef privée (d)   : 561485811714486201295445402 23
Texte ASCII de 6 caractères : "Alfred" -> 71933831046500
Message chiffré   : 450555239454106302498031269 60
Message déchiffré : 71933831046500 -> "Alfred"
```

- Le texte sera d'abord codé par un entier $< n$, conformément à la RFC 2437.
- Vous utiliserez évidemment la bibliothèque GMP afin de manipuler des entiers de taille suffisamment grande.
- Vous pourrez également utiliser les BigIntegers si vous préferez travailler en Java.

Ce qu'il faut retenir

① Le chiffrement RSA nécessite de réaliser **de manière efficace** des calculs assez simples (multiplications, divisions euclidiennes, puissances) sur des entiers comportant plus de mille bits significatifs, à l'aide de bibliothèques particulières.

② La validité du système RSA repose sur des résultats classiques d'arithmétique, plutôt abordables : principalement le théorème d'Euler.

③ La fabrique de nombres premiers met en œuvre des techniques probabilistes qui s'appuient sur des résultats de la théorie des nombres assez pointus permettant d'estimer la proportion de nombres premiers, la proportion de témoins de Miller, ou encore la proportion d'éléments inversibles.

Nous vérifions expérimentalement ces théorèmes en TP !

④ Les algorithmes de chiffrement sont simples et efficaces en pratique, ce qui permet à RSA d'être l'un des systèmes de chiffrement asymétrique les plus rapides de nos jours.

- ✔ *L'algorithme RSA*
- ✔ *Un exemple pour comprendre*
- ✔ *Validité et sécurité*
- ✔ *La technique du padding*
- ✔ *La fabrique de clefs RSA*
- ☛ *L'encodage des suites d'octets*

Le RFC 2437

La règle d'encodage numérique d'une *suite d'octets* est spécifiée ainsi par le RFC 2437 :

```
OS2IP converts an octet string to a nonnegative integer.
Input: X, octet string to be converted
Output: x, corresponding nonnegative integer
Steps:
1. Let X_1 X_2 ... X_l be the l octets of X from first to last,
   and let x_{l-i} have value X_i for 1<= i <= l.
2. Let x = x_{l-1}*256^{l-1}+x_{l-2}*256^{l-2}+...+x_1*256+x_0.
3. Output x.
```

Autrement dit, le texte X formé par une suite d'octets $X[0], ..., X[l-1]$ (dans cet ordre) sera codé par l'entier

$$x = \sum_{k=0}^{l-1} X[l-1-k] \times 256^k$$

c'est-à-dire que x est simplement le nombre s'écrivant X *en base 256*...

Le RFC 2437

La règle de décodage d'un nombre suit le procédé bien connu de l'écriture en base $b = 256$ d'un nombre x à l'aide de *divisions euclidiennes successives.*

```
I2OSP converts a nonnegative integer to an octet string of
a specified length.
Input: x, nonnegative integer to be converted
       l, intended length of the resulting octet string
Output: X, corresponding octet string of length l; or
        "integer too large"
Steps:
1. If x>=256^l, output "integer too large" and stop.
2. Write x in its unique l-digit representation base 256:
   x = x_{l-1}*256^{l-1}+x_{l-2}*256^{l-2}+...+x_1*256+x_0
   where 0 <= x_i < 256 (note that one or more leading
   digits will be zero if x < 256^{l-1}).
3. Let the octet X_i have the value x_{l-i} for 1 <= i <= l.
   Output the octet string: X = X_1 X_2 ... X_l.
```

- ✔ *L'algorithme RSA*
- ✔ *Un exemple pour comprendre*
- ✔ *Validité et sécurité*
- ✔ *La technique du padding*
- ✔ *La fabrique de clefs RSA*
- ✔ *L'encodage des suites d'octets*
- ☛ *Techniques d'accélération*

Pour chiffrer plus vite

Pour chiffrer rapidement, il est intéressant d'avoir une clef publique e petite, car le nombre de multiplications (et donc le temps de calcul) est proportionnel à la taille de e.

En pratique, on choisit donc e, plutôt petit, puis d.

La plus petite clef possible est $e = 3$ (car e est nécessairement impair et $e \neq 1$).

C'est cependant une mauvaise idée car des attaques particulières sont connues pour ce cas précis. C'est pourtant un choix assez souvent adopté...

L'ANSSI recommande de prendre pour e une valeur « strictement supérieure à $2^{16} = 65536$. »

En réduisant le nombre de bits égaux à 1 dans l'écriture de e en binaire, on réduit aussi le nombre de multiplications. Or, $65537 = 2^{16} + 1$. C'est donc une valeur intéressante à utiliser si possible (quitte à changer de valeurs pour p ou q).

Pour déchiffrer plus vite

Nous disposons naturellement pour déchiffrer des valeurs de d et n (c'est le minimum) mais aussi de e (qui est publique). Ces trois données permettent, à elles seules, de retrouver p et q et donc de calculer $d_p = d \pmod{p-1}$ et $d_q = d \pmod{q-1}$. Nous aurons également besoin de $\tilde{q} = q^{-1} \pmod{p}$, l'inverse de q modulo p.

Ces données supplémentaires sont en général fournies avec la clef privée : il est inutile les recalculer à chaque déchiffrement.

Pour déchiffrer rapidement, on calcule d'abord

- $a = c^{d_p} \pmod{p}$;
- $b = c^{d_q} \pmod{q}$;
- $h = \tilde{q} \times (a - b) \pmod{p}$.

Et alors il suffit d'appliquer la **formule de Garner** :

$$m = b + h \times q$$

Forme standard d'une clef privée

Selon la norme PKCS#1 (ou la RFC 3447), une clé privée RSA peut se représenter de plusieurs façons. Pour l'une d'elles, la structure ASN.1 est effectivement la suivante :

```
RSAPrivateKey ::= SEQUENCE {
  version           Version,
  modulus           INTEGER,  -- n
  publicExponent    INTEGER,  -- e
  privateExponent   INTEGER,  -- d
  prime1            INTEGER,  -- p
  prime2            INTEGER,  -- q
  exponent1         INTEGER,  -- d mod (p-1)
  exponent2         INTEGER,  -- d mod (q-1)
  coefficient       INTEGER,  -- (inverse of q) mod p
  otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

Forme standard d'une clef publique

Les clefs publiques sont en revanche sans surprise :

```
RSAPublicKey ::= SEQUENCE {
   modulus            INTEGER,   -- n
   publicExponent     INTEGER    -- e
}
```