Cryptographie

Sandrine Caruso

10 mai 2009

Introduction

Lors d'une guerre, on peut avoir besoin de transmettre des messages, sans que l'ennemi ne soit capable de les déchiffrer. C'est ainsi que, dans l'Histoire, de nombreuses méthodes de cryptographie ont été mises au point. Avec l'apparition des technologies actuelles, l'utilisation de la cryptographie a explosé, non plus seulement à des fins guerrière, mais pour protéger toutes sortes d'informations circulant sur des réseaux comme internet ; ainsi a-t-elle pris une place importante dans notre vie de tous les jours, lorsque nous faisons un achat par carte bancaire, ou retirons de l'argent, etc...

En outre, la puissance des ordinateurs permet de briser de nombreux codes, et oblige à trouver des systèmes bien plus performants que ceux utilisés par le passé.

Voici un aperçu de la cryptographie à travers les siècles.

1 La méthode de Jules César

La méthode de cryptographie qu'utilisait Jules César repose sur un principe très simple. Il s'agissait, tout simplement, de décaler l'alphabet d'un certain nombre de lettres. Par exemple, remplacer A par D, B par E, ..., Z par C. Les deux personnes qui veulent s'envoyer des messages conviennent d'un nombre n secret (la *clé*) ; celui qui code décale toutes les lettres de n, et celui qui décode n'a plus qu'à décaler toutes les lettres de $-n$ pour retrouver le message initial.

Exercice 1. Suppose que tu interceptes le message ci-dessous envoyé par tes ennemis, et que tu sais qu'il est codé avec la méthode de Jules César, en utilisant notre alphabet de 26 lettres. Saurais-tu le décoder ?

```
uvbzzvttlzlujpuxbhualhchuaqlzbzjoypzaavbalshnhbsll
zavjjbwllwhyslzyvthpuzavbaluvubuwlapacpsshnlwlbwsl
kpyylkbjapislznhbsvpzylzpzallujvyllaavbqvbyzhsluch
opzzlbylashcplulzawhzmhjpslwvbyslznhyupzvuzyvthpul
zcvpzpulzklihihvybthxbhypbtshbkhubtlawlapivubt
```

Cette méthode de cryptographie est peut-être suffisamment efficace lorsque on ne peut écrire qu'en gravant dans la pierre, mais dès qu'on a une feuille et un crayon (ou *a fortiori* un ordinateur), le déchiffrement est très facile.

Une méthode pour améliorer ce code est la suivante. Au lieu de se contenter de décaler chaque lettre du même nombre de lettres, on peut choisir une permutation quelconque des lettres.

Exercice 2. Montre qu'il existe $26!$ permutations différentes de 26 lettres.

Si l'on intercepte un message codé à l'aide d'une permutation quelconque, on pourrait essayer de le déchiffrer en générant tous les messages obtenus avec toutes les permutations possibles... Mais le nombre $26!$ est de l'ordre de 10^{26}. Les ordinateurs les plus puissants existant actuellement atteignent au plus 10^{15} opérations par seconde. Ainsi, générer tous les messages possibles prend environ 10^{11} secondes, c'est-à-dire plus de 3000 ans ! Bref, ce n'est pas très raisonnable[1].

Pourtant, une technique efficace a été élaborée pour briser tout de même très rapidement un tel code. Supposons que l'on connaisse la langue dans laquelle a été rédigée le message. Il se trouve que, dans une langue donnée, toutes les lettres n'apparaissent pas à la même fréquence ! Par exemple, en français, les pourcentages d'apparition des lettres sont, au centième de % près, les suivants :

e	a	i	s	t	n	r	u	l	o	d	m	p
15,87	9,42	8,41	7,90	7,26	7,15	6,46	6,24	5,34	5,14	3,39	3,24	2,86
c	v	q	g	b	f	j	h	z	x	y	k	w
2,64	2,15	1,06	1,04	1,02	0,95	0,89	0,77	0,32	0,30	0,24	0,00	0,00

Si l'on intercepte un message suffisamment long, les fréquences d'apparition des lettres devraient être proches de celles ci-dessus. Par exemple, il y a de fortes chances pour que la lettre qui apparaît le plus souvent soit un e. Celle qui arrive en deuxième n'est pas forcément un a, car les fréquences d'apparition des lettres suivantes (i, s, t...) sont assez proches. Mais on peut tenir compte d'autres critères. Par exemple, si une lettre arrive souvent doublée, ce n'est probablement pas un a ou un i, mais plutôt un s ou un t... De proche en proche, on peut ainsi reconstituer assez rapidement tout le texte.

Exemple 3. Dans le message codé de l'exercice 1, les fréquences des lettres étaient les suivantes :

l	z	h	b	p	a	u	v	y	s	t	w	j
13,82	9,76	8,94	8,54	7,32	7,32	7,32	6,91	5,69	5,28	3,25	3,25	2,85
c	k	n	i	x	q	o	m	g	e	f	r	d
2,03	1,63	1,63	1,63	0,81	0,81	0,81	0,41	0	0	0	0	0

Ces fréquences sont-elles cohérentes avec celles du tableau précédent ? Aurais-tu pu deviner immédiatement de combien les lettres avaient été décalées ?

[1] En réalité, on pourrait arriver à un temps raisonnable en faisant travailler des milliers d'ordinateurs en même temps... Mais passons...

2 Un code un peu plus sophistiqué...

Pour éviter la méthode de déchiffrement utilisant la fréquence des lettres, on peut faire en sorte qu'une même lettre ne soit pas toujours remplacée par la même autre. Un tel code est le suivant. La clé secrète, au lieu d'être un nombre (le décalage, dans le code de Jules César), ou une permutation, est un mot. Prenons, par exemple, le mot « tutorat ». Le principe du code est alors le suivant. On regarde le rang de chacune des lettres de ce mot dans l'alphabet (ici, 20, 21, 20, 15, 18, 1, 20), et on décale la première lettre du message de 20, la deuxième de 21, la troisième de 20, etc... Lorsqu'on arrive à la fin du mot-clé, on recommence. Par exemple, voici un message (en bleu) et la manière dont il sera codé (en rouge) :

lecarredelalongueurdelhypothenuseestegalalasomme
tutorattutorattutorattutorattutorattutorattutora
fzwpjsyxzfpdphapyjjeyfcsegubyiohwfmnzapdbfunibef

descarresdeslongueursdesdeuxautrescotes
ttutorattutorattutorattutorattutorattut
xynwpjsymyyhdphapyjjtxynxtmyuooltkdinzm

Dans l'exemple ci-dessus, on peut observer un piège qui permettra à une personne mal intentionnée de décoder facilement notre message. En effet, dans le message d'origine, les suites de caractères « carre » et « longueur » sont présentes deux fois, et il se trouve que la longueur du mot « tutorat » fait que les deux occurrences de « carre » et les deux occurrences de « longueur » sont codées de la même façon : « wpjsy » et « dphapyjj » respectivement. Dans notre exemple, il ne serait pas difficile d'éviter ce problème : il suffit de choisir une clé qui a une lettre de plus, par exemple. Mais en pratique, lorsque le texte envoyé est long, ce genre de « coïncidence » (disons, des groupes de trois ou quatre lettres consécutives que l'on retrouve souvent) a en fait de fortes chances de se produire.

Exercice 4. Supposons que tu interceptes le texte en rouge, et que tu ne connaisses ni le texte en bleu, ni la clé. Comment déduirais-tu, de la répétition des chaînes de caractères « wpjsy » et « dphapyjj » , les longueurs possibles du mot-clé ?

Remarque que les chaînes de caractères « wpjsy » et « dphapyjj » sont espacées du même nombre de caractères la première et la deuxième fois, de sorte que le fait d'avoir deux chaînes répétées ne nous donne pas plus d'information sur la longueur de la clé que si on en avait qu'une. Mais si l'espacement n'était pas le même, vois-tu comment on pourrait avoir encore plus de précision sur la longueur de la clé ?

Si le texte intercepté est suffisamment long et que tu as réussi à trouver la longueur de la clé, comment utiliser la technique de la partie précédente pour déchiffrer le message ?

Exercice 5. Sauras-tu décoder le message suivant ?

qcrfpgjbtqfnumijljxctxrtlyaxgyjltryqzqtfjpxmbrcsjq
rfpgjqsjqsjsmfsiwchhmijq

Ce code n'est donc pas encore suffisamment sûr. Pourtant, pour résoudre le problème ci-dessus, il suffit de choisir comme clé un « mot » (une suite de caractères) qui soit à peu près de la même longueur que le texte à coder. Les problèmes de répétition disparaissent alors. Ce code-là est sûr (et est encore actuellement utilisé par les militaires). Son principal inconvénient par rapport aux codes précédents est que la clé est longue, et plus difficile à stocker.

3 Codes à clé publique

Les codes présentés ci-dessus ont cependant un inconvénient supplémentaire : si l'on connaît la clé pour coder un message, on sait aussi décoder celui-ci, et il faut donc être très prudent lorsqu'on échange une clé avec son interlocuteur. C'est d'autant plus compliqué si l'interlocuteur habite de l'autre côté du monde ! Pendant longtemps, on a dû transporter ces clés dans des mallettes verrouillées et ultra-protégées.

Mais le développement des technologies, et l'utilisation de la cryptographie dans la protection des données bancaires, le commerce en ligne, *etc*, rend impossible de telles transmissions de clés. Prenons l'exemple du commerce en ligne. Pour que les paiements soient sécurisés, il faudrait que l'entreprise donne une clé différente à chacun de ses clients... Clé qui devrait être échangée de manière parfaitement sûre (donc certainement pas par mail ou par téléphone !).

Quels systèmes de cryptographie ont donc rendu le commerce en ligne possible ? L'idée géniale est la suivante : la clé pour chiffrer un message ne doit pas être la même que la clé pour déchiffrer. Une seule personne est en possession de la clé de déchiffrement (appelée *clé privée*), et il divulge publiquement la clé de chiffrement (appelée *clé publique*). Ainsi, tout le monde peut lui envoyer un message chiffré (par exemple, le code de sa carte banquaire), mais il est le seul à pouvoir le déchiffrer. Bien sûr, pour que cela soit sûr, il faut qu'on ne puisse pas retrouver la clé privée en connaissant la clé publique ! Ou du moins, que cela ne soit pas faisable en un temps raisonnable ! Comment est-ce possible ?

3.1 Notion de complexité. Fonctions à sens unique

La *complexité* est une notion d'informatique qui évalue le nombre d'*opérations élémentaires* nécessaires pour exécuter un programme. Notons que le terme d'*opération élémentaire* est un peu flou... Il peut s'agir des manipulations « de base » réalisées « physiquement » par l'ordinateur (lecture d'un bit, modification d'un bit), mais il est en général compliqué de calculer la complexité selon ces paramètres, et on la calcule souvent en choisissant pour « opérations élémentaires » des opérations un peu plus compliquées (des additions, des multiplications, par exemple). Un informaticien patenté qui veut faire des calculs précis de complexité doit être conscient que toutes ces opérations ne prennent pas le même temps (une multiplication « coûte plus cher » qu'une addition, qui « coûte plus cher » qu'une lecture de bit...), mais nous ne nous en préoccuperons pas trop.

Exemple 6. Voici un algorithme qui, étant donné un réel a et un entier positif n, calcule a^n.

```
Entrée : a et n
Sortie : a^n
r <- 1
pour i de 1 à n faire
  r <- r*a
fin pour
renvoyer r
```

Exercice 7. Pourquoi cet algorithme renvoie-t-il bien a^n ? Combien de multiplications effectue-t-il ?

Exemple 8. On peut calculer a^n de manière plus efficace, à l'aide de l'algorithme suivant. Nous le programmons de manière *récursive*, c'est-à-dire que le programme s'appelle lui-même.

```
Entrée : a et n
Sortie : a^n
puiss(a,n) =
  si n = 0 alors renvoyer 1
  sinon
   p <- puiss(a,[n/2])
   si n est pair renvoyer p*p
   sinon renvoyer p*p*a
   fin si
  fin si
```

$[n/2]$ désigne la partie entière de $n/2$. Remarque l'appel récursif à la ligne 6. Expliquons un peu ce qu'il se passe.

Par exemple, que va faire le programme si on lui demande de calculer 2^2 ? Il commence à lire l'algorithme. Un premier test : n est-il égal à 0 ? Non, il est égal à 2. On passe donc à la suite. L'algorithme demande de calculer $puiss(a, [n/2])$.

Bien, on recommence donc à lire depuis le début, mais avec les paramètres $a = 2$ et $n = [2/2] = 1$. Est-ce que n est égal à 0 ? Toujours pas, n est égal à 1.

On essaie donc à nouveau de calculer $puiss(a, [n/2])$. On entre une troisième fois dans le programme avec les paramètres $a = 2$ et $n = [1/2] = 0$. Ah, cette-fois ci, le test $n = 0$ est vrai ! Bien, on sait alors quoi renvoyer : 1.

Mais n'oublions pas que 1, c'était égal à $puiss(a, [n/2])$ où a et n étaient les paramètres de la fois d'avant, la deuxième fois où l'on est entré dans le programme : $a = 2$ et $n = 1$, et donc $puiss(2, [1/2]) = 1$. Que dit la suite du programme ? Tester si n est pair... n est pour le moment égal à 1, il n'est pas pair. On renvoie $p \times p \times a$, c'est-à-dire $1 \times 1 \times 2 = 2$.

Bon, maintenant, ce résultat, 2, est ce qu'on voulait calculer en demandant $puiss(2, [2/2])$, la première fois que l'on est entré dans le programme, avec $a = 2$ et $n = 2$. Alors, maintenant que l'on a obtenu ce résultat, on continue. Est-ce que n est pair ? Ici, n est égal à 2. Il est bien pair. On renvoie donc $p \times p$, c'est-à-dire $2 \times 2 = 4$. Cette fois-ci, ce résultat 4 est bien le résultat que l'on cherchait.

Ainsi, le programme stocke en mémoire, au fur et à mesure, ce qu'il doit calculer, jusqu'à arriver à quelque chose qu'il sait calculer directement. Alors, il remonte petit à petit, calculant de proche en proche tout ce qu'il avait stocké.

Cet algorithme paraît bien compliqué, et au vu de l'exemple de 2^2, pas plus efficace que le précédent. C'est vrai que pour de petites valeurs de n, il sera plus lent. Mais dès que n devient un peu plus grand, il est effectivement plus rapide.

Exercice 9. Pourquoi cet algorithme se termine-t-il forcément au bout d'un moment ?

Supposons pour simplifier que n est une puissance de 2 : $n = 2^k$. Notons M_k le nombre de multiplications qu'il faut effectuer pour calculer a^{2^k}. Est-ce que M_k dépend de a ? Exprime M_k en fonction de M_{k-1}, et calcule M_0. Déduis-en M_k en fonction de k.

Compare M_k au nombre de multiplications nécessaires pour calculer a^{2^k} à l'aide de l'algorithme précédent. À partir de quelle valeur de k est-il plus intéressant d'utiliser le deuxième algorithme ?

Ces exemples montrent que, pour calculer la même chose, il existe des algorithmes plus efficaces que d'autres.

La notion de complexité est très importante en cryptographie. En effet, la complexité est directement reliée au temps de calcul : plus il y a d'opérations à faire, plus l'ordinateur mettra du temps. Et il peut parfois mettre beaucoup, beaucoup de temps ! (Rappelle-toi les permutations de la partie 1. Calculer $26!$ prenait énormément de temps !)

Lorsqu'on chiffre un message, on applique en fait une *application bijective*, c'est-à-dire une fonction f qui va d'un ensemble E dans un ensemble F, telle que :

– à tout élément x de E, on associe un unique élément $f(x)$ de F (f est une *application*),

– pour tout élément y de E, il existe un unique $x \in E$ tel que $f(x) = y$ (f est *bijective*)

Pour déchiffrer le message, on applique *l'application réciproque* de f, c'est-à-dire *l'unique application bijective* g telle que pour tout $x \in E$, $g(f(x)) = x$ et pour tout $y \in F$, $f(g(y)) = y$.

Dans nos exemples, l'ensemble E est l'ensemble des messages que l'on peut chiffrer, et l'ensemble F est l'ensemble des messages chiffrés que l'on peut obtenir. Récapitulons les applications f de chiffrage et g de déchiffrage que l'on a utilisées :

1. Code de Jules César : f est la fonction qui décale toutes les lettres d'un message de n, g est la fonction qui décale toutes les lettres d'un message de $-n$.

1 bis. Permutation : f est la fonction qui applique à toutes les lettres d'un message une certaine permutation σ, g est la fonction qui applique à toutes les lettres d'un message la permutation réciproque de σ.

Remarque : j'emploie ici le vocabulaire *permutation réciproque*. Ce n'est pas un hasard. Une permutation σ des 26 lettres de l'alphabet n'est rien d'autre qu'une application bijective allant de l'ensemble des lettres de l'alphabet dans ce même ensemble : à toute lettre de l'alphabet, elle en associe une unique autre. Pour toute lettre y de l'alphabet, il y a une unique lettre x telle que $\sigma(x) = y$.

2. Décalage variable : notons $n_1, n_2, \ldots, n_k$ le rang des lettres du mot-clé. f est l'application qui décale la première lettre de n_1, la deuxième de n_2, ..., la k-ième de n_k, la $(k+1)$-ième de n_1, etc... g est l'application qui décale la première lettre de $-n_1$, la deuxième de $-n_2$, ..., la k-ième de $-n_k$, la $(k+1)$-ième de $-n_1$, etc...

Dans les exemples précédents, il n'était pas difficile, connaissant f, de calculer g (essaie de calculer rapidement la permutation réciproque d'une permutation donnée).

Une *fonction à sens unique* est une fonction qui est facile à calculer (c'est-à-dire dont la complexité est peu élevée), mais dont il est difficile à calculer un inverse (cela prend trop de temps). C'est une telle fonction dont nous avons besoin pour mettre en place un système de chiffrage *asymétrique*, ou *à clé publique*.

3.2 Un peu d'arithmétique

Soient $x, y \in \mathbb{Z}$, $n \in \mathbb{N}^*$. On dit que x *est congru à* y *modulo* n, et on écrit $x \equiv y \pmod{n}$, s'il existe un entier $k \in \mathbb{Z}$ tel que $x = y + kn$.

Exercice 10. Montre que $x \equiv y \pmod{n}$ si et seulement si x et y ont le même reste dans la division euclidienne par n.

On appelle *classe de* x *modulo* n l'ensemble des nombres qui sont congrus à x modulo n. La classe de x modulo n sera parfois notée $\overline{x}$, s'il n'y a pas d'ambiguïté sur l'entier n.

Exercice 11. Combien y a-t-il de différentes classes modulo n ?

Exercice 12. Montre que tout entier $x \in \mathbb{Z}$ est dans une certaine classe modulo n.

Montre que, si deux classes modulo n ont un élément commun, alors elles sont égales (c'est-à-dire qu'elles ont exactement les mêmes éléments).

On note $\mathbb{Z}/n\mathbb{Z}$ l'ensemble des classes modulo n.

Addition dans $\mathbb{Z}/n\mathbb{Z}$. Soient X et Y deux classes modulo n. On voudrait définir $X + Y$. Soit x un élément de X, y un élément de Y (x et y sont donc des nombres entiers). On a envie de définir $X + Y$ comme la classe modulo n de l'entier $x + y$. Mais il faut vérifier que cela ne dépend pas des éléments x et y que l'on a choisis ! Soient $x' \in X$ et $y' \in Y$ deux autres éléments des classes X et Y.

Pour vérifier que l'on peut bien définir $X + Y$ comme expliqué ci-dessus, il suffit de vérifier que la classe de $x' + y'$ est la même que la classe de $x + y$. En effet, comme x' et y' sont quelconques dans X et Y, cela montrera que pour n'importe quel choix de x dans X et de y dans Y, la classe de $x + y$ est toujours la même. On pourra donc bien définir $X + Y = \overline{x + y}$.

Montrons donc que les classes $\overline{x + y}$ et $\overline{x' + y'}$ sont égales. On a vu, à l'exercice 12, que si deux classes ont un élément commun, alors elles sont en fait égales. Il suffit donc de montrer que ces deux classes $\overline{x + y}$ et $\overline{x' + y'}$ ont un élément commun. On a $x + y = x + y + 0 \times n$, donc $x + y \in \overline{x + y}$. Maintenant, comme x et x' sont dans la même classe X, il existe $k_1 \in \mathbb{Z}$ tel que $x = x' + k_1 n$. De même, comme y et y' sont dans la même classe Y, il existe $k_2 \in \mathbb{Z}$ tel que $y = y' + k_2 n$. Alors $x + y = x' + y' + (k_1 + k_2)n$, et donc $x + y \in \overline{x' + y'}$. Ainsi, $x + y$ est un élément commun à $\overline{x + y}$ et $\overline{x' + y'}$, donc ces deux classes sont égales.

On peut définir la soustraction dans $\mathbb{Z}/n\mathbb{Z}$ exactement de la même façon.

Produit dans $\mathbb{Z}/n\mathbb{Z}$. Soient X et Y deux classes modulo n. Soient $x \in X$ et $y \in Y$. Montre que l'on peut définir le produit XY comme la classe de xy.

Inverse. Essayons maintenant de voir si l'on peut définir une division. En fait, puisqu'on sait définir la multiplication, on peut ramener notre problème de division à un problème *a priori* plus simple, celui de l'inversion. En effet, si x et y sont deux nombres ($y \neq 0$), alors $\frac{x}{y} = x \times \frac{1}{y}$. Si l'on sait trouver un nombre qui est l'inverse de y, on peut multiplier x par ce nombre pour obtenir $\frac{x}{y}$.

Mais ici, nous avons un problème. En effet, si y est un entier, son inverse $\frac{1}{y}$, en revanche, ne reste pas entier (sauf si $y = 1$ ou $y = -1$) ! Or, lorsqu'on étudie la congruence et les classes modulo n, c'est uniquement avec des entiers que l'on travaille. Impossible, donc, de définir un inverse comme on l'a fait précédemment avec l'addition et la multiplication !

Mais réfléchissons un peu à ce qu'est, au fond, l'inverse d'un nombre y. C'est tout simplement un nombre z tel que $yz = 1$. Bien sûr, ce nombre est forcément égal à $\frac{1}{y}$... Mais dans $\mathbb{Z}/n\mathbb{Z}$, on essaye de faire des opérations sur des objets, les classes modulo n, qui ne sont pas des nombres, mais des ensembles... Maintenant que nous avons défini la multiplication, est-il possible, étant donnée une classé Y, de trouver une classe Z telle que $YZ = \overline{1}$ (la classe de 1) ? Même si nous ne pouvons pas définir la « classe de $\frac{1}{y}$ » (avec $y \in Y$), puisque $\frac{1}{y}$ n'est pas entier, peut-être existe-t-il quand même une classe Z qui vérifie $YZ = \overline{1}$, et que l'on pourrait alors appeler *l'inverse de* Y. C'est ce que nous allons essayer d'étudier.

Exercice 13. Soient x et y deux entiers. Quelle relation de congruence doivent vérifier x et y (modulo n) pour que l'on ait $\overline{x} \times \overline{y} = \overline{1}$?

Voici un théorème très important en arithmétique :

Théorème (Bézout)**.** *Soient a et b deux entiers relatifs. Alors a et b sont premiers entre eux si et seulement si il existe deux entiers relatifs u et v tels que $au + bv = 1$.*

Ces deux nombres u et v sont appelés des *coefficients de Bézout* de a et b.

Exercice 14. Remarque que si a et b ne sont pas premiers entre eux, alors il existe des entiers u et v (également appelés *coefficients de Bézout* de a et b) tels que $au + bv = \text{PGCD}(a, b)$. Saurais-tu le montrer ? Que penser de la réciproque (*ie*, s'il existe des entiers u et v tels que $au + bv = d$, est-ce que d est le PGCD de a et b) ?

Exercice 15. Soit x un entier. En utilisant le théorème de Bézout, montre que la classe $\overline{x}$ de x modulo n admet un inverse dans $\mathbb{Z}/n\mathbb{Z}$ si et seulement si x est premier avec n. Comment s'exprime cet inverse en fonction des coefficients de Bézout de x et n ?

On note Y^{-1} l'inverse de Y, s'il existe.

Ainsi, dans $\mathbb{Z}/n\mathbb{Z}$, certaines classes admettent des inverses, et d'autres pas... Pourtant, il existe certaines valeurs de n pour lesquelles tout se passe bien :

Exercice 16. Montre que si n est un nombre premier, alors toute classe modulo n différente de la classe de 0 admet un inverse dans $\mathbb{Z}/n\mathbb{Z}$.

Voici un algorithme que tu connais probablement bien : l'algorithme d'Euclide. Soient a et b deux entiers positifs, $b \neq 0$. On veut calculer leur PGCD. On suppose que l'on dispose d'une fonction *reste*, qui prend deux arguments, et qui calcule le reste de la division euclidienne du premier par le deuxième.

```
Entrée : a et b
Sortie : pgcd(a,b)
x <- a
y <- b
tant que r non nul faire
```

```
  r <- reste(x,y)
  x <- y
  y <- r
fin tant que
renvoyer x
```

Exercice 17. Pourquoi cet algorithme se termine-t-il toujours au bout d'un moment ? Pourquoi donne-t-il bien le PGCD de a et b ?

On peut modifier l'algorithme ci-dessus pour qu'il ne renvoie pas seulement le PGCD de a et b, mais également des coefficients de Bézout. (Ainsi, d'après l'exercice 15, on sera capable de calculer un inverse d'une classe modulo n, si cet inverse existe.) Voici cet algorithme, qu'on appelle *algorithme d'Euclide étendu*. On suppose qu'en plus de la fonction *reste*, on dispose d'une fonction *quotient* qui calcule le quotient de la division euclidienne.

```
Entrée : a et b
Sortie : u, v, pgcd(a,b) avec au + bv = pgcd(a,b)
x <- a
y <- b
u1 <- 1
v1 <- 0
u2 <- 0
v2 <- 1
tant que r non nul faire
  r <- reste(x,y)
  q <- quotient(x,y)
  u <- u1 - q*u2
  v <- v1 - q*v2
  u1 <- u2
  v1 <- v2
  u2 <- u
  v2 <- v
  x <- y
  y <- r
fin tant que
renvoyer u, v, x
```

Exercice 18. Montre qu'après chaque étape de la boucle *tant que*, les paramètres vérifient $au + bv = r$. Déduis-en que l'algorithme renvoie bien des coefficients de Bézout et le PGCD.

Si l'on veut seulement un inverse de la classe de a modulo b, est-il vraiment utile de calculer à chaque étape v, $v1$ et $v2$?

L'ensemble des classes *inversibles* modulo n est noté $(\mathbb{Z}/n\mathbb{Z})^\times$ (ou $(\mathbb{Z}/n\mathbb{Z})^*$).

Exercice 19. Montre que $\overline{1} \in (\mathbb{Z}/n\mathbb{Z})^\times$, et que si X est dans $(\mathbb{Z}/n\mathbb{Z})^\times$, son inverse aussi. Montre que si X et Y sont dans $(\mathbb{Z}/n\mathbb{Z})^\times$, alors XY aussi.

3.3 Logarithme discret

Soit p un nombre premier.

Exercice 20. Pourquoi $(\mathbb{Z}/p\mathbb{Z})^\times$ a-t-il $p-1$ éléments ?

On admet qu'il existe un élément $B \in (Z/p\mathbb{Z})^\times$ tel que tout élément de $(\mathbb{Z}/p\mathbb{Z})^\times$ s'écrit sous la forme B^k pour un certain entier k (où B^k est égal à $B \times B \times \cdots \times B$, k fois). On dit que B est un *générateur* de $(\mathbb{Z}/p\mathbb{Z})^\times$.

Exercice 21. Montrer que B^{p-1} est égal à la classe de 1 modulo p. En déduire que si k' est dans la classe de k modulo $p-1$, alors $B^{k'} = B^k$.

Exercice 22. Déduire de l'exercice précédent qu'on peut définir une application f de $\mathbb{Z}/(p-1)\mathbb{Z}$ dans $(\mathbb{Z}/p\mathbb{Z})^\times$, qui à une classe $\overline{k}$ modulo $p-1$ associe B^k. Montrer que f est bijective.

La fonction inverse de f est appelée *logarithme discret* (en *base* B).

Cette fonction f est une fonction à sens unique. En effet, si l'on connaît B et $\overline{k}$ (et p), il est facile (et rapide !) de calculer B^k, avec le deuxième algorithme présenté dans la partie 3.1. En revanche, si l'on connaît B^k et B (et p), il n'est pas facile de trouver k ; si le nombre premier p est assez grand, c'est impossible en un temps raisonnable, car la complexité des algorithmes connus actuellement est trop élevée.

Voici donc comment le cryptosystème par *logarithme discret* fonctionne. Traditionnellement, on appelle Alice la personne qui possède la clé privée, et Bob la personne qui veut envoyer un message codé à Alice.

Alice choisit un grand nombre premier p, et un générateur B de $(\mathbb{Z}/p\mathbb{Z})^\times$. Puis elle choisit un entier k, et elle calcule B^k et obtient une classe C. Elle envoie alors à Bob la clé publique (p, B, C), et garde secret k, qui est la clé privée.

Bob représente le message à coder par une classe (non nulle) modulo p (en pratique, par un entier naturel non nul strictement plus petit que p. Pourquoi cela revient-il au même ?), que l'on note m. Puis il choisit un nombre entier i (qu'il garde secret) et calcule les classes $A_1 = B^i$ et $A_2 = m \times C^i$ modulo p. Il envoie à Alice A_1 et A_2.

Pour décoder le message, Alice calcule $(A_1^k)^{-1} \times A_2$ et retrouve ainsi m.

Exercice 23. Vérifie que $(A_1^k)^{-1} \times A_2$ est bien égal à m. Si un tiers (qui ne connaît donc ni k ni i) intercepte le message chiffré (A_1, A_2), pourquoi ne peut-il *a priori* pas en déduire m ?

L'algorithme ci-dessus peut aussi être utilisé pour signer un message, afin de s'assurer que la personne qui nous a envoyé le message est bien celle que l'on pense. Voici comment Alice signerait un message. Alice et Bob possèdent chacun leur propre clé secrète, respectivement k et i, modulo le même nombre premier p, avec la même base B. Ils se communiquent leurs clés publiques $C_A = B^k$ et $C_B = B^i$ respectivement. Puis Alice envoie un message anodin m à Bob, ainsi que le message chiffré à l'aide de la clé publique de Bob et de sa propre clé privée, $m \times C_B^k$. Alors Bob code m avec la clé publique d'Alice et sa propre clé privée et obtient donc $m \times C_A^i$. Il vérifie que cela donne la même chose que le deuxième message, et en déduit que c'est bien Alice qui l'a envoyé.

Exercice 24. Explique ce système de signature.

3.4 RSA

Le système de cryptographie RSA a pour nom les initiales de ses inventeurs, R. Rivest, A. Shamir et L. Adleman. Voici son principe.

Alice choisit deux grands nombres premiers p et q, et elle calcule $n = pq$.

Exercice 25. Montre que l'ensemble $(\mathbb{Z}/n\mathbb{Z})^\times$ des classes inversibles modulo n a $(p-1)(q-1)$ éléments.

De manière générale, si r est un entier, le nombre d'éléments de $(\mathbb{Z}/r\mathbb{Z})^\times$ est noté $\varphi(r)$, et est appelé la *caractéristique d'Euler* de r. Ainsi, si $n = pq$, alors $\varphi(n) = (p-1)(q-1)$. On admet le théorème suivant :

Théorème. *Soit r un entier, et soit $X \in (\mathbb{Z}/r\mathbb{Z})^\times$. Alors $X^{\varphi(r)} = \bar{1}$ (classe de 1 modulo r).*

Alice choisit un entier positif e, strictement plus petit que $(p-1)(q-1)$, qui soit premier avec $(p-1)(q-1)$ (cela revient à choisir une classe dans $(\mathbb{Z}/(p-1)(q-1)\mathbb{Z})^\times$. Pourquoi ?) Elle calcule ensuite l'entier positif $k < (p-1)(q-1)$ tel que la classe de k modulo $(p-1)(q-1)$ soit l'inverse de la classe de e modulo $(p-1)(q-1)$.

Exercice 26. Quelle relation de congruence vérifient e et k ?

Ensuite, Alice envoie sa clé publique (n, e) à Bob, et conserve pour elle sa clé privée k. Elle ne dévoile pas non plus p et q, qui permettraient de retrouver k à partir de e. Si Bob veut envoyer un message à Alice, il l'écrit sous forme d'un élément de $\mathbb{Z}/n\mathbb{Z}$ (en pratique, d'un entier strictement plus petit que n), que l'on note m.

Exercice 27. Si p et q ont environ 200 chiffres, peux-tu donner un ordre de grandeur de la probabilité que m ne soit pas premier avec n ?

Ainsi, il y a de très fortes chances pour que le message m soit premier avec n, donc que sa classe modulo n soit dans $(\mathbb{Z}/n\mathbb{Z})^\times$. On suppose que c'est bien le cas.

Pour chiffrer son message, Bob l'élève à la puissance e dans $\mathbb{Z}/n\mathbb{Z}$ (en utilisant l'algorithme de l'exemple 8). Il obtient un message m' qu'il envoie à Alice.

Pour déchiffrer, Alice calcule alors m'^k dans $\mathbb{Z}/n\mathbb{Z}$.

Exercice 28. Pourquoi le calcul de m'^k modulo n redonne-t-il bien le message m ?

3.5 Quelques remarques

Faisons le bilan des opérations à effectuer pour chiffrer et déchiffrer un message avec la méthode RSA, ainsi que des opérations qu'un tiers devrait effectuer pour découvrir la clé secrète en étant en possession de la clé publique.

Choisir deux grands nombres premiers. Les deux nombres premiers doivent rester secrets. Comme des millions de personnes ont besoin régulièrement de clés privées personnelles, cela signifie qu'il faut être capable de générer énormément de nombres premiers, et des nombres premiers très grands (en pratique, actuellement, ils doivent avoir quelques milliers de chiffres). Déjà, il faut savoir qu'il y a « beaucoup » de nombres premiers (de l'ordre de $\frac{n}{\log n}$ nombres premiers plus petits que n...). Ensuite, il y a des algorithmes très efficaces pour tester si des nombres sont premiers, et en pratique, on tire un nombre au hasard et on teste s'il est premier.

Calculer le produit pq. Cette étape est également assez rapide. Bien sûr, comme les nombres premiers sont très grands, les multiplier prend tout de même un certain temps. Mais des algorithmes utilisant le même genre de principe que l'algorithme 8 ont été mis au point pour multiplier rapidement deux très grands entiers.

Choisir e. Choisir un entier qui soit premier avec $(p-1)(q-1)$, c'est facile. On tire un entier au hasard, et on vérifie qu'il est premier avec $(p-1)(q-1)$ (il y a de fortes chances que ce soit le cas) en utilisant l'algorithme d'Euclide. Cet algorithme est rapide. Si on a pas de chance et que le nombre choisi n'est pas premier avec $(p-1)(q-1)$, on en choisit un autre et on réessaie.

Calculer k. Là encore, on utilise l'algorithme d'Euclide (étendu) pour calculer un inverse de e modulo $(p-1)(q-1)$.

Élever à la puissance e ou k. Comme indiqué précédemment, pour élever à la puissance e ou k modulo n, on utilise l'algorithme 8.

Tous les algorithmes utilisés ci-dessus ont ce qu'on appelle une *complexité polynomiale*. Ce qu'il faut retenir, c'est que de tels algorithmes s'effectuent en un temps raisonnable, même s'ils sont appliqués à des nombres très grands. Voyons maintenant ce qu'il faudrait faire pour briser le code.

Trouver k connaissant e et n. Pour calculer k, on a besoin de connaître e et $\varphi(n) = (p-1)(q-1)$. Si l'on développe cette expression, on obtient $pq - p - q + 1 = n - p - q + 1$. Pour calculer ce nombre, il semblerait donc que l'on ait vraiment besoin de connaître p et q.

Factoriser n. Trouver p et q, cela revient à *factoriser* le nombre n : trouver ses facteurs premiers. Il y a bien sûr des algorithmes pour cela, le plus naïf que l'on puisse imaginer étant d'essayer de diviser n par tous les nombres premiers plus petits que n, et même plus petits que $\sqrt{n}$ (Pourquoi ?). Pour cela, on utilise l'algorithme de division euclidienne, qui est très rapide. Mais, trouver tous les nombres premiers plus petit que $\sqrt{n}$, en revanche, c'est beaucoup plus long (on utilise pour cela le crible d'Ératosthène : on entoure 2 et on raye les multiples de 2, on entoure le prochain nombre non rayé : 3 et on raye les multiples de 3, etc...).

Exercice 29. Trouve ainsi tous les nombres premiers plus petits que 100. Essaie d'imaginer le temps que cela prendrait de trouver tous les nombres premiers plus petits que $\sqrt{n}$, si $\sqrt{n}$ est un nombre d'environ 200 chiffres...

Il existe de nombreux autres algorithmes plus rapides pour factoriser n, mais même les plus efficaces actuellement connus ont une complexité qualifiée d'*exponentielle*. Exécuter de tels algorithmes prend beaucoup trop de temps lorsqu'on les applique à de très grands nombres.