Les dessous de la cryptographie à clé publique

Christiane Rousseau
Université de Montréal

De tout temps l'homme a cherché des moyens de transmettre des messages secrets et a inventé des codes secrets de plus en plus sophistiqués. Historiquement on observe très souvent que les meilleurs codes secrets ont été percés par l'adversaire : ce fut le cas avec le code Enigma utilisé par les Allemands et décodé par les Alliés durant la dernière guerre mondiale. La cryptographie à clé publique, ou code RSA, a été introduite en 1978 [RSA] et est abondamment utilisée sur Internet. Le mode de fonctionnement du système est public. La renommée de ce code est si grande que tout chercheur qui réussirait à le briser obtiendrait une gloire immédiate. Pourtant ce système résiste encore depuis plus de 25 ans !

On commencera par expliquer ce système de cryptographie, basé sur la théorie des nombres, plus particulièrement l'arithmétique $(+, .)$ modulo n, et on montrera pourquoi il fonctionne : c'est la partie élémentaire. La clé du système RSA est un entier n qui est le produit de deux grands nombres premiers p et q. Pour briser le code, il suffit de factoriser n. Le code est encore inviolé parce que les meilleurs ordinateurs ne sont pas capables de factoriser de grands nombres entiers dans un temps raisonnable.

Par contre, il est facile de fabriquer une clé avec un ordinateur, c'est-à-dire de construire deux grands nombres premiers et de les multiplier. La naissance du système RSA a donné naissance à de nombreux algorithmes, surtout probabilistes, parfois déterministes, pour générer de grands nombres premiers et tenter de factoriser de grands nombres entiers. On parlera de ces algorithmes anciens et modernes permettant de construire une clé. Les chercheurs mettent donc toute leur énergie sur un algorithme qui permettrait de factoriser de grands nombres entiers en un temps raisonnable (les informaticiens diront en temps polynomial). Shor a publié en 1997 un tel algorithme mais sur un ordinateur quantique. L'ordinateur quantique n'est plus tout à fait une fiction : on a pu construire en 2003 un

ordinateur quantique factorisant le nombre 15. Comme on le voit, la recherche se poursuit sous nos yeux. On terminera par une brève introduction à l'algorithme de Shor.

Le code RSA est un système à clé publique. Ceci signifie que le fonctionnement du code et la clé sont publics ! Les avantages sont les suivants :

- Il n'y a plus de danger que le code soit connu : il est public ! C'est pourquoi c'est le seul type de code qui puisse fonctionner lorsqu'il y a des millions d'utilisateurs.
- Il est possible de « signer » un message de telle sorte qu'on soit sûr de sa provenance.

L'ingrédient de base est la théorie des nombres, plus particulièrement l'arithmétique $(+, .)$ modulo n. En particulier, on utilise le théorème de Fermat généralisé par Euler.

La méthode fonctionne parce que la théorie et la pratique sont très différentes en théorie des nombres :

- Il est difficile en pratique pour un ordinateur de factoriser un grand nombre ;
- Il est facile en pratique de construire de grands nombres premiers ;
- Il est facile en pratique de décider si un grand nombre est premier.

Définition 1 Soit a, b, n des entiers. On dit que « a est congru à b modulo n » si $n|(a-b)$, i.e. il existe un entier c tel que $a - b = cn$. On note $a \equiv b \pmod{n}$.

1 Le principe du code RSA [RSA] :

- On choisit p et q deux grands nombres premiers (plus de 100 chiffres).
- On calcule $n = pq$. Le nombre n, la "clé", a environ 200 chiffres ou plus. Il est public alors que p et q sont gardés secrets.
- On calcule $\varphi(n)$, où φ est la fonction d'Euler définie comme suit : $\varphi(n)$ est le nombre d'entiers premiers dans $E = \{1, 2, \ldots, n\}$ qui sont relativement premiers avec n. Alors $\varphi(n) = (p-1)(q-1)$.
- Calculer $\varphi(n)$ sans connaître p et q est aussi difficile que de factoriser n.
- On choisit $e \in E$ relativement premier avec $\varphi(n)$, i.e. (Le PGCD de e et $\varphi(n)$ est 1). e est la *clé de cryptage*. Elle est publique et sert à l'expéditeur pour encoder son message.

- Il existe $d \in \{1, \ldots, n\}$ tel que $ed \equiv 1 \pmod{\varphi(n)}$. L'existence de d découle de l'algorithme d'Euclide pour trouver le PGCD de e et $\varphi(n)$. d est la *clé de décryptage*. Elle est secrète et permet au receveur de décoder le message.
- L'expéditeur veut envoyer un message m qui est un nombre de $\{1, 2, \ldots, n\}$ relativement premier avec n.
- Il code $m^e \equiv a \pmod{n}$, i.e. $a \in \{1, \ldots, n\}$ et envoie a.
- Le receveur reçoit a. Pour décoder, il calcule $a^d \pmod{n}$. Le petit théorème de Fermat, généralisé par Euler, assure que $a^d \equiv m \pmod{n}$.

Nous allons montrer toutes les étapes ci-dessus qui requièrent une preuve. Nous aurons alors complètement montré que le code RSA fonctionne.

Proposition 1 *Si p et q sont deux nombres premiers distincts et $n = pq$, alors $\varphi(n) = (p-1)(q-1)$.*

Preuve Soit $F = \{1, 2, \ldots, n-1\}$. F contient $n-1$ éléments. Pour calculer $\varphi(n)$ on doit compter le nombre d'éléments qui restent dans F une fois qu'on a enlevé tous les nombres qui ne sont pas premiers avec n, soit tous les multiples de $p : p, 2p, \ldots (q-1)p$ et tous les multiples de $q : q, 2q, \ldots (p-1)q$. Il reste donc $(pq-1) - (q-1) - (p-1) = (p-1)(q-1)$ éléments. □

Proposition 2 *Si $e \in E = \{1, \ldots, n\}$ satisfait $(e, \varphi(n)) = 1$, alors il existe $d \in E$ tel que $ed \equiv 1 \ (mod\, \varphi(n))$.*

Preuve Puisque $(e, \varphi(n)) = 1$, alors par l'algorithme d'Euclide il existe deux entiers a et b tels que $1 = ae + b\varphi(n)$. Donc $ae \equiv 1 \pmod{\varphi(n)}$. Soit $d \in E$ le reste de la division de a par $\varphi(n)$ (ceci a du sens même si $a < 0$). Alors $a = q\varphi(n) + d$ avec $d \in E$. On en déduit que $ae \equiv de \equiv 1 \pmod{\varphi(n)}$. □

Théorème d'Euler : Si m est relativement premier avec n, alors $m^{\varphi(n)} \equiv 1 \pmod{n}$. (Fermat a prouvé le théorème quand n est premier.)

Preuve Soit $S = \{a \in N \mid a < n \text{ et } (a, n) = 1\}$. Alors S contient $\varphi(n)$ éléments : $S = \{a_1, \ldots, a_{\varphi(n)}\}$. Soit m tel que $(m, n) = 1$. Multiplions chaque a_i par m et soit b_i le reste de la division de $a_i m$ par n. Alors $a_i m \equiv b_i \pmod{n}$. Comme $(a_i, n) = 1$ et $(m, n) = 1$, alors $(b_i, n) = 1$. Donc $b_i \in S$. De plus, on peut montrer facilement (exercice) que $a_i \neq a_j$

implique que $b_i \neq b_j$. Donc les éléments $b_1, \ldots, b_{\varphi(n)}$ forment une permutation des éléments de S, ce qui implique que $S = \{b_1, \ldots, b_{\varphi(n)}\}$.
D'où

$$\prod_{i=1}^{\varphi(n)} a_i = \prod_{i=1}^{\varphi(n)} b_i.$$

Or $a_i m \equiv b_i \pmod{n}$. Donc $\prod_{i=1}^{\varphi(n)} b_i \equiv m^{\varphi(n)} \prod_{i=1}^{\varphi(n)} a_i \pmod{n}$. Ceci entraîne que n divise $(m^{\varphi(n)} - 1) \prod_{i=1}^{\varphi(n)} a_i$. Comme $(n, a_i) = 1$, alors n divise $m^{\varphi(n)} - 1$. □

Proposition 3 *Le procédé de décodage du code RSA fonctionne et permet de récupérer le message initial.*

Preuve Soit m le message qui est un nombre de $E = \{1, 2, \ldots, n\}$ relativement premier avec n. Pour coder le message, l'expéditeur calcule $a \in E$ tel que $m^e \equiv a \pmod{n}$. Le receveur calcule :

$$a^d \equiv (m^e)^d = m^{ed} = m^{b\varphi(n)+1} = m^{b\varphi(n)}.m = (m^{\varphi(n)})^b.m \equiv 1.m = m \pmod{n}.$$ □

Signature d'un message avec le code RSA : chacun publie sa clé n et sa clé e :

- expéditeur : n_A, e_A publiques, d_A secrète ;
- receveur : n_B, e_B publiques, d_B secrète.

L'expéditeur veut envoyer un message m relativement premier avec n_A et n_B. Il calcule : $m \mapsto m^{d_A} \equiv m_1 \pmod{n_A} \mapsto m_1^{e_B} \equiv m_2 \pmod{n_B}$.
Il envoie m_2.
Pour décoder le message, le receveur fait : $m_2 \mapsto m_2^{d_B} \equiv m_1 \pmod{n_B} \mapsto m_1^{e_A} \equiv m \pmod{n_A}$.

Exemple 1 (Nous avons utilisé Mathematica pour construire l'exemple suivant). Une compagnie veut instaurer un système de commandes sur Internet. Elle instaure donc un cryptage à clé publique pour la transmission du numéro de carte de crédit.
Le numéro de carte de crédit est un numéro de 16 chiffres auquel on ajoute les 4 chiffres qui correspondent à la date d'expiration, soit un nombre de 20 chiffres.
Elle choisit donc p et q deux grands nombres premiers. Nous fonctionnerons dans notre exemple avec des nombres de 25 chiffres, ce qui donne pour n un nombre de 50 chiffres

environ. Dans notre exemple, prenons

$$p = 976095975111111 2041886431$$

et

$$q = 834552399867834125649111 1.$$

Ceci donne

$$n = 814603238530311544121578649434490335599002230148 41$$

$$\varphi(n) = 814603238530311544121578468369652837704469246373 00.$$

La compagnie choisit $e = 45879256903$ et fait calculer d par Mathematica :

$$d = 614249316518661714502675899921801756121674757401 67.$$

A priori on ne peut envoyer que des messages premiers avec n. Ici aucun problème : les seuls diviseurs de n ont 25 chiffres et donc tout nombre de 20 chiffres est relativement premier avec n. Un client a le numéro de carte de crédit : 1234 5678 9098 7654 et la date d'expiration de sa carte est le 01/06.
On doit donc envoyer le message $m = 12345678909876540106$. Le programme d'envoi calcule :

$$m^e \equiv a = 625176510626059110979407460361990023455526694648 5 (\mathrm{mod}\, n).$$

Le nombre a est transmis. À la réception la compagnie calcule :

$$a^d \equiv 12345678909876540106 = m \ (\mathrm{mod}\, n).$$

Dans cet exemple, les entiers p et q choisis ne sont pas assez grands et un ordinateur pourrait factoriser n.

2 La construction de grands nombres premiers

Nous avons affirmé qu'il est facile de construire de grands nombres premiers. Cela vient du théorème des nombres premiers qui donne la distribution asymptotique des nombres premiers. Pour construire un nombre premier de 100 chiffres, i.e. un élément de $E =$

$\{1, \ldots, 10^{100}\}$, on génère au hasard des nombres entiers de 100 chiffres et on teste s'ils sont premiers. Nous énonçons sans preuve le théorème des nombres premiers. Sa preuve est d'un niveau très avancé.

Théorème des nombres premiers : Si on choisit n au hasard dans $E = \{1, \ldots, N\}$ alors

$$\text{Prob}(n \text{ premier}) \approx \frac{\frac{N}{\ln N}}{N} = \frac{1}{\ln N}.$$

Proposition 4 *On génère au hasard des nombres impairs dans $E = \{1, \ldots, 10^{100}\}$ et on teste s'ils sont premiers. Alors, après en moyenne 115 essais, on trouve un nombre premier.*

Preuve Si $N = 10^{100}$, alors $\text{Prob}(n \text{ premier}) \approx \frac{1}{\ln 10^{100}} = \frac{1}{100 \ln 10} \approx \frac{1}{230}$. Comme un entier sur deux est impair, alors $\text{Prob}(n \text{ premier si } n \text{ impair}) \approx \frac{1}{115}$. Soit X le nombre d'essais nécessaires avant d'obtenir un nombre premier. Alors X est une variable aléatoire géométrique de paramètre $p = \frac{1}{115}$. Son espérance est donc $E(X) = \frac{1}{p} = 115$. □

Pour que la méthode fonctionne, il faut qu'il existe un moyen rapide de tester si un nombre entier n est premier, qui soit plus simple que de factoriser n. On va donc discuter de la « taille » d'un algorithme.

Taille d'un algorithme appliqué à un entier n de m chiffres. On a alors $n \approx 10^m$.

1. Les algorithmes pour factoriser n fonctionnent en *temps exponentiel* par rapport à la « taille » m de n. L'algorithme classique demande de tester si les nombres $2, 3, \ldots, d \leq \sqrt{n}$ sont des diviseurs de n. Le nombre de tests est donc de l'ordre de $10^{\frac{m}{2}}$. Il existe de bien meilleurs algorithmes mais pas au point de menacer le code RSA.
2. Pour être utilisable en pratique, un algorithme doit fonctionner en *temps polynomial* : Cm^r avec r entier.
3. Il existe depuis longtemps des *algorithmes probabilistes* pour décider en temps polynomial si un nombre est premier. Un tel algorithme ne peut affirmer avec certitude qu'un nombre est premier. Il permet d'affirmer qu'avec une très grande probabilité le nombre est premier.
4. Un tout nouvel algorithme déterministe (appelé algorithme AKS) pour décider en temps polynomial si un nombre est premier vient d'être annoncé en 2003 par Agrawal, Kayal et Saxena ([AKS] et [B]). Ce résultat a eu un grand retentissement, mais

cet algorithme est moins rapide que les algorithmes probabilistes. C'est donc une percée théorique, mais les informaticiens continuent de lui préférer les algorithmes probabilistes.

Un algorithme probabiliste pour tester si n est premier :
Le principe sous-jacent est que n laisse ses « empreintes » partout, si bien que, si n n'est pas premier, au moins la moitié des nombres de l'ensemble $E = \{1, \ldots, n\}$ savent que n n'est pas premier. Le test utilise le symbole de Jacobi. Le symbole de Jacobi est une fonction $J(a, b)$ définie sur les couples d'entiers (a, b) à valeurs dans $\{-1, 1\}$. Nous donnerons sa définition ci-dessous mais celle-ci est sans intérêt, si ce n'est qu'il est facile pour un ordinateur de calculer $J(a, b)$ même si a et b sont grands.

Théorème :

1. Si n est premier, alors

$$(*) \qquad (a, n) = 1 \quad \text{et} \qquad J(a, n) \equiv a^{\frac{n-1}{2}} (\mathrm{mod}\, n).$$

   Si a satisfait $(*)$, on dit que a « passe le test ».

2. Si n n'est pas premier, alors moins de la moitié des éléments $a \in E$ premiers avec n satisfont $(*)$. Si a relativement premier avec n ne satisfait pas $(*)$, on dit que a «échoue le test».

Preuve Nous ne ferons pas la preuve de 1. Nous ne dirons qu'un mot sur la preuve de 2 pour montrer l'élégance d'un argument algébrique. Soit $S = \{a \in E \mid (a, n) = 1\}$. Alors si on définit sur S la multiplication modulo n, S devient un groupe pour cette multiplication. Soit $F \subset S$ le sous-ensemble des éléments de S qui vérifient $(*)$. Alors F est un sous-groupe de S. Soit $|F|$ le nombre d'éléments de F et $|S|$ le nombre d'éléments de S. Le théorème de Lagrange en théorie des groupes affirme que $|F|$ divise $|S|$. Alors $|F| \leq \frac{|S|}{2} \leq \frac{|E|}{2}$ dès que S a un élément qui ne vérifie pas $(*)$. □

Algorithme :

1. On choisit a_1 au hasard et on vérifie si $(*)$ est satisfaite. Si oui, a_1 a passé le test. Sinon on sait que n n'est pas premier et on arrête. **Mais on n'a pas identifié de facteur de n !**

2. On choisit a_2 au hasard et on vérifie si $(*)$ est satisfaite. Si oui, a_2 a passé le test. Sinon on sait que n n'est pas premier et on arrête.
3. ...
4. Si $a_1, \ldots, a_k$ réussissent le test avec k assez grand, on sait que n a une très grande chance d'être premier. En effet, si n n'est pas premier, chacun des a_i a au moins une chance sur deux d'échouer le test.

Remarque : On voit donc que l'algorithme permet d'affirmer avec certitude que n n'est pas premier dès qu'un des a_k a échoué le test. Par contre il ne permet jamais d'affirmer que n est premier : on peut seulement conclure que n est presque sûrement premier. C'est le propre d'un algorithme probabiliste.

Que veut dire «k assez grand» ? C'est un exercice avec la formule de Bayes. Nous nous contenterons de donner le résultat sans faire les calculs. Soit p_k la probabilité que n soit premier sachant que $a_1, \ldots, a_k$ ont passé le test. Si n a 100 chiffres, i.e. est de l'ordre des 10^{100}, alors :

$$p_{15} \geq 0,996533$$

$$p_{35} \geq 0,9999999967.$$

Définition du symbole de Jacobi $J(a,n)$ pour n impair :

$$J(a,n) = \begin{cases} 1 & a = 1 \\ J\left(\frac{a}{2}, n\right)(-1)^{\frac{n^2-1}{8}} & a \text{ pair} \\ J(n \pmod a, a)(-1)^{\frac{(a-1)(n-1)}{4}} & a \text{ impair} \end{cases}$$

où $n \pmod a$ désigne le reste de la division de n par a.
On voit donc que le calcul de $J(a,n)$ se fait par itération successive. Son calcul est facile à programmer et le calcul se fait en temps polynomial.

Exemple : prenons $a = 130$ et $n = 207$. Alors

$$J(130,207) = J(65,207)(-1)^{\frac{42848}{8}} = J(65,207)(-1)^{5356} = J(65,207) = J(12,65)(-1)^{\frac{64\times 206}{4}}$$
$$= J(12,65) = J(6,65)(-1)^{\frac{4224}{8}} = J(6,65)(-1)^{528} = J(6,65) = J(3,65)(-1)^{528} = J(3,65)$$
$$= J(2,3)(-1)^{\frac{2\times 64}{4}} = J(2,3) = J(1,3)(-1)^{\frac{8}{8}} = -J(1,3) = -1$$

Ce calcul peut sembler long et fastidieux. Mais ce qui est important c'est que pour un ordinateur c'est un calcul simple.

Pour vérifier si a passe le test, on doit aussi calculer $a^{\frac{n-1}{2}} \pmod{n}$, soit $130^{103} \pmod{207}$. On doit être plus astucieux que de demander à l'ordinateur de calculer 130^{103} et de le réduire ensuite modulo 207. Pour cela on décompose 103 en puissances de 2 :

$$103 = 64 + 32 + 4 + 2 + 1 = 1 + 2^1 + 2^2 + 2^5 + 2^6.$$

On calcule :

$$\begin{aligned}
130^2 &= 16900 \equiv 133 \pmod{207} \\
130^4 &= (130^2)^2 \equiv 133^2 = 17689 \equiv 94 \pmod{207} \\
130^8 &= (130^4)^2 \equiv 94^2 = 8836 \equiv 142 \pmod{207} \\
130^{16} &= (130^8)^2 \equiv 142^2 = 20164 \equiv 85 \pmod{207} \\
130^{32} &= (130^{16})^2 \equiv 85^2 = 7225 \equiv 187 \pmod{207} \\
130^{64} &= (130^{32})^2 \equiv 187^2 = 34969 \equiv 193 \pmod{207}
\end{aligned}$$

Finalement

$$\begin{aligned}
130^{103} &= 130^{64} \times 130^{32} \times 130^4 \times 130^2 \times 130^1 \\
&\equiv 193 \times 187 \times 94 \times 133 \times 130 \equiv 67 \pmod{207}
\end{aligned}$$

On voit que $J(130, 207)$ n'est pas congru à $130^{\frac{207-1}{2}}$. On en conclut que 207 n'est pas premier. Ici c'était facile à voir : $207 = 3^2 \cdot 23$.

Discussion de la valeur du code RSA : Le code a été introduit en 1978. Il a stimulé les chercheurs à trouver de meilleurs algorithmes pour factoriser de grands nombres entiers mais sans succès : la méthode tient toujours si l'entier n est assez grand. En 1978, on évaluait à 74 ans le temps pour factoriser un nombre de 100 chiffres, à $3,8 \times 10^9$ années le temps pour factoriser un nombre de 200 chiffres et à $4,2 \times 10^{25}$ années le temps pour factoriser un nombre de 500 chiffres. Une clé de 100 chiffres est donc vulnérable maintenant. C'est d'ailleurs ce qui est arrivé avec la clé de 100 chiffres du système bancaire européen, laquelle a été factorisée par un particulier au début de l'an 2000. Mais, même en tenant compte de l'augmentation de la puissance des ordinateurs et du fait qu'on peut en mettre beaucoup

en parallèle, une clé de 200 chiffres tient encore le coup tant qu'on n'a pas de meilleur algorithme de factorisation.

Où en est-on par rapport aux évaluations de 1978 ? Les améliorations sont de deux ordres : la puissance des ordinateurs et de meilleurs algorithmes. La loi de Moore (du nom de Gordon Moore, cofondateur d'Intel) prédisait en 1965 que la densité des transistors doublerait tous les 18 mois à 2 ans, et elle s'est révélée étonnamment exacte. Quel est le lien avec la vitesse de calcul ? Les précisions suivantes viennent de Paul Rousseau travaillant chez TSMC : la vitesse des transistors augmente d'un facteur 1,4 tous les 2-3 ans. En fait les compagnies annoncent que la vitesse de l'horloge d'un circuit est multipliée par 2, mais le circuit fait moins de travail par cycle, donc ce facteur est artificiel. La vraie mesure est la capacité de faire du « vrai travail ». Pour un algorithme de factorisation où le travail peut être fait en parallèle, l'augmentation de la capacité de travail est de l'ordre de 2,8, soit 1,4 par transistor et un facteur 2 dû à l'augmentation du nombre de transistors. Il s'est écoulé 27 ans depuis 1978. Si l'on prend des générations ayant en moyenne 2,5 années, cela donne 10,8 générations, soit un facteur de 67500, inférieur à 10^5.

L'amélioration au niveau des algorithmes est non moins spectaculaire. Déjà Gauss au 19$^{\text{e}}$ siècle avait qualifié le problème pratique de la factorisation de grands nombres premiers de problème fondamental en théorie des nombres. Les algorithmes les plus importants sont :

- le crible quadratique de Pomerance ;
- la méthode des courbes elliptiques de Lenstra ;
- le crible des corps de nombres de Pollard, Adleman, Bulher, Lenstra et Pomerance.

En 1996, on factorisait des nombres de 130 chiffres. Un bon article sur le sujet est l'article de Carl Pomerance [P]. L'auteure n'exclut pas de faire un prochain article sur le sujet. Malgré toutes ces améliorations, le code RSA n'est pas encore menacé et une clé de 200 chiffres est encore suffisante.

La méthode de cryptage-décryptage pour le code RSA est longue et fastidieuse. La cryptographie à clés publiques n'est donc pas utilisée pour transmettre de longs textes et on lui préfère d'autres méthodes, surtout quand le texte transmis n'a pas besoin d'être tenu secret longtemps. Pour de plus longs textes, on va parfois préférer un algorithme à clé symétrique, par exemple le DES (Data Encryption System), i.e. l'expéditeur et le receveur ont la même clé. Ils peuvent utiliser le code RSA pour se transmettre la clé.

Application du code RSA :

- envoyer un numéro de carte de crédit sur la toile ;
- coder des NIP dans les systèmes bancaires.

Pour casser le code, on pense en général qu'il faut pouvoir factoriser la clé en temps polynomial. Rien ne prouve cependant qu'il ne puisse exister un autre algorithme fonctionnant en temps polynomial et permettant de retrouver le message m à partir de l'information publique (n, e, m^e) sans pour autant factoriser n. Un algorithme polynomial permettant de factoriser la clé existe, mais sur un ordinateur quantique. Nous allons discuter brièvement cet algorithme.

3 L'algorithme de Shor pour factoriser de grands nombres

Avant de discuter cet algorithme, on va commencer par se convaincre que les raffinements de l'algorithme classique de factorisation ne permettent pas de diminuer significativement le temps de factorisation. On considère un nombre n de 200 chiffres, i.e. un nombre de l'ordre de 10^{200}. L'algorithme classique consiste à chercher s'il existe un diviseur $d \leq \sqrt{n}$. On doit donc faire environ 10^{100} essais. Essayons quelques astuces :

- Si on se limite aux nombres d impairs, on a $m_1 = \frac{10^{100}}{2}$ tests à faire.
- Si on se limite à des grands diviseurs (des nombres de 100 chiffres), alors on a $m_2 = \frac{9}{10} m_1$ tests à faire (exercice).
- Si on met en parallèle un milliard d'ordinateurs, on a $m_3 = 10^{-9} m_2$ tests à faire sur chaque ordinateur.
- Si chacun des milliards d'ordinateurs est un super-ordinateur de 5000 processeurs pouvant faire 5000 opérations en parallèle (c'est la puissance maximum des super-ordinateurs en 2004), on limite le nombre d'opérations successives à faire sur chaque processeur à $m_4 = \frac{m_3}{5000}$.
- Avec ces tests, on aurait encore $m_5 \geq 10^{86}$ opérations sucessives à faire.
- Et supposons qu'on arrive tout juste à s'approcher avec d'autres astuces d'une factorisation de la clé, alors il suffirait d'allonger la clé de quelques dizaines de chiffres pour voir nos efforts anéantis.

On voit donc que pour factoriser des grands nombres il nous faut absolument un meilleur algorithme. L'algorithme de Shor a été introduit en 1997 et permet de factoriser des nombres entiers.

- Cet algorithme fonctionne en temps exponentiel sur un ordinateur classique.
- Il fonctionne en temps polynomial sur un ordinateur quantique.

C'est un algorithme probabiliste : si n n'est pas premier, l'algorithme a une très grande probabilité de trouver un diviseur d de n. L'algorithme ne permet donc pas de décider avec certitude si n est premier. Par contre, dès qu'on a trouvé un diviseur d de n, on sait que n n'est pas premier. Nous nous contenterons de donner les grandes lignes de cet algorithme, sans en montrer tous les détails.

Le principe de l'algorithme de Shor ([K & al] et [S]) :

Première étape : **On cherche un entier r tel que $n \mid r^2 - 1$, mais ni $r-1$ ni $r+1$ ne sont divisibles par n.**

En effet : $r^2 - 1 \equiv 0 \pmod{n}$, ce qui implique que $(r-1)(r+1) = mn$ pour un entier m. Alors si p est un facteur premier de n, nécessairement $p \mid r-1$ ou $p \mid r+1$. Si $p \mid r-1$, alors $(r-1, n) = d > 1$. Donc d est un diviseur de n. De même si $p \mid r+1$.

Exemple : Si $n = 65, r = 14$, alors $r^2 = 196 = 3 \times 65 + 1 \equiv 1 \pmod{65}$. $r - 1 = 13$ est un diviseur de 65. Par contre, si on prend $s = 64 \equiv -1 \pmod{65}$, alors $s^2 \equiv (-1)^2 = 1 \pmod{65}$. On voit que $s+1 = 65$ est divisible par 65. Donc s ne nous est d'aucun secours pour trouver un diviseur propre de 65.

Deuxième étape : **Comment trouver r ?**
On prend a au hasard dans $E = \{1, \ldots, n\}$ et on calcule les puissances de a : $a, a^2, a^3, \ldots$ que l'on réduit modulo n : $a^k \equiv a_k \pmod{n}$ avec $a_k \in E$.

- Si $(a, n) = d$, on a trouvé un diviseur de n.
- Si $(a, n) = 1$, comme E est fini, il existe k et l tels que $a_k = a_l$. On peut supposer $k > l$. Alors $a_{k-l} \equiv a^{k-l} \equiv 1 \pmod{n}$.
- Donc il existe s minimum tel que $a^s \equiv 1 \pmod{n}$. Ce nombre s est appelé l'*ordre* de a. On a $s \leq n$.

- Si s est pair : $s = 2m$, on prend $r \equiv a^m \pmod{n}$ avec $r \in E$. Alors $r^2 \equiv a^{2m} = a^s \equiv 1 \pmod{n}$.
- Si ni $r-1$, ni $r+1$ ne sont divisibles par n, on a terminé par la première étape.

Sinon on recommence avec un $a' \neq a$ choisi au hasard dans E.
On peut montrer qu'il y a beaucoup de $a \in E$ d'ordre impair qui font l'affaire, donc c'est un bon algorithme.

Rapidité de l'algorithme : la seule partie de l'algorithme qui ne s'effectue pas en temps polynomial est de calculer l'ordre de a. Un algorithme simpliste consiste à calculer tous les a_k jusqu'au premier qui est égal à 1. Le nombre d'opérations est de l'ordre de n, donc l'algorithme est exponentiel. C'est pour cette seule partie de l'algorithme qu'un ordinateur quantique prend la relève.

Calcul de l'ordre de a modulo n avec un ordinateur quantique (nous nous contenterons de donner quelques idées) : on écrit les nombres en base 2. Si n s'écrit avec m chiffres dans $\{0,1\}$, alors $n \leq 2^m$. On écrira les entiers k en base 2 : $k = [j_{m-1}j_{m-2}\dots j_0] = j_{m-1}2^{m-1} + j_{m-2}2^{m-2} + \dots + j_0 2^0$. Pour calculer l'ordre de a, on voudrait pouvoir calculer a^k pour tous les $k \in E$ simultanément, c'est-à-dire pour tous les $[j_{m-1}, \dots, j_0] \in \{0,1\}^m$. Se donner k revient donc à se donner m bits dans $\{0,1\}$. Essayer tous les $k \in E$ c'est essayer toutes les possibilités $j_i = 0$, et $j_i = 1$, pour $i = 0, \dots, m-1$, soit 2^m possibilités. C'est là que l'ordinateur quantique vient à la rescousse. On remplace les bits classiques $j_{m-1}, \dots, j_0$ par des bits quantiques.

Les bits quantiques Un bit quantique a la propriété de pouvoir se mettre dans un état superposé. Il est dans l'état $|0\rangle$ avec probabilité $|\alpha|^2$ et dans l'état $|1\rangle$ avec probabilité $|\beta|^2$ où $|\alpha|^2 + |\beta|^2 = 1$. ($\alpha$ est l'« amplitude » de $|0\rangle$ et β est l'« amplitude » de $|1\rangle$. Ce sont tous deux des nombres complexes.) En mécanique quantique, on dira que son état est $\alpha|0\rangle + \beta|1\rangle$. Pour se donner une analogie, pensons à un sou : il a probabilité 1/2 de tomber sur pile et 1/2 de tomber sur face. Avant le lancer, notre sou est donc dans un état superposé. Par contre, quand on le lance, on observe soit pile, soit face. C'est la même chose avec un bit quantique. Si on le mesure, on obtient 0 avec probabilité $|\alpha|^2$ et 1 avec probabilité $|\beta|^2$.

Le grand parallélisme d'un ordinateur quantique Si on met tous les bits $j_{m-1}, \ldots, j_0$ dans un état superposé en même temps, alors, en calculant $a^{|k\rangle} \pmod n$ où $|k\rangle$ est une superposition de tous les $k \in E$, on fait le calcul de a^k pour tous les $k \in E$ simultanément ! Comme le calcul quantique est linéaire et réversible, on peut voir $a^{|k\rangle} \pmod n$ comme un superposition de tous les $a_k \equiv a^k \pmod n$, chacun étant lié à la valeur de $k \in E$ associée. Toute l'information qui nous est nécessaire se trouve maintenant dans cet état, mais on ne peut y accéder sans le mesurer. En le mesurant, on ne mesure qu'une seule valeur $k \in E$, soit un $m-$tuplet $(j_{m-1}, \ldots, j_0)$. La mesure d'un état quantique est un processus aléatoire qui suit une distribution de probabilité dictée par les amplitudes de la superposition. Il faut donc user d'ingéniosité pour augmenter nos chances de lire un k qui nous intéresse. En particulier on utilise les probabilités conditionnelles et on mesure k sous la condition que $a^k \equiv 1 \pmod n$. On a un peu triché dans cette présentation et ce n'est pas aussi simple que cela, mais les grandes idées sont là.

Remarque : On a déjà montré dans la section précédente qu'il n'est pas difficile pour un ordinateur de calculer $a^k \pmod n$. En effet, si $k = j_{m-1}2^{m-1} + j_{m-2}2^{m-2} + \ldots + j_0 2^0$, alors $a^k = \prod_{j_i=1} a^{2^i}$. Il suffit donc de calculer les a^{2^i} modulo n, pour $i = 0, \ldots, m-1$. Ce calcul se fait de proche en proche :

- $a^2 \equiv a_1 \pmod n$ avec $a_1 \in E$;
- $a^4 \equiv (a_1)^2 \equiv a_2 \pmod n$ avec $a_2 \in E$;
- ...
- $a^{2^{m-1}} \equiv (a_{m-2})^2 \equiv a_{m-1} \pmod n$ avec $a_{m-1} \in E$.

Finalement $a^k \equiv \prod_{j_i=1} a_i \pmod n$.

Où en est-on avec l'ordinateur quantique ? Isaac Chuang et ses collègues du centre de recherche Almaden d'IBM ont pu construire un ordinateur quantique avec 7 bits quantiques simultanément dans un état superposé, lequel a permis de factoriser le nombre 15. Leur technique ne se généralise pas à un grand nombre de bits quantiques. Mais la recherche se poursuit sur d'autres techniques...

Remerciements Je tiens à remercier Isabelle Ascah-Coallier qui m'a intéressée à l'ordinateur quantique et Valérie Poulin qui m'a aidée à comprendre son fonctionnement.

Références

[AKS] M. Agrawal, N. Kayal and N. Saxena, *Primes is in P*, prépublication sur le site de Manindra Agrawal à www.cse.iitk.ac.in

[B] F. Bornemann, *Primes is in P*, Notices of the American Mathematical Society, (2003), **50** No 5, page 545-552.

[K & al] E. Knill, R. Laflamme, H. Barnum, D. Dalvit, J. Dziarmaga, J. Gubernatis, L. Gurvits, G. Ortiz, L. Viola and W. H. Zurek, *Introduction to quantum Information Processing*, 2002.

[RSA] R. L. Rivest, A. Shamir and L. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, Communications of the ACM, (1978), **21** No. 2, 120-126.

[P] C. Pomerance, *A tale of two sieves*, Notices of the American Mathematical Society, (1996), **43** No. 12, 1473-1485.

[S] P. W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM J. Computation, **26** (1997), 1484-1509.