Factorisation d'entiers

Mines de Nancy
Pépites algorithmiques
Séances des 19 et 26 mai 2009

Gaëtan BISSON

Adresser vos questions et remarques à : gaetan.bisson@loria.fr

0 Présentation

Définition. *Un entier $n > 1$ est dit premier si ses seuls diviseurs sont 1 et n.*

Théorème. *Tout entier peut s'écrire comme produit de facteurs premiers, de façon unique à l'ordre des termes près.*

Exemple. $91 = 7 \cdot 13$; le saviez-vous ?

L'objectif de ces deux séances est de mettre ce résultat bien connu sous forme effective, c'est-à-dire de concevoir un algorithme qui prend en entrée un entier et renvoie sa décomposition en produit de facteurs premiers.

Il existe de nombreuses méthodes pour ce faire, dont l'efficacité et la sophistication varient grandement ; c'est d'ailleurs un sujet qui suscite beaucoup de recherches. Les quelques-unes que nous présenterons ici ont été choisies pour leur élégance — ce qui rime avec simplicité.

Objectifs. Il n'est pas attendu que toutes les techniques présentées en cours soient implantées. Le but premier est d'avoir, à l'issue des deux séances, écrit un programme *fonctionnel* de factorisation, aussi simple soit-il. S'il reste du temps, on pourra ensuite améliorer les programmes en incorporant des techniques plus avancées.

Déroulement. La première séance motivera et présentera le problème de la factorisation ; elle décrira ensuite des tests de primalité ainsi que les techniques sous-jacentes d'arithmétique multiprécision. La seconde séance sera dédiée aux algorithmes de factorisation proprement dits.

1 Motivation

Le problème de la factorisation est récurrent en mathématiques algorithmiques, c'est-à-dire lorsque l'on calcule toute sorte d'objets mathématiques. Toutefois, son application la plus visible est très certainement liée au cryptosystème RSA.

Le cryptosystème RSA [6] spécifie une manière de chiffrer et de signer des messages électroniques, ce qui permet de communiquer de façon privée et authentifiée. Il est de nos jours largement utilisé pour *sécuriser* des communications, notamment dans le cadre du commerce électronique. Mentionnons les quelques applications « grand publique » que sont les cartes bancaires, les certificats Web et les communications militaires.

Description. Alice veut envoyer un message à Bob. Elle souhaite le *chiffrer* de sorte que seul Bob soit capable d'en lire le contenu, et pas des espions éventuels qui auraient intercepté ce message en chemin.

1. Bob choisit deux nombres premiers p et q ainsi qu'un entier e ; il calcule $n = pq$ et $d = e^{-1} \bmod (p-1)(q-1)$. Il envoie à Alice les entiers n et e.
2. Supposons que le message d'Alice soit un entier $m \in \{1, \dots, n\}$; elle calcule $m' = m^e \bmod n$ et envoie m' à Bob.
3. Bob évalue $m'^d \bmod n$; il retrouve ainsi le message m.

En pratique, Alice découpera son messages de façon à pouvoir représenter chaque morceau par un entier de $\{1, \dots, n\}$. Les entiers p, q et d constituent la *clef privée* que Bob garde pour lui ; les entiers n et e constituent la *clef publique* qui est connue de tous. Mentionnons aussi qu'à fin de simplification cette description omet quelques subtilités.

Proposition. *Un espion qui sait factoriser est capable de déchiffrer les messages.*

Démonstration. L'espion connaît n, e et m'. S'il parvient à factoriser n, c'est-à-dire à trouver les valeurs de p et q, il peut alors calculer d (comme Bob l'a fait à l'étape 1) et déchiffrer le message en évaluant $m'^d \bmod n$. □

Conjecture. *Déchiffrer les messages est aussi difficile que de factoriser n.*

En pratique, un ordinateur standard peut factoriser un entier de 200 bits en quelques secondes, un entier de 400 bits en quelques jours et un entier de 700 bits en quelques années. La majorité des applications utilise un n de 1024 bits, ce qui est donc relativement sûr, ne serait-ce que pour encore quelques décennies.

Exercice. Un message a été chiffré avec les paramètres $n = 51983$ et $e = 30109$, donnant $m' = 34869$. Sachant que $51983 = 227 \cdot 229$, retrouver ce message.

Solution. On rappelle : l'algorithme d'Euclide étendu qui permet de calculer d connaissant e et $N = (p-1)(q-1)$, ainsi que l'exponentiation rapide qui évalue m'^d en $\log d$ étapes.

Algorithme EUCLIDEÉTENDU(e, N) :
1. Poser $(a, a') \leftarrow (e, N)$ et $(x, x') \leftarrow (1, 0)$.
2. Tant que $a \neq 0$:
3. Poser $(a, a') \leftarrow (a' \bmod a, a)$;
   poser $(x, x') \leftarrow (x' - qx, x)$ où $q = \lfloor a'/a \rfloor$.
4. Vérifier que $a' = 1$ et renvoyer x'.

Algorithme EXPONENTIATIONRAPIDE(m', d) :
1. Calculer l'écriture binaire $d = \sum_{i=0}^{s} d_i 2^i$ et poser $r \leftarrow 1$.
2. Pour $i \leftarrow s$ jusqu'à 0 (décroissant) :
3. Poser $r \leftarrow r^2 \cdot m'^{d_i}$.
4. Renvoyer r.

Comme seul le résultat modulo n nous intéresse, on pensera à réduire modulo n à chaque étape, de sorte que la taille des résultats intermédiaires ne grossisse pas inutilement.

2 Primalité

Un cas particulier du problème de la factorisation est de savoir reconnaître les nombres premiers ; cela permet de décider, dans un algorithme de factorisation, si une décomposition est achevée ou s'il faut la poursuivre.

Pour s'assurer de la primalité d'un nombre n, la méthode naïve consiste à vérifier qu'aucun entier $d \leqslant \sqrt{n}$ autre que 1 ne divise n. Sa complexité est exponentielle en la taille de n (i.e. $\log n$), elle n'est donc efficace que pour de petites valeurs de n.

On connaît aussi des méthodes déterministes de complexité polynomiale, mais on leur préfère en pratique des algorithmes probabilistes plus rapides basés sur le fait suivant.

Proposition (petit théorème de Fermat)**.** *Soit p un nombre premier. Quelque soit l'entier x on a $x^p = x \bmod p$.*

Malheureusement la réciproque est fausse : il existe de (rares) entiers composés n (dits de Carmichael) tels que pour tout x on ait $x^n = x \bmod n$, par exemple $n = 561$. Cela dit, la modification à apporter pour corriger ce problème est minime ; elle aboutit à l'algorithme de Miller–Rabin [5, 7], décrit ci-dessous.

Algorithme MILLERRABIN(n,k) :

1. Écrire $n-1$ sous la forme $2^s t$ avec t impair.
2. Répéter k fois :
3. Choisir un entier $x \in \{2,\ldots,n-1\}$ et poser $x \leftarrow x^t \bmod n$.
4. Si $x \neq 1$:
5. Tant que $x \neq 1, n-1$ et au plus $s-1$ fois :
6. Poser $x \leftarrow x^2 \bmod n$.
7. Si $x \neq n-1$, renvoyer `composé`.
8. Renvoyer `premier`.

Proposition. *Si n est premier,* MILLERRABIN(n,k) *renvoie systématiquement* `premier`. *Si n est composé,* MILLERRABIN(n,k) *renvoie* `composé` *avec probabilité* $1-4^{-k}$.

On peut donc ajuster la valeur de k selon la certitude désirée.

Exercice. Se convaincre que 51991 est premier mais que 51997 ne l'est pas.

3 Arithmétique multiprécision

Les *entiers machine*, de type `int`, sont limités à 32 (voire 64) bits. Pour manipuler de *grands entiers* (n se compose typiquement de 1024 bits), on utilisera une bibliothèque dont la tache sera de représenter les grands entiers par des collections d'entiers machines.

Solution Java. La bibliothèque `java.math.BigInteger` est documentée en ligne :

`http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html`

On peut très bien utiliser une telle bibliothèque de façon transparente mais il n'en est pas moins intéressant d'en comprendre le fonctionnement. Si le temps le permet, on pourra même écrire soi-même une petite classe Java qui implantera une ou plusieurs des techniques exposées ci-dessous.

Représentation. Fixons un entier B, la *base*, e.g. $B = 2^{16}$. L'entier n dont la représentation en base B est $\sum_{i=0}^{s} n_i B^i$ sera alors représenté par un tableau de taille $s+1$ constitué des entiers machine n_i.

Addition. La façon évidente d'additionner deux entiers est optimale et s'implante très simplement. Il en va bien sûr de même pour la soustraction.

Par contre, presque toutes les autres opérations ne peuvent être implantées efficacement qu'en ayant recours à des techniques moins évidentes, à commencer par la multiplication.

Multiplication. Soient deux entiers a et b dont on supposera pour simplifier les tailles s égales, quitte à rajouter des « 0 » en tête du plus petit.

La méthode naïve consiste à écrire $ab = \sum_k 2^k \sum_{i+j=k} a_i b_j$, ce qui comporte $O(s^2)$ opérations élémentaires ; cela est, comme on s'en doute, optimal pour de petites valeurs de s. Asymptotiquement toutefois, la méthode de la transformée de Fourier [2] donne une complexité de $O(s \log s \log\log s)$; malheureusement, sa sophistication est telle qu'elle n'est intéressante en pratique que pour de très grands entiers.

Dans notre cas, i.e. $s \approx 64$, la méthode de Karatsuba [3] peut être envisagée. Elle exploite l'égalité

$$(aB^k + a')(bB^k + b') = abB^{2k} + ((a+a')(b+b') - ab - a'b')B^k + a'b'$$

qu'il s'agit d'appliquer récursivement pour $k = s/2$, utilisant ainsi 3 multiplications là où la méthode naïve en utiliserait 4.

Division. Pour calculer a mod b où b est un entier de taille s et a un entier de taille $s + s'$, la méthode naïve nécessite $O(ss')$ opérations.

Une façon de rendre cette opération aussi rapide que la multiplication est d'utiliser le principe « diviser pour régner ». Il s'agit de découper le diviseur et le dividende en deux, et d'appliquer récursivement l'algorithme pour calculer les divisons de ces deux morceaux. Formellement, cela donne :

Algorithme DIVISION(a, b) :
1. Si $s' < 2$, appliquer l'algorithme naïf.
2. Fixer $k \leftarrow s'/2$ et écrire $b_1 B^k + b_0 \leftarrow b$.
3. Calculer $q_1, r_1 \leftarrow$ DIVISION(a_1, b_1) avec $a_1 B^{2k} + a_0 \leftarrow a$.
4. Poser $a' \leftarrow r_1 B^{2k} + a_0 - q_1 b_0 B^k$ en ajustant q_1 pour que $a' > 0$.
5. Calculer $q_0, r_0 \leftarrow$ DIVISION(a'_1, b_1) avec $a'_1 B^k + a'_0 \leftarrow a'$.
6. Poser $a'' \leftarrow r_0 B^k + a'_0 - q_0 b_0$ en ajustant q_0 pour que $a' > 0$.
7. Renvoyer $(q_1 B^k + q_0, a'')$.

PGCD. L'algorithme EUCLIDEÉTENDU implanté directement a une complexité de $O(s^2)$ (où s est la taille des entrées). Les meilleurs algorithmes ont une complexité de $O(s \log^2 s \log\log s)$ mais nous n'en parlerons pas ici.

Fin de la première séance.

Le minimum est d'avoir implanté MILLERRABIN ; il est souhaité d'avoir codé RSA. Si le temps le permet, écrivez votre propre bibliothèque multiprécision.

*
* *

4 Méthodes exponentielles

Tout comme pour le problème de primalité, la méthode naïve pour factoriser un entier n consiste à énumérer les entiers $d \leqslant \sqrt{n}$ et à tester lesquels divisent n ; en suivant l'ordre croissant et en divisant n autant de fois que possible par les entiers trouvés, on est assuré de les avoir ainsi sous forme première. La complexité de ce procédé est de $O(\sqrt{n}) = O(\exp \frac{1}{2}s)$.

Pour faire tomber ce coût, on peut mettre en œuvre le très général principe des collisions.

Paradoxe des anniversaires. Contrairement à ce que l'intuition peut laisser croire, il y a 70% de chance que dans un groupe de 30 élèves deux aient leur anniversaire le même jour.

On dit que ces deux élèves sont *en collision*.

Proposition. *Le nombre de fonctions injectives de* $\{1,\dots,x\}$ *dans* $\{1,\dots,y\}$ *est* $\prod_{i=0}^{x-1}(y-i)$*, pour un total de* y^x *fonctions.*

Utilité. Posons $x = \sqrt{y}$; la formule de Stirling prouve que la proportion de fonctions injectives admet une limite finie, qui vaut $e^{-1/2} \approx 0{,}6$. En d'autres termes, si les années comportaient y jours, il faudrait considérer $\sqrt{y}$ élèves pour avoir 40% de chance que deux d'entre eux aient le même anniversaire.

Dans le contexte de la factorisation, pour trouver des facteurs non triviaux de n, on cherche des couples d'entiers $(\alpha,\beta) \in \{1,\dots,n\}^2$ pour lesquels $\operatorname{pgcd}(\alpha-\beta,n) \neq 1,n$. C'est-à-dire que les élèves sont des entiers modulo n et que leurs anniversaires sont leurs résidus modulo les facteurs de n.

Pour le plus petit facteur, on a $y = \sqrt{n}$ dans le pire des cas ; il faut donc considérer $x = \sqrt{y} = n^{1/4}$ entiers afin d'avoir 40% de chance de trouver un facteur de n.

En termes plus algorithmiques, cela donne la méthode de Shanks [1] :

Algorithme SHANKS(n) :
1. Stocker une liste L composée de $n^{1/4}$ entiers de $\{1,\dots,n\}$.
2. Tant que nécessaire :
3. Choisir un entier $\alpha \in \{1,\dots,n\}$.
4. Pour $\beta \in L$:
5. Poser $k \leftarrow \operatorname{pgcd}(\alpha-\beta,n)$; si $k \neq 1,n$, renvoyer $(k,n/k)$.

Bien entendu, cela ne permet que de « casser » l'entier n en deux bouts plus petits ; il faut ensuite itérer récursivement cette construction jusqu'à ne plus avoir que des bouts premiers.

L'inconvénient majeur de cet algorithme est son coût en mémoire très élevé.

Méthode ρ de Pollard. L'idée est de chercher les couples (α, β) parmi les valeurs d'une *fonction pseudo-aléatoire* $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$, en utilisant l'observation suivante :

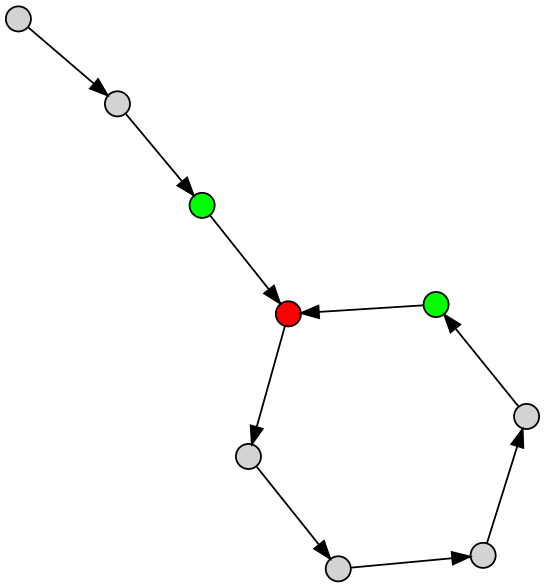
$$\text{Si } f^a(1) = f^b(1) \text{ et } a \geqslant 2b, \text{ alors } f^{2(a-b)}(1) = f^{(a-b)}(1).$$

On peut donc se contenter de tester les couples $(\alpha = f^k(1), \beta = f^{2k}(1))$, ce que fait efficacement l'algorithme ci-dessous [4].

Algorithme POLLARD(n) :
1. Poser $\alpha \leftarrow 1$, $\beta \leftarrow 1$ et $f : x \mapsto x^2 + 1 \bmod n$.
2. Tant que nécessaire :
3. Poser $\alpha \leftarrow f(f(\alpha))$ et $\beta \leftarrow f(\beta)$.
4. Si $\text{pgcd}(\alpha - \beta, n) \neq 1, n$, crier victoire.

Dans le cas où $\text{pgcd}(\alpha - \beta, n) = n$, il sera indiqué de relancer l'algorithme avec différentes valeurs initiales pour α et β, et éventuellement f.

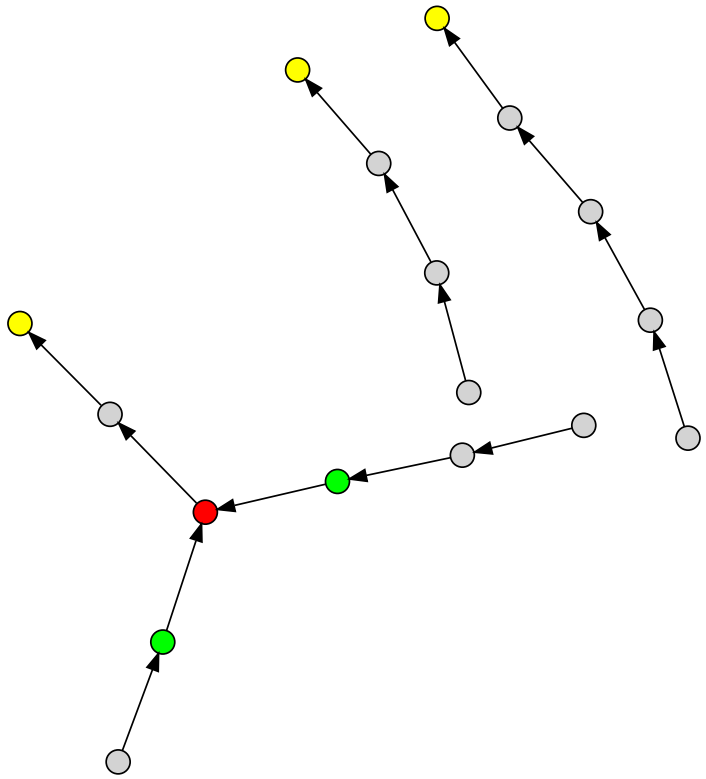
En vertu de l'observation, après $n^{1/4}$ itérations, on aura testé une bonne partie des collisions entre $n^{1/4}$ éléments. La complexité de cette méthode est donc comparable à celle de la précédente, mais elle a l'avantage de ne requérir que peu de mémoire.



MÉTHODE ρ DE POLLARD
(modulo le facteur trouvé)

Version parallèle. On peut distribuer de façon efficace le calcul sur plusieurs machines de la façon suivante. On fixe tout d'abord un sous-ensemble S d'éléments « distingués ». Alors, chaque machine choisit aléatoirement un entier x et itère l'opération $x \leftarrow f(x)$ jusqu'à ce que $x \in S$; elle stocke alors les valeurs initiale et finale de x dans une table partagée. Lorsque deux valeurs finales, α et β sont telles que $\text{pgcd}(\alpha - \beta, n) \neq 1$, on peut retrouver une collision.

Exercice. Discuter de la taille de S.



VERSION PARALLÈLE
(modulo le facteur trouvé)

Digression. Ces dernières méthodes sont extrêmement générales ; elles ont des applications bien au delà du problème de la factorisation. On peut citer par exemple le calcul de la structure de groupes abéliens ou encore la recherche de collisions de fonctions de hachage.

5 Méthode sous-exponentielle

Asymptotiquement, l'algorithme de factorisation le plus rapide connu est le crible des corps de nombres [9]. Il a comme complexité

$$L_{1/3}(n) \qquad \text{où } L_\alpha(x) = \exp\left(O(1)\cdot(\log x)^\alpha(\log\log x)^{1-\alpha}\right),$$

ce qui est infiniment mieux qu'une complexité exponentielle, sans pour autant être polynomial, ce que l'on qualifie de *sous-exponentiel*.

Ici, nous présenterons la méthode de Dixon [8] qui contient déjà quelques idées clefs tout en restant assez simple. Elle a pour complexité $L_{1/2}(n)$ et c'est historiquement le premier algorithme de factorisation à atteindre une complexité sous-exponentielle.

Il s'agit de « fabriquer » des relations de la forme $\alpha^2 = \beta^2 \bmod n$ (avec $\alpha \neq \pm\beta \bmod n$), de sorte qu'on ait une décomposition non triviale $n \mid (\alpha-\beta)(\alpha+\beta)$. L'idée pour ce faire est de combiner des relations plus faciles à trouver pour en éliminer les facteurs non carrés.

Comme il faut contrôler ces facteurs, on se restreint à ne considérer qu'une *base de facteurs* S de taille raisonnable, qui consiste en pratique en les nombres premiers plus petits que $L_{1/2}(n)$. On entreprend ensuite d'écrire autant de relations (modulo n) entre des carrés et les facteurs de cette base. L'étape finale consiste à éliminer les facteurs non carrés par des techniques d'algèbre linéaire.

Algorithme DIXON(n) :

1. Poser $N \leftarrow \lfloor \exp(\sqrt{2\log n \log\log n}) \rfloor$.
2. Former la liste S constituée des nombres premiers $p \leqslant N$.
3. Répéter N fois :
4. Choisir un entier z aléatoirement.
5. Si possible, écrire $z^2 \bmod n$ comme $\prod_p p^{e_{z,p}}$; sinon aller en 4.
6. Dans la matrice $e_{z,p}$, trouver une relation non triviale de la forme $\sum_z \lambda_z e_{z,p} = 0 \bmod 2$ pour certains $\lambda_z \in \{0,1\}$.
7. Poser $\alpha \leftarrow \prod_z z^{\lambda_z}$ et $\beta \leftarrow (\prod_p p^{\frac{1}{2}\sum_z \lambda_z e_{z,p}})$; on a ainsi $\alpha^2 = \beta^2 \bmod n$.
8. Si $\operatorname{pgcd}(\alpha-\beta, n) \neq 1, n$, crier victoire.

Application. On pourra essayer de factoriser des nombres de Fermat, $2^{2^n}+1$, et/ou des nombres de Mersenne, 2^n-1, pour n aussi grand que possible.

Références

[1] Daniel Shanks. Class number, a theory of factorization and genera. In *Proceedings of Symposia in Pure Mathematics*, volume 20, pages 415–440, 1970.

[2] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3–4) :281–292, 1971.

[3] Anatolii A. Karacuba. Berechnungen und die Kompliziertheit von Beziehungen. *Elektronische Information Verarbeitung und Kybernetic*, 11 :603–606, 1975.

[4] John M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3) :331–334, 1975.

[5] Gary L. Miller. Riemann's hypothesis and a test for primality. *Journal of Computing and Systems Science*, 13 :300–317, 1976.

[6] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[7] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12 :128–138, 1980.

[8] John D. Dixon. Asymptotically fast factorization of integers. *Mathematics of Computation*, 36 :255–260, 1981.

[9] Arjen K. Lenstra and Hendrik W. Lenstra, editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer, 1993.