Quand des mathématiques des XVIIe, XVIIIe, XIXe (et XXe) siècles m’aident à résoudre un problème de test

Remarque préliminaire: ce texte a été écrit par un humain, l'IA n'a fait que le relire et suggérer des améliorations/corrections.

> « Il y a cinq opérations élémentaires en arithmétique : l’addition, la soustraction, la multiplication, la division et l’exponentiation modulaire. »
> Auteur indéterminé

> « Le vériable artiste martial ne se bat pas pour les applaudissements, il se bat pour perfectionner sa technique. »
> (probablement) Kenshiro San, illustre disciple de l’école du Hokuto Shinken

Table des matières

1 Introduction

Après y avoir réfléchi suffisamment longtemps (comme Lone Watie et ses amis) j’ai décidé d’écrire un court article expliquant comment des techniques issues de l’arithmétique – la reine des mathématiques ou plutôt *die Königin der Mathemati*k devrais-je écrire – m’ont permis de résoudre un problème de test. Je devais tester une classe qui traite des fichiers arbitrairement grands, je ne parle pas de la photo de votre chat qui occupe (inutilement ?) 7Mo d’espace disque, je parle, par exemple, du volume VeraCrypt de 521Go dans lequel vous faites les sauvegardes de votre smartphone. C’est le genre de “grand” dont je

parle.
Pour tester cette classe j'ai donc besoin de simuler des fichiers potentiellement gigantesques, les options usuelles comme stocker les fichiers comme données de test, les générer sur le disque à la volée, etc. sont inenvisageables (as-tu déjà essayé de mettre un fichier binaire de 3Go dans ton dépôt Git ?). Soudain une idée me vint : pourquoi ne pas générer des données aléatoires mais avec un niveau d'aléa maîtrisé et surtout reproductible d'une exécution des tests à l'autre ? La solution est toute trouvée : l'exponentiation modulo un nombre premier et j'allais redécouvrir tout seul la notion de GCL… *Say what*? Pas de panique je vous explique tout cela, il s'agit d'arithmétique modulaire (si vous savez ce qu'est un nombre premier et une exponentiation modulaire vous pouvez passer à la section 2.3).

2 Vers l'exponentiation modulaire

2.1 Introduction à l'arithmétique modulaire

L'arithmétique modulaire c'est quoi ? L'idée de base est en fait assez simple, au lieu de faire les calculs comme vous les faites habituellement on ignore la partie des entiers qui est multiple d'un entier n donné, un exemple, si on travaille modulo 4, au lieu de 2 + 3 = 5 nous avons 2 + 3 = 1 (modulo 4) car 5 = 4 +1. Vous faites de l'arithmétique modulaire quasiment tous les jours (peut-être) sans le savoir : modulo 12 pour l'heure qu'il est (17h = 5h de l'après-midi), modulo 30 ou 31 quand on est le 29 du mois et qu'il faut calculer quel jour on sera dans 7 jours, ou, encore plus simple : si on est jeudi, quel jour sera-t-on dans 19 jours ? Comme 19 = 7*2+5 = 5 modulo 7, la réponse est jeudi + 5 = jeudi – 2 = mardi (et oui ! 5 = -2 modulo 7).
De même qu'il existe une arithmétique usuelle avec les quatre opérations +, -, x, /, il existe une arithmétique modulaire : on peut additionner, soustraire, etc. modulo un nombre entier donné. Qui dit multiplication dit élévation à une puissance : on peut multiplier un nombre par lui même n fois, c'est l'exponentiation : a^n = a * a *….* a (n fois a, soit n-1 multiplications), on parle alors d'exponentiation modulaire lorsque la multiplication est faite modulo un entier donné.

2.2 L'exponentiation modulaire

L'arithmétique modulaire a ses lois, par exemple nous avons :

$$(a \text{ modulo } b) * (c \text{ modulo } b) = (a * c) \text{ modulo } b$$
$$(a \text{ modulo } b) + (c \text{ modulo } b) = (a + c) \text{ modulo } b$$

Avec ces lois nous pouvons calculer, par exemple, 6^5 mod 17, plutôt que de calculer d'abord $6^5 = 7776$ et faire la division euclidienne par 17 (7776 = 457 * 17 + 7) pour en déduire 6^5 mod 17 = 7, nous réduisons à chaque étape :

$$6 * 6 = 36 = 17*2 + 2 = 2 \text{ mod } 17$$
$$6^3 = 2 * 6 = 12 \text{ mod } 17$$
$$6^4 = 12 * 6 = 4 * 17 + 4 = 4 \text{ mod } 17$$
$$6^5 = 4 * 6 = 24 = 7 \text{ mod } 17$$

On voit que si p est petit les valeurs intermédiaires restent petites et qu'en conséquence, par exemple, le calcul de 6^1279 mod 17 n'exige pas de calculer d'abord 6^1279 (un nombre à 996 chiffres) avant de faire une division euclidienne par 17. Bien, maintenant que vous savez ce qu'est une exponentiation modulaire voyons quel usage de celle-ci m'a permis de résoudre mon problème de test.

L'exponentiation modulaire est une opération très importante à la base de toute la cryptographie qui sécurise nos communications sur internet car elle possède certaines propriétés très utiles. Un exemple : si je vous donne un nombre n et que je vous demande quel nombre élevé à la puissance k donne n modulo un nombre p autrement dit si je vous demande de résoudre l'équation d'inconnue x suivante

$$x^k = n \text{ (modulo } p)$$

dans nombre de cas il vous sera extrêmement difficile de trouver la réponse, c'est le problème dit « du logarithme discret ». Mais ce qui nous intéresse ici c'est le caractère apparemment imprédictible des valeurs de a^n mod p quand on fait varier n, voici quelques exemples :

		n										
a	p	0	1	2	3	4	5	6	7	8	9	10
2	3	1	2	1	2	2	1	2	1	2	1	2
3	5	1	3	4	2	1	3	4	2	1	3	4
4	7	1	4	2	1	4	2	1	4	2	1	4
6	9	1	6	0	0	0	0	0	0	0	0	0

On voit que dans certains cas la suite des valeurs est périodique dans d'autre la suite devient nulle à partir d'une certaine valeur de n, c'est le cas de la dernière ligne car comme 6 = 2x3 et que 9 = 3x3 = 3^2, 6^n = 2^n*3^n est divisible par 9 dès que n>=2. On voit donc que les cas intéressants sont les cas où p est premier, dans ces cas là le petit théorème de Fermat (mathématicien français, notable toulousain du XVIIe siècle qu'on ne présente plus) nous dit que si a n'est pas divisible par p alors a^(p-1) = 1 mod p, donc la suite des valeurs de a^n mod p est périodique de période au plus p-1. Nous allons voir dans des exemples que la période est parfois inférieure à p-1, c'est déjà le cas dans le tableau ci-dessus avec a = 4 et p =7, la période est 3 et non 6 (remarquer que 3 est un diviseur de 6), nous donnons d'autres exemples :

		n												
a	p	0	1	2	3	4	5	6	7	8	9	10	11	12
5	11	1	5	3	4	9	1	5	3	4	9	1	5	3
6	11	1	6	3	7	9	10	5	8	4	2	1	6	3
2	13	1	2	4	8	3	6	12	11	9	5	10	7	1
3	13	1	3	9	1	3	9	1	3	9	1	3	9	1

Dans le cas p = 11, avec a = 5 nous avons une suite de période 5 < p – 1 = 10, 5 divisant 10, mais avec a = 6, la suite est de période maximale soit 10. Dans le cas p = 13, avec a = 2, nous avons une suite de période maximale soit p – 1 = 12, mais dans le cas a = 3, la période est de 3 (qui divise p – 1 = 12).
Ces exemples correspondent en fait à des résultats bien connus en théorie des nombres (la branche des mathématiques qui étudie ces sujets) : pour tout nombre premier p le groupe multiplicatif Z/pZ est cyclique d'ordre p-1 et on appelle racine primitive modulo p tout élément a du groupe tel que la suite a^k génère tous les éléments du groupe.

2.3 Racines primitive modulo p et GCL

Nous avons un exemple de racine primitive modulo un nombre premier dans le tableau précédent avec a=3 et p=13, on voit que 3 est une racine primitive modulo 13, un autre exemple plus simple s'obtient avec a=3 et p=7 : 3^0=1, 3^1=3, 3^2=2, 3^3=6, 3^4=4, 3^5=5, 3^6=1 (mod 7), on obtient tous les nombres de 1 à 6 donc 3 est une racine primitive modulo 7.
Il n'existe pas à ce jour de formule connue permettant de trouver les racines primitives modulo p, autrement dit, étant donné un nombre premier p pour trouver une racine primitive a de p, il n'y a d'autres choix que de calculer a^k mod p pour un certain nombre de valeurs de k pour vérifier que la période est maximale. En effet en vertu d'un résultat hyper-connu de théorie des groupes, le théorème de Sylow (Peter Ludwig de ses prénoms, mathématicien norvégien 1832-1918), l'ordre d'un élément dans un groupe divise l'ordre du groupe, en conséquence il nous suffit de vérifier que pour tout d diviseur de p-1 tel que d < p – 1, a^d est différent de 1 modulo p. En réalité nous n'avons pas besoin de tester tous les diviseurs car, d'après un théorème dû à Leonhard Euler (mathématicien suisse, bâlois même, du 18ième siècle, passé à l'Est en 1727), il suffit de tester seulement les diviseurs de la forme (p – 1)/q où q est un facteur premier de p – 1.
Au bout du compte nous voyons que nous avons mis la main sur un moyen de générer une suite de nombres semblant aléatoires et périodique d'une période aussi grande que l'on veut mais qui est en réalité parfaitement reproductible : il suffit de choisir un nombre premier p aussi grand que l'on veut et de trouver un entier a racine primitive modulo p, la suite à considérer est alors (a^n mod p)n.

Dans mes cas de test je peux générer une suite basée sur cette mécanique, et en jouant sur la valeur de a et de p je peux faire varier la période de ma suite, suite qui constituera en fait les octets de mon fichier simulé.

Pour ne pas refaire une recherche à chaque exécution des tests on peut calculer une bonne fois pour toutes un ensemble de couples d'entiers positifs (a, p) tels que p est premier et a est une racine primitive modulo p, ces couples seront nos données de test, plus besoin de stocker des fichiers. Le tableau page suivante donne des exemples de tels couples, en plus de ces couples il contient des couples (b, p) tels que b = a^k (mod p) pour un certain entier k > 0, ce qui permet d'obtenir des suites ayant des périodes plus petites.
On remarque rapidement que moins p – 1 a de facteurs premiers (et plus ceux-ci sont petits) plus la recherche d'une racine primitive modulo p est facilitée car il y a moins d'exponentiations modulaires à évaluer. Nous retrouvons la notion d'entier friable (ou lisse), un entier est dit B-friable (ou B-lisse) si et seulement si ses facteurs premiers sont inférieurs à une base donnée B. Ces entiers sont particulièrement important en cryptographie et, selon le contexte, on voudra des nombres premiers p tels que p-1 est friable ou, au contraire, p-1 n'est pas friable (on parle de nombres premiers sûrs).

En fait, nous venons de décrire ce que l'on appelle un générateur à congruence linéaire (GCL), introduit par Derrick Henry Lehmer (mathématicien américain né à Berkeley en 1905) en 1948 : on choisi un premier terme X_0, dit la graine ou germe (*seed* en anglais), on fixe a (le multiplicateur), c (l'incrément) et le module m, la suite des valeurs générées est alors définie par

$$X_{n+1} = (a*X_n + c) \bmod m$$

Dans les cas que j'ai décrit ci-dessus j'ai choisi m = p premier, c = 0, a une racine primitive modulo p.

a	p	période	commentaire
3	257	256	257 est un nombre de Fermat premier (nombres de la forme Fn = 2^(2^n)+1) donc pour savoir si a est une racine primitive modulo Fn, il n'y a qu'une quantité à évaluer: a^(2^n-1) mod Fn, ici 3^128 mod 257 = 256 différent de 1, donc 3 est bien une racine primitive modulo 257.
2	257	16	2 = 3^48 mod 257
136	257	32	136 = 3^8 mod 257
2	10501	10500	2 est bien une racine primitive modulo 10501: a^(10500/q) mod p = 10500, 4730, 9780, 4100 pour q = 2, 3, 5, 7 respectivement
3	65537	65536	65537 est un nombre de Fermat premier F4 = 2^(2^4)+1. 3^32768 mod 65537 = 65536 différent de 1 donc 3 est bien une racine primitive modulo 65537.
81	65537	16384	81 = 3^4, donc l'ordre est bien 2^16/2^2 = 2^14
6561	65537	8192	6561 = 3^8, donc l'ordre est bien 2^16/2^3 = 2^13 = 8192
5	19131877	19131876	On a p-1 = 4*3^14, comme 5^(2*3^14) mod p = 19131876 et 5^(4*3^13) mod p = 18954689, 5 est bien une racine primitive modulo p.
2	86093443	86093442	p = 2*3^16+1 2^((p-1)/2) mod p = 86093442 et 2^((p-1)/3) mod p = 29680860, donc 2 est bien une racine primitive modulo p.
5	258280327	258280326	p = 2*3^17+1 5^((p-1)/2) mod p = 258280326 5^((p-1)/3) mod p = 216758952

2	2441406251	2441406250	p = 2*5^13+1 2^((p-1)/2) mod p = 2441406250 2^((p-1)/5) mod p = 13991924
3	20100080249	20100080248	p = 2^3*2512510031+1
22	206158430209	206158430208	p = 3*2^36+1
2	15258789062501	15258789062500	p = 4*5^18+1
3	411782264189299	411782264189298	p = 2*3^30+1

3 Usage dans les tests

Voyons à présent comment ce qui a été exposé précédemment m'a facilité l'écriture des tests d'une classe traitant des fichiers arbitrairement gros. Comme un fichier F que l'on traite peut être de taille arbitrairement grande on le traite par morceaux : tant qu'on n'a pas atteint la fin du fichier F, on lit des blocs de F consécutifs de taille suffisamment petite pour ne pas épuiser la mémoire disponible. Pour tester cela il suffit de simuler un fichier en créant des blocs contenant les valeurs consécutive d'une suite obtenue via un GCL, cela permet de satisfaire aux conditions suivantes :
- les tests doivent être reproductibles (mêmes données simulées générées quelque soit la machine)
- les valeurs utilisées pour la simulation ne doivent pas être "trop simples" (i.e. suite de zéros ou suite de uns, etc.)
- il n'est pas nécessaire de stocker des fichiers volumineux à utiliser en entrée des tests
- le niveau d'aléa des données dans les fichiers en entrée des tests doit être configurable

Finissons par le code que j'utilise pour mon GCL :

```
// La table
const GCL_PARAMS : [(u64, u64) ; 11] = [
        (3, 257),
        (2, 10501),
        (3, 65537),
        (5, 19131877),
        (2, 86093443),
        (5, 258280327),
        (2, 2441406251),
        (3, 20100080249),
        (22, 206158430209),
        (2, 15258789062501),
        (3, 411782264189299)
] ;

// Le GCL
```

```
pub struct Gcl { a: u64, p: u64, xn: u128 }

impl Gcl {
   pub fn new(a: u64, p: u64) -> Self { Gcl{a: a, p: p, xn: 1u128} }
   pub fn next(&mut self) -> u64 {
      self.xn = (self.xn * self.a as u128) % self.p as u128;
      self.xn as u64
   }
   pub fn scrambled_next(&mut self) -> u64 {
         Self::scramble(self.next())
   }
   fn scramble(x: u64) -> u64 {
         // Effet boule de neige (ou avalanche) chaque bit influence tous les autres
         // Étape 1: ou exclusif avec le nombre d'or (partie entière de phi * 2^64)
         let mut y = x ^ 0x9E3779B97F4A7C15;
         y = y.wrapping_mul(0x9E3779B97F4A7C15); // Étape 2: multiplication par le nombre d'or
         y ^= y >> 32; // Etape 3: mélanger bits hauts/bas
         y
   }
}

// Utilisation dans un test :


// Production de données simulées avec un GCL
let my_gcl = Gcl::new(GCL_PARAMS[4].0, GCL_PARAMS[4].1) ;
let data = Vec::new();
for (i = 0 ; i < 4*1024*1024 ; ++i) {
 data.push(my_gcl.scrambled_next().to_le_bytes()) ;
}

// Traitement
let result = file_processor.process_data(data) ;

// Vérifications
…. snip ….
```

Et voilà, nous avons sous la main une approche qui permet de satisfaire les critères énoncés :

- déterminisme : même séquence à chaque exécution
- reproductibilité : séquence identique sur toutes les machines
- configurabilité : choix de période via (a, p)

- efficacité/utilisabilité : pas de stockage de fichiers volumineux
- contrôle de l'entropie : elle est ajustable via le scrambling

4 Conclusion, remarques finales

Nous avons vu dans cet article comment des concepts mathématiques développés il y a des siècles et qui peuvent sembler au premier abord abstraits et inutiles, car issus de la théorie des nombres qui est réputées être une des branches les plus abstraites des mathématiques, peuvent permettre de résoudre un problème bien concret du XXIe siècle. J'espère que cette lecture vous aura été utile, je termine par quelques remarques :

1. Dans cet article nous n'avons fait qu'effleurer l'écume qui est à la surface d'un domaine extraordinairement riche et varié : la théorie des nombres, la cryptographie et les interactions entre ces deux disciplines. J'ai réalisé un véritable tour de force car j'ai réussi à ne pas évoquer les choses suivantes : les corps finis, le théorème de Fermat-Wiles, les courbes elliptiques (sur un corps fini), les méthodes de décomposition en facteurs premiers d'entiers comme le crible de corps de nombres ou la méthode des courbes elliptiques, l'exponentiation modulaire par échelle binaire, la multiplication modulaire par la méthode de Barett, la loi de réciprocité quadratique, l'algorithme de Shor, la cryptographie post-quantique etc.
2. Il existe un outil très utile pour trouver les racines primitives modulo un nombre premier p, il s'agit de WolframAlpha, rendez-vous sur le site et taper une question du style "*Is 5 a primitive root modulo 257?*" (oui en anglais…) vous obtenez la réponse quasi-instantanément : "*True*" (vrai), "*5 is a primitive root of 257*" (5 est une racine primitive de 257). Si l'on veut travailler en local on pourra utiliser un outil comme PARI/GP.
3. Il n'y a rien de révolutionnaire dans ce qui a été présenté, bon nombre de développeurs ou testeurs auraient simplement utilisé un germe (*seed*) fixé mis en entrée d'un générateur de nombres pseudo-aléatoires (comme le *Mersenne twister*, *ChaCha20*, etc...) fourni par une bibliothèque de fonction disponible dans le langage dans lequel ils programment. Mais auraient-ils autant de contrôle sur les données générées qu'avec la méthode présentée ici ? Avec un GCL, on choisit précisément la période, la distribution, et le niveau d'entropie - un contrôle que les générateurs standard ne permettent pas.
4. Pour les aficionados de la performance pure dont il m'arrive de faire partie, oui la réduction modulo p c'est lent, oui il y a probablement des générateurs plus rapides, oui avec a = 2 c'est probablement plus rapide parce que tout le monde sait qu'une multiplication par une puissance de 2 c'est un décalage de bits et c'est donc beaucoup plus rapide qu'une multiplication (est-ce si sûr avec des processeurs modernes ?). Oui, certains choix de p permettent d'accélérer la réduction modulo p (nombre de Mersenne et nombre de Fermat), et on pourrait même aller jusqu'à utiliser la réduction de Barrett ou celle de Montgomery mais je pense que dans le cas présent tout cela serait exagéré et introduirait une grande complexité pour des gains minimes, il s'agit juste de générer des données de test, les performances sont amplement acceptables dans ce contexte.
5. Les valeurs générées par un GCL sont des entiers qui lorsqu'ils sont codés en 64-bit par exemple peuvent avoir beaucoup de zéros initiaux (e.g. l'entier 2 codé en 64-bit s'écrit 0000000000000000000000000000000000000000000000000000000000000010), on peut donc se retrouver avec un fichier simulé ayant beaucoup de zéros ce qui peut être indésirable selon le traitement que l'on teste, c'est pour cela qu'il faut parfois "mélanger les bits" de façon déterministe c'est l'objet de

la fonction scramble_next dans l’exemple de code donné, on multiplie par un nombre choisi de façon à avoir un mélange des bits qui excède un minimum requis, on augmente ainsi l’entropie des données générées.

Youcef Lemsafer, octobre 2025.