DM: Tests de primalité de Fermat et de Miller-Rabin

christina.boura@uvsq.fr

8 avril 2020

Les tests de primalité sont des algorithmes indispensables pour la cryptographie à clé publique. Parmi leurs utilisations les plus courantes sont la phase de génération de clés pour le cryptosystème RSA ou l'établissement d'un échange de clés basé sur le protocole Diffie-Hellman.

Les tests de primalité les plus efficaces sont les tests de nature probabiliste. Ces tests affirment qu'un nombre est composé avec une certitude absolue, mais ils peuvent se tromper en déclarant qu'un nombre est premier. Pour diminuer la probabilité d'erreur, le test est répété plusieurs fois (avec des paramètres à chaque fois différents). Si après plusieurs itérations, le nombre testé n'a pas été reconnu comme composé, on conclue alors qu'il est probablement premier.

Le but de ce DM est d'implanter en `C`, en utilisant la librairie `GMP`, deux tests de primalité très populaires : le test de Fermat et le test de Miller-Rabin.

1 Test de Fermat

Le test de Fermat est basé sur le Petit Théorème de Fermat, qui déclare que si p est un nombre premier, alors pour tout entier a tel que $\text{pgcd}(a, p) = 1$

$$a^{p-1} \equiv 1 \mod p. \tag{1}$$

Si pour un entier a,

$$a^{n-1} \not\equiv 1 \mod n,$$

alors n est sûrement composé. De l'autre côte, si l'équation (1) est vérifiée pour un entier a, on ne peut pas conclure avec certitude que n est premier puisque la réciproque du théorème de Fermat est fausse. En effet, il peut avoir des nombres composés vérifiant l'équation (1). Si pour un entier a, l'équation (1) est vérifiée et n est un nombre composé, alors n est appelé un nombre *pseudo-premier de base a*. Plus de détails sur les nombres pseudo-premiers peuvent être trouvés à la page Wikipedia `http://fr.wikipedia.org/wiki/Nombre_pseudopremier`.

Cependant les nombres pseudo-premiers sont relativement rares et on peut donc envisager d'adopter ce critère comme un test probabiliste de primalité, appelé test de Fermat. Ce test est décrit par l'algorithme 1.

Algorithme 1: Test de Fermat

Données : Un entier n et le nombre de répétitions souhaité k
pour $i = 1$ *jusqu'à* k **faire**
  Choisir aléatoirement a tel que $1 < a < n - 1$;
  si $a^{n-1} \not\equiv 1 \mod n$ **alors**
    renvoyer `composé` ;
renvoyer `premier` ;

L'entier k correspond au nombre de fois que ce test sera répété. À chaque itération, on effectue le test avec une base a différente. Plus le nombre de répétitions est grand, plus la probabilité que la réponse du test est correcte augmente.

2 Le test de Miller-Rabin

Le test de Miller-Rabin est lui aussi basé sur le Petit Théorème de Fermat, mais exploite quant à lui quelques propriétés supplémentaires.

Si n est un nombre impair dont on souhaite tester la primalité, on peut écrire $n-1 = 2^s t$, où t est un nombre impair. Dans le cas où n est un nombre premier nous avons par le Petit Théorème de Fermat

$$\begin{aligned} & 1 \equiv a^{n-1} \mod n \\ \Leftrightarrow \quad & 0 \equiv a^{2^s t} - 1 \mod n \end{aligned}$$

Puisque $s \geq 1$, le nombre $a^{2^s t} = (a^{2^{s-1} t})^2$ est un carré. Donc,

$$a^{2^s t} - 1 = (a^{2^{s-1} t} + 1)(a^{2^{s-1} t} - 1).$$

Si $s - 1 > 0$ alors le dernier terme est de nouveau une différence de carrés qui peut être donc factorisée. En continuant de la même manière, on obtient au final l'expression suivante :

$$0 \equiv (a^{2^{s-1} t} + 1)(a^{2^{s-2} t} + 1) \cdots (a^t + 1)(a^t - 1) \mod n. \tag{2}$$

On sait maintenant que pour un nombre premier p, si $ab \equiv 0 \mod p$ alors $a \equiv 0 \mod p$ ou $b \equiv 0 \mod p$. Par conséquent, si n est premier, alors l'équation (2) est vraie si et seulement si un des termes de la partie droite est $0 \mod n$. Autrement dit, si n est premier alors

$$\begin{aligned} a^{2^j t} \quad & \equiv \quad -1 \mod n, \text{ pour au moins un } j = 0, \ldots, s-1 \\ & \text{ou} \\ a^t \quad & \equiv \quad 1 \mod n. \end{aligned}$$

Si pour un nombre entier n, une des équations ci-dessus est vérifiée, alors l'algorithme conclue que n est probablement premier et termine. Si aucune de ces équations n'est vérifiée, alors l'algorithme renvoie que n est composé. Nous pouvons maintenant apporter à cette première approche l'**amélioration** suivante. Si on trouve que

$$a^{2^j t} \equiv 1 \mod n$$

pour un $j = 0, \ldots, s-1$, on peut directement conclure que n est composé et terminer l'exécution de l'algorithme. Ceci est basé sur le fait que si p est premier, les seuls éléments pour lesquels $x^2 \equiv 1 \mod p$ sont 1 et -1. Or, 1 et -1 sont les deux seuls racines carrés de l'unité.

La version du test de Miller-Rabin, prenant en compte cette dernière astuce est décrite par l'algorithme 2.

Algorithme 2: Test de Miller-Rabin

Données : Un entier n et le nombre de répétitions souhaité k

Décomposer $n - 1 = 2^s t$, t impair ;
pour $i = 1$ *jusqu'à* k **faire**
  Choisir aléatoirement a tel que $0 < a < n$;
  $y \leftarrow a^t \mod n$;
  si $y \not\equiv 1 \mod n$ *et* $y \not\equiv -1 \mod n$ **alors**
    pour $j = 1$ *jusqu'à* $s - 1$ **faire**
      $y \leftarrow y^2 \mod n$;
      si $y \equiv 1 \mod n$ **alors**
        renvoyer `composé` ;
      si $y \equiv -1 \mod n$ **alors**
        arrêter la boucle de j. Continuer avec le i suivant ;
    renvoyer `composé` ;
renvoyer `premier` ;

Si le test de Miller-Rabin renvoie `composé`, alors le nombre est effectivement composé. Il peut être démontré que si le test de Miller-Rabin dit que n est premier, le résultat est faux avec une probabilité inférieure à 1/4. En effet, il existe des valeurs de a qui produiront de manière répétée des menteurs, qui indiqueront donc que n est premier alors qu'il est composé. On appelle un *témoin fort* pour n un entier a pour lequel

$$a^t \not\equiv 1 \mod n \text{ et } a^{2^s j} \not\equiv -1 \mod n \text{ pour tous les } 0 \leq j \leq s-1.$$

Il peut être montré qu'il existe toujours un témoin fort pour n'importe quel composé impair n, et qu'au moins 3/4 de ces valeurs pour a sont des témoins forts pour la composition de n. Si on répète ce test k fois, la probabilité que le résultat est toujours faux décroît très rapidement. La probabilité que le test renvoie `premier` à tort après k itérations est $\frac{1}{4^k}$.

3 L'algorithme `square and multiply`

L'opération cruciale dans les deux tests précédents est l'exponentiation modulaire. Cette opération très coûteuse, si implantée de manière naïve, peut exploser le temps de calcul. Par exemple, si on souhaite calculer $x^{2^{1024}}$ et on procède en calculant d'abord $x^2 = x \cdot x$, ensuite $x^3 = x \cdot x^2$, jusqu'à $x^{2^{1024}} = x \cdot x^{2^{1024}-1}$, on va devoir effectuer $(2^{1024} - 1)$ multiplications !

Un algorithme qui permet de calculer l'exponentiation modulaire efficacement est la méthode `square and multiply`, qui utilise la décomposition binaire de l'exposant. Cette méthode, pour le cas de l'exponentiation modulaire est décrite par l'algorithme suivant.

Algorithme 3: `square and multiply`

Données : Un entier a, le module n et un exposant $H = \sum_{i=0}^{t} h_i 2^i$, avec $h_i \in \{0, 1\}$ et $h_t = 1$
Résultat : $a^H \mod n$
$r = a$;
pour $i = t-1, \ldots, 0$ **faire**
  $r = r^2 \mod n$;
  si $h_i = 1$ **alors**
    $r = r \cdot a \mod n$;
Sorties : r

Pour calculer par exemple $x^{26} = x^{11010_2}$ (on s'intéresse pas ici aux opérations modulaires), les étapes à effectuer sont les suivantes :

`SQ`	$x \cdot x = x^2$
`MUL`	$x \cdot x^2 = x^3$
`SQ`	$x^3 \cdot x^3 = x^6$
`SQ`	$x^6 \cdot x^6 = x^{12}$
`MUL`	$x \cdot x^{12} = x^{13}$
`SQ`	$x^{13} \cdot x^{13} = x^{26}$

4 La bibliothèque `GNU MP`

Le langage `C` possède plusieurs types pour représenter des nombres entiers. Cependant, tous ces types ont une précision fixe et ne peuvent pas dépasser un certain nombre d'octets. Le type le plus grand est `long long int` qui peut contenir des entiers d'une taille maximale de 64 bits. Or, tous ces types sont beaucoup trop courts pour les applications cryptographiques qui nécessitent la manipulation des données d'au moins 512 bits.

`GNU MP`, pour `GNU Multi Precision`, souvent appelée `GMP` est une bibliothèque C/C++, de calcul multiprécision sur des nombres entiers, rationnels et à virgule flottante qui permet en particulier de manipuler des très grands nombres.

Vous pouvez télécharger l'archive la plus récente contenant cette bibliothèque `gmp-6.2.0.tar.lz` à l'adresse

`https://gmplib.org/#DOWNLOAD`

et le manuel d'utilisation à l'adresse

`https://gmplib.org/gmp-man-6.2.0.pdf`

La page 3 de ce guide vous explique comment installer cette bibliothèque sous `Linux`. La bibliothèque a été principalement conçue pour des systèmes `Unix`, mais il est possible de l'installer et la configurer pour `Windows` et `Mac`. `Google` pourrait toujours vous aider si nécessaire.

Vos programmes doivent inclure le fichier '`gmp.h`' et la compilation doit se faire avec le `-lgmp`, par exemple

```
gcc monprogramme.c -lgmp
```

4.1 Manipulation de nombres entiers

Le type d'un entier avec `GMP` est `mpz_t`. Pour déclarer un entier `n` vous écrivez

```
mpz_t n;
```

Avant d'affecter un nombre entier à la variable `n`, cette variable doit être initialisée, en utilisant une des fonctions d'initialisation, par exemple :

```
mpz_init(n);
```

Si vous n'avez plus besoin de cette variable, vous devez alors la libérer :

```
mpz_clear(n);
```

La bibliothèque contient un très grand nombre de fonctions, (`mpz_add(), mpz_sub(), mpz_div()`,...) qui permettent d'effectuer des opérations classiques et moins classiques sur les nombres entiers. Le manuel de `GMP` contient la description et le mode d'utilisation de toutes les fonctions dont vous aurez besoin. Vous pouvez visiter la page de `Wikipedia`,

`http://en.wikipedia.org/wiki/GNU_Multiple_Precision_Arithmetic_Library`

pour voir un exemple d'un programme simple.

5 À vous maintenant

Vous devez implanter les algorithmes de Fermat et de Miller-Rabin décrits dans les sections précédentes en utilisant la librairie `GMP`. L'utilisateur doit fournir au programme l'entier n à tester ainsi qu'un entier k, indiquant le nombre d'itérations. L'algorithme `square and multiply` doit être implanté pour l'exponentiation modulaire.

Pour que la demarche ne soit pas trop simple, vous n'êtes pas autorisés à utiliser les fonctions de la libairie `GMP` suivantes : `mpz_probab_prime_p`, `mpz_nextprime`, `mpz_gcd`, `mpz_gcd_ui`, `mpz_gcdext`, `mpz_powm`, `mpz_powm_ui`, `mpz_powm_sec`, `mpz_pow_ui` et `mpz_ui_pow_ui`.

Pour tester vos programmes voici une page qui contient un grand nombre de premiers

`http://en.wikipedia.org/wiki/List_of_prime_numbers`

Testez également vos programmes avec certains des nombres pseudo-premiers fournis ici

`http://fr.wikipedia.org/wiki/Nombre_pseudo-premier`

Vous devez m'envoyer vos programmes par mail. Pour faciliter la démarche, j'attends de vous une archive compressée, nommée de vos prénoms et noms, contenant le code source. Votre code doit être bien commenté et facile à utiliser. Vous pouvez include un fichier README.txt expliquant comment tester votre code. Vous pouvez travailler **individuellement** ou **par équipe de trois personnes au plus**. Bien évidement, chaque équipe doit travailler seule.

La date limite pour m'envoyer vos programmes est le **mercredi 29 avril à 20h**. Un point de pénalité sera attribué à chaque heure de retard.