Tests de primalité

Mikaël Benhaiem
François Parodi

14 mai 2015

Table des matières

Introduction

Les nombres premiers sont les entiers naturels qui admettent exactement deux diviseurs distincts entiers et positifs, qui sont 1 et lui-même. Ils jouent un rôle fondamental en cryptographie. Il est donc intéressant de pouvoir tester la primalité d'un nombre donné.

Un test de primalité naïf qui testerait la primalité d'un nombre en essayant de le diviser par des entiers successifs entre 2 et le nombre en question n'est pas envisageable pour des très grands nombres, car beaucoup trop longs (Malgré des améliorations, comme le fait de tester jusqu'à $\sqrt{nombre}$, et de ne tester que les nombres impairs une fois que la division par 2 a été échouée). Il est ce qu'on appelle pseudo-polynomial (c'est à dire que sa complexité en temps est un polynôme en la valeur numérique de l'entrée), tout en étant en temps exponentiel. Des tests plus efficaces existent et peuvent être classé en deux différentes catégories : probabiliste, c'est-à-dire qu'un nombre qui passe le test a de très fortes chances d'être premier (test de Miller-Rabin, test de primalité de Fermat, test de Solovay-Strassen...), et déterministe, c'est-à-dire qui, à l'inverse d'un test probabiliste, donne la preuve de la primalité ou non d'un nombre (test AKS, certificat de Pratt...).

Le but de ce projet est d'étudier quatre différents tests de primalité déterministes : le test de Lucas-Lehmer pour les nombres de Mersennes, le test Lucas-Lehmer-Riesel, les certificats de Pratt, et le test de Pocklington.

Ce projet est réalisé en C et utilise la bibliothèque GMP pour la manipulation de très grands nombres.

1 Les deux tests de Lucas-Lehmer (Nombres de Mersenne et Riesel)

1.1 Théorème de Lucas-Lehmer sur les nombres de Mersenne et explications

Le test de Lucas-Lehmer est un test de primalité qui fonctionne très rapidement pour les nombres de Mersenne. Les nombres de Mersenne, non nécessairement premiers, constituent la suite de nombre $M_p = 2^p - 1$ avec p premier (Cette condition est nécessaire pour que M_p soit premier). Ces nombres sont candidats à être premiers. Ce test s'effectue en temps polynomial, en fonction de la variable p. Le théorème correspondant au test est le suivant :

Théorème 1.1 *Soient p un nombre premier, et $M_p = 2^p - 1$ le nombre de Mersenne à tester. On défini la suite $\{s_i\}$ pour tous les $i \geq 0$ par*

$$s_i = \begin{cases} 4, & si\ i = 0; \\ s_{i-1}^2 - 1 & sinon. \end{cases}$$

Alors M_p est premier si et seulement si :

$$s_{p-2} \equiv 0\ (M_p).$$

Dans le cas contraire, M_p est un nombre composé. Le nombre s_{p-2} mod M_p est appelé le "résidu de Lucas-Lehmer" de p. A noter que si p n'est pas premier, cela implique que M_p n'est pas premier. En effet, soit a divise p. On a alors les suites $M_a = 2^a - 1$ et $M_p = 2^p - 1$. Il existe un q tel que $p = aq$. On a alors

$$M_p = 2^{aq} - 1 = (2^a - 1)(1 + 2^a + 2^{2a} + 2^{3a} + ... + 2^{(q-1)a}).$$

D'où, M_a divise M_p, ce qui implique que M_p n'est pas premier.

1.2 Théorème de Lucas-Lehmer-Riesel et explications

Le test de Lucas-Lehmer-Riesel est un test de primalité pour les nombres de la forme $N = k \cdot 2^n - 1$, avec $2n > k$. Tout comme celui sur les nombres de Mersenne, il est rapide et s'effectue en temps polynomial par rapport au n. Il repose sur le théorème suivant :

Théorème 1.2 *Soit une suite $\{u_i\}$ pour tout $i > 0$ définie par $u_i = u_{i-1}^2 - 2$. Alors N est premier si et seulement si N divise u_{n-2}.*

On va choisir la valeur de départ u_0 en fonction de N :
- Si $k = 1$ et n est impair, on prend $u_0 = 4$. Si $n \equiv 3 \bmod 4$, on peut prendre $u_0 = 3$.
- Si $k = 3$ et $n \equiv 0$ ou $3 \bmod 4$, alors on prend $u_0 = 5778$.
- Si $k \equiv 1$ ou $5 \bmod 6$, et que 3 ne divise pas N, alors on prend $u_0 = (2 + \sqrt{3})^k + (2 - \sqrt{3})^k$
- Sinon, k est un multiple de 3 et il est difficile de sélectionner une bonne valeur pour u_0.

Le test fonctionne de la façon suivante : on démontre qu'un nombre est premier en montrant qu'un groupe a l'ordre qu'il aurait si ce nombre était premier, et on le montre en prenant un élément de ce groupe avec cet ordre-là.

1.3 Implémentation de Lucas-Lehmer-Mersenne

Algorithme de la fonction :

```
int lucasLehmerMersenne(int p)
{
        m = 2^p - 1;
        s = 4;
        i = 0;

        if(p <= 1)
                return 0;
        while(i < p - 2)
        {
                s = s*s - 2 mod m
                i = i + 1;
        }
        if(s == 0 ||p == 2)
                return 1;
        else
                return 0;
}
```

La fonction `int lucasLehmerMersenne(int p)` prend en entrée un entier $p \geq 2$. Le but de cette fonction est de renvoyer 1 si le nombre de Mersenne correspondant au p est premier, et 0 sinon. Si $p \leq 1$, un message d'erreur est affiché, et la fonction renvoie 0. Sinon, elle calcule une variable $m = 2^p - 1$, qui est le nombre de Mersenne correspondant au p. Ensuite, on effectue $p - 2$ fois l'opération $s = s^2 - 2$, en initialisant $s = 4$. Du coup, dans le cas de $p = 2$, le nombre de Mersenne $M_2 = 2^2 - 1 = 3$ est premier, mais on ne rentre pas dans la boucle. A la place, dans le test final qui est si $s = 0$ on teste également si $p = 2$. Si une de ces deux conditions est vérifiée, alors on renvoie 1, ce qui signifie que le nombre de Mersenne M_p est premier. Sinon, on renvoie 0.

On peut noter que dans notre implémentation, on peut donner en entrée n'importe quel nombre $p \leq 2$, c'est à dire que le p donné en entrée n'est pas forcément premier. En effet, p premier étant une condition nécessaire pour que M_p soit premier, si le nombre en entrée n'est pas premier, alors la fonction renverra tout simplement 0, ce qui indique que le nombre $M_p = 2^p - 1$ correspondant n'est pas premier.

Après des tests effectués, on peut affirmer que le programme est capable de prouver que le nombre $M_{86243} = 2^{86243} - 1$ correspondant à $M28$ et composé de 25962 chiffres est premier en environ une minute. Ce n'est pas la limite du programme, mais cela donne une approximation de la puissance de ce test.

1.4 Implémentation de Lucas-Lehmer-Riesel

La fonction `int lucasLehmerRiesel(int k, int n)` renvoie vrai si le nombre $N = k \cdot 2n - 1$ est premier, faux sinon. Si k est un multiple de 3, la fonction ne fait pas de calculs et déclare le résultat indéterminé. Si le k en entrée est pair,

la fonction procède à un ajustement de k et n tel qu'on ait bien $N = k \cdot 2n - 1$ avec k impair. Voyons l'algorithme plus en détail :

```
int lucasLehmerRiesel(int k, int n)
{
        //Ajustement de k et n si k est pair
        while (k % 2 == 0)
        {
                k = k / 2;
                n = n + 1;
        }

        //Il faut que 2^n > k
        if (2^n <= k)
        {
                Erreur: 2^n <= k
                return 0;
        }
        N = k * 2^n - 1
        u0 = -1;
        if (k == 1)
        {
                if (n % 2 == 1)
                {
                        u0 = 4;
                }
                else if (n % 4 == 3)
                {
                        u0 = 3;
                }
        }
        if (k == 3)
        {
                n_ = n % 4;
                if ((n_ == 0) || (n_ == 3))
                {
                        u0 = 5778;
                }
        }
        else
        {
                k_ = k % 6;
                if ((k_ == 1) || (k_ == 5))
                {
                        r3 = N/3;
                        if (r3 != 0)
                        {
                                u0 = (2+sqrt(3))^k +
                                    (2-sqrt(3))^k
                        }
```

```
                }
                else
                {
                        Erreur: k multiple de 3, u0
                           indeterminé
                        return 0;
                }
        }
        if (u0 == -1)
        {
                Erreur: u0 indeterminé
                return 0;
        }

        //Calcul de u indice n-2
        int i = 0;
        u = u0
        while (i < (n - 2))
        {
                u = u*u - 2;
                i = i + 1;
        }

        //N est premier s'il divise u
        if (N | u)
        {
                //N est premier
                return 1;
        }
        else
        {
                //N n'est pas premier
                return 0;
        }
}
```

La première boucle consiste à ajuter le k par rapport au n. Puis, plusieurs tests vont déterminer la valeur du $u0$, qui est le premier terme de la suite u_n. Ensuite, le test est assez similaire à celui de Lucas-Lehmer pour les nombres de Mersenne. En effet, on effectue $n-2$ fois l'opération $u = u^2 - 2$, et ensuite on teste simplement si $N = k \times 2^n - 1$ divise u. Si oui, N est premier, sinon N n'est pas premier.

2 Certificats de Pratt

2.1 Théorème du Certificat de Pratt et explications

Historiquement, le concept des certificats de primalité a été introduit par le Certificat de Pratt. Son but est de prouver qu'un nombre donné en entrée est premier en un temps polynomial. Ce problème est dans NP car il doit utiliser un algorithme de factorisation, qui lui même est dans NP. Il est basé sur le test de primalité de Lucas-Lehmer, qui lui même utilise le test de Miller-Rabin, qui est essentiellement la réciproque du Petit théorème de Fermat. Pour le rendre vrai, on rajoute une condition. Ainsi on a le théorème suivant (pour tester si un nombre n donné en entrée est premier ou non) :

Théorème 2.1 *On considère un entier a tel que :*

- *a et n sont premiers entre eux ;*
- *$a^{n-1} \equiv 1\ (n)$;*
- *Pour chaque facteur premier q de $n-1$, on a : $a^{\frac{n-1}{q}} \not\equiv 1\ (n)$.*

Alors, n est premier.

Le choix du a est généralement fait aléatoirement en respectant la condition $2 \leq a \leq n-1$. Si les deux premières conditions sont respectées, il y a de très fortes chances que la troisième conditions soit également respectée. En effet les deux premières conditions sont nécessaires pour qu'un nombre soit premier, mais pas suffisante. La troisième rend l'ensemble de ces tests suffisants pour prouver qu'un nombre est premier. Si on tombe dans les très rares cas où elle n'est respectée, en général on relance le certificat avec un a différent. Algorithmiquement, il a une assez grande complexité, car pour chaque facteur premier, on veut être sur que ce nombre est bien premier : Il faut ainsi que chaque facteur vérifie ce théorème.

Le test

$$pgcd(a, n) = 1$$

prouve que a et n sont premiers entre eux. Le deuxième test

$$a^{n-1} \equiv 1\ mod(n)$$

correspond au test de Miller-Rabin (Donc en arrivant ici, le test probabiliste de Miller Rabin affirme qu'il y a une très forte chance que le nombre soit premier), qui lui même utilise le petit théorème de Fermat. Ce théorème est le suivant :

Théorème 2.2 *Si p est un nombre premier et si a est un entier non divisible par p, alors*

$$a^{p-1} - 1 \equiv 0\ mod(p).$$

Enfin, le dernier test

$$a^{\frac{n-1}{q}} \not\equiv 1$$

pour chaque facteur premier q, correspond à la condition du test de Lucas-Lehmer qui le rend déterministe. Cette condition se démontre grâce à la théorie des ensembles. L'idée de cette démonstration est que si l'ordre de a dans l'ensemble (Z/nZ) est égal à $n-1$, cela implique que l'ordre de ce groupe est $n-1$,

ce qui implique que n est premier. Réciproquement, si n est premier, il existe une racine primitive modulo n, et n'importe quelle racine primitive (un entier g tel que, modulo n, chaque autre entier de l'ensemble soit simplement une puissance de g) de ce genre passera les deux étapes de l'algorithme avec succès.

Ce certificat est rapide pour les nombres premiers de Fermat (nombres de la forme $F_n = 2^{2^n} + 1$), mais bien plus lent que d'autres tests de primalités pour d'autres nombres.

2.2 Implémentation des fonctions du certificat et de factorisation

La fonction `int certificatPratt(mpz_t p, gmp_randstate_t state)` renvoie 1 si le nombre p est premier, 0 sinon (Où si l'on a pas pu prouver qu'il était premier), et -1 si une erreur a été rencontrée. Cette fois le type en entrée est un mpz_t tout simplement car on veut gérer de très grands nombres. l'argument state sert uniquement à déterminer la graine qui permet de choisir l'entier a aléatoire. Dans notre code, l'élément qui permet d'arrêter de certificat, est lorsque $n = 2$. Pour bien illustrer le fonctionnement du programme, voici un exemple tiré du site Wikipédia pour le nombre 229 (Les a sont choisis aléatoirement) :

$n = 229$ ($a = 6$, $n - 1 = 228 = 2^2 \times 3 \times 19$)

1. $n = 2$: n est premier.
2. $n = 3$: ($a = 2$, $n - 1 = 2$)
   (a) $n = 2$: n est premier.
3. $n = 19$ ($a = 2$, $n - 1 = 18 = 2 \times 3^2$)
   (a) $n = 2$: n est premier.
   (b) $n = 3$ ($a = 2$, $n - 1 = 2$)
      i. $n = 2$: n est premier.

Dans les fichiers `certificatPratt.h` et `certificatPratt.c`, on a 3 fonctions supplémentaires ainsi qu'une nouvelle structure de données. Cette structure est défini de la manière suivante :

```
struct facteursPremiers
{
        mpz\_t \*facteurs;
        int longueur;
} typedef facteursPremiers;
```

Cette structure va être utilisée pour stocker un tableau de mpz_t ainsi qu'un entier qui détermine la longueur du tableau. La fonction `facteursPremiers factorisation(mpz_t f)` est la fonction qui va justement préparer une variable du type facteursPremiers, en décomposants en produits de facteurs premier la variable f donnée en entrée (un algorithme naif de factorisation est utilisé ici). Voici le fonctionnement du coeur de la fonction :

```
facteursPremiers factorisation(mpz_t f)
{
```

```
        i = 0;
        marqueur = 0;
        fact.longueur = 0;
        while(i <= sqrt(f))
        {
                if(f % i == 0)
                {
                        while(f % i == 0)
                        {
                                f = f/i;
                        }
                        fact.longueur ++;
                        fact.facteurs[marqueur] = i;
                        marqueur++;
                }
                i = i + 1;
        }
}
```

Tant que f est divisible par i, on le divise par i sans incrémenter i. Ansi, si par exemple f est divisible par 2^3, on divisera f par 2 à 3 reprise, ce qui permettra de stocker dans notre tableau uniquement un seul 2. En effet on ne cherche pas à stocker le nombre de fois que chaque diviseur apparaît, car nous n'en avons pas besoin dans notre programme. Suite à cela, si jamais f vaut une valeur différente de 1, cela signifie qu'il reste à stocker ce diviseur : ce qui est fait à la suite du programme, après avoir vérifié si on a bien $fact.longueur \geq 1$. En effet, si $fact.longueur == 0$, cela signifie qu'un diviseur n'a été trouvé : dans ce cas, on stocke dans le tableau f lui même. La suite de la fonction consiste juste à allouer de la mémoire pour la variable $fact.facteurs$, et libérer la mémoire des variables mpz_t.

Les deux autres fonctions `void clearFacteursPremiers(facteursPremiers f)` et `static void clearAll(mpz_t a, mpz_t p_1, mpz_t resultatPgcd, mpz_t resultatMod, mpz_t resultatCalcul, mpz_t puis)` sont simplement des fonctions qui vont libérer la mémoire du tableau des variables facteursPremiers ainsi que chaque variable du type mpz_t.

Voyons la fonction principale `int certificatPratt(mpz_t p, gmp_randstate_t state)` plus en détail. Tout d'abord, on teste simplement si $p = 2$. Dans ce cas la fonction renvoie directement 1. Sinon, on va prendre un a aléatoire, et vérifier les deux premières conditions, c'est à dire :

$$pgcd(a, p) = 1$$

et

$$a^{n-1} \equiv 1 \ (n).$$

Si une de ces condition n'est pas respectée, la fonction renvoie 0 (Car comme dit précédemment, ces conditions sont nécessaires). Sinon, on va stocker dans une variable facteursPremiers fact, les facteurs de $p - 1$. Pour chaque facteur (Pour i allant de 0 à fact.longueur), on teste si le certificat de Pratt renvoie

bien 1 sur chacun de ses facteurs fact.facteurs[i] (Il y a donc ici la récursivité du programme, et des calculs pris à la source du site Wikipédia montrent qu'il y a au maximum $4log_2 n - 4$ appels récursifs). Si un de ces certificats renvoie 0, cela veut dire que la fonction de factorisation à renvoyé un facteur non premier : erreur s'affiche, et on sort de la fonction en renvoyant -1. Sinon, on teste la dernière condition :

$$a^{\frac{n-1}{fact.facteurs[i]}} \not\equiv 1.$$

Si cette condition n'est pas respectée (C'est à dire que le résultat de ce calcul est 1), cela veut dire qu'on rentre dans le cas très rare que les deux premières conditions ont été respectés, mais pas la troisième : On relance donc un certificat sur p. Sinon, rien n'est effectué : le but est d'ici de s'assurer que chaque fact.facteur[i] respecte la dernière condition. Si on sort de la boucle en ayant testé chaque facteur, on peut alors renvoyer 1, car cela veut dire tous les facteurs respectent la relation

$$a^{\frac{n-1}{fact.facteurs[i]}} \not\equiv 1.$$

Si la fonction renvoie 1, on certifie alors que le nombre p donné en entrée est premier.

3 Amélioration du certificat : Pocklington

3.1 Explications de l'amélioration de Pocklington

Le test de primalité de Pocklington a été mis au point par Henry Cabourn Pocklington et Derrick Henry Lehmer.

Ce test repose sur le théorème de Pocklington (ou critère de Pocklington) :

Théorème 3.1 *Soit $N > 1$ un entier.*
S'il existe des nombres a et q tels que :

- *q est premier, $q|N-1$ et $q > \sqrt{N} - 1$ (1)*
- *$a^{N-1} \equiv 1 \bmod N$ (2)*
- *$pgcd(a^{(N-1)/q} - 1, N) = 1$ (3)*

Alors N est premier.

On prouve ce théorème avec un raisonnement par l'absurde : on suppose N non premier, ce qui implique qu'il existe un nombre premier $p \leq \sqrt{N}$ qui divise N. Donc $q > p - 1$, ce qui implique $pgcd(q, p-1) = 1$. Il existe donc un entier u tel que

$$uq \equiv 1. \bmod p - 1. \quad (4)$$

De ceci découle :
$1 \equiv a^{N-1} \bmod p$ par la propriété (2) et p|N
$\equiv (a^{N-1})^u \equiv a^{u(N-1)} \equiv a^{uq(N-1)/q} \equiv (a^{uq})^{(N-1)/q} \bmod p$
$\equiv a^{(N-1)/q} \bmod p$ par la propriété (4) et le petit théorème de Fermat.
Ceci montre que p divise le pgcd de la propriété (3), et que ce pgcd n'est donc pas égal à 1, ce qui est une contradiction.

Utiliser ce théorème pour un test de primalité est simple : pour un N donné, il faut chercher a et q tel que les 3 conditions sont validées. Si on y parvient, N est premier et (a, q) constituent un certificat de primalité qui peut être vérifié rapidement pour confirmer la primalité de N.

Le plus dur est de trouver un q qui vérifie les conditions, ce qui n'est pas toujours possible. Le test sera dans certains cas incapable de déterminer si un nombre est premier ou non.

3.2 Implémentation de la fonction de pocklington

La fonction `int pocklington(mpz_t n, facteursPremiers *f)` prend en entrée le nombre à tester n et une liste de facteurs premiers de $n-1$, et renvoie 2 si n est premier, 0 s'il ne l'est pas, et 1 si la primalité est indécidée. On commence par calculer le facteur minimum f_{imin} de la liste de facteurs premiers de $n-1$ tel que $f_{imin} > \sqrt{N} - 1$. Ensuite, on recherche des valeurs a et q en utilisant une double boucle imbriquée. a va prendre les valeurs successives de nombres

premiers, q va aller de f_{imin} jusqu'à f_{imax}. On regarde pour ces valeurs si on a bien les conditions (2) et (3), et on sort en renvoyant 2 dès qu'un couple (q, a) marche. Si on va au bout de la boucle sans trouver de couple (q, a) qui marche, on renvoie 1.
L'algorithme de la fonction est le suivant :

```
int pocklington(mpz_t n, facteursPremiers *f) {
        res = -1;
        i_min = 0;
        for (i = 0; i < f->longueur; i++) {
                if (n-1 < f->facteurs[i]) {
                        i_min = i;
                        break;
                }
        }
        for (j = 0; j < LIMIT; j++) {
                //On verifie si primes[j] = n (si oui,
                    n est premier)
                if (n == primes[j]) {
                        res = 2;
                        break;
                }
                //On verifie si a|n (si oui, n n'est
                   pas premier)
                if (pgcd(primes[j],n)=1) {
                        res = 0;
                        break;
                }
                //Test de Fermat entre a et n (si non,
                    n n'est pas premier)
                if (a^(n-1) mod n != 1) {
                        Echec du test de Fermat
                        res = 0;
                        break;
                }
                for (i = i_min; i < f->longueur; i++)
                   {
                        //On verifie si pgcd(a^((n-1)/
                           f->facteurs[i])-1, n) = 1 (
                           si oui, n est premier)
                        if (pgcd(a^((n-1)/f->facteurs[
                           i])-1,n) != 1) {
                                if (pgcd(a^((n-1)/f->
                                   facteurs[i])-1, n)
                                   = 1) {
                                        Echec du test
                                           de
                                           Pocklington
                                        res = 0;
                                        break;
```

```
                                }
                        }
                        else {
                                Succes du test de
                                   Pocklington
                                res = 2;
                                break;
                        }
                }
                if (res != -1) {
                        break;
                }
        }
        if (res == -1) {
                La primalité est indécidée.
                res = 1;
        }

        return res;
}
```

On effectue une série d'insctructions pour chaque valeurs de primes[j]. Dans cette boucle, on vérifie tout d'abord si $primes[j] = n$ (Car dans ce cas on aura pas a effectuer tout le certificat). Ensuite, on vérifie la première condition, c'est à dire si $pgcd(a, n) = 1$. Si oui, la variable res vaudra 0 (Ce qui indiquera que le nombre n'est pas premier), et on sort de la boucle. Ensuite on effectue le test de Fermat entre a et n, c'est à dire la condition suivante :

$$primes[j]^{n-1} \equiv 1 \ mod \ n.$$

Si cette condition est fausse, res vaut 0 et on sort de la boucle. Ensuite, pour chaque facteur premier, on vérifie :

$$pgcd(a^{(n-1)/f->facteurs[i]} - 1, n) = 1.$$

Si oui, alors on met res à 2. Sinon, on met res a 0. Si le res n'a jamais été modifié après chaque test avec tous les $primes[j]$, alors on ne peut pas conclure sur la primalité du nombre, mais c'est un cas très rare.

Conclusion

Le programme obtenu à l'issue du projet satisfait l'ensemble des critères fixés initialement, qui étaient de programmer les deux tests de Lucas-Lehmer (Pour les nombres de Mersenne et les nombres de Riesel) ainsi que le certificat de Pratt et son amélioration de Pocklington.

Les deux premières fonctions ont été assez rapidement comprises au niveau des algorithmes, puis codés facilement. Etant donné que nous savions coder avec la bibliothèque gmp.h, nous n'avons pas perdu de temps à l'apprentissage de l'utilisation de cette bibliothèque. Nous n'avons pas eu de problèmes particuliers au niveau de ces deux fonctions. Les réelles difficultés ce sont situés au niveau du certificat de Pratt ainsi que de son amélioration de Pocklington. Concernant le certificat de Pratt, l'algorithme à été bien plus dur à saisir au début, et même une fois saisi, le code n'a pas été simple à écrire. En effet, c'est surtout la récursivité qui posait problème : il fallait que le programme puisse être récursif, sans qu'il ne puisse se bloquer à l'infini.

Au final, l'efficacité du certificat de Pratt (où de son amélioration de Pocklington) pourrait être améliorée dans le futur, en se concentrant sur l'amélioration de la fonction de factorisation. On se reporte donc ici au problème NP la décomposition en produit de facteurs premiers.