Miller_Rabin

February 17, 2019

1 Tests de primalité

Marc Lorenzi
Juillet 2016

1.1 1 Le test naïf

La première idée qui vient à l'esprit pour savoir si un nombre p est premier est de chercher le plus petit diviseur de p (autre que 1, évidemment). En remarquant que si a divise p alors $\frac{p}{a}$ divise p, on voit que si un entier p est composé alors p possède un diviseur inférieur à $\sqrt{p}$.

La fonction ci-dessous renvoie le plus petit diviseur d'un entier p. Elle effectue dans le pire cas, en particulier lorsque p est premier, $\mathcal{O}(\sqrt{p})$ opérations.

```
In [1]: def plus_petit_diviseur(p):
            k = 2
            while k * k <= p and p % k != 0: k += 1
            if k * k > p: return p
            else: return k
```

```
In [2]: plus_petit_diviseur(91)
```

```
Out[2]: 7
```

On en déduit immédiatement une fonction prenant en paramètre un entier p et renvoyant True si et seulement si p est premier.

```
In [3]: def is_prime(p):
            if p <= 1: return False
            else: return plus_petit_diviseur(p) == p
```

```
In [4]: print([p for p in range(100) if is_prime(p)])
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Quels sont les plus grands nombres p dont on puisse décider la primalité avec la fonction **est_premier** en un temps raisonnable ? En acceptant de faire, disons, de l'ordre de un milliard d'opérations, on peut tester des nombres p inférieurs à 10^{18} soit des nombres d'une vingtaine de chiffres.

Comment faire, alors, pour savoir si un nombre de 100, 1000, 1000000 de chiffres est premier ? Certainement pas en utilisant cet algorithme. Il va valloir essayer autre chose.

1.2 2 Première tentative ... et échec

1.2.1 2.1 Le petit théorème de Fermat

La petit théorème de Fermat affirme que si p est un nombre premier alors pour tout entier a on a $a^p \equiv a[p]$.

Soit p un entier naturel. Soit a un entier. Nous dirons que a est un témoin de non-primalité de p lorsque $a^p \not\equiv a[p]$. Si un entier p possède un témoin de non-primalité alors, d'après le petit théorème de Fermat, il n'est pas premier.

La réciproque est-elle vraie ? Cela vaut la peine de tenter quelques expériences.

1.2.2 2.2 Le test de Fermat

Définissons tout d'abord une fonction d'exponentiation rapide modulo p. La fonction ci-dessous prend en paramètres trois entiers x, n, p. Elle renvoie x^n modulo p en effectuant un nombre de multiplications de l'ordre de $\log p$. D'où son appellation de "rapide". C'est la clé de tout ce qui va suivre : on peut élever un nombre à des puissances énormes quasi-instantanément.

```
In [5]: def power_mod(x, n, p):
            z = 1
            m = n
            y = x
            while m != 0:
                if m % 2 == 1: z = (z * y) % p
                m = m // 2
                y = (y * y) % p
            return z
```

```
In [6]: power_mod(2, 3 ** 1000, 561)
```

```
Out[6]: 2
```

Définissons ensuite la fonction **temoin_fermat**. Cette fonction prend un entier a en paramètre. Elle renvoie True si l'entier a est un témoin de non-primalité de p, et False sinon.

```
In [7]: def temoin_fermat(a, p):
            x = power_mod(a, p, p)
            return x != a
```

Voici enfin la fonction **test_fermat**. Cette fonction prend un entier p en paramètre. Elle tire 20 nombres au hasard entre 2 et $p-1$. Si l'un d'entre eux est un témoin de non primalité de p, on sait que p est composé. Si aucun n'est un témoin de primalité de p ... eh bien on ne sait pas trop.

```
In [8]: import random
```

```
In [9]: def test_fermat(p):
            c = 20
            while c > 0:
                a = random.randint(2, p - 1)
                c = c - 1
                if temoin_fermat(a, p): return False
            return True
```

```
In [10]: test_fermat(13)

Out[10]: True

In [11]: def is_fermat_prime(p):
             if p <= 1: return False
             elif p == 2: return True
             else: return test_fermat(p)

In [12]: print([p for p in range(100) if is_fermat_prime(p)])

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

1.2.3 2.3 Les nombres de Carmichael

Tout a l'air d'aller pour le mieux. Essayons jusqu'à 10000 ...

```
In [13]: print([p for p in range(10000) if is_fermat_prime(p) and not is_prime(p)])

[561, 1105, 1729, 2465, 2821, 6601, 8911]
```

Malheur ! Il y a 7 nombres entre 2 et 10000 qui ne sont pas premiers et qui, pourtant, passent le test. Nous n'y pouvons pas grand chose : ils font partie de la famille des nombres de Carmichaël. Bien que "rares" il y en a quand même une infinité : ce résultat a été démontré en 1994 par Alford, Granville et Pomerance. Pour les courageux, voici l'article original.

Tout est-il perdu pour autant ? Non, car en adaptant ce qui vient d'être fait on obtient quelque chose qui fonctionne : l'algorithme de Miller-Rabin.

1.3 3 Miller-Rabin

1.3.1 3.1 Le principe

Soit p un nombre impair, $p \geq 3$. On écrit tout d'abord $p = 1 + q2^t$ où q est un nombre impair et t un entier naturel, $t \geq 1$.

```
In [14]: def factor2(p):
             q = p - 1
             t = 0
             while q % 2 == 0:
                 q = q // 2
                 t = t + 1
             return (q, t)

In [15]: p = 10013
         print(is_prime(p))
         q, t = factor2(p)
         print(q, t)
```

```
False
2503 2
```

Supposons que $p = 1 + q2^t$ est un nombre premier. Soit a un entier entre 2 et $p-1$. D'après le petit théorème de Fermat, on a $a^{p-1} \equiv 1[p]$, ou encore $a^{q2^t} \equiv 1[p]$. Soit $x = a^{q2^{t-1}}$. On a $x^2 \equiv 1$ mais comme p est premier, l'anneau $\mathbb{Z}/p\mathbb{Z}$ est un corps. Donc, de $x^2 - 1 = (x-1)(x+1) = 0$ on déduit $x = \pm 1$. Si $x \equiv 1$, soit $y = a^{q2^{t-2}}$. On a $y^2 \equiv 1$ donc, encore une fois, $y \equiv \pm 1$. Ainsi de suite ... Ainsi, si p est premier, alors - $a^q \equiv 1[p]$, ou alors - il existe $e \in [0, t-1]$ tel que $a^{q2^e} \equiv -1[p]$.

Nous définissons donc une fonction **temoin_miller** qui prend en paramètres les entiers a, q, t, p, et qui renvoie True si aucune des deux conditions ci-dessus n'est réalisée, et False sinon. Si, pour un certain entier a, la fonction renvoie False, alors a est dit témoin de Miller (de non primalité) pour p : l'existence d'un tel a prouve que p n'est pas premier.

```
In [16]: def temoin_miller(a, q, t, p):
             b = power_mod(a, q, p)
             if b == 1: return False
             e = 0
             while b != 1 and b != p - 1 and e <= t - 2:
                 b = (b * b) % p
                 e = e + 1
             return b != p - 1
```

1.3.2 3.2 Cette fois-ci, cela fonctionne

La fonction **temoin_miller** est un raffinement de la fonction **temoin_fermat**. Et alors ? Eh bien nous avons le résultat suivant (que nous admettrons) :

Soit $p \geq 3$ un nombre impair non premier. Soit L l'ensemble des entiers entre 2 et $p-1$ *qui ne sont pas* des témoins de Miller de non primalité pour p. Alors, le cardinal de L vérifie $|L| \leq \frac{p-1}{4}$.

Ainsi, - Si p est premier, la fonction **temoin_miller** renvoie toujours False - Si p est composé, la fonction **temoin_miller(a)**, avec en argument un entier a aléatoire entre 2 et $p-1$, renvoie False avec une probabilité inférieure à $1/4$.

En admettant que les non-témoins de Miller sont uniformément répartis entre 2 et $p-1$, si l'on exécute la fonction **temoin_miller** en prenant c nombres a tirés au hasard, la probabilité que l'on obtienne tout le temps False alors que p est composé est donc inférieure à $(\frac{1}{4})^c$. Pour $c = 20$, par exemple, cette probabilité est inférieure à :

```
In [17]: 0.25 ** 20
```

```
Out[17]: 9.094947017729282e-13
```

À titre de comparaison, c'est environ la probabilité de gagner cent mille fois de suite au loto.

```
In [18]: p = 8191 * 23
         print(p)
         print(is_prime(p))
         q, t = factor2(p)
         s = [a for a in range(2, p) if not temoin_miller(a, q, t, p)]
         print(len(s) / float(p))
```

```
188393
False
5.308052846974144e-06
```

Regroupons tout cela dans une unique fonction.

```
In [19]: def test_miller(p):
             q, t = factor2(p)
             c = 20
             while c > 0:
                 a = random.randint(2, p - 1)
                 c = c - 1
                 if temoin_miller(a, q, t, p): return False
             return True
```

```
In [20]: def is_probably_prime(p):
             if p <= 1: return False
             elif p == 2: return True
             elif p % 2 == 0: return False
             else: return test_miller(p)
```

```
In [21]: s1 = [p for p in range(10000) if is_probably_prime(p)]
         s2 = [p for p in range(10000) if is_prime(p)]
         print(s1 == s2)
```

```
True
```

```
In [23]: p = 119506876879526579251836131572511635189824558 1
         print(is_probably_prime(p))
```

```
False
```

```
In [24]: a = 24444516448431392447461
         b = 48889032896862784894921
         print(is_probably_prime(a))
         print(is_probably_prime(b))
         print(a * b - p)
```

```
True
True
0
```

1.3.3 3.3 Test sur de “grands” nombres

Posons, pour tout entier n, $M_n = 2^n - 1$. Il n'est pas difficile de montrer que si M_n est premier, alors n est aussi un nombre premier.

```
In [25]: def mersenne(p):
             return 2 ** p - 1

In [26]: for p in range(103):
             if is_prime(p):
                 print(p, mersenne(p))
```

```
2 3
3 7
5 31
7 127
11 2047
13 8191
17 131071
19 524287
23 8388607
29 536870911
31 2147483647
37 137438953471
41 2199023255551
43 8796093022207
47 140737488355327
53 9007199254740991
59 576460752303423487
61 2305843009213693951
67 147573952589676412927
71 2361183241434822606847
73 9444732965739290427391
79 604462909807314587353087
83 967140655691703339764940 7
89 618970019642690137449562111
97 158456325028528675187087900671
101 25353012004564588029934064107 51
```

Les nombres de Mersenne deviennent vite grands lorsque n augmente. Par exemple, le 1000 ème nombre de Mersenne possède 1000 chiffres en base 2, soit environ 300 chiffres décimaux.

```
In [27]: from math import log
         1000 * log(2) / log(10)
```

```
Out[27]: 301.02999566398114
```

```
In [28]: print(mersenne(1000))
```

```
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156 9
```

Quels sont, les nombres de Mersenne M_p probablement premiers, pour p premier, $p \leq 2000$?

```
In [29]: s = [p for p in range(2000) if is_prime(p)]
         print(s)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
```

```
In [30]: s1 = [(p, mersenne(p)) for p in s]
```

```
In [33]: s2 = [p for (p, n) in s1 if is_probably_prime(n)]
         print(s2)
```

```
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279]
```

```
In [35]: print(mersenne(1279))
```

```
104079321946643990819252403273640855386152622472667048053191123504036080596733602980122394417323
```

Un dernier essai, avec M_{4253} qui est le premier nombre de Mersenne premier ayant plus de 1000 chiffres.

```
In [36]: p = mersenne(4253)
         print(p)
```

```
190797007524439073807468042969529173669356994749940177394741882673528979787005053706368049835514
```

```
In [37]: is_probably_prime(p)
```

```
Out[37]: True
```

À titre de comparaison, l'algorithme naïf sur une machine effectuant 10^{10} opérations par secondes demanderait un temps de ... 10^{3000} millénaires.

1.4 4 Fabriquer de grands nombres premiers

1.4.1 4.1 Créer un nombre premier de taille donnée

Comment fabriquer un nombre premier ayant une certaine taille, disons L bits ? Un algorithme naïf consite à tirer un entier n au hasard entre 2^{L-1} et 2^L. Puis, tant que n n'est pas premier, à incrémenter n. Quoique naïf, cet algorithme fonctionne plutôt bien jusqu'à des valeurs de L de l'ordre du millier. Je n'entrerai pas plus avant dans les détails.

```
In [38]: def random_prime(L):
             a = 2 ** (L - 1)
             b = 2 * a - 1
             n = random.randint(a, b)
             if n % 2 == 0: n = n + 1
             while not is_probably_prime(n):
                 n = n + 2
             return n
```

```
In [39]: p = random_prime(30)
         print(p)
```

```
573742219
```

```
In [40]: print(is_probably_prime(p))
```

```
True
```

1.4.2 4.2 Créer des clés RSA

Le chiffrement crypographique RSA nécessite la création de clés. Créer une clé de taille N (bits) consiste à créer deux nombres premiers de taille $N/2$. Vous désirez créer votre clé RSA-2048 ? rien de plus facile ...

```
In [42]: p = random_prime(1024)
         q = random_prime(1024)
         n = p * q
         print(n)
```

```
16393884042692445230727591532992543074648901984371329861645477461074537225986368015037302783212
```

La connaissance de n permet le cryptage de messages. La connaissance de p et q permet le décryptage. Avis aux amateurs : factorisez le nombre ci-DESSOUS. Parce que ci-dessus c'est facile : print(p, q) :-).

```
In [43]: n = random_prime(1024) * random_prime(1024)
         print(n)
```

```
18971922618517022695439966888497800882617802659512893876663543759501932932866523367120165646364
```

```
In [ ]:
```