Factorisation d'entiers

Stéphane Horte & Gabriel Lewertowski

24 mars 2014

Table des matières

Résumé

Nous présentons trois algorithmes de factorisation d'entiers : les algorithmes ρ et $(p-1)$ proposés par J.M Pollard en 1974 dans [1] ainsi que l'algorithme *ECM* (Elliptic Curve Method) décrit par H. W. Lenstra, Jr dans [4]. Nous testons ensuite nos implémentations sur des grands nombres choisis aléatoirement, et nous comparons avec les résultats obtenus par GMP-ECM.

1 Présentation des algorithmes

1.1 Algorithme ρ de Pollard

Introduit en 1975 par John M. Pollard [1], l'algorithme ρ (parfois Monte-Carlo) est très efficace pour déterminer des petits facteurs premiers (de l'ordre de 10 chiffres). Des améliorations furent proposées, notamment en 1985 par Brent [3] pour réduire le nombre de calculs effectués. Nous nous intéressons ici à la version initiale de John M. Pollard, qui trouve un facteur premier p en $O(\sqrt{p})$ itérations.

1.1.1 Principe de fonctionnement

Soit N un entier quelconque à factoriser.

Nous nous plaçons dans $\mathbb{Z}/N\mathbb{Z}$. Soit f une fonction polynomiale d'ordre au moins 2 et à coefficients dans $\mathbb{Z}$. Par compatibilité de l'addition et de la multiplication avec la réduction modulo N, f définit une fonction de $\mathbb{Z}/N\mathbb{Z}$ dans $\mathbb{Z}/N\mathbb{Z}$. Pour toutes fonctions polynomiales réduites modulo N et pour tout x_0 élément de $\mathbb{Z}/N\mathbb{Z}$, on définit la suite des itérations $(x_i)_{i\in\mathbb{N}}$ avec $x_i = f^i(x_0)$. La suite des $(x_i)_{i\in\mathbb{N}}$ est à valeurs dans un ensemble fini. Par conséquent il existe forcément deux indices $i \neq j$ tels que $x_i = x_j$. Soit (i,j) le plus petit couple (pour l'ordre lexicographique) qui vérifie cette propriété. La période de la suite est donc $j - i = \lambda$. On remarque aussi que pour tout k et l dans $\mathbb{N}$ on a $x_{i+k} = x_{j+k+l\lambda}$. Par conséquent on peut trouver un entier t tels que $x_t = x_{2t}$. Le plus petit e vérifiant cette propriété s'appelle **l'épacte** et est noté e. Il est démontré les résultats suivants :

$$\mathbb{E}(\lambda) = \sqrt{\frac{N\pi}{8}} + O(1) \approx 0.627\sqrt{N} + O(1)$$

$$\mathbb{E}(e) = \sqrt{\frac{N\pi^5}{288}} + O(1) \approx 1.03\sqrt{N} + O(1)$$

où $\mathbb{E}$ désigne l'espérance probabiliste.

Soit p un facteur de N encore inconnu. La fonction f se réduit à une fonction de $\mathbb{Z}/p\mathbb{Z}$ dans $\mathbb{Z}/p\mathbb{Z}$ en prenant sa réduction modulo p de sa réduction modulo N. Par les théorèmes énoncés précédemment nous sommes capables de trouver en un nombre d'étapes $O(\sqrt{p})$ un entier l tel que $x_l = x_{2l}$ modulo p. Pour un tel indice, p divise $x_l - x_{2l}$ et par conséquent p divise aussi le pgcd de N et de $x_l - x_{2l}$. C'est dans ce résultat que réside le principe de la méthode ρ de Pollard : Pour chercher un facteur de N il nous suffit alors de calculer les pgcd successifs de N et de $x_i - x_{2i}$ pour $i = 1, 2, \ldots$.

L'avantage de cet algorithme par rapport à un test trivial de divisibilité par tous les nombres inférieurs à $\sqrt{N}$ est immédiat : on obtient une complexité $O(\sqrt{p})$ au lieu de $O(\sqrt{N})$. Remarquons que nous n'avons pas besoin de connaître explicitement p pour appliquer la méthode ρ.

1.1.2 Notre code

Pour chacun des algorithmes ρ, $p-1$ et *ECM*, nous implémentons en python une fonction *un_facteur* qui trouve un facteur premier. Cette fonction peut avoir recours au test de primalité de Miller-Rabbin, dont l'implémentation est donnée en Annexe. Enfin, pour chaque algorithme, une fonction *factorise* (voir également le code en Annexe) permet de factoriser totalement un entier.

```python
# -*- coding: utf-8 -*-
# pollard.py
import fractions
import utils


def un_facteur(N, f=lambda x, m: (x*x + 1) % m, k=3):
    """
    Calcule un facteur de N avec la méthode Pollard-rho.

    :Parameters:
        - N: l'entier à factoriser
        - f: la fonction d'itération à utiliser
        - k: le nombre d'appels à AMR pour déterminer si un entier est premier
    """

    if utils.AMR(N, k):
        # N est probablement premier
        return N

    x, y, g = 1, 1, 1
    while (g == 1):
        x, y = f(x, N), f(f(y, N), N)
        g = fractions.gcd(x - y, N)

    if (g == N):
        raise Exception("Impossible de trouver un facteur")
    else:
        return g
```

1.2 La méthode $p-1$

La méthode ρ que nous venons d'étudier ne présuppose aucune hypothèse sur l'entier N à factoriser. Dans le cas où N possède un facteur premier p tel que $p-1$ est **friable**, c'est-à-dire qu'il ne possède que des petits facteurs premiers, la méthode $p-1$, également inventée par John M. Pollard, permet de trouver p plus rapidement.

1.2.1 Le principe de fonctionnement

Le principe de l'algorithme $p-1$ repose sur le théorème de Fermat : si p est un nombre premier et si a est un entier non divisible par p alors $a^{p-1} \equiv 0 \pmod{p}$.

Considérons encore une fois N le nombre que nous cherchons à décomposer et p un de ses facteurs à priori inconnu. Soit $a \in \{2, \ldots, N-2\}$ premier avec N.

La première phase

Soit B_1 une borne, typiquement $B_1 = 10^6$. Soit $R = ppcm(1, 2, ..., B_1)$. On calcule ensuite le nombre $b \equiv a^R \pmod{N}$ en un temps $O(B_1)$. Si $p-1$ divise R alors, par le théorème de Fermat on a p qui divise $b-1$, et donc le pgcd de $b-1$ et de N. $pgcd(b-1, N)$ est donc un facteur de N, non nécessairement premier puisque tous les facteurs q de N tels que $q-1$ divise R apparaissent dans $pgcd(b-1, N)$. La première étape a échoué si $d = pgcd(p, a^{R-1})$ vaut 1 ou N. Si $d = 1$ alors deux solutions existent :

— recommencer avec une borne B_1 plus grande
— passer à l'étape 2 (voir plus loin)

Remarque : si $d = N$, tous les facteurs q de N sont tels que $q-1$ divise R, il faut donc recommencer avec B_1 plus petit.

Cette étape nécessite que tous les facteurs de $p-1$ soient petits. Cette technique n'est donc clairement pas efficace si $p-1$ est, par exemple, la multiplication de deux nombres premiers de tailles équivalentes. On peut cependant remarquer que la première étape finira toujours par renvoyer un facteur p de N quitte à augmenter B_1.

La seconde phase

La seconde phase requiert une autre borne $B_2 \gg B_1$, par exemple $B_2 = 1000B_1$. On suppose que $p-1$ n'a qu'un seul grand facteur premier dans sa décomposition, c'est à dire $p-1 = sh$ avec h qui divise R de la première phase et $B_1 \leqslant s \leqslant B_1$. Pour tous les s nombre premier entre B_1 et B_2 on calcule $T_s = b^s = a^{Rs}$. On calcule ensuite $d_s = pgcd(N, T_s)$ en espérant que celui ci soit différent de 1. Comme expliqué dans la première phase, si $p-1$ divise $R \cdot s$ alors p divise T_s et p divise d_s. Comme dans la première phase on remarque que dans le cas où $d_s > 1$ alors d_s est un facteur de N mais pas obligatoirement un facteur premier. La seconde phase à une complexité de l'ordre de $\log(B_2)^2$. Encore une fois, dans le cas où $p-1$ possède au moins deux grands nombres premiers dans sa décomposition, la méthode $p-1$, même accompagnée de la seconde phase, n'est pas très efficace.

Commentaires

Cette méthode repose, tout comme le théorème de Fermat, sur l'ordre du groupe multiplicatif $(\mathbb{Z}/p\mathbb{Z})^*$ qui vaut $p-1$ et sur le théorème de Lagrange qui stipule que l'ordre d'un élément divise forcément l'ordre du groupe. L'idée de se déplacer dans un groupe associé à p est très intéressante. Les méthodes $p+1$ et *ECM* (Elliptic Curve Method) sont très largement inspirées de cette technique en se plaçant sur d'autres groupes, faisant ainsi d'autres hypothèses sur l'écriture des facteurs de N.

1.2.2 Notre code

```python
# -*- coding: utf-8 -*-
# pmoinsun.py
import utils
import math
import fractions


def un_facteur(n, b1=10**6, phase2=True, b2=5*10**8, k=3):
    """
    Renvoie un facteur de n avec la méthode p-1

    """

    if utils.AMR(n, k):
        # n probablement premier
        return n

    b = 2
    primes_up_to_bound = utils.primes_from_file(1, b1)
    for p in primes_up_to_bound:
        q = utils.valuation(b1, p)
        b = pow(b, q, n)
        g = fractions.gcd(b-1, n)
        if g > 1:
            break

    if g == 1:
        if not phase2:
            raise Exception('Aucun facteur calculé : il faut augmenter B1 ou faire la phase 2')

        # Phase 2, continuation standard
        # On tabule b^i mod n pour i <= log(b2)^2
```

```
33          c = [pow(b, i, n) for i in range(int(math.log(b2)**2) + 1)]
34
35          primes = utils.primes_from_file(b1, b2)  # nombres premiers entre b1 et b2
36          s1 = primes.next()
37          t = pow(b, s1, n)  # t va itérativement contenir b^s pour tous les s premiers
                , b1 <= s <= b2
38          for s2 in primes:
39              pgcd = fractions.gcd(t - 1, n)
40              if pgcd > 1:
41                  return pgcd
42
43              t = (t*c[s2 - s1]) % n  # s1, s2 = primes[i], primes[i+1]
44              s1 = s2
45
46          pgcd = fractions.gcd(t, n)
47          if pgcd == 1:
48              raise Exception('Aucun facteur trouvé aprés la phase 2')
49
50      elif g == n:
51          raise Exception('Aucun facteur calculé : Il faut réduire b1')
52      else:
53          return g
```

1.3 La méthode *ECM*

La méthode *ECM* propose le même raisonnement que $p-1$ mais en se plaçant sur les courbes elliptiques sur $\mathbb{Z}/N\mathbb{Z}$. Pour N non premier, les points d'une courbe elliptique sur $\mathbb{Z}/N\mathbb{Z}$ ne forment pas un groupe car l'addition de deux points de la courbe nécessite l'inversion d'un facteur, qui peut être non inversible dans $\mathbb{Z}/N\mathbb{Z}$. Cependant, si ce facteur est non inversible, c'est qu'il n'est pas premier avec N : un facteur de N est dès lors trouvé.

1.3.1 Les courbes elliptiques

Nous prendrons couramment l'écriture de Weierstrass pour définir les courbes elliptiques. Une courbe elliptique dans $\mathbb{Z}/N\mathbb{Z}$ est définie par la donnée de deux coefficients A et B tels que $4A^3 + 27B^2 \neq 0$. Ce sont les points (X, Y) de $(\mathbb{Z}/N\mathbb{Z})^2$ qui vérifient

$$Y^2 = X^3 + AX + b \tag{1}$$

Il est évident, pour les raisons exposées en 1.1.1, qu'une courbe elliptique dans $\mathbb{Z}/N\mathbb{Z}$ définit aussi une courbe elliptique dans $\mathbb{Z}/p\mathbb{Z}$. Cette courbe elliptique définit bien un groupe car p est premier. Ce groupe est d'ordre $p + 1 - t_{A,B}$ où $t_{A,B}$ dépend de A et B avec $|t_{A,B}| \leqslant 2\sqrt{p}$ (théorème de Hasse). Tout comme l'algorithme $p-1$ réussit si $p-1$ est **friable**, la méthode *ECM* réussit si $p+1-t_{A,B}$ est friable. La grande force de la méthode *ECM* est qu'il est peu probable que deux courbes elliptiques aient le même ordre. En effectuant l'algorithme avec une courbe choisie aléatoirement, l'ordre du groupe varie à chaque itération, ce qui multiplie la probabilité de succès par rapport à $p-1$.

Dans ce cas il est même possible de calculer le point $k \cdot (x, y)$ en un temps $O(\log_2(k))$. Si un point est écrit grâce à la forme homogénéisé (x, y, z) alors c'est le 0 de la courbe modulo p si et seulement si p divise z. C'est ce que nous serons amenés à tester.

À nouveau, l'algorithme se présente comme une succession de deux phases.

La première phase

La première phase est équivalente à la première phase de la méthode $p-1$. Soit B_1 une borne. Nous choisissons (à priori aléatoirement) une courbe elliptique et un point sur cette courbe. Dans la pratique, il est plus facile de choisir un point $P_0 = (x_0, y_0, 1)$ et un coefficient A puis chercher B tel que la courbe définie grâce à A et B passe par le point P_0. Nous posons de manière analogue

$R = ppcm(1, 2, .., B_1)$ puis nous calculons le point $R \cdot P_0$ en un nombre d'opérations de l'ordre de $\log(R) = O(B_1)$. Pour cela, l'utilisation de l'équation homogène

$$ZY^2 = X^3 + AZ^2X + Z^3b$$

permet une multiplication rapide en évitant le calcul d'inverses.

Si l'ordre du groupe des points de la courbe elliptique divise R, alors le point $R \cdot P_0 = (x, y, z)$ est égal à O_E, l'élément neutre de la courbe (i.e $p|z$). Le test de fin de la première phase est donc le calcul de $d = pgcd(N, z)$, qui fournit un diviseur de N lorsque $1 < d < N$. On dit que la première phase a échoué si $d = 1$ ou $d = N$. Si $d = N$, on peut recommencer avec cette courbe elliptique et une borne B_1 plus petite ou recommencer avec une autre courbe. Si $d = 1$ on peut recommencer avec une borne B_1 plus grande, passer à la seconde phase ou changer de courbe et reprendre à la phase 1.

La seconde phase

Pour la seconde phase nous nous donnons une borne $B_2 \gg B_1$. Nous repartons du point $Q = R \cdot P_0$ calculé durant la phase précédente. Pour tous les nombres premiers s compris entre B_1 et B_2 nous remplaçons Q par $s \cdot Q$. Une fois tous les nombres premiers compris entre B_1 et B_2 passés il ne reste qu'à calculer $d = pgcd(z, N)$ ou z représente la troisième coordonnée de Q. À nouveau, si $1 < d < N$, on a trouvé un diviseur de N. Si $d = 1$ il faut augmenter les bornes ou bien recommencer avec une autre courbe de façon à changer le cardinal du groupe sur lequel nous travaillons. Si $d = N$, on peut à souhait recommencer sur une autre courbe ou baisser la borne B_2 sur la même courbe. La seconde phase est une version allégée de la première phase. En effet, cela revient à supposer qu'il n'est plus nécessaire de prendre en compte les puissances de "petits" nombres premiers trop grandes. Par exemple, la plus grande puissance de 2 considérée est $2^{[\log(B_1)]}$. Contrairement à la méthode $p - 1$, le pire cas n'est pas celui où le cardinal du groupe s'écrit comme produit de deux "grands" nombres premiers mais le cas où un "grand" nombre premier apparaît à une puissance au moins 2 dans la décomposition de l'ordre du groupe.

1.3.2 Notre code

```python
# -*- coding: utf-8 -*-
# ECM.py

import random
import utils
import fractions


def un_facteur(n, B1=10**6, nb_essais=1, phase2=True, B2=5*10**8):
    """
    Calcule un facteur de n avec la méthode ECM.
    k est le nombre d'essais.

    """
    for i in range(nb_essais):
        if(n % 2 == 0):
            return 2
        delta = n
        while (delta == n):
            x0 = random.randint(0, n-1)
            y0 = random.randint(0, n-1)
            a = random.randint(0, n-1)
            b = ((y0**2) - (x0**3) - a * x0) % n
            delta = fractions.gcd(4*a**3 + 27*b**2, n)
        if(delta != 1):
            return delta
        P = (x0, y0)
```

```
28          d = (utils.XGCD(4, n)[1]) % n
29          d = (d * (a+2)) % n
30          for k in utils.primes_from_file(1, B1):
31              u = k
32              while(u <= B1):
33                  u = u*k
34                  P = ECM_mult(P, k, d, n)
35                  g = fractions.gcd(n, P[1])
36                  if(g != 1):
37                      return g
38
39          if phase2:
40              for p in utils.primes_from_file(B1, B2):
41                  P = ECM_mult(P, p, d, n)
42                  g = fractions.gcd(P[1], n)
43                  if g > 1:
44                      return g
45
46      raise Exception('Aucun_facteur_trouvé')
47
48
49  def ECM_mult((x, z), k, d, n):
50      """
51      Multiplication rapide sur une courbe elliptique
52
53      :Parameters:
54       -`(x, z)`: un point de la courbe
55       -`k`: un entier
56       -`d`: l'inverse de 4 mod n
57       -`n`: la caractéristique de l'anneau sur lequel la courbe est définie
58      :return:
59       - k.P
60
61       """
62
63      xp = 1
64      xq = x
65      zp = 0
66      zq = z
67
68      i = k.bit_length()
69      for l in range(i):
70
71          # regarde si le i-l eme bit de k est un 0 ou un 1
72          f = (k >> (i-1-l))-((k >> (i-l)) << 1)
73          if(f == 0):
74              xv = (pow((xp+zp), 2, n)*pow((xp-zp), 2, n)) % n
75              zv = ((4*xp*zp)*(pow((xp-zp), 2, n)+d*4*xp*zp)) % n
76              xw = (z*pow((xq-zq)*(xp+zp)+(xq+zq)*(xp-zp), 2, n)) % n
```

2 *ECM*, hier et aujourd'hui

La méthode *ECM* a été proposée par H. W. Lenstra en 1985 [4], et peu d'améliorations ont été apportées depuis. Alors qu'à ses débuts, l'algorithme permet de trouver des facteurs premiers de 30 chiffres au maximum, R. Brent prédit qu'*ECM* permettra à l'avenir de trouver des facteurs premiers de 50 chiffres. En effet, dix ans plus tard en 1995, P. Montgomery trouve un facteur de 47 chiffres de $5^{256}+1$. L'actuel record est un facteur de 83 chiffres, trouvé en septembre 2013 par R. Propper en factorisant $7^{337}+1$ [8].

P. Montgomery et R. Propper se sont intéressés à ces nombres dans le cadre du Projet Cunnin-

gham, lancé en 1925 et visant à factoriser des entiers de la forme $b^n \pm 1$ avec $b \in \{2, 3, 5, 6, 7, 10, 11, 12\}$ et n grand. *ECM* fait partie, avec le crible algébrique et le crible quadratique à polynômes multiples, des trois seuls algorithmes utilisés actuellement pour factoriser ces nombres.

Aujourd'hui, l'algorithme *ECM* est le meilleur algorithme de factorisation connu parmi ceux pour lesquels la complexité dépend de la taille du facteur trouvé et non de la taille de l'entier à factoriser : on ne considère donc pas le crible quadratique et le crible algébrique, dont les complexités respectives en fonction de l'entier n à factoriser valent $O\left(e^{\sqrt{\log n \log\log n}}\right)$ et $O\left(e^{(\frac{64}{9}\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}}\right)$.

L'implémentation la plus rapide de l'algorithme *ECM* est GMP-ECM, un programme développé en C par des chercheurs de l'INRIA, permettant de factoriser un entier avec les méthodes $p-1$, $p+1$ et *ECM*.

2.1 Fonctionnement de GMP-ECM

Le programme prend en paramètre la borne B_1 de l'étape 1 et un ou plusieurs entiers à factoriser depuis l'entrée standard. Par exemple, pour factoriser un entier avec $B_1 = 10^6$:

```
gabriel@gabriel-Inspiron-1545:~$ echo "127414639345398724455633733782600403037059161312 67" | ecm 1e6
GMP-ECM 6.4.2 [configured with GMP 5.1.2, --enable-asm-redc] [ECM]
Input number is 127414639345398724455633733782600403037059161312 67 (50 digits)
Using B1=1000000, B2=1045563762, polynomial Dickson(6), sigma=2197245383
Step 1 took 1928ms
********** Factor found in step 1: 6465167595551812530 4781
Found composite factor of 23 digits: 6465167595551812530 4781
Probable prime cofactor 19707863324852242832543 3807 has 27 digits
gabriel@gabriel-Inspiron-1545:~$
```

Par défaut, la méthode utilisée est *ECM*. On peut utiliser $p-1$ avec l'option -pm1 :

```
1  echo "12652209139612535291" | ecm -pm1 1e6
```

L'utilisateur peut également spécifier une valeur de B_2 pour la phase 2. Par exemple, pour utiliser $B_1 = 10^6$ et $B_2 = 10^9$:

```
1  echo "12652209139612535291" | ecm 1e6 1e9
```

Si aucune valeur de B_2 n'est spécifiée, GMP-ECM détermine automatiquement la valeur de B_2 optimale en fonction de la taille de l'entier à factoriser.

Nb de chiffres de N	B_1 optimal	B_2 optimal	Nb de courbes à utiliser
20	11e3	1.9e6	74
25	5e4	1.3e7	214
30	25e4	1.3e8	430
35	1e6	1.0e9	904
40	3e6	5.7e9	2350
45	11e6	3.5e10	4480
50	43e6	2.4e11	7553
55	11e7	7.8e11	17769
60	26e7	3.2e12	42017
65	85e7	1.6e13	69408

2.2 Détails de l'implémentation

L'algorithme *ECM* tel que présenté en 1985 par Lenstra sélectionnait aléatoirement la courbe elliptique sur laquelle travailler. P. Montgomery a proposé en 1987 dans [5] une amélioration pour travailler avec une famille infinie de courbes dont l'ordre est divisible par 12, ce qui augmente la

probabilité de succès puisque l'algorithme *ECM* réussit si l'ordre du groupe n'a que des petits facteurs. C'est cette famille de courbes qu'utilise GMP-ECM, avec la paramétrisation

$$by^2z = x^3 + ax^2z + xz^2$$

qui permet une addition et une multiplication rapides comme nous l'avons vu en 1.3.1.

Lors de la phase 2, GMP-ECM a recours à l'extension de Brent-Suyama qui, au lieu de calculer b^s pour s un nombre premier dans l'intervalle $[B_1, B_2]$, calcule $b^{(6k)^e-1}$ avec k un entier et e un petit entier pair. Cette extension utilise la remarque suivante : tout nombre premier strictement supérieur à 3 est de la forme $6k \pm 1$. Pour plus de détails sur l'extension de Brent-Suyama et l'évaluation multipoints, voir [6].

3 Résultats expérimentaux

3.1 Protocole

Pour tester les différents algorithmes de factorisation, nous avons cherché à factoriser des entiers "aléatoires" comportant $10, 20, \ldots, 200$ chiffres. Pour cela, nous avons utilisé les décimales de π :

```
1  def decimales_pi(n, offset=0):
2      """
3      Renvoie n décimales de pi en partant de la offset-iéme
4
5      """
6
7      with open('pi.txt') as f:
8          f.seek(offset)
9          return int(f.read(n))
```

où *pi.txt* est un fichier contenant les 500 premières décimales de π. Nous avons ainsi effectué 100 tests différents pour chaque taille d'entiers et pour les algorithmes suivants :

- $p-1$ sans phase 2
- $p-1$ avec phase 2
- *ECM* testé sur 5 courbes
- *ECM* testé sur 50 courbes

Le test est réussi si l'entier a été entièrement factorisé. Ici, seule nous intéresse la faisabilité et non la rapidité : nous avons donc utilisé GMP-ECM plutôt que nos implémentations.

3.2 Résultats pour $B_1 = 10^4$

4 Annexe

Nous détaillons ici l'implémentation des fonctions arithmétiques utilisées.

4.1 AMR

```
1  def AMR(n, k=3):
2      """
3      Test Miller-Rabin pour déterminer si un entier est premier
4
5      :Parameters:
6          -'n' : l'entier dont on veut savoir s'il est premier
7          -'k' : le nombre de fois qu'on lance le test
8
9      :return:
```

```
        False : n est un nombre composé
        True : n est probablement un nombre premier (mais n peut être composé)

    """

    if (n % 2 == 0):
        return False

    if (n == 3):
        return True

    d, s = n-1, 0
    while (d % 2 == 0):
        d, s = d/2, s+1

    def WitnessLoop():
        a = random.randint(2, n-2)
        x = pow(a, d, n)
        if (x == 1 or x == n-1):
            return True
        for j in range(s-1):
            x = pow(x, 2, n)
            if x == 1:
                return False
            if x == n-1:
                return True
        return False

    for i in range(k):
        if not WitnessLoop():
            return False

    return True  # n est probablement premier
```

4.2 Crible d'Eratosthène segmenté

```
def segmented_eratho(l, r):
    """
    Renvoie les nombres premiers p tels que l <= p <= r

    http://programmingpraxis.com/2010/02/05/segmented-sieve-of-eratosthenes/

    Cette fonction stocke en mémoire une liste de taille (r-l)/2.
    Si (r-l)/2 > 10^7, on utilisera big_segmented_eratho.

    """

    # On se ramène au cas où l et r sont impairs
    if l % 2 == 0:
        l += 1

    if r % 2 == 0:
        r -= 1

    primes = eratho(int(math.sqrt(r)))[1:]  # on enléve 2
    q = [(-(l+p)/2) % p for p in primes]
    table = ((r-l)/2 + 1)*[True]
    for i, p in enumerate(primes):
        for j in range(q[i], ((r-l)/2 + 1), p):
            table[j] = False

    return [l+2*i for (i, b) in enumerate(table) if b]
```

Références

[1] J. M. Pollard, *A Monte Carlo method for factorization*, BIT, 1975, pp.331-334

[2] J. M. Pollard, *Monte Carlo methods for index computation (mod p)*, Mathematics of computation, Vol. 32, No.143 (Juillet 1978), pp. 918-924

[3] R. P. Brent, *An improved Monte Carlo factorization algorithm*, BIT, 1980, pp.176-184

[4] H. W. Lenstra, Jr, *Factoring integers with elliptic curves*, Annals of Mathematics, 1987, pp.649-673

[5] P. Montgomery, *Speeding the Pollard and Elliptic curve methods of factorization*, American Mathematical Society, jan. 1987, pp. 243-264

[6] P. Zimmerman, *GMP-ECM : yet another implementation of the Elliptic Curve Method*, conférence.

[7] Paul Zimmermann & Bruce Dodson, *20 years of ECM*

[8] `http://www.loria.fr/~zimmerma/records/top50.html`

[9] ECM README, `http://srawlins.ruhoh.com/ecm_readme/`