TIPE ENS: Étude de méthodes pour la factorisation en facteurs premiers

1 Introduction

La sécurité des méthodes de cryptographie comme le protocole RSA se base sur la difficulté à factoriser un entier en ses facteurs premiers. Nous verrons ici trois méthode permettant la résolution de ce problème, chacune ayant une spécificité par rapport à l'autre. Et nous discuterons de la complexité de ces algorithmes.

2 Méthodes de factorisations en facteurs premiers

On se place dans le contexte suivant: on dispose d'un entier n qui est composite pour sur, et qui est, également, produit de deux facteurs premiers $p, q \in \mathcal{P}$ ($n = p \cdot q$). On ne connait pas ses entiers premiers.

2.1 Méthode naïve

Une méthode naïve pour obtenir de tels facteurs serait de tester tous les diviseurs de n non-triviaux afin d'obtenir un facteur premier sur les deux, puis les deux.

Proposition 1. *La méthode naïve est* ***très*** *mauvaise, en effet, il va falloir tester* $\mathcal{O}(\sqrt{n})$ *entiers.*

C'est cependant une méthode très facile à implémenter.

2.2 Méthode du Rho de Pollard

La méthode du Rho de Pollard améliore **considérablement** la recherche de facteurs.

On va considérer $(X_n)_{n\in\mathbb{N}}$ une famille de vad indépendantes dans $[\![1, n]\!]$ sur un espace probabilisé adapté $(\Omega, \mathcal{A}, P)$. Cette famille portera la propriété suivante: chaque X_i sera uniforme modulo p, avec p le plus petit facteur de n.

Il va exister un entier i qui vérifie la propriété suivante:

$$\exists j \in [\![1, i-1]\!] \text{ tel que } \overline{X_i(\omega)} = \overline{X_j(\omega)} \text{ dans } \mathbb{Z}/p\mathbb{Z} \tag{1}$$

pour un certain $\omega \in \Omega$.

Définition 1 (Vad L)**.** *On définit la vad L sur* $(\Omega, \mathcal{A}, P)$ *comme la valeur du petit entier qui vérifie (1).*

Proposition 2 (Majoration de la loi de L)**.** *Soit* $i \in \mathbb{N}\backslash\{0, 1\}$ *La probabilité* $P(L \geq i) \leq e^{-\frac{i^2}{2p}}$ *avec* $i \ll n$.

Proposition 3 (Majoration de l'espérance)**.** $\mathbf{E}(L) = \mathcal{O}(p^{1/2})$.

Définition/Proposition 1 (Définition d'une suite particulière)**.** *On définit $f : x \mapsto x^2 + 1 \ mod \ n$. Puis, on pose $(x_{n\in\mathbb{N}^*})$ de la manière suivante:*

- $x_1 \in [\![0, n-1]\!]$
- $\forall n \in \mathbb{N} \quad x_{n+1} = f(x_n) \ mod \ n$

On va considérer que $(x_{n\in\mathbb{N}^*})$ est définit pseudo-périodique grâce à f, et que chaque "tirage" de x_n se fait uniformément modulo p.

Proposition 4 (Principe théorique de l'algorithme du Lièvre et de la Tortue)**.** *Si, $\exists i, j \in \mathbb{N}^* \ i > j$, tel que: $x_i = x_j$, alors, $\exists k \in \mathbb{N}^*$ tel que $x_{2k} = x_k$*

2.2.1 Principe de l'algorithme

Le principe de l'algorithme est le suivant: On cherche des cycles parmi $(x_{n\in\mathbb{N}^*})$, de manière à avoir la propriété (1).

Proposition 5 (Principe de l'algorithme du Rho de Pollard)**.** *Si il existe (i, j) qui vérifie (1), alors $pgcd(x_i, x_j)$ peut être un facteur non-trivial de n*

Avant toutes choses, on va explicité l'algorithme du Lièvre et de la Tortue:

Algorithme 1 Algorithme du Lièvre et de la Tortue

1: Soit $(u_{n\in\mathbb{N}})$ une suite quelconque à valeur dans un ensemble fini
2: On pose k un indice
3: **Tant que** $u_k \neq u_{2k}$ **faire**
4: On incrémente k
5: **Fin Tant que**
6: Renvoyer k

Cet algorithme calcule le cycle explicité par la propositon plus haut. On prouve dès lors la correction de l'algorithme facilement.

Proposition 6 (Terminaison de l'algorithme)**.** *L'algorithme du Lièvre et de la Tortue s'arrête.*

L'algorithme du Rho de Pollard s'écrit dès lors de la façon suivante:

Algorithme 2 Algorithme du Rho de Pollard

1: On pose x_0 un entier pris aléatoirement dans $[\![0, n-1]\!]$
2: On pose k un indice ≥ 1
3: **Tant que** si $pgcd(|x_k, x_{2k}|) \in \{1, n\}$ **faire**
4: On calcule x_{k+1} et x_{2k+2}
5: On incrémente k
6: **Fin Tant que**
7: Renvoyer $pgcd(|x_k, x_{2k}|)$

On reconnait ici la condition de l'algorithme du Lièvre et de la Tortue ($pgcd(x_k, x_{2k}) \in \{1, n\}$) mais où l'on fait l'égalité modulo p.

Proposition 7 (Complexité moyenne de l'algorithme du Rho de Pollard)**.** *L'algorithme du Rho de Pollard a complexité moyenne en $\mathcal{O}(\sqrt{p}) = \mathcal{O}(n^{1/4})$.*

Proposition 8 (Subtilité sur le choix de x_0)**.** *Il s'avère probable lors de lancement de différentes instances de l'algorithme que celles-ci bouclent à l'infini, ce qui prouve que l'algorithme ne se termine pas forcément !*

Bien souvent, prendre une nouvelle valeur de x_0 règle le problème, et cela n'arrive quasiment pas sur les grands entiers.

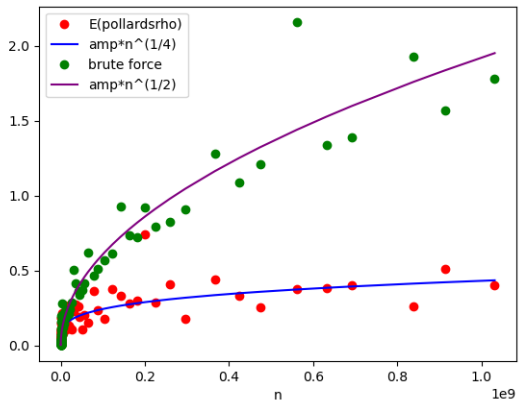

Figure 1: Temps mis par l'algortihme du Rho de Pollard et l'algorithme naïf

2.3 La méthode du crible quadratique

Comparé au Rho de Pollard, la méthode du crible quadratique est plus sophistiquée, et nécessite une implémentation plus difficile que celle de l'autre algorithme.

Le but de cette méthode est de trouver des entiers u et v satisfaisant la propriété suivante:

$$u^2 \equiv v^2 \ [n] \text{ et } u \not\equiv v \ [n] \tag{2}$$

Proposition 9. *Si (2) est vérifiée, on a une chance non-négligeable d'avoir* $pgcd(u-v,n)$ *ou* $pgcd(u+v,n)$ *qui est non-trivial, donc qui représente un facteur de* n*.*

On a donc un nouveau moyen de trouver des facteurs. Cependant, trouver des entiers comme cela n'est pas mince affaire.
Pour essayer de trouver de pareils entiers, il nous faut développer les entiers lisses:

Définition 2 (Entier B-friable ou B-lisse)**.** *Un entier* $n = p_1^{\alpha_1} \dots p_k^{\alpha_k}$ *est* B*-friable (ou* B*-lisse) si:*

$$\forall i \in [\![1,k]\!] \quad p_i \leq B$$

On donne également la définition suivante de polynômes qui seront d'une grande importance dans la théorie développée ci-après:

Définition 3 (Polynôme de Kraitchik)**.** $Q(X) = X^2 - n \in \mathbb{Z}[X]$ *est un polynôme que nous appellerons polynôme de Kraitchik.*

2.3.1 Méthode du crible (phase de collecte)

On va se donner ici une borne M, que nous développerons plus en détail plus tard.

On va ici prendre les $M+1$ entiers de l'intervalle: $I = [\![\lceil\sqrt{n}\rceil \lceil\sqrt{n}\rceil + M]\!]$, que l'on notera $x_0, \dots, x_M$. On va filtrer (pour des raisons que nous expliciterons plus tard) les $Q(x_i)$ qui sont B-friable. Pour règler ce problème, on pourrait le faire de façon naïve, et tester tous les premiers $\leq B$ avec tous les entiers x_i.

On voit bien que cette méthode va être vite limitante, surtout que M, et B dépendront de n, et seront assez grands.

Une autre méthode consisterait en la mise en place d'un crible, on va mettre en place un tableau T qui va associer pour chaque $i = 0, \dots, M$ leur $Q(x_i)$ correspondant. Dès lors que l'on repère un entier

$Q(x_i)$ étant divisible par un des nombres premiers de la base de facteur (entiers inférieurs à B), on va pouvoir "cascader" la divisibilité grâce à la propriété suivante:

Proposition 10 (Itérés d'un $Q(x_i)$ divisible)**.** *On suppose que $Q(x_i)$ est divisible par un certain entier premier p. Alors, $\forall j \in \mathbb{N}^*$ avec $j \cdot p + x_i$ dans I, $Q(x_i + j \cdot p) \equiv 0\ [p]$.*

Si un $Q(x_i)$ a pour facteur une puissance d'un premier, on répétera l'opération sur de la même manière sur p, autant de fois que possible (voir algorithme 3 ci-dessous).

Les entiers B-friable resultant seront les entiers d'indice i tel que $T[i] = 1$. On notera par la suite I l'indices des entiers tels que $Q(x_i)$ est B-friable.

Proposition 11 (Complexité du crible)**.** *Le crible effectue en tout $\mathcal{O}(Mln(ln(B)))$ divisions (ici M et B dépendent de n).*

On admettra cette propriété.

On voit que cette complexité est nettement meilleure que celle de la méthode naïve (qui serait ici en $\mathcal{O}(MB)$.

Algorithme 3 Algorithme du crible pour filtrer les $Q(x_i)$

```
1:  Pour i = 0, ..., M faire
2:      Si T[i] = 1 alors
3:          Aller à la prochaine itération
4:      Fin Si
5:      Pour p premier inférieur à B faire
6:          Tant que p | T[i] faire
7:              T[i] = T[i]/p
8:              k = 1
9:              Tant que i + k·p ≤ M faire
10:                 Si p | T[i + k·p] alors
11:                     T[i + k·p] = T[i + k·p]/p
12:                 Fin Si
13:                 On incrémente k
14:             Fin Tant que
15:         Fin Tant que
16:     Fin Pour
17: Fin Pour
```

Cependant, on a pas encore trouver nos entiers u et v qui vérifient (2).

On va pouvoir montrer la propriété suivante:

Proposition 12 (Existence probable d'un carré)**.** *Si on parvient à trouver $J \subset [\![0, M]\!]$ tel que $\prod_{j \in J} Q(x_j)$ est un carré, alors on va pouvoir construire des entiers qui ont une chance de vérifier (6).*

Pour identifier une telle famille, on va utiliser va devoir faire appelle à l'algèbre linéaire.

2.3.2 Résolution matricielle pour obtenir un carré modulo n (ou phase d'exploitation)

On dénote par π la fonction indicatrice des nombres premiers.

Proposition 13 (Propriété importante)**.** *On suppose que l'on a k et B deux entiers tels que: $k \geq \pi(B)$. On a également k entiers $m_1, \ldots, m_k$ B-friable.*
Il va exister $J \subset [\![1, k]\!]$ tel que $\prod_{j \in J} m_i$ est un carré.

Pour notre cas, on va utiliser $(Q(x_i))_{i\in I}$ comme famille d'entiers B-friables.

On s'assurera plus tard qu'une certaine borne M nous permettra d'avoir $Card(I) \geq \pi(B) + 1$.
Pour obtenir donc une combinaison linéaire de lignes nulle, on va devoir déterminer le noyau de la matrice transposée (cf. preuve de la prop 13). Pour cela, on peut utiliser une méthode simple mais efficace: le pivot de Gauss.
Une fois la base du noyau déterminée, pour chaque vecteur de celle-ci, on va définir J comme l'ensemble des composantes ayant un 1 sur ce vecteur. Et ainsi, on va déterminer $\dim\ker A$ candidats possibles pour être des carrés, en vertu de la proposition 12.

2.3.3 Analyse de complexité de l'algorithme

On sait d'après la proposition 11, que la complexité du crible est en $\mathcal{O}(Mln(ln(B)))$. Cependant, nous n'avons pas encore d'expression donnant une borne fiable en fonction de n, ni même pour B...

Nous allons donner une heuristique pour B et M afin d'avoir un algorithme le plus optimal

Proposition 14 (Heuristique sur B et M)**.** *Si on pose $B = L(n)^{1/\sqrt{2}}$, alors on va pouvoir approximativement garantir que, au bout de $L(n)^{\sqrt{2}}$ test de nombres, on va avoir plus de $\pi(B)$ nombres B-friable. On pourra notamment poser $M = L(n)^{\sqrt{2}}$*

On admettra cette proposition.

Finalement, on peut, avec ces heuristiques, avoir un algorithme du crible qui s'exécute en $\mathcal{O}\left(L(n)^{\sqrt{2}}\right)$.

3 Implémentation sur les grands entiers

Les algorithmes que l'on vient d'expliciter sont utiles comme on l'a dit pour déchiffrer l'algorithme de chiffrement RSA, hors, à l'heure actuelle, les entiers utilisés comme clé publique sont de l'ordre de 1024 bits. Or, la plupart des ordinateurs personnels ne sont capables de gérer que 64 bits. On va voir que l'on peut créer ce type d'entiers, mais avec un coup temporel...

On va définir l'implémentation de tels entiers: on génére un tableau de taille fixe n pointant vers un entier de 32 bits, on aura alors un entier de $32n$ bits au final.

3.1 Addition et soustraction

L'addition de tels entier se fait indice par indice. Il faut s'assurer cependant de bien ajouter la retenue à chaque indice de rang supérieur. Or, le langage de programmation C dans lequel est réalisé ce projet ne permet pas de tel connaissance sur la retenue, qui est habituellement renseignée comme "flag" dans un des registres du CPU.
Pour cela, nous utliserons des "mask" permettant de nous renseigner sur cette retenue:

(a & 0x80000000 && b & 0x80000000) $\|(!(a+b+c\&0x80000000)\&\&(a\&0x80000000\|\|b\&0x80000000))$

avec a, b les entiers que l'ont ajoute entre eux, et c la valeur de la retenue précèdente (0 ou 1).
Pour la soustraction, il suffit de faire le complément à deux, pour donner le négatif dans certain entier, et ajouter les deux nombres une fois cela fait.
Dans les deux cas, on se retrouve ici avec une complexité en $\mathcal{O}(n)$ avec la taille du tableau précédemment explicitée.

3.2 Multiplication

Pour multiplier deux entiers entre eux, on va se servir du shift. Le shift est une opération qui va décaler d'un certain nombre les bits de chaque entier du tableau, soit à gauche, soit à droite.
On va représenter nos entiers que l'on veut multiplier de la façon suivante, en base 2^{32}:

$$a = \sum_{i=1}^{n} a_i \cdot 2^{32i} \; et \; b = \sum_{i=1}^{n} b_i \cdot 2^{32i}$$

L'addition des deux devrait nous donner:

$$a \cdot b = \sum_{i=1}^{n} \sum_{j=1}^{n} a_i \cdot b_j \cdot 2^{32 \cdot (i+j)}$$

Le shift nous permettra alors de "multiplier" les $a_i \cdot b_j$ par $2^{32(i+j)}$ en décalant de $32(i+j)$ bits vers la gauche.
Cette opération se fait en $\mathcal{O}(n^3)$.

3.3 Division euclidienne

Pour implémenter la division euclidienne, nous considérerons l'algorithme suivant:

Algorithme 4 Division euclidenne

On veut ici $a = b \cdot q + r$

On pose $r = 0$ et $q = 0$ des entiers de $32n$ bits

1: **Pour** $i = 32n - 1$ jusque 0 **faire**
2: On shift r de 1 bit vers la gauche.
3: On met le bit de poids faible en la valeur du $i^{ème}$ bit de a.
4: **Si** $r \geq b$ **alors**
5: $r = r - b$
6: Le $i^{ème}$ bit de q est mis à 1.
7: **Fin Si**
8: **Fin Pour**=0

L'algorithme suivant s'exécute en $\mathcal{O}(32n)$, clairement meilleur qu'un algorithme naïf qui aurait pu s'exécuter en $\mathcal{O}(q)$.

4 Conclusion

On a vu que le problème de factorisation d'un nombre en ses facteurs premiers est un problème complexe à résoudre pour $n = pq$. Certaines méthodes, comme le Rho de Pollard sera plus efficace en terme de complexité spatial, ou le crible sera meilleur en temps pour de grands entiers. Il y a cependant de meilleurs algorithmes, comme le crible algébrique, ou des méthodes sur les courbes elliptiques, et également des améliorations sur les implémentation qui peuvent grandement améliorer le temps d'exécution.

5 Annexe

Preuve (proposition 2): Soit $i \in \mathbb{N} \backslash \{0, 1\}$,

$$\begin{aligned} P(L \geq i) &= \frac{(p - (i-1)) \dots (p-1)}{p} \\ &= \left(1 - \frac{1}{p}\right) \dots \left(1 - \frac{i-1}{p}\right) \end{aligned} \tag{3}$$

car ici $(X_{j \in [\![1, i-1]\!]})$ sont uniformes modulo p.

Puis,

$$
\begin{aligned}
P(L \geq i) &= \left(1 - \frac{1}{p}\right) \dots \left(1 - \frac{i-1}{p}\right) \\
&\leq exp\left(-\frac{1}{p}\right) \dots exp\left(-\frac{i-1}{p}\right) \\
&\leq exp\left(-\frac{i(i-1)}{2p}\right)
\end{aligned} \tag{4}
$$

Or, ici, $i \ll n$, on peut donc considérer que: $exp\left(-\frac{i(i-1)}{2p}\right) \approx exp\left(-\frac{i^2}{2p}\right)$
D'où le résultat attendu. □

Preuve (proposition 3): Premièrement,

$$
\begin{aligned}
P(L \geq i) &\leq exp\left(-\frac{i^2}{2p}\right) \\
&\leq \int_{i-1}^{i} exp\left(-\frac{t^2}{2p}\right) dt
\end{aligned} \tag{5}
$$

Puis,

$$
\begin{aligned}
\mathbf{E}(L) &= \sum_{i=3}^{\infty} P(L \geq i) \\
&\leq \sum_{i=3}^{\infty} exp\left(-\frac{i^2}{2p}\right) \\
&\leq \sum_{i=1}^{\infty} exp\left(-\frac{i^2}{2p}\right) \\
&\leq \int_{0}^{\infty} exp\left(-\frac{t^2}{2p}\right) dt \\
&\leq \sqrt{2p} \int_{0}^{\infty} e^{-u^2} du \\
&\leq \sqrt{\frac{\pi p}{2}}
\end{aligned} \tag{6}
$$

Donc, finalement, $\mathbf{E}(L) = \mathcal{O}(\sqrt{p})$ □

Preuve (proposition 4): On pose $T = i - j$ la période de la suite, puis $x_{i+s} = x_{j+\lambda T+s}$, avec $\lambda \geq 0$ et $s \geq 0$.

On pose $k = i + s$ et on cherche s et λ tel que $2k = i + \lambda T + s$. Ceci admet un couple de solutions, on prend s et λ les plus petits possible. □

Preuve (proposition 5): D'après (1), on va avoir dès lors une couple (i, j) tel que $p \mid x_i - x_j$, puis $p \mid n$, donc on peut se retrouver avec $pgcd(x_i, x_j) = p$. □

Preuve (proposition 6): En effet, l'algorithme prend ses valeurs dans un ensemble fini, d'après le principe des tiroirs: on devrait trouver deux indices tel que $u_i = u_j$ et dès lors on prouve la terminaison d'après la proposition du principe théorique de l'algorithme du Lièvre et de la Tortue. □

Preuve (proposition 7): On a considéré que la famille $(x_{n\in\mathbb{N}^*})$ était calculer de manière pseudo-aléatoire. On va également assumer ici qu'ils sont tous uniformes lors de leur tirage de sorte à ce que l'on fasse le parallèle à la suite $(X_n)_{n\in\mathbb{N}}$ dont les membres suivent une loi uniforme modulo p.

D'après la proposition plus haut, on a $\mathbf{E}(L) = \mathcal{O}(\sqrt{p})$, or ici L est lié au nombre de fois que l'on va itérer dans la boucle.

En effet, L donne directement l'information du premier candidat qui satisfait (1), ici, grâce au principe du Lièvre et de la Tortue, la différence entre (i, j) et $(2k, k)$ est constante, ce qui ne viendra pas modifié la borne de L.

Finalement, cet algorithme de Las Vegas s'exécute en moyenne en $\mathcal{O}(n^{1/4})$. □

Preuve (proposition 8): Il n'est pas imposible de tirer un x_0 tel que le cycle calculé par l'algorithme donne un *pgcd* égal à n ! En effet, si $x_i - x_j = 0$ par exemple, on a dès lors $pgcd(|x_i, x_j|, n) = n$. □

Preuve (proposition 9): On a:

$$u^2 \equiv v^2 \ [n] \Longleftrightarrow (u - v)(u + v) \ [n]$$

□

Preuve (proposition 10): On sait que $Q(x_i) \equiv 0 \ [p]$, or $Q(x_i + kp) = (x_i + kp) - n = x_i^2 + 2x_ikp + k^2p^2 - n \equiv 0 \ [p]$. □

Preuve (proposition 12): On pose $u = \prod_{i\in I} x_i$ et $\prod_{i\in I} Q(x_i) = v^2$ par hypothèse.

On a donc,

$$\overline{v^2} = \prod_{j\in J} \overline{Q(x_j)} = \prod_{j\in J} \overline{x_j^2} = \overline{u^2}$$

On alors une chance sur deux d'avoir (6), car on est pas assurer d'avoir $v \not\equiv u \ [n]$ □

Preuve (proposition 13): Soit $p_1, \ldots, p_{\pi(B)}$ les $\pi(B)$ premiers $\leq B$.

$$\forall i \in [\![1, j]\!], \ \forall j \in [\![1, \pi(B)]\!], \ \exists \alpha_{i,j} \in \mathbb{N} \mid m_i = \prod_{j=1}^{\pi(B)} p_j^{\alpha_{i,j}}$$

On pose $A = (\overline{\alpha_{i,j}})_{i,j} \in M_{k,\pi(B)}(\mathbb{F}_2)$.
On a, $rg(A) \leq min(k, \pi(B)) = \pi(B)$, donc forcément les lignes seront liées, ce qui amène à avoir $\epsilon_1, \ldots, \epsilon_k \in \mathbb{F}_2^k$ tel que $\epsilon_1 L_1 + \cdots + \epsilon_k L_k = 0$

Puis, on pose I l'ensemble des indices de ϵ_i non nul.
On obtient donc: $\forall j \in [\![1, \pi(B)]\!] \ \sum_{i\in I} \alpha_{i,j}$ pair.
Et finalement:

$\prod_{i\in I} m_i = \prod_{j=1}^{\pi(B)} p_j^{\sum_{i\in I} \alpha_{i,j}}$ est un carré. □

Listing 1: Code Python

```
from random import randint
import sys
import time
from math import ceil,sqrt, exp, log
import matplotlib.pyplot as plt

def pgcd(a, b):
        q = a // b
        r = a % b
        if r == 0:
                return b
        return pgcd(b, r)


def pow_mod(b, p, n):
        acc = b % n
        out = 1
        while p != 0:
                if p % 2 == 1:
                        out = out*acc % n
                acc = acc*acc % n
                p = p // 2
        return out


def fermat_test(n, k):
        for _ in range(k):
                a = randint(2, n-1)

                if pow_mod(a, n-1, n) != 1:
                        return False
        return True


def pollardsrho(n, f):
        x = randint(0, n)
        y = x

        #print("choix: "+str(x))

        while pgcd(y-x, n) == 1 or pgcd(y-x, n) == n:
                x = f(x, n) % n
                y = f(f(y, n) % n, n) % n

        return pgcd(y-x, n)


def generate_prime(size_max, size_min, k):
        n = randint(int(size_min), int(size_max))

        if n % 2 == 0:
                n+= 1

        while not fermat_test(n, k):
                n+= 2
```

```
        return n


def pollardsrho_time(n, f):
        before = time.time()
        pollardsrho(n, f)
        after = time.time()
        return (after-before)*1000


def brute_force(n):
        for i in range(2, int(sqrt(n))):
                if n % i == 0:
                        return i

        return n


def brute_force_time(n):
        before = time.time()
        brute_force(n)
        after = time.time()
        return (after-before)*1000


def is_quadratic_residue(p, n):
    if p == 2:
        return True

    return pow_mod(n, (p-1)/2, p) == 1


def factor_base(B, n):
    T = [2]

    for i in range(3, B+1):
        if fermat_test(i, 10) and is_quadratic_residue(i, n):
            T.append(i)
    return T


def crible(B, X):
    T = [i for i in range(X+1)]

    for j in range(2, X+1):
        p = T[j]
        if p > B:
            continue

        T[j] = 1

        i = 1

        while j+i*p <= X:
            if T[j+i*p] % p == 0:
                T[j+i*p]//= p
```

```
            i+= 1

    return T


def quadratic_sieve(B, M, n, FB):
    Q = lambda x: x**2 - n

    T = [Q(ceil(sqrt(n)) + i) for i in range(M+1)]

    for i in range(M+1):
        if T[i] == 1:
            continue

        for p in FB:
            while T[i] % p == 0 and T[i] != 0:
                T[i]//= p

                k = 1

                while i + k*p <= M:
                    if T[i+k*p] % p == 0:
                        T[i+k*p]//= p
                    k+= 1

    return T


def filter_sieve(B, M, n, FB):
    T = quadratic_sieve(B, M, n, FB)

    a = ceil(sqrt(n))

    Q = lambda x: x**2 - n

    res = []

    for i in range(M+1):
        if T[i] == 1:
            res.append((a+i, Q(a+i)))

    return res


def add_list(l1, l2):
    res = []
    for i in range(len(l1)):
        res.append((l1[i] + l2[i]) % 2)
    return res

def empty_list(l):
    for i in range(len(l)):
        if l[i] != 0:
            return False
    return True
```

```
def calc_ker(T, B, n):
    FB = factor_base(B, n)

    A = []

    for i in range(len(T)):
        line = []

        for j in range(len(FB)):
            a = 0
            _, k = T[i]

            while k % FB[j] == 0:
                a+= 1
                k//= FB[j]

            line.append(a % 2)
        A.append(line)


    nc = len(FB)
    nl = len(A)

    op_mat = []

    for i in range(nl):
        op_mat.append([0]*i+[1]+[0]*(nl-i-1))

    for j in range(nc):
        i = j

        while i < nl and A[i][j] == 0:
            i+= 1

        if i == nl:
            continue

        A[i], A[j] = A[j], A[i]
        op_mat[i], op_mat[j] = op_mat[j], op_mat[i]

        for k in range(i+1, nl):
            if A[k][j] == 0:
                continue
            for q in range(j, nc):
                A[k][q] = (A[k][q] + A[j][q]) % 2
            op_mat[k] = add_list(op_mat[k], op_mat[j])

    ker = []

    for i in range(len(A)):
        if empty_list(A[i]):
            ker.append(op_mat[i])

    return ker
```

```
def L(n):
    return exp(sqrt(log(n)*log(log(n))))


def calc_factor(n, FB):

    B = ceil((L(n))**(1/sqrt(2)))

    M = ceil((L(n))**(sqrt(2)))

    T = filter_sieve(B, M, n, FB)

    ker = calc_ker(T, B, n)

    FB = factor_base(B, n)

    s1 = []
    s2 = []

    for i in range(len(ker)):
        u = 1
        v = 1

        for j in range(len(ker[i])):
            if ker[i][j]:
                u*= T[j][1]
                v*= T[j][0]

        for p in FB:
            if u % p == 0:
                u//= p

        s1.append(u)
        s2.append(v)

    print(s1)
    print(s2)

    for i in range(len(s1)):
       print((s1[i]**2) % n)
       print((s2[i]**2) % n)

    candidates = []

    for i in range(len(s1)):
        candidates.append(pgcd(n, s2[i]-s1[i]))
        candidates.append(pgcd(n, s1[i]+s2[i]))

    return candidates


def crible_time(n):
    FB = factor_base(ceil((L(n))**(1/sqrt(2))), n)
    before = time.time()
    calc_factor(n, FB)
```

```
    after = time.time()
    return (after-before)*1000
```

References

[1] Donald E. Knuth : The Art of Computer Programming, vol. II: Seminumerical Algorithms. 1997

[2] J. M. Pollard : A monte carlo method for factorization. 1975

[3] J. Hoffstein, J. Pipher, J. H. Silverman : An introduction to Mathematical Cryptography. 2008