Université Paris Diderot
Double Licence Math-Info

Projet de Mathématiques-Informatique

Présenté par :
LEMOINNE Marianne et VOVARD hugo

Factorisation des entiers

Des divisions successives au crible quadratique, les outils que nous donnes l'algèbre pour factoriser des entiers de l'ordre de 10^{50}

Projet encadré par :
BRUNAT Oliviet

Contents

I. Introduction

Du code césar à la machine Enigma, nombreuses ont été les innovations de l'Homme pour rendre illisible ses messages aux non averties. Aujourd'hui, à l'ère de l'information et des intelligences artificielles, il devient quasiment impossible de construire un système d'encodage qui ne risque pas d'être craqué rapidement. Pourtant, un système semble rester efficace depuis sa création en 1983, l'algorithme inventé par Ron Rivest, Adi Shamir et Léonard Adleman en 1977 surnommé depuis le RSA. Son principe semble pourtant simple:

Une personne créé une paire de clefs, l'une qu'il garde secrète (on la dit clefs privée) et la seconde qu'il rend publique (on la dit clef publique). Si une personne souhaite envoyer un message au créateur des clefs, il lui suffit d'encoder son message grâce à la clef publique. Ainsi, seul le détenteur de la clef privée sera capable de décoder le message.

Le RSA repose sur le principe de chiffrement asymétrique, c'est à dire qu'il n'est pas possible de retrouver la clef privée à partir de la clef publique. Or tout cela repose sur l'incapacité actuelle des scientifiques à factoriser un entier grand (de l'ordre de 10^{50}) de façons efficace.

Dans ce projet nous allons étudier différentes méthodes pour factoriser un entier, nous verrons les différences de complexité des algorithmes qui en découlent en nous intéressant en particulier au crible quadratique.

A. n **est il premier ?**

Commençons par nous poser les bonnes questions. Soit $n \leq 2$ un nombre entier que l'on souhaite factoriser. Il faut alors se demander :

1. Est ce que n est un nombre premier ou une puissance d'un nombre premier ? Dans ce cas la factorisation est évidente.
2. Peut on trouver $2 \leq d \leq n$ tel que d divise n?

Rappelons quelques définitions et quelques méthodes qui nous serons utiles pour la suite.

Définition : Soit n un entier naturel, n est un nombre premier si et seulement si il admet exactement deux diviseurs, 1 et lui même.

Pour implémenter les programmes de factorisation, nous aurons besoin de tester si un nombre est premier. Pour cela, nous avons implémentez la méthode "classique" des divisions successives:

```
#méthode des divisions succéssives
def div_suc (x):
    global petitspremiers
    if x<=1024:
        if x in petitspremiers:
            return True
        else:
            return False
    for i in range (2, math.trunc(math.sqrt(x))+1):
        if (x%i==0):
            return False
    return True
```

Cette méthode n'étant pas la plus efficace, nous avons récupéré le code de la méthode probabiliste de Miller Rabin (cf bibliographie):

```python
def millerRabin(n, k=20):
    """Test de primalité probabiliste de Miller-Rabin"""
    global petitspremiers

    # éliminer le cas des petits nombres <=1024
    if n<=1024:
        if n in petitspremiers:
            return True
        else:
            return False

    # éliminer le cas des nombres pairs qui ne peuvent
    #pas être 1ers!
    if n & 1 == 0:
        return False

    # recommencer le test k fois: seul les nb ayant
    #réussi k fois seront True
    for repete in range(0, k):
        # trouver un nombre au hasard entre 1 et n-1
        #(bornes inclues)
        a = random.randint(1, n-1)
        # si le test echoue une seule fois => n est composé
        if not _millerRabin(a, n):
            return False
    # n a réussi les k tests => il est probablement 1er
    return True
```

```python
#  Test de primalité probabiliste de Miller-Rabin issue de
#http://python.jpvweb.com/python/mesrecettespython/doku.php
#?id=est_premier
def _millerRabin(a, n):
    """Ne pas appeler directement (fonction utilitaire).
    Appeler millerRabin(n, k=20)"""
    # trouver s et d pour transformer n-1 en (2**s)*d
    d = n - 1
    s = 0
    while d % 2 == 0:
        d >>= 1
        s += 1

    # calculer l'exponentiation modulaire (a**d)%n
    apow = lpowmod(a,d,n) # =(a**d)%n

    # si (a**d) % n ==1 => n est probablement 1er
    if apow == 1:
        return True

    for r in range(0,s):
        # si a**(d*(2**r)) % n == (n-1) => n est
        #probablement 1er
        if lpowmod(a,d,n) == n-1:
            return True
        d *= 2

    return False
```

Pour la suite de ce projet nous prendrons n tel que n n'est pas divisible par des petits nombre premiers (comme 2, 3, 5, 7, 11, 13..) plus précisément nous prendrons n impair. Pour arriver à cela en pratique, dans notre code, nous vérifierons avant toute chose si n n'a pas de diviseur présent dans un tableau petitspremiers, qui contient les nombres premiers inférieurs à 1021.

B. La méthode des divisions successives

La méthode la plus simple pour factoriser un nombre est la méthode des divisions successives, cette méthode consiste à diviser n (le nombre dont on cherche la factorisation) successivement par tous les nombres entiers successifs jusqu'à trouver les nombres premiers qui divisent n.
Ci dessous le code de la fonction, ainsi que quelques exemples de factorisation par cette fonction.

```python
#Retourne la factorisation de n avec la méthode des
#divisions succéssives
def div_suc_fact (n):
    global petitspremiers
    if (millerRabin(n)):
    #si n est premier, on renvoie n
        return [n]
    else:
        for i in petitspremiers:
        #On commence par tester si de petits entiers
        #divisent n (entre 2 et 1021)
            if(n%i==0):
                return div_suc_fact(int(n/i))+[i]
        for i in range (1021, math.trunc(sqrt(n))) :
        #Si l'on n'a pas trouvé les facteurs dans les
        #petits permiers, on vérifie les entiers de
        #1021 à sqrt(n)
            if(n%i==0):
                return div_suc_fact(int(n/i))+[i]
```

```
Avec la méthode des DIVISIONS SUCCESSIVES

La méthode à mis 2.9000000000001247e-05 pour trouver que :
2041 = 157*13
La méthode à mis 0.08388399999999996 pour trouver que :
34624234323236231 = 48695893*1002863*709
La méthode à mis 0.005249999999999977 pour trouver que :
967140655691703339649407 = 7078369*59377*191*43*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*167
La méthode à mis 84.387818 pour trouver que :
1099998619700431613 = 1099999349*999999337
La méthode à mis 0.0009890000000041255 pour trouver que :
140737488355327 = 13264529*4513*2351
La méthode à mis 0.0003020000000490945 pour trouver que :
4333801 = 6761*641
La méthode à mis 0.3953810000000004 pour trouver que :
39335476910299 = 8251183*4767253
```

Cette méthode n'est pas du tout optimiser pour les très grand nombre non divisible par les premiers entiers. Par exemple on peux voir que pour obtenir la factorisation de 1099998619700431613 le programme a mis environ 85 seconde.

II. La méthode de Fermat

Au XVII siecle, Pierre de Fermat, mathématicien français met au point un algorithme afin de décomposer un nombre entier impair non premier.

La méthode de Fermat consiste à écrire n (le nombre dont on cherche la factorisation) comme une différence de carré parfait pour pouvoir le factoriser grâce à l'identité remarquable : $(a+b)(a-b) = a^2 - b^2$ cette méthode repose sur le Lemme suivant :

L'ensemble des couples $(a, b) \in \mathbb{N}^2$ tels que $n = ab$ avec $a \leq b$, et celui des couples $(r, s) \in \mathbb{N}^2$ tels que $n = r^2 - s^2$,sont en bijection.

Proof. Soit $A = \{(a, b) \in \mathbb{N}^2 \mid n = ab \text{ et } a \geq b\}$ et $B = \{(r, s) \in \mathbb{N}^2 \mid n = r^2 - s^2\}$. Montrons alors que les applications défini ci dessous sont réciproque l'une de l'autre :

$$\begin{aligned} f : A &\to B & g : B &\to A \\ (a, b) &\to (\frac{a+b}{2}, \frac{a-b}{2}) & (r, s) &\to (r+s, r-s) \end{aligned}$$

On a alors $(\frac{a+b}{2})^2 - (\frac{a-b}{2})^2 = ab = n$
et $(r+s)(r-s) = r^2 - s^2 = n$ avec $r+s \geq r-s$

On a donc bien f et g réciproque l'une de l'autre et donc A et B en bijection.

□

On applique la méthode ainsi:
Supposons que l'on ai $n = ab$, posons alors:

$$r = \frac{a+b}{2} \qquad s = \frac{a-b}{2}$$

On a donc $n = r^2 - s^2$ autrement dis $r^2 - n = s^2$. Plus les entiers a et b sont proche, plus s est un petit nombre et donc plus r est proche de $\sqrt{n}$ tout en lui étant supérieur. On prend alors $r = \lfloor \sqrt{n} \rfloor + u$ avec $u \in \mathbb{N}$ et on cherche u tel que $r^2 - n$ soit un carré parfait.
On peut alors écrire $n = (r+s)(r-s)$. Il s'agit ensuite de vérifier que $r+s$ et $r-s$ sont des nombre premier et si ce n'est pas le cas on peux appliquer à nouveau la méthode de Fermat.

C'est ce que fait l'algorithme suivant :

```
#donne une factorisation en 2 entiers(non forcément premier de n)
def fermat(n):
    r=int(sqrt(n))+1     #On part de sqrt(n)+1
    while (int(sqrt((r*r)-n))!=sqrt((r*r)-n)):
    #tant que r²-n n'est pas un carrée, on rajoute 1 a r
        r=r+1
    s=int(sqrt(r*r-n))
    #s prend la valeur r²-n (qui est un entier de part
    #le while précédent)
    return [r, s]
```

La méthode de Fermat est extrêmement efficace si les facteurs de la factorisation sont proches. En effet plus a et b sont éloignés l'un de l'autre, plus r sera éloigné de $\sqrt{n}$ et plus il faudra d'itération pour trouver une factorisation.

Pour avoir une idée de la complexité de l'algorithme de Fermat, on résonne ainsi : On par de $\sqrt{(n)}$ et l'on itère de 1 tant que l'on a pas trouvé r, i.e $r - \sqrt{n}$ fois :

$$r - \sqrt{n} = \frac{a+b}{2} - \sqrt{n} = \frac{a+b-2\sqrt{ab}}{2} = \frac{(\sqrt{b}-\sqrt{a})^2}{2} = \frac{(\sqrt{n}-a)^2}{2a}$$

Si n est premier, i.e $a = 1$ alors la complexité est de l'ordre $O(n)$, Fermat est donc très peux efficace pour prouver la primalité d'un entier. Mais lorsque a est proche de $\sqrt{n}$ alors la complexité est très faible.

Exemple obtenu grâce à un algorithme qui utilise fermat et vérifie ensuite que les facteurs trouvés son premier et les factorise sinon.

```
Avec la méthode de FERMAT

La méthode à mis 0.00018100000001197714 pour trouver que :
2041 = 13*157
La méthode à mis 0.002724000000000615 pour trouver que :
318857230604106212019172455805819400840087099741220133337 = 564674446565546484548476167*564674446565546484548476511
La méthode à mis 0.37147599999998704 pour trouver que :
140737488355327 = 2351*4513*13264529
La méthode à mis 0.006904000000005794 pour trouver que :
4333801 = 641*6761
La méthode à mis 1.2199679999999944 pour trouver que :
39335476910299 = 4767253*8251183
```

Reprenons l'exemple de 2041:

$$\sqrt{2041} = 45.1$$

On prend regarde donc si $46^2 - 2041$ est un carré parfait, ce n'est pas le cas, on recommence donc avec 47 puis 48... jusqu'à trouver que

$$85^2 - 2041 = 5184$$

qui est un carré parfait : $72^2 = 5184$. L'algorithme renvoi donc

$$2041 = (85 - 72)(85 + 71) = 13 * 157$$

III. De Kraitchik au crible quadratique

A. L'approche de Gauss Kraitchik

Au tout début du XIX siècle, Gauss reprend les idée de Fermat et propose une nouvelle méthode qui sera ensuite remise au goût du jour par Kraitchik au début du XX siècle. Leur idée est de trouver une différence de carré égale à un multiple de n, i.e. deux entiers u et v tel que

$$u^2 \equiv v^2[n] \; et \; u \not\equiv v[n] \tag{1}$$

En effet, dans ce cas on aura que n divise $(u - v)(u + v)$ sans diviser ni $u - v$ ni $u + v$, ainsi les valeurs $pgcd(u - v, n)$ et $pgcd(u + v, n)$ fournissent des diviseurs non triviaux de n.

Il faut donc trouver u et v qui vérifie la condition (1). Pour cela, partons du polynôme de Kraitchik: $Q(X) = X^2 - n \in \mathbb{Z}[X]$ L'idée va être ici de trouver une famille de $(x_i)_{i\in[1,k]}$ tel que le produit des $Q(x_i)$ soit un carrée. On pose ainsi

$$v^2 = Q(x_1) \cdot ... \cdot Q(x_k)$$

$$u = x_1 \cdot ... \cdot x_k$$

Ainsi on obtient

$$u^2 \equiv \prod_{i=1}^{k} x_i^2 [n]$$

$$u^2 \equiv \prod_{i=1}^{k} x_i^2 - n[n]$$

$$u^2 \equiv \prod_{i=1}^{k} Q(x_i)[n] \equiv v^2[n]$$

Toute la problématique va maintenant être de déterminer de tel entier x_i de façons systématique et efficace, pour ainsi pouvoir l'implémenter.

B. La recherche de congruences de carrées

Définition : x est B-friable si tous les diviseurs premiers de x sont (inf ou égale) à B. Savoir si un nombre est B-friable joue un rôle important dans plusieurs méthodes de factorisation. Nous avons donc implémenté le programme si dessous pour obtenir la liste des entiers B-friables compris entre 1 et X :

```
#Retourne un tableau contenant tous les nombres B-friable
#entre 1 et X
def friable(B, X):
    tab=[i for i in range(1,X+1)] #On créé un tableau
    #contenant les
    #entiers    de 1 a X
    for i in range (1, X):
        if(tab[i]!=1 and millerRabin(tab[i]) and tab[i]<=B):
        #Si l'entier est premier, différent de 1 et inférieur
        #à B, alors c'est un facteurs premiers qui nous intéresse
            for j in range(2*i+1, X, i+1):
            #On fait des pats de taille i+1
                tab[j]=math.trunc(tab[j]/tab[i])
                #On divise chaque multiple de tab[i] par tab[i]
            tab[i]=1

    return [p+1 for p in range (0, X-1) if tab[p]==1]
    #On prend les indices du tableau pour lequel la valeur est 1,
    #il s'ait des entiers B-friables
```

Lemmes : Soient k et B des entiers naturels tels que $k \geq \pi(B)+1$. Soient $m_1, ..., m_k$ des entiers naturels B-friables. Il existe une sous-famille non vide des m_i dont le produit est un carré.

Proof. On a m_i B-friable donc les diviseurs premiers de m_i sont inférieur ou égaux à B. Posons p_j le j-ième nombre premier. Pour tout entier i compris entre 1 et k on peut noter la décomposition de m_i en nombres premiers comme ceci:

$$m_i = \prod_{j=1}^{\pi(B)} p_j^{\alpha_{i,j}}$$

Avec $\alpha_{i,j} \geq 0$ et $v(m_i) = (\alpha_{i,1}, ..., \alpha_{i,\pi(B)})$

On peut alors construire la matrice M de taille $(k, \pi(B))$, où l'élément à la place (i, j) correspond à $\alpha_{i,j}$ modulo 2. Le rang de M est donc au plus $\pi(B)$.
On a $k \geq \pi(B) + 1$ donc les vecteurs lignes de M forment un système lié de $F_2^{\pi(B)}$ (F_2-espace vectoriel).
On a donc $i_1, ..., i_t$ tel que $v(m_{i1}), ..., v(m_{it})$ sont tous paires. Donc le produit $m_{i1} \cdot ... \cdot m_{i1}$ est un carré.

□

Voyons maintenant comment déterminer les congruence carré modulo n.
Soit B un entier naturel, on suppose que l'on connais déjà $k \geq \pi(B)+1$ entiers naturels x_i tel que $Q(x_i)$ soit B-friable.
D'après le lemme précédent il existe une sous famille des $Q(x_i)$ dont le produit est un carré.

On peut donc se demander comment mettre en évidence une telle famille.On pose pour tout entier i compris entre 1 et k

$$Q(x_i) = \prod_{j=1}^{\pi(B)} p_j^{\alpha_{i,j}}$$

la décomposition en facteurs premiers de $Q(x_i)$ avec $\alpha_{i,j} \geq 0$
Soit M la matrice de taille $(k, \pi(B))$, où l'élément à la place (i, j) correspond à $\alpha_{i,j}$ modulo 2 et l_i le i-eme vecteur de M. Comme $k \geq \pi(B) + 1$ on a :

$$(\varepsilon_1, ..., \varepsilon_k) \in F_2^k$$

tel que

$$\sum_{i=1}^{k} \varepsilon_i l_i = 0$$

Donc $(\varepsilon_1, ..., \varepsilon_k)$ appartient au noyau de la transposée de M. Soit $I \subseteq \{1, ..., k\}$ l'ensemble des i tels que :

$$\varepsilon_i = 1$$

on a donc

$$\sum_{i \in I} l_i = 0$$

Et donc, pour tout $i \in I, Q(x_i)$ est un nombre premier avec une puissance paire. D'où:

$$\prod_{i \in I} Q(x_i)$$

est un carré.

Il suffi donc de poser :

$$u = \prod_{i \in I} x_i \qquad v^2 = \prod_{i \in I} Q(x_i)$$

On obtient ainsi $u^2 \equiv v^2[n]$.

Voici un exemple de code permettant d'obtenir u et v a partir d'une matrice construite grâce à une liste de $Q(x-i)$ passé en argument:

```

def cree_mat(liste_Q, B):
    tab = get_puissance_decomposition (liste_Q, B)
    for i in range (len(tab)):
        for j in range (len(tab[i])):
            tab[i][j] = tab[i][j]%2
    M = mat(tab)
    return M


```

```
def completer_matrice (M):
    A = mat(zeros(len(array(M)[0])))
    At = transpose(mat(zeros(len(M))))
    while(len(M)<len(array(M)[0])):
        M = concatenate((M, A), axis=0)
    while(len(M)>len(array(M)[0])):
        M = concatenate((M, At), axis=1)
    return M

def extraction_ligne (M):
    M = completer_matrice(M)
    A = array(M)
    M = Matrix(A)
    Mt = M.transpose()
    res = []
    i=0
    while(len(res)<=1):
        X = Mt.nullspace()[i]
        for j in range(len(X)):
            if (int(X[j])%2==1):
                res += [j]
        i+=1
    return res

def get_uv (l, M, n):
    list_x = extraction_ligne(M)
    u = 1
    v = 1
    for i in list_x:
        u = u*l[i]
        v = v*(l[i]**2 - n)
    return [u, get_racine(v)]
```

Reprenons l'exemple de 2041: Prenons $B = 7$ (nous verrons plus loin comment le choisir efficacement). $\pi(7) = 4$, il nous faudrait donc au moins 5 entiers de la forme $Q(x_i)$ 7-friable pour être sur de trouver la famille libre qui forme un carré parfait. Il y a cinq entiers x_i qui vérifie la condition précédente entre 46 et 53:

- $Q(46) = 75 = 3 * 5^2$
- $Q(47) = 168 = 2^3 * 3 * 7$
- $Q(49) = 360 = 2^3 * 3^2 * 5$
- $Q(51) = 560 = 2^4 * 5 * 7$
- $Q(53) = 768 = 2^8 * 3$

En prenant les puissances dans $\mathbb{Z}/2\mathbb{Z}$ on obtient la matrice :

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

On a directement que $l_1 + l_5 = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}$ donc le vecteur $\begin{pmatrix} 1 & 0 & 0 & 0 & 1 \end{pmatrix}$ appartient au noyau de la matrice transposé M^t. On à donc $Q(46) * Q(53) = 2^8 * 3^2 * 5^2$ qui est un carré parfait. Ce qui conduit à

$$(46 * 53)^2 \equiv (2^4 * 3 * 5)^2 [2041] \iff 397^2 \equiv 240^2 [2041]$$

Comme $pgcd(397 - 240, 2041) = 157$, on trouve au final que $2041 = 13 * 157$

Il nous reste maintenant à déterminer comment choisir la constante B de façon optimal et comment trouver les $\pi(B)$ x_i tel que les $Q(x_i)$ soit B-friable.

C. Le crible quadratique

Le crible quadratique en lui même se base sur l'approche de Kraitchik, et la recherche de congruences de carrées. En effet, nous avons vue avec l'approche de Kraitchik que pour trouver la factorisation d'un entier n, il fallait chercher deux entier u et v vérifiant (1). Pour cela, nous sommes passés par le pôlynome de Kraitchik, en posant v^2 comme le produit de $Q(x_i)$ tel que ce produit soit un carrée parfait, et en posant u comme la somme des x_i correspondant. Nous avons ensuite vue comment trouver cette congruence de carrée en posant un problème d'algèbre linéaire relativement simple.

Nous sommes donc maintenant ramené à deux autres problèmes:

1. Comment choisir la constante B utilisé plus haut?
2. Comment trouver $\pi(B)$ entiers x tel que $Q(x)$ soit friable?

Ces deux problèmes sont en faite liée: si l'on choisit une constante B trop petite, il suffira de peu d'entier pour conclure, mais trouver ces entiers ne sera pas choses aisé, puisque l'on réduit le choix des nombres premiers pouvant intervenir dans leurs décompositions. A contrario, choisir un B trop grand garantie de trouver des entiers satisfaisant, mais en trouver un nombre suffisant devient très coûteux en temps.

Nous admettrons que la constante B doit être de l'ordre de

$$exp(\frac{1}{2}\sqrt{log(n)log(log(n))})$$

Il reste maintenant à trouver les $x_i \in I$ tel que $Q(x_i)$ soit B-friable avec

$$I = [\sqrt{n}, \sqrt{n} + A]$$

avec A une constante tel que l'on trouve au moins $\pi(B+1)$ valeurs. Nous admettrons le lemme suivant:
Soit p un nombre premier impaire.

1. Alors $Q(X)$ a exactement 2 racines modulo p.
2. Soit p un nombre premier et a un entier tel que $Q(a) \equiv 0[p^k]$. Alors il existe $b \in [|1; p-1|]$ tel que $2ab \equiv 1[p]$. De plus on a :

$$Q(a + (\sqrt{n} + a^2) * b) \equiv 0[p]$$

L'idée est semblable à celle pour trouver les entiers B-friables entre 1 et X. On construit un tableau avec tous les nombres qui pourrait convenir, puis on crible petit à petit par les premiers inférieurs à B.

Partons d'un tableau T contenant les $Q(x_i)x_i \in I$, i.e. à l'indice i, nous aurons l'entier $(\sqrt{n}+i)^2 - n$. Pour tout premier impaire p inférieur à B, on commence par trouver les 2 racines de $Q(X)$ modulo p, i.e. x_1, x_2 tel que $Q(x_1) \equiv 0[p]$ et $Q(x_2) \equiv 0[p]$. On commence par criblé pour x_1. On va diviser tous les $Q(x_1 + jp)$ $j \in \{0, 1, 2, ...\}$ par p. Puis on fait de même pour x_2. On doit maintenant s'occuper des $Q(X)$ divisible par p^2. Pour cela, on trouve les deux racines b_1 et b_2 de $Q(X)$ modulo p^2 grâce au lemme précédant. On crible pour b_1 et b_2, puis on s'occupe des $Q(X)$ divisible par p^3, et ainsi de suite.
Voici un algorithme qui permet de faire cela:

```
def friable_bis(B, X, n):
    r = get_racine(n)
    T = [int(((r+i)**2-n)) for i in range (1, X)]
    P = [p for p in range (3,B+1) if millerRabin(p)]
    x = 1
    go = True
    puissance = 1

    for p in P :
        racine = find_racine(T, p)
        for i in racine :
            for j in range(i, len(T), p):
                T[j] = int(T[j]/p)
        racine_bis=racine
        while (go):
            racine_bis = find_racine_bis(racine_bis, p, n,puissance)
            puissance += 1
            go = False
            for i in racine_bis :
                for j in range(i, len(T), p**puissance):
                    T[j] = int(T[j]/p)
                    go = True
        go = True
        puissance=1

    p=2

    if( ((r+1)**2-n)%2 == 0 ):
        for i in range (0, len(T), 2):
            T[i] = int(T[i]/2)
    else :
        for i in range (1, len(T), 2):
            T[i] = int(T[i]/2)

    if( n%4 == 3 ):
        return [r+t+1 for t in range (len(T)) if (T[t] == 1)]
    else :
```

```python
        if(n%8==5):
            racine=find_racine(T,2)
            for i in racine :
                for j in range(i, len(T), 4):
                    T[j] = int(T[j]/2)
            return [r+t+1 for t in range (len(T)) if (T[t] == 1)]
        else:
            racine=find_quatre_racine(T, 2)
            for i in racine :
                for j in range(i, len(T), 4):
                   T[j] = int(T[j]/2)

            if( n%8 == 1 ):
                racine = find_quatre_racine(T, 2)
                for i in racine :
                    for j in range(i, len(T), 8):
                        if(T[j] != 1):
                            T[j] = int(T[j]/2)
                puissance =4

                while (go):
                    racine = find_quatre_racine(T, 2)
                    go = False
                    for i in racine :
                        for j in range(i, len(T), 2**puissance):
                            if(T[j]!=1):
                                T[j] = int(T[j]/2)
                            go = True
                    puissance += 1
            go = True
            puissance = 2

    return [r+t+1 for t in range (len(T)) if (T[t] == 1)]
```

Et voici un exemple d'utilisation de l'algorithme:

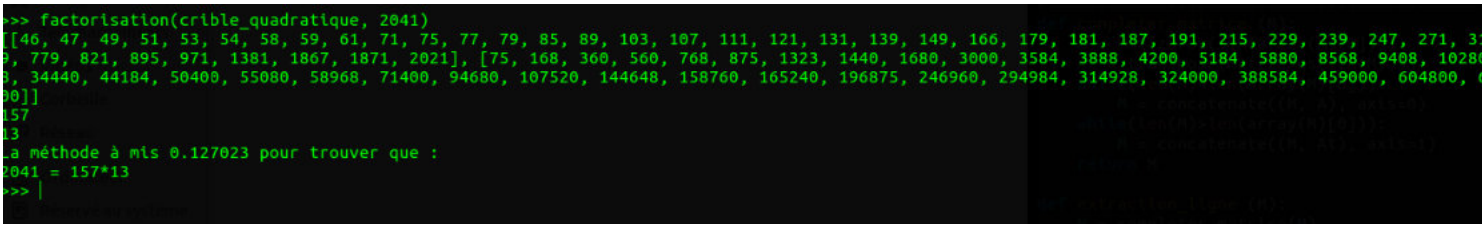
```
>>> friable_bis(7, 2000, 2041)
[46, 47, 49, 51, 53, 54, 58, 59, 61, 71, 75, 77, 79, 85, 89, 103, 107, 111, 121
, 779, 821, 895, 971, 1381, 1867, 1871, 2021]
>>> n = 39335476910299
>>> friable_bis(179, 3*10**5, n)
[6274946, 6277786, 6277808, 6278707, 6280186, 6283157, 6283293, 6283753, 628397
13, 6392977, 6407082, 6419668, 6420749, 6423277, 6448368, 6470993, 6523322, 655
```

Grâce à tous ce que nous avons vue dans ce dossier, nous avons pus esquisser un algorithme effectuant le crible quadratique:

```python
def cree_liste(n, B):
    r = get_racine(n)
    X = 2000
    A = int(B/log(B))
    liste_x = []
    liste_Q = []
    liste_friable = friable_bis(B, X, n)
    X = X*10
    for x in liste_friable:
        liste_Q.append(x**2-n)
        liste_x.append(x)
        A -= 1
    return [liste_x, liste_Q]

def crible_quadratique(n):
    B = int(exp((1/2)*sqrt(log(n)*log(log(n)))))
    listes = cree_liste(n, B)
    print(listes)
    M = cree_mat(listes[1], B)
    uv = get_uv(listes[0], M, n)
    u = uv[0]%n
    v = uv[1]%n
    p = gcd(max(u, v)-min(u, v), n)
    q = int(n/p)
    print(p)
    print(q)
    if(millerRabin(p) and millerRabin(q)):
        return [p, q]
    elif(millerRabin(p)):
        return [p]+crible_quadratique(q)
    elif(millerRabin(q)):
        return crible_quadratique(p)+[q]
    return crible_quadratique(p)+crible_quadratique(q)
```

Cette algorithme n'est pas terminé, il subsiste quelques bug et il n'a pas été optimisé. Néanmoins, il fonctionne:



Conclusion : Durant ce projet nous nous somme rendu compte que certain nombre, malgrès leur taille peuvent être factorisé très rapidement, et cela grâce à des mathématiciens qui ont su mélanger algèbre linéaire et algorithmique. Outre le liens étroit entre mathématiques et informatique, nous avons pus voir dans ce dossier que le chant de l'impossible pour les supercalculateur que sont aujourd'hui nos ordinateur personnel, se réduit de jours en jours.