

Rapport du Travail Encadré de Recherche :
Factorisation et corps finis

Louis COUMAU, Axel DURBET, Fivos REYRE, Sid Ali ZITOUNI TERKI

28 septembre 2022

Table des matières

1	Introduction	3
2	Pré-requis théoriques	4
2.1	Arithmétique de base	4
2.2	Anneaux et corps fini	7
3	Tests de primalité	10
3.1	Test combinatoire	10
3.2	Test de Fermat	12
3.2.1	Algorithme	12
3.2.2	Probabilité d'erreur	13
3.3	Test de Miller-Rabin	14
3.3.1	Algorithme	14
3.3.2	Probabilité d'erreur	15
3.4	Comparaison	15
3.5	AKS	16
4	Méthodes élémentaires de factorisation	17
4.1	Premières idées	17
4.2	Puissance d'un nombre premier	18
4.3	Variante du crible d'Eratosthène	19
4.4	Remarque sur l'indicatrice d'Euler	21
5	Trois algorithmes de factorisation	23
5.1	Algorithme $p - 1$ de Pollard	23
5.2	Méthode $p + 1$ de Williams	28
5.3	Courbes elliptiques	29
5.3.1	Définitions et théorèmes	30
5.3.2	Algorithme de Lenstra	34
6	Analyse de Complexité	36
6.1	Complexité	36
6.1.1	La notation L	36
6.1.2	L et la friabilité	36
6.1.3	Le choix de B dans les courbes elliptique	37

7	Implémentation	39
7.1	Ossature de l'algorithme et fonctions phares	39
7.2	Le problème des grands nombres	39
7.3	Une bibliothèque de cryptographie : GMP	40
7.4	Éliminer les petits facteurs	41
7.5	Test de primalité	41
7.6	Es-tu un carré ou plus?	42
7.7	Pollard	42
7.8	Courbes elliptiques	42
7.8.1	Création et destruction de la courbe elliptique	42
7.8.2	Choix de la courbe elliptique	43
7.8.3	Structure des points	43
7.8.4	Somme sur la courbe	43
7.8.5	Produit sur la courbe	43
7.8.6	Le choix de B	43
7.8.7	Méthode de Lenstra	44
7.9	Comment les assembler	44
7.10	Tests et résultats	44
7.10.1	Tests $P - 1$ Pollard en C	44
7.10.2	Test de l'algorithme complet en C	48
7.10.3	Tests complémentaires	49
8	Conclusion	51
8.1	Bibliographie	52
A	Le code	53
A.1	Python	53
A.1.1	Pollard python	53
A.1.2	ECM python	54

Chapitre 1

Introduction

La factorisation est l'un des plus grands problèmes mathématiques sur lequel se basent certains systèmes cryptographiques tel que RSA.

Pour notre projet de recherche, nous allons nous intéresser au problème suivant : Comment factoriser un entier de façon efficace ?

Dans un premier temps, nous verrons quelques pré-requis théoriques et deux tests de non-primalité. Nous donnerons quelques exemples d'algorithmes de test de non-primalité. Une fois ceux-ci établis, nous parlerons de quelques algorithmes naïfs de factorisation.

Dans un second temps, nous verrons comment exhiber un diviseur non trivial¹ d'un entier n de façon moins évidente. Pour cela, nous évoquerons la méthode de Pollard dite $p - 1$ et d'une variante dite $p + 1$.

Ce qui nous mènera aux courbes elliptiques et à leurs utilisations afin de factoriser un entier.

Suite à cela, nous proposerons un algorithme en C qui répond au problème de la factorisation.

Pour finir, nous ferons quelques tests sur nos algorithmes et analyserons les résultats.

1. Un diviseur non trivial de n est un entier différent de 1 et de n

Chapitre 2

Pré-requis théoriques

2.1 Arithmétique de base

Dans cette partie, nous rappellerons les principes de base de l'arithmétique. Puis, nous verrons en fin de partie les groupes. Le but de cette partie est de poser les notions qui nous permettront par la suite de travailler notre sujet.

Définition 2.1.1. *On dit qu'un entier a non nul divise un entier b lorsqu'il existe k un entier tel que $a = k \times b$. On note alors $a \mid b$ (a divise b).*

Définition 2.1.2. *Soient a et b non nuls, on note $PGCD(a, b)$ le plus grand entier k qui divise à la fois a et b . Si le $PGCD(a, b) = 1$, on dit que a est premier avec b ou encore que a et b sont premiers entre eux.*

Définition 2.1.3 (Nombre premier). *On dit qu'un entier $p \geq 2$ est un nombre premier si $\forall 0 < a \leq p - 1, PGCD(p, a) = 1$. Les seuls diviseurs d'un nombre premier sont 1 et lui-même.*

Théorème 2.1.1 (Gauss). *Soit a, b, c des entiers positifs non nuls tel que a divise $b \times c$. Si a est premier avec b alors il divise c .
 $\forall a, b, c \in \mathbb{N}^*; PGCD(a, b) = 1 \implies a \mid c$*

Corollaire 2.1.1. *Si a est premier avec b et c , alors a est premier avec $b \times c$ (a, b et c non nuls).
 $\forall a, b, c \in \mathbb{N}^*, PGCD(a, b) = 1$ et $PGCD(a, c) = 1 \implies PGCD(a, bc) = 1$*

Théorème 2.1.2 (Bézout). *Soit a et b deux entiers premiers entre eux non nuls. Il existe une infinité de couples (u, v) avec u et v des entiers relatifs tels que l'égalité suivante est vérifiée : $a \times u + b \times v = 1$
On appelle u et v les coefficients de Bézout.
Plus généralement, pour tout a et b entiers non nuls, il existe une infinité de*

couples (u, v) tel que :
 $a \times u + b \times v = PGCD(a, b)$

On trouve une solution particulière à partir de l'algorithme d'Euclide étendu.

Démonstration. Soit a et b deux entiers non nuls premiers entre eux. Grâce à l'algorithme d'Euclide étendu, on trouve un couple (u, v) tel que $a \times u + b \times v = 1$.
 On cherche les couples (s, t) tels que $a \times s + b \times t = 1$.
 On écrit donc $a \times s + b \times t = a \times u + b \times v$.
 On a donc (*) $a(s - u) = b(v - t)$
 D'où $a \mid b(v - t)$
 D'après Gauss $a \mid v - t$ car $PGCD(a, b) = 1$
 De ce fait, il existe k dans \mathbb{Z} tel que : $a \times k = v - t$
 Ainsi $t = v - ak$, on remplace dans (*)
 On a alors : $a(s - u) = b(v - v + ak)$
 Donc $s = bk - u$.
 Ainsi les solutions à $a \times s + b \times t = 1$ sont :
 $S = \{(bk - u; v - ak), k \in \mathbb{Z}\}$ □

Définition 2.1.4 (PPCM). Soit a et b deux entiers. On définit le PPCM(a, b) le plus petit multiple commun de a et b .

Il vérifie l'égalité suivante : $PPCM(a, b) = \frac{|a \times b|}{PGCD(a, b)}$

Définition 2.1.5 (DFP). Décomposition en facteurs premiers (DFP). Tout entier k positif non nul peut s'écrire comme produit de puissances de nombres premiers.

Cette décomposition est unique pour chaque entier à l'ordre des éléments près.

Définition 2.1.6 (Congruence). Classe de congruence modulo k . On note a modulo k ou encore $a \bmod k$ ou $a [k]$ le reste de la division euclidienne de a par k . Si les restes de la division euclidienne de a par k et de b par k sont égaux alors on dit que a et b sont dans la même classe de congruence modulo k .

On dit que a est "congru" à b modulo k (noté $a \equiv b [k]$).

Il existe une autre façon de le dire, on dit que a et b sont congrus modulo un entier n strictement positif si $a - b$ est un multiple de n .

Définition 2.1.7 (Groupe). Un groupe est un couple composé d'un ensemble G et d'une loi de composition sur cet ensemble $*$. Celle-ci, à deux éléments $(a, b) \in G^2$, associe un autre élément $a * b$.

Cette loi doit vérifier les propriétés suivantes :

1) Loi de composition interne
 $\forall (a, b) \in G^2, a * b \in G$.

2) Associativité

$$\forall (a, b, c) \in G^3, (a * b) * c = a * (b * c)$$

3) Élément neutre

$$\exists e \in G / \forall a \in G, e * a = a * e = a$$

"e" est l'élément neutre du groupe $(G, *)$.

4) Inverse

Soit e l'élément neutre. On a donc : $\forall a \in G, \exists b \in G / a * b = b * a = e$
b est l'inverse de a pour la loi *.

Définition 2.1.8 (Ordre). Soit a un élément du groupe, on appelle ordre de a le plus petit entier positif k tel que $a^k = 1$.

Définition 2.1.9 (Sous-groupe). On dit que H est un sous-groupe de G si :
H est un groupe pour la même loi que G, l'élément neutre de H est le même que celui de G et l'inverse de tout élément de H est dans H

Définition 2.1.10 (Sous-groupe distingué). On dit que H est un sous-groupe distingué ou normal si chaque élément $g \in G, gHg^{-1} \in H$.

Définition 2.1.11 (Ensemble Quotient). Soit G un ensemble et H un sous-groupe de G.

On définit G/H l'ensemble quotient comme l'ensemble G/\sim avec \sim la relation d'équivalence suivante : $(x, y) \in G^2, x \sim y \iff x^{-1} \times y \in H$.

G/H n'est en général pas un groupe. Ceci n'est vrai seulement si le sous-groupe H est distingué.

Définition 2.1.12 (Indice). Soit G un groupe et H un sous-groupe de celui-ci. L'indice de H dans G (noté $[G : H]$) est le cardinal de l'ensemble quotient G/H .

Théorème 2.1.3. Soit G un groupe fini et H un sous-groupe de G. Alors, l'ordre de H divise l'ordre de G.

Autrement dit $\#H \mid \#G$.

Démonstration. Par définition, l'indice $[G : H]$ de H dans G est le cardinal de l'ensemble G/H des classes à droite suivant H des éléments de G. On note les classes à droite comme suit : $H \times a$.

Or ces classes forment une partition de G.

$$\text{Ainsi, } \#G = \sum_{Ha \in G/H} \#Ha.$$

Chacune des classes à droite a le même cardinal que H.

Ainsi, on en déduit :

$$\#G = \#H \times [G : H].$$

Ainsi $\#H \mid \#G$

□

Corollaire 2.1.2. Soit G un groupe fini. Soit a un élément de G . Alors l'ordre de a divise l'ordre de G .

Ce corollaire est aussi connu sous le nom de théorème de Lagrange.

Définition 2.1.13 (Indicatrice d'Euler). L'indicatrice d'Euler est une fonction qui à tout entier naturel n non nul associe le cardinal (=le nombre d'éléments) de l'ensemble des nombres $k \in]0, n[$ premiers avec lui. La fonction est notée φ . Si $n = \prod_{i=1}^r p_i^{k_i}$ alors $\varphi(n) = \prod_{i=1}^r (p_i - 1)p_i^{k_i - 1}$. Notons qu'avec p premier, $\varphi(p) = p - 1$.

Théorème 2.1.4 (Fermat-Euler). Soit a et k deux entiers premiers entre eux. Alors $a^{\varphi(k)} \equiv 1 [k]$.

Démonstration. Soit $(a, k) \in \mathbb{N}$ tel que $\text{PGCD}(a, k) = 1$.

Alors $a \in (\mathbb{Z}/k\mathbb{Z})^\times$ de cardinal $\varphi(k)$.

Donc, d'après le théorème de Lagrange, l'ordre de a divise $\varphi(k)$, on le note d .

Ainsi, $\varphi(k) = d \times n$ avec $n \in \mathbb{N}$.

De ce fait, on sait que $a^d \equiv 1 [k]$.

D'où : $(a^d)^n \equiv 1^n \equiv 1 [k]$.

Ainsi, $a^{\varphi(k)} \equiv 1 [k]$

□

Cas particulier : Petit théorème de Fermat.

Théorème 2.1.5 (Petit théorème de Fermat). Soit p un nombre premier. Alors, $\forall a \in \mathbb{N}^*$, $a^{p-1} \equiv 1 [p]$.

Théorème 2.1.6 (Théorème des restes chinois). Soit $a \in \mathbb{N}^*$ tel que :
 $a = p_1 \times p_2 \times \dots \times p_n$ avec $\text{PGCD}(p_i, p_j) = 1$ ($i \neq j$), $p_i \in \mathbb{N}^*$.
Alors $\mathbb{Z}/a\mathbb{Z}$ est isomorphe à $(\mathbb{Z}/p_1\mathbb{Z}) \times \dots \times (\mathbb{Z}/p_n\mathbb{Z})$.

2.2 Anneaux et corps fini

Définition 2.2.1 (Anneau). Un anneau est un ensemble A muni de deux lois (notons les $+$ et \times) telles que :

- $(A, +)$ est un groupe commutatif
- \times est associative et possède un élément neutre noté 1
- \times est distributive par rapport à $+$

On note A le groupe multiplicatif de A . Il est constitué de tous les éléments inversibles de A .

Définition 2.2.2 (caractéristique). Soit $\varphi(\mathbb{Z}) \rightarrow A$ l'application définie par : $\forall n \in \mathbb{Z}, \varphi(n) = 1_A + 1_A + \dots + 1_A$ (n fois).
 $\text{Ker}(\varphi) = \{n, \varphi(n) = 0\}$ définit un idéal, $m\mathbb{Z}$. On appelle alors m la caractéristique de A .
Autrement dit, m est le plus petit entier tel que la somme de m fois l'élément neutre pour la multiplication vaut 0.

Définition 2.2.3 (Corps). On dit que A est un corps lorsque $A^* = A - \{0\}$.

Définition 2.2.4. Soit K un corps. On dit que L est une extension de K (que l'on note $L : K$) lorsque K est un sous-corps de L .

On note $[L : K] = \dim_K(L)$ le degré de l'extension $L : K$.

Théorème 2.2.1. Si $L : K$ et $M : L$ (M et L deux ensembles finis), on a $[M : K] = [M : L] \times [L : K]$

Démonstration. Soit (m_1, \dots, m_a) une base de M sur L et (l_1, \dots, l_b) une base de L sur K .

Alors $(m_i, l_j)_{i \in \{1, \dots, a\}, j \in \{1, \dots, b\}}$:

$$x \in M, x = \sum_{i=1}^a \lambda_i m_i, \lambda_i \in L \Rightarrow x = \sum_{i,j} k_{i,j} m_i l_j$$

$$\sum_{i,j} k_{i,j} m_i l_j = 0, \sum_i (\sum_j k_{i,j} l_j) m_i = 0 \Rightarrow \forall i, \sum_j k_{i,j} l_j = 0 \Rightarrow \forall j, k_{i,j} = 0 \quad \square$$

Proposition 2.2.1. Pour tout nombre premier p , $\mathbb{Z}/p\mathbb{Z}$ est un corps.

Démonstration. p étant premier, tout élément non nul de $\mathbb{Z}/p\mathbb{Z}$ est inversible. On a donc $(\mathbb{Z}/p\mathbb{Z})^* = \mathbb{Z}/p\mathbb{Z} - \{0\}$. \square

Proposition 2.2.2. Pour tout corps fini K il existe un premier p et un entier n tel que K soit une extension de $\mathbb{Z}/p\mathbb{Z}$, de degré n .

On note ce corps \mathbb{F}_{p^n} .

Définition 2.2.5. Soient K, L deux corps finis, L étant une extension de K . Soit $\alpha \in L$. On note $K(\alpha)$ le plus petit sous-corps de L qui contient K et α .

Définition 2.2.6. Si $L = K(\alpha)$, on dit que L est extension simple (ou encore primitive) de K .

Définition 2.2.7. Si $[K(\alpha) : K] < \infty$, on appelle polynôme minimal de α le polynôme P_α unitaire de plus petit degré tel que α soit racine.

Proposition 2.2.3. Soient K, L deux corps et $\alpha \in L$, $[K(\alpha) : K] < \infty$, P_α le polynôme minimal de α .
 P_α est irréductible.

Démonstration. Supposons par l'absurde qu'il existe deux polynômes $Q, R \in K[X]$ tels que $P_\alpha = P \times Q$. Alors $P_\alpha(\alpha) = Q(\alpha) \times R(\alpha)$. et $\deg(Q) < \deg(P)$. Donc on a trouvé un polynôme de plus petit degré que P tel que α est racine. Ceci est absurde par définition donc P_α est irréductible. \square

Proposition 2.2.4. *Soient K, L deux corps, L une extension de K , $\alpha \in L$ et $[K(\alpha) : K]$ fini. $K(\alpha)$ est isomorphe à $K[X] / \langle P_\alpha \rangle$, où $\langle P_\alpha \rangle = \{P \in K[X], P(\alpha) = 0\}$.*

Définition 2.2.8 (élément primitif). *Soit K un corps fini et K^* son groupe multiplicatif. On dit que α est un élément primitif de K si α engendre K^* .*

Théorème 2.2.2. *Tout corps fini K admet un élément primitif.*

Chapitre 3

Tests de primalité

Nous avons vu précédemment la définition d'un nombre premier (*Définition 2.1.3*). Ainsi, avant de vouloir factoriser un entier, il est important de déterminer s'il est premier ou non.

3.1 Test combinatoire

Proposition 3.1.1. $\forall k \in [1, n-1], \binom{n}{k} \equiv 0 \pmod n \iff n \text{ est premier.}$

Démonstration. Commençons par le sens trivial :

\Leftarrow

On a n premier. Aussi, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Si n est premier et $0 < k < n$, alors $n \nmid k!$ et $n \nmid (n-k)!$. Donc, d'après le Corollaire 2.1.1 $n \nmid k!(n-k)!$. Ainsi, pour $1 \leq k \leq n-1$,

$$\binom{n}{k} = n \times \frac{(n-1)!}{k!(n-k)!} \equiv 0 \pmod n$$

\Rightarrow

Raisonnons par contraposé, ie,

$$n \text{ non premier} \implies \exists k \in [1, n-1], \binom{n}{k} \not\equiv 0 \pmod n$$

Si n non premier, il existe un nombre premier p tel que $1 < p < n$ et un entier d tel que $n = pd$.

Voyons le cas ou $k = p$

$$\begin{aligned}
\binom{n}{p} &= \frac{n!}{p!(n-p)!} \\
&= \frac{pd!}{p!(pd-p)!} \\
&= \frac{p \times d \times (pd-1) \times \dots \times (pd-p+1) \times (pd-p)!}{p \times (p-1)! \times (pd-p)!} \\
&= \frac{d \times (pd-1) \times \dots \times (pd-p+1)}{(p-1)!} \\
&\equiv \frac{d \times (-1) \times \dots \times (-(p-1))}{(p-1)!} \pmod{pd}
\end{aligned}$$

On a $(-(p-1))! = (-1)^{p-1}(p-1)!$, ce qui nous donne :

$$\begin{aligned}
&\equiv \frac{d \times (-1)^{p-1} \times (p-1)!}{(p-1)!} \pmod{pd} \\
&\equiv (-1)^{p-1} \times d \pmod{pd}
\end{aligned}$$

Donc, pour conclure, si $p|n$ avec p premier et $d = n/p$, alors

$$\binom{n}{p} = (-1)^{p-1} \times d \not\equiv 0 \pmod{n}$$

Nous avons donc la connaissance d'un entier $k \in [1, n-1]$ tel que $\binom{n}{k} \not\equiv 0 \pmod{n}$ dans le cas ou n n'est pas premier.

□

Malheureusement, cette propriété est inutile en pratique car nous ne pouvons pas nous permettre de tester les coefficients binomiaux pour toutes les valeurs de k . Surtout que cela implique la formule du factoriel. Cependant, on peut en déduire ceci :

Proposition 3.1.2. n premier $\implies 2^n \equiv 2 \pmod{n}$

Démonstration.

$$n \text{ premier} \iff \forall k \in [1, n-1], \binom{n}{k} \equiv 0 \pmod{n} \quad (3.1)$$

$$\implies \sum_{k=1}^{n-1} \binom{n}{k} \equiv 0 \pmod{n} \quad (3.2)$$

$$\iff \binom{n}{0} + \sum_{k=1}^{n-1} \binom{n}{k} + \binom{n}{n} \equiv 2 \pmod{n} \quad (3.3)$$

$$\iff \sum_{k=0}^n \binom{n}{k} \equiv 2 \pmod{n} \quad (3.4)$$

Or, on sait que :

$$\sum_{k=0}^n \binom{n}{k} = 2^n$$

Ainsi,

$$n \text{ premier} \implies 2^n \equiv 2 \pmod{n}$$

□

3.2 Test de Fermat

Voyons maintenant une propriété plus forte que la précédente. Nous avons vu que le petit théorème de Fermat (*Théorème 2.1.5*) énonce que :

$$p \text{ premier} \iff \forall a \in \llbracket 0, p \llbracket, a^{p-1} \equiv 1 \pmod{p}$$

Dont on peut en déduire :

$$p \text{ non premier} \iff \exists a \in \llbracket 0, p \llbracket, a^{p-1} \not\equiv 1 \pmod{p}$$

Le test de primalité de n consiste donc à élever à la puissance $n-1$ un nombre a premier¹ avec n , tiré aléatoirement dans l'intervalle $[2, n-1]$ et vérifier le résultat modulo n .

3.2.1 Algorithme

1. On notera que si l'entier n'est pas premier avec n , alors ils ont un diviseur commun et donc n n'est pas premier.

Algorithme 3.2.1.1. [TEST DE FERMAT]*Entrée : n un entier impair.**Sortie : "composé"², "peut être premier"*

1. tirer aléatoirement a dans $[2, n - 1]$
2. Si $\text{pgcd}(a, n) \neq 1$
3. renvoyer "composé"
4. Sinon
5. $r \leftarrow a^{n-1} \pmod n$
6. Si $r \neq 1$
7. renvoyer "composé"
8. Sinon, renvoyer "peut être premier"

Évidemment, si on a trouvé que $a^{n-1} \equiv 1 \pmod n$, on ne peut pas vraiment déduire que n est premier puisque l'on a testé qu'un seul entier sur tout l'intervalle $[2, n - 1]$ or, il faudrait tous les tester pour affirmer que l'entier est bien premier. Cependant, il existe des nombres n composés appelés nombres de Carmichael, dont tous les nombres inférieurs et premiers avec n sont de faux témoins. Le test de Fermat ne parviendra donc jamais à déterminer si n est un composé de Carmichael. On peut cependant réitérer l'opération un certain nombre de fois.

3.2.2 Probabilité d'erreur

Soit n un entier composé et p_n la probabilité d'échec du test. Soit M_n l'ensemble des faux témoins :

$$M_n = \{x \in (\mathbb{Z}/n\mathbb{Z})^* \mid x^{n-1} = 1\}$$

On a donc :

$$p_n = \frac{\text{card}(M_n)}{n - 2}$$

Aussi, on sait que M_n est un sous-groupe de $(\mathbb{Z}/n\mathbb{Z})^*$ donc $\text{card}(M_n) \mid \varphi(n)$. Nous avons deux cas :

1. $M_n \neq (\mathbb{Z}/n\mathbb{Z})^*$

$$\begin{aligned} \varphi(n) \neq \text{card}(M_n) &\implies \varphi(n) > \text{card}(M_n) \\ &\implies \exists d \geq 2 \in \mathbb{Z} \mid \varphi(n) = d \times \text{card}(M_n) \end{aligned}$$

2. Dans les algorithmes de tests de primalité qui vont suivre nous décidons de renvoyer "composé" dans le cas où la propriété est satisfaite et "peut être premier" si la propriété n'est pas satisfaite.

On peut donc minorer d par 2,

$$p_n = \frac{\text{card}(M_n)}{n-2} \leq \frac{\text{card}(M_n)}{\varphi(n)} \leq \frac{1}{2}$$

2. $M_n = (\mathbb{Z}/n\mathbb{Z})^*$

$$p_n = \frac{\text{card}(M_n)}{n-2} = \frac{\varphi(n)}{n-2} \sim 1$$

La probabilité dépend donc de l'entier n . Si ce n'est pas un nombre de Carmichael, la probabilité d'échec est majorée par $\frac{1}{2}$ sinon elle tend vers 1. On ne peut donc pas parler d'algorithme probabiliste.

3.3 Test de Miller-Rabin

Le test de Fermat peut être affiné avec un lemme pour être capable de détecter les nombres de Carmichael.

Lemme 3.3.1. *Soit A un anneau intègre et $x \in A$.*

$$x^2 = 1 \iff x \in \{\pm 1\}$$

Ainsi, le théorème principal est le suivant :

Théorème 3.3.1 (Miller-Rabin). *Soit n un nombre premier impair. On pose $n-1 = 2^e m$ avec m un entier impair. Pour tout entier a premier à n ,*

$$\begin{cases} \text{soit } a^m \equiv 1 \pmod{n} \\ \text{soit il existe } i \in [0, e-1] \text{ tel que } a^{2^i} \equiv -1 \pmod{n} \end{cases}$$

3.3.1 Algorithme

Algorithme 3.3.1.1. [MILLER-RABIN]

Entrée : n : entier impair, $e, m : n-1 = 2^e m$

Sortie : "composé", "peut être premier"

1. On tire a aléatoirement sur $[1, n-1]$
 2. $d \leftarrow \text{pgcd}(a, n)$
 3. Si $d \neq 1$:
 4. On renvoie "composé"
 5. $b \leftarrow a^m \pmod{n}$
 6. Si $b \equiv 1 \pmod{n}$ ou s'il existe $i \in [0, e-1]$ tel que $b^{2^i} \equiv -1 \pmod{n}$:
 7. On renvoie Vrai
 8. On renvoie "composé"
-

3.3.2 Probabilité d'erreur

Reprenons comme pour le test de Fermat, p_n la probabilité de tirer un menteur. Voyons M_n l'ensemble de ces menteurs :

$$M_n = \{x \in (\mathbb{Z}/n\mathbb{Z})^* : \left\{ \begin{array}{l} \text{soit } x^m \equiv 1 \pmod{n} \\ \text{soit il existe } i \in [0, e-1] \text{ tel que } x^{2^i m} \equiv -1 \pmod{n} \end{array} \right\}$$

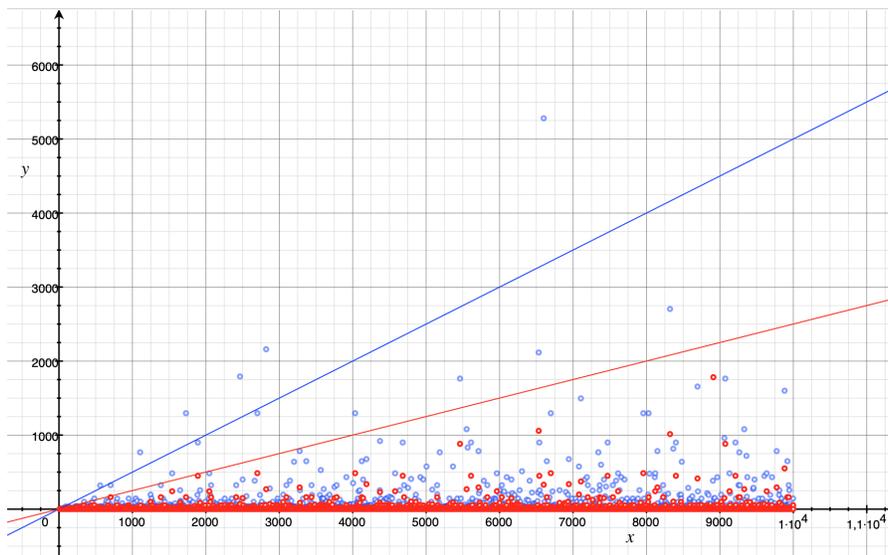
Théorème 3.3.2. *Si $n \geq 9$ est un nombre composé impair, alors*

$$\text{card}(M_n) \leq \frac{n-1}{4}$$

Ainsi, $p_n \leq \frac{1}{4}$.

3.4 Comparaison

Voyons sur un plan un nuage de points (*Figure 3.1*) représentant $y = \text{card}(M_x)$. En bleu sont représentés les $\text{card}(M_x)$ du test de Fermat accompagnés de la fonction représentative³ $y = \frac{x}{2}$ et en rouge, le $\text{card}(M_x)$ du test de Miller-Rabin avec la fonction $y = \frac{x}{4}$.



Cardinaux des ensembles des menteurs M_x du test de Fermat (en bleu) et de Miller-Rabin (en rouge) avec $x \in [1; 10^4]$

FIGURE 3.1 – Comparaison

3. On remarquera que tous les points bleus au dessus de cette courbe sont les nombres de Carmichael.

On peut constater qu'aucun des points rouges ne dépassent la fonction $\frac{x}{4}$. On peut donc en conclure que le test de Miller-Rabin est environ deux fois plus efficace que celui de Fermat.

3.5 AKS

Il existe un algorithme de test de primalité déterministe. Un exemple d'un tel algorithme est AKS. Celui-ci se base sur une généralisation du petit théorème de Fermat.

Proposition 3.5.1 (AKS). *Pour tout entier $n \geq 2$ et tout entier a premier avec n ,*

$$n \text{ premier} \iff (X + a)^n \equiv X^n + a \pmod{n}$$

La complexité d'un tel algorithme est en $\mathcal{O}(\log^{12}(n))$ mais sur certaines variantes, celle-ci n'est plus qu'en $\mathcal{O}(\log^{10,5}(n))$. Cette dernière valeur nous donne un algorithme polynomial pour savoir avec certitude si un nombre est composé ou non.

Chapitre 4

Méthodes élémentaires de factorisation

4.1 Premières idées

Maintenant que nous avons suffisamment de bagages, voyons comment nous pourrions factoriser un entier. L'objectif est de faire passer un entier quelconque n sous sa forme de produit de facteurs premiers.

On sait que nécessairement, les diviseurs non triviaux de n (*s'ils existent*) sont strictement inférieurs à celui-ci. Ainsi, l'algorithme naïf est de parcourir tous les entiers de 2 à n et de vérifier si chaque élément est un diviseur de n . On s'aperçoit que, si n est grand, les calculs deviennent longs. Soit, $n - 2$ étapes de calculs.

Pour améliorer cela, étudions la façon avec laquelle sont distribués les diviseurs de n .

Proposition 4.1.1. *Soit n un entier naturel. Si n est le produit de deux nombres entiers $d_i \geq 2$, avec $i \in \mathbb{N}$ tel que $0 \leq i \leq 1$, alors exactement l'un des deux est dans l'intervalle $[2, \sqrt{n}]$.*

Démonstration. Résonnons par l'absurde : On pose $n = ab$ avec $a > \sqrt{n}$ et $b > \sqrt{n}$. Ainsi,

$$\begin{aligned} ab &> \sqrt{n}\sqrt{n} \\ ab &> n \end{aligned}$$

Donc, $ab > n$ ce qui est absurde. □

De même,

Proposition 4.1.2. *Soit n un entier naturel. Si n est le produit de trois nombres entiers $d_i \geq 2$, avec $i \in \mathbb{N}$ tel que $0 \leq i \leq 2$, alors au moins l'un des trois est dans l'intervalle $[2, n^{\frac{1}{3}}]$.*

La démonstration se fait de la même façon que la précédente. Ainsi, nous pouvons généraliser :

Proposition 4.1.3. *Soit n un entier naturel. Si n est le produit de k nombres entiers $d_i \geq 2$ tel que $k \in \mathbb{N}_{\geq 2}$ et $i \in \mathbb{N}$ tel que $0 \leq i \leq k - 1$, alors, il existe au moins un diviseur dans l'intervalle $[2, n^{\frac{1}{k}}]$.*

Exemple :

$$6191893 = 17 \times 457 \times 797$$

$$6191893^{\frac{1}{3}} \approx 183 \text{ or } 17 \in [2, 183]$$

Nous pouvons donc dire que plus un entier a de diviseurs, moins il faudra d'étapes pour en trouver un. Si nous supposons que notre nombre n a au moins deux facteurs premiers distincts, il suffit donc de parcourir l'intervalle $[2, \sqrt{n}]$ afin d'en trouver un.

Mais est-ce suffisant pour trouver tous les diviseurs ? Effectivement, il n'est pas un problème de parcourir l'intervalle $[2, \sqrt{n}]$ pour trouver tous les diviseurs. En effet, si on en trouve un, il suffit de le "retirer" en divisant n par ce dernier. Ce qui nous donne un entier plus petit, et donc, potentiellement plus facile à factoriser.

À partir de là, pour factoriser un entier nous allons plutôt nous intéresser à chercher un seul diviseur non trivial.

4.2 Puissance d'un nombre premier

Voyons un autre cas intéressant de nombres à factoriser.

Proposition 4.2.1. *Soit n un nombre entier quelconque. Le nombre d'étapes pour déterminer si n est une puissance d'un nombre premier p est au plus $\log_2(n)$.*

Démonstration. L'algorithme à appliquer dans ce cas est : Avec $k = 2$, tant que $n^{\frac{1}{k}}$ n'est pas un entier et est supérieur à 2, on incrémente k . Sinon, on s'arrête. Deux cas se présentent à nous :

- Soit $n = p^i$, et dans ce cas, on aura fait $i - 1$ étapes.
- Soit n n'est pas une puissance d'un nombre entier et dans ce cas, on s'arrête si $n^{\frac{1}{k}} \leq 2$.

$$\begin{aligned}
n^{\frac{1}{k}} &\leq 2 \\
\frac{1}{k} \log(n) &\leq \log(2) \\
\frac{1}{k} &\leq \frac{\log(2)}{\log(n)} \\
k &\geq \frac{\log(n)}{\log(2)} \\
k &\geq \log_2(n)
\end{aligned}$$

Ainsi, dans le pire des cas, l'algorithme fait au plus $\log_2(n)$ étapes pour déterminer si n est une puissance de nombre premier. \square

4.3 Variante du crible d’Eratosthène

Nous avons vu plus haut qu’il suffisait de parcourir $[2, \sqrt{n}]$ pour trouver un diviseur de n . Mais peut-on faire mieux ?

Le crible d’Eratosthène est basé sur le principe d’élimination des valeurs non pertinentes. Si on trouve p , un entier premier tel que $p \nmid n$, alors $\forall k, k \in \mathbb{N} \setminus \{0\}$, pk ne divise pas n . Ainsi, il est inutile de tester tous les multiples de p . Donc l’idéal serait de tester uniquement les nombres premiers qui sont dans $[2, \sqrt{n}]$. Deux possibilités d’implémentation sont possibles :

- Soit on élimine tous les multiples des entiers rencontrés sur un intervalle, et donc, on les conserve en mémoire,
- Soit on utilise une liste de nombres premiers, donnée au préalable.

Dans les deux cas, ces implémentations deviennent vite problématiques quand le nombre à factoriser devient grand.

Regardons de plus près la distribution des nombres premiers : (*Figure 4.3*)
Voyons ce qui se produit si on change la disposition : (*Figure 4.3*)

On s’aperçoit que, dans une colonne, tous les nombres qui sont en dessous de 2,3,4 et 6 sont nécessairement des multiples de 2, et 3. Avec cette disposition, il est facile de représenter chaque élément d’une colonne par une suite $(S)_{i,j}$ pour $0 \leq i$ et $0 \leq j \leq 6$ représentant respectivement la ligne et la colonne.

4	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

FIGURE 4.1 – Distribution des premiers nombres premiers
On décide de façon arbitraire de les représenter de cette façon.

4	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

FIGURE 4.2 – Distribution intéressante des premiers nombres premiers

Ainsi,

$$\begin{aligned}
 (S)_{i,1} &= 6i + 1 \\
 (S)_{i,2} &= 6i + 2 \\
 &\vdots \\
 &\vdots \\
 (S)_{i,j} &= 6i + j
 \end{aligned}$$

On vérifie facilement que pour $j = 0, 2, 3, 4$, $(S)_i$ est divisible par 2 ou 3. Donc les suites pour $j = 1$ et $j = 2$ génèrent tout les éléments non divisibles par 2 et 3.

$$\begin{aligned}
 (S)_{i,1} &= 6i + 1 \\
 (S)_{i,5} &= 6i + 5
 \end{aligned}$$

Ce qui nous amène à une remarque intéressante :

$$n \geq 5 \text{ et } n \text{ premier} \Rightarrow n \equiv \pm 1 \pmod{6} \quad (4.1)$$

On notera que la réciproque est fausse.

Pour en revenir à la factorisation, on peut établir un pseudo algorithme :

Algorithme 4.3.0.1. [VARIANTE DU CRIBLE D'ERATOSTHÈNE]

Entrée : n un entier

Sortie : d un diviseur non trivial de n

1. *Si 2 divise n ,*
2. *on renvoie 2*
3. *Si 3 divise n ,*
4. *on renvoie 3*
5. *$i \leftarrow 1, a \leftarrow 0, b \leftarrow 0$*
6. *Tant que $a \leq \sqrt{n}$*
7. *$a \leftarrow 6i - 1$*
8. *$b \leftarrow 6i + 1$*
9. *Si a divise n ,*
10. *on renvoie a*
11. *Si b divise n ,*
12. *on renvoie b*
13. *$i \leftarrow i + 1$*
14. *fin tant que*

Dans le pire des cas, le test s'arrête si $6i - 1 > \sqrt{n}$

$$\begin{aligned}6i - 1 &> \sqrt{n} \\6i &> \sqrt{n} + 1 \\i &> \frac{\sqrt{n} + 1}{6} \approx \frac{\sqrt{n}}{6}\end{aligned}$$

Plus haut nous parlions d'un algorithme testant \sqrt{n} valeurs. À présent, l'algorithme teste $\frac{\sqrt{n}}{6}$ valeurs. Il en teste donc 6 fois moins.

4.4 Remarque sur l'indicatrice d'Euler

Dans cette section on s'intéressera uniquement à la forme $n = p \times q$ tel que p et q sont deux nombres premiers distincts. (aussi appelé "nombre RSA")

On a vu que l'indicatrice d'Euler $\varphi(n)$ (*Définition 2.1.13*) est le nombre d'éléments inférieurs et premiers avec n . Aussi, nous pouvons émettre la proposition suivante :

Proposition 4.4.1. *Connaître n et $\varphi(n)$ est équivalent à connaître p et q .*

Démonstration. Si p et q sont deux nombres premiers, alors,

$$\begin{aligned}\varphi(n) &= (p-1)(q-1) \\ &= pq - p - q + 1 \\ &= n - p - q + 1 \\ 0 &= n - p - q + 1 - \varphi(n) \\ &= pn - p^2 - pq + p - p\varphi(n) \\ &= -p^2 + pn - n + p - p\varphi(n) \\ &= -p^2 + p(n+1 - \varphi(n)) - n \\ 0 &= p^2 - p(n+1 - \varphi(n)) + n\end{aligned}$$

Ainsi, nous avons un polynôme dont p est racine. On peut donc calculer le discriminant :

$$\begin{aligned}\Delta &= b^2 - 4ac \\ &= (n+1 - \varphi(n))^2 - 4n\end{aligned}$$

On a pour finir deux solutions :

$$\begin{aligned}\frac{-b - \sqrt{\Delta}}{2a} &= p \\ \frac{-b + \sqrt{\Delta}}{2a} &= q\end{aligned}$$

Si $p < q$.

□

Chapitre 5

Trois algorithmes de factorisation

5.1 Algorithme $p - 1$ de Pollard

L'algorithme de Pollard est basé sur le petit théorème de Fermat.

Soit p un nombre premier tel que $p|n$. Soit d un diviseur non trivial de n quelconque. D'après le petit théorème de Fermat, $\forall a \in \mathbb{N}$:

$$\begin{aligned} a^p &\equiv a \pmod{p} \\ \text{si } \text{pgcd}(a, p) &= 1, a^{p-1} \equiv 1 \pmod{p} \\ a^{(p-1)k} &\equiv 1 \pmod{p}, \forall k \in \mathbb{Z} \end{aligned}$$

On peut donc poser $M = (p - 1)k$. Ainsi,

$$a^M - 1 \equiv 0 \pmod{p} \iff p|(a^M - 1)$$

Nous avons donc :

$$p|(a^M - 1) \text{ et } p|n \implies \text{pgcd}(a^M - 1, n) = d \quad (5.1)$$

Avec d un diviseur de n et un multiple de p .

Par conséquent, il suffit de trouver un multiple de $p - 1$. Pour ce faire, nous allons voir deux définitions intéressantes sur les entiers : la B -friabilité et la B -ultra-friabilité.

Pour la suite, on prendra n un entier dont la décomposition en produit de facteurs premiers est

$$n = \prod_{i=0}^l p_i^{\alpha_i}$$

tel que $\forall i, 0 \leq i \leq l, p_i$ est premier et $\alpha_i \in \mathbb{N}$.

Définition 5.1.1 (*B-friable ou B-lisse*). On dit qu'un entier n est *B-friable* si $\forall i, p_i \leq B$ i.e. si B est supérieur ou égal au plus grand diviseur premier de n .

Exemple : $126 = 2 \times 3^2 \times 7 \Rightarrow 126$ est 7-friable.

Voyons maintenant la deuxième définition :

Définition 5.1.2 (*B-ultra-friable ou B-super-lisse*). On dit qu'un entier n est *B-ultra-friable* si $\forall i, p_i^{\alpha_i} \leq B$ i.e. si B est supérieur ou égal à l'entier étant la plus grande puissance de premier divisant n .

Exemple : $126 = 2 \times 3^2 \times 7 \Rightarrow 126$ est 9-ultra-friable

On peut déjà énoncer quelques propriétés évidentes.

Proposition 5.1.1. *Un entier n est toujours n -friable.*

Démonstration. Le principal argument est, en raisonnant par l'absurde, de dire qu'il existe un des diviseurs premiers de n supérieur ou égal à celui-ci. Ce qui donnerait un produit supérieur à n , ce qui serait absurde. □

Évidemment, cette propriété est peu intéressante car on souhaite obtenir une borne minimale¹ idéalement égale au maximum de tous les diviseurs premiers. En revanche, on peut remarquer que les nombres premiers p sont exactement p -friables.

Revenons à notre problème. On souhaite trouver M , un multiple de $p - 1$. C'est là que la notion de friabilité entre en jeu.

Proposition 5.1.2. *Si n est B -ultra-friable, alors $n \mid B!$.*

Démonstration. On rappelle que $n = \prod_{i=0}^l p_i^{\alpha_i}$. D'après la définition,

$$\begin{aligned} n \text{ est } B\text{-ultra-friable} &\Rightarrow \forall i, p_i^{\alpha_i} \leq B \\ &\Rightarrow \forall i, \exists j \mid p_i^{\alpha_i} < p_j^{\alpha_j} \end{aligned}$$

1. Car on souhaite que la borne B soit la plus précise possible.

On a donc un entier étant supérieur à toutes les autres puissances de nombres premiers divisant n que l'on appellera m .

De ce fait, nous aurons dans $m!$ tous les autres entiers inférieurs à m dont tous les $p_i^{\alpha_i}$ et donc, $n|m!$. Or, $B \geq m$, ainsi, $n|B!$. \square

Ainsi, nous arrivons donc à la propriété suivante :

Proposition 5.1.3. *Si $p - 1$ est B -ultra-friable, alors $PGCD(a^{B!} - 1, n) = d$ tel que $d > 1$.*

Démonstration. On a vu plus haut qu'il suffisait de trouver M , un multiple de $p - 1$ pour que $PGCD(a^M - 1, n) = d$. Or, on a aussi vu que si $p - 1$ est B -ultra-friable, alors $B!$ est un multiple de $p - 1$, soit le M recherché. Donc $PGCD(a^{B!} - 1, n) = d$ \square

Cependant, $B!$ est une valeur qui peut vite devenir très grande. De ce fait, il faut prendre B le plus petit possible. Si on reprend l'exemple sur la B -friabilité et la B -ultra-friabilité donné plus haut, on remarque que B -friable $<$ B -ultra-friable. Effectivement, en général, on aura $\forall i \in \mathbb{N}, \forall j \in \mathbb{N}, p_i \leq p_j^{\alpha_i}$. Mais, avons nous les mêmes propriétés que B -ultra-friable ?

Reprenons un exemple :

$$126 = 2 \times 3^2 \times 7 \text{ est } 7\text{-friable}$$

Or,

$$\begin{aligned} 7! &= 7 \times 6 \times 5 \times 4 \times 3 \times 2 \\ &= 7 \times (3 \times 2) \times 5 \times 4 \times 3 \times 2 \end{aligned}$$

On retrouve donc 126 dans 7!

$$\boxed{7} \times (\boxed{3} \times 2) \times 5 \times 4 \times \boxed{3} \times \boxed{2}$$

De ce fait, on peut penser que si n est B -friable, alors $n|B!$. Mais voyons un contre exemple :

$$294 = 2 \times 3 \times 7^2 \text{ est } 7\text{-friable}$$

Or, 7 apparaît qu'une seule fois dans 7!. Donc la propriété de B -friable ne se comporte pas de la même façon que celle de B -ultra-friable.

Ainsi, nous allons plus jouer sur la probabilité que l'entier $p - 1$ soit B -friable. Dans un premier temps, nous pouvons dire que la probabilité pour qu'un entier n soit n -friable est 1. (*premier critère*).²

2. C'est une conséquence d'une propriété précédemment énoncée.

Pour mieux voir la densité d'entier B -friable dans un intervalle, nous allons effectuer des statistiques.

Commençons par faire des expérimentations sur la densité des entiers B -friables. Pour ce faire, posons l'ensemble des entiers y -friables ne dépassant pas x ,

$$S(x, y) := \{\forall k \leq x \mid k \text{ est } y\text{-friable}\}$$

et son cardinal,

$$\Psi(x, y) = |S(x, y)|.$$

Pour estimer la proportion de valeurs y -friables inférieures à x , nous allons diviser le nombre de valeurs respectant le critère par le nombre de valeurs analysées. Soit $D_{x,y}$ représentant cette densité définie par,

$$D_{x,y} = \frac{\Psi(x, y)}{x}.$$

Nous pouvons d'abord affirmer que le rapport $\frac{\ln(x)}{\ln(y)}$ va jouer un rôle essentiel dans cette densité. Ainsi nous posons,

$$u = \frac{\ln(x)}{\ln(y)}$$

Il y a différentes méthodes et formules pour approcher $\Psi(x, y)$, cependant, comme nous allons manipuler de grands entiers, nous utiliserons la formule asymptotique,

$$\Psi(x, y) = xu^{-u}$$

Nous avons donc :

$$D_{x,y} = u^{-u}$$

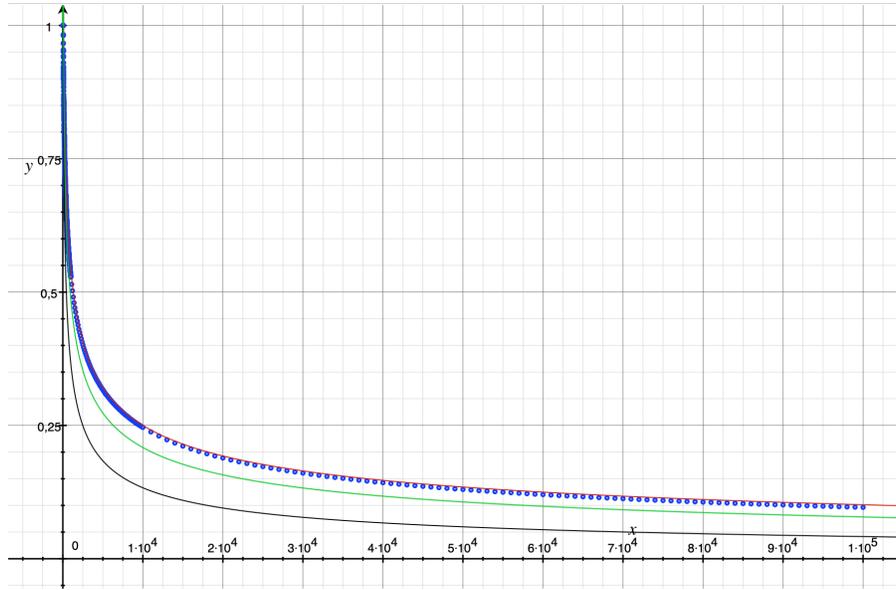
Cette formule permet d'appréhender l'ordre de grandeur de $D_{x,y}$

Pour des raisons d'expérimentations, nous représenterons 3 formules de $D_{x,y}$ sur les graphiques³ :

$$\begin{aligned} \text{noir} : D_{x,y} &= u^{-u}, \\ \text{rouge} : (D_{x,y})_1 &= u_1^{-u_1}, u_1 = \frac{\ln(x)}{\ln(2y)} \\ \text{vert} : (D_{x,y})_2 &= u_2^{-u_2}, u_2 = \frac{\ln(x)}{\ln(ky)}, k = \frac{\log(B) + 1}{\log(B)} \end{aligned}$$

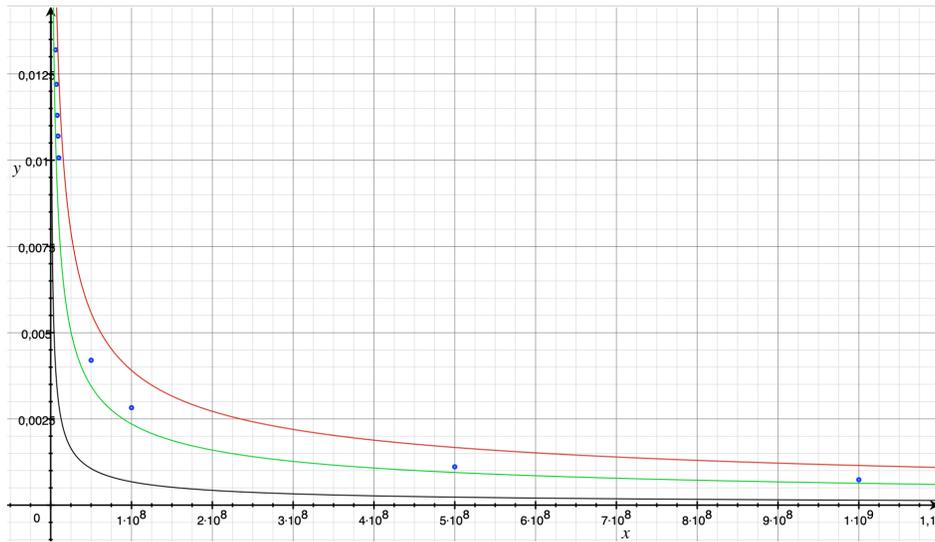
Regardons le comportement des fonctions quand x devient plus grand dans la figure $B.2$

3. Ces formules ont été trouvées par interpolation sur des points allant jusqu'à 10^9



Densité d'entiers 50-friables sur $[1, x]$ avec $x \in [1; 10^5]$

FIGURE 5.1 – Suite de points représentant la densité d'entiers 50-friables compris entre 1 et x , avec la densité en ordonné. Par exemple, on voit qu'il y a environ 25% d'éléments 50-friables sur l'intervalle $[1, 10^4]$



Densité d'entiers 50-friables sur $[1, x]$ avec $x \in [1; 10^9]$

FIGURE 5.2 –

La conclusion est donc qu'il faudrait choisir une base B différente suivant la taille de n . Effectivement, on peut approximativement estimer la taille de p en s'intéressant à la taille de \sqrt{n} . Ainsi, on aurait, un ordre d'idée de la probabilité pour que $p - 1$ soit B -friable avec la relation :

$$\Psi(\sqrt{n}, B) = \left(\frac{\ln(\sqrt{n})}{\ln(B)} \right)^{\frac{\ln(\sqrt{n})}{\ln(B)}}$$

Après avoir vu la théorie, voyons à présent l'algorithme :

Algorithme 5.1.0.1. [P-1 POLLARD]

Entrée : n un entier et B une borne

Sortie : d un diviseur de n

1. on tire a aléatoirement sur $[2, n - 1]$
2. $d \leftarrow \text{pgcd}(a, n)$
3. si $d \neq 1$:
4. retourner d
5. pour q allant de 2 à B
6. $a \leftarrow a^q \pmod n$
7. retourner $\text{pgcd}(a - 1, n)$

La complexité de $P - 1$ est $O(B \times \log(B) \times (\log(N))^{1+\epsilon})$ avec B la borne choisie et N l'entier à factoriser.

5.2 Méthode $p + 1$ de Williams

C'est une variante de la méthode $p - 1$ de Pollard, qui utilise le Tore de Lucas pour réaliser une factorisation rapide si un facteur $p \in \mathbb{N}$ a une décomposition de $p + 1$ en petits facteurs premiers.

Tout d'abord, nous allons nous intéresser à la définition Tore de Lucas puis au théorème qui en découle.

Définition 5.2.1 (Tore de Lucas). *Soit un groupe noté $T_D(k)$ avec K un corps. Ce groupe nommé le Tore de Lucas est défini par l'équation $\{(x, y) \in K^2 \text{ tel que } x^2 - Dy^2 = 1\}$, le 1 est le 1 du corps K . La loi $*$ sur $T_D(K)$ est :*

$$(x_1, y_1) * (x_2, y_2) = (x_1x_2 + Dy_1y_2, x_1y_2 + x_2y_1)$$

avec l'opposé de $(x_1, y_1) = (x_1, -y_1)$ et son neutre $(1, 0)$.

Théorème 5.2.1. Soit p premier impair et D qui est non nulle et n'est pas un carré dans \mathbb{F}_p , il y a $p + 1$ couples $(x, y) \in \mathbb{F}_p^2$ tel que $x^2 - D \times y^2 = 1$.

Remarque 5.2.1. Si on prend un n entier impair et D quelconque, on peut définir $T_D(\mathbb{Z}/n\mathbb{Z})$. De plus si $n = a \times b$ et leur $\text{pgcd}(a, b) = 1$ alors on a :

$$T_D(\mathbb{Z}/n\mathbb{Z}) \simeq T_D(\mathbb{Z}/a\mathbb{Z}) \times T_D(\mathbb{Z}/b\mathbb{Z})$$

La méthode $p + 1$ est un algorithme de factorisations d'entiers ressemblant à l'algorithme $p - 1$ de Pollard décrit au 5.1. Le but de la méthode de $p + 1$ est de trouver des factorisations différentes à la méthode $p - 1$ de Pollard.

Algorithme 5.2.0.1. [$p + 1$ DE WILLIAMS]

Entrée : n un entier

Sortie : d un diviseur de n

1. On Choisit B
 2. On Définit un point $u = (x, y)$ tel que $x \in [1, n]$ et $y = 1 \pmod n$.
 3. On Calcule $D = (x^2 - 1)/y^2$
 4. Pour k allant de 2 à B
 5. On Remplace u par u^k dans le Tore de Lucas.
 6. On calcule $\text{pgcd}(n, y)$
 7. Si $d \neq 1$ on obtient un diviseur non trivial de n
 8. On relance avec B plus grand
-

Remarque 5.2.2. Pour que cet algorithme soit plus intéressant, il faut qu'il soit fait simultanément avec l'algorithme $p - 1$ de Pollard. Certes, en appliquant les deux algorithmes, ça serait très coûteux mais négligeable par rapport à son efficacité.

5.3 Courbes elliptiques

En 1985, Hendrik Lenstra mis au point un algorithme de factorisations qui se base sur les courbes elliptiques. Celui-ci a la particularité de permettre de trouver les petits facteurs premiers d'un nombre. De plus, là où la plupart des algorithmes de factorisations dépendent de la taille du nombre à factoriser celui-ci dépend principalement de la taille du facteur à trouver (comme P-1).

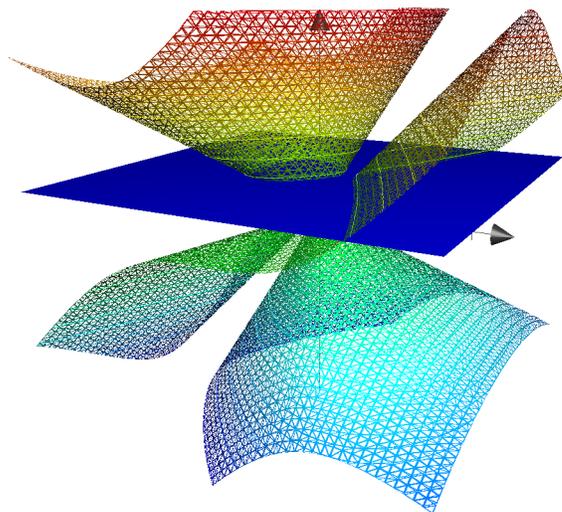
5.3.1 Définitions et théorèmes

Définition 5.3.1. Soit A un anneau (typiquement $A = \mathbb{Z}/n\mathbb{Z}$).

On appelle plan projectif sur A , noté $\mathbb{P}^2(A) = E/\sim$, où E est l'ensemble des triplets (x, y, z) dans A premiers avec n et où la relation d'équivalence \sim est définie par $(x, y, z) \sim (ux, uy, uz)$ pour tous $x, y, z \in A$ et $u \in A^*$:

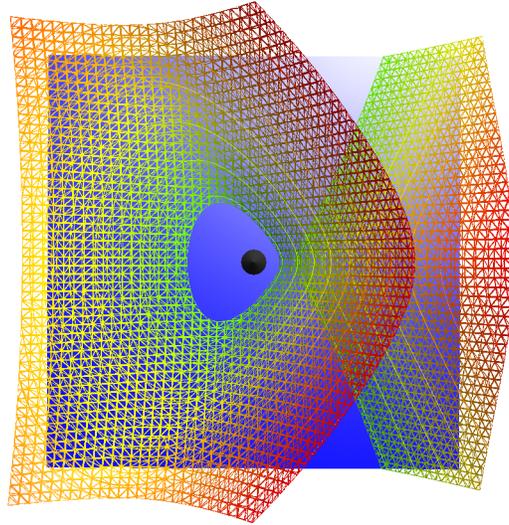
$$\mathbb{P}^2(A) = \{(x, y, z) \in A \mid \text{pgcd}(x, y, z, n) = 1\} / \sim$$

Remarque 5.3.1. L'ensemble E étant quotienté par cette classe d'équivalence, tout triplet (x, y, z) peut être représenté de façon unique par $(\frac{x}{z}, \frac{y}{z}, 1)$, si z est inversible dans A . Ainsi, tous les points équivalents à (x, y, z) sont situés sur la droite passant par l'origine et un point (représentant) du plan d'équation $z = 1$.



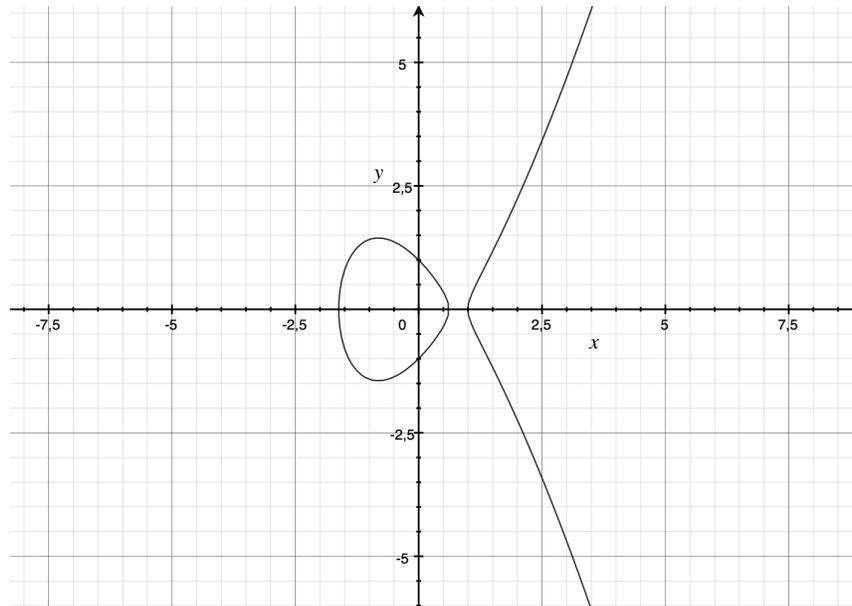
Graphes de la fonction $zy^2 = x^3 + axz^2 + bz^3$ avec $a = -2$ et $b = 1$ en trois dimensions avec, en bleu, le plan d'équation $z = 1$

FIGURE 5.3 – Graphe en 3D



Graphe de la fonction $zy^2 = x^3 + axz^2 + bz^3$ avec $a = -2$ et $b = 1$ en trois dimensions vu de l'axe z .

FIGURE 5.4 – Graphe en 3D



Graphe de la fonction $zy^2 = x^3 + axz^2 + bz^3$ avec $a = -2$, $b = 1$ et $z = 1$ en deux dimensions.

FIGURE 5.5 – Graphe en 2D

Définition 5.3.2. Soit A un anneau dans lequel 6 est inversible. Soient a et b deux éléments inversibles de A .

On définit une courbe elliptique sur A par l'équation (dite de Weierstrass) suivante :

$$E : \{(x, y, z) \in \mathbb{P}^2(A) \mid y^2 z = x^3 + axz^2 + bz^3\}$$

Notons que l'on définit ici des équations de courbes elliptiques sur des corps de caractéristiques différentes de 2 et 3 en raison de leur complexité.

Proposition 5.3.1. Soit K un corps de cardinal p et E une courbe elliptique définie sur K . On définit l'addition sur la courbe elliptique E de la manière suivante :

Soient P, Q deux points de E (P et Q de coordonnées respectives (x_P, y_P, z_P) et (x_Q, y_Q, z_Q)).

Sur la représentation graphique de la courbe elliptique (sur le plan d'équation $z = 1$), on note O le point de coordonnées $(0, 1, 0)$ du graphe ($x = +\infty$).

La droite passant par P et Q coupe la courbe E en un troisième point R .

Enfin, la droite passant par O et R recoupe la courbe en un point R' .

On a alors les équations suivantes :

$$P + Q + R = O$$

$$P + Q = R'$$

$$R + R' = O$$

$$O + O = O = -O$$

Ainsi, si R est de coordonnées $(u, v, 1)$ alors $P + Q$ est de coordonnées $(u, -v, 1)$.

On définit ainsi l'addition entre deux points P et Q de la courbe de la manière suivante :

$$P + Q = R'$$

D'un aspect plus algébrique, on définit la loi d'addition de la manière suivante sur une courbe elliptique E de caractéristique p :

Soit P, Q, R trois points de la courbe E tels que $P + Q = R$ de coordonnées respectives $(x_P, y_P), (x_Q, y_Q), (x_R, y_R)$.

On a les relations suivantes :

- Si $P = Q = O$ alors $R = O$
- Si $Q = O$ alors $P = R$
- Si $x_P = x_Q$ et $y_Q = -y_P$ alors $R = O$

- Si $x_P \neq x_Q$ alors
$$\begin{cases} x_R = \lambda^2 - x_P - x_Q \text{ mod } p \\ y_R = \lambda \times (x_P - x_R) - y_P \text{ mod } p \\ \lambda = (y_Q - y_P) \times (x_Q - x_P)^{-1} \text{ mod } p \end{cases}$$

- Si $P = Q$ et $y_P \neq 0$ alors
$$\begin{cases} x_R = \lambda^2 - 2 \times x_P \text{ mod } p \\ y_R = \lambda \times (x_P - x_R) - y_P \text{ mod } p \\ \lambda = (3x_P^2 + a) \times (2 \times y_P)^{-1} \text{ mod } p \end{cases}$$

Ce dernier point correspond à la multiplication du point P par 2.

Voici un schéma illustrant l'addition dans E :

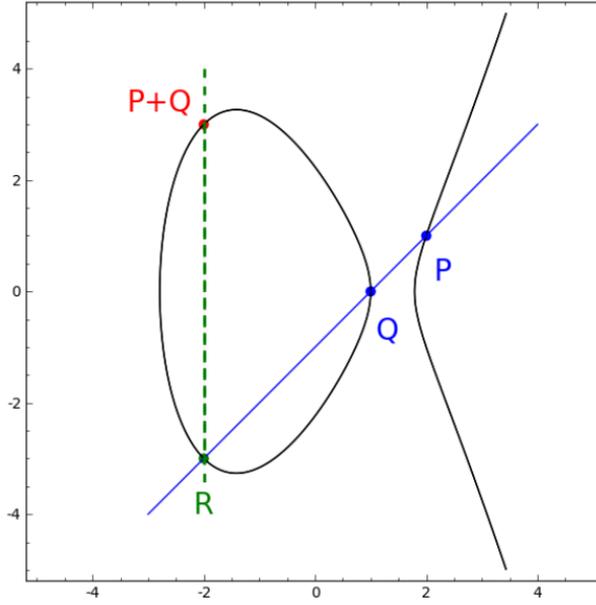


FIGURE 5.6 – Addition sur les courbes elliptiques

Remarque 5.3.2. Si $A = \mathbb{Z}/n\mathbb{Z}$ n'est pas un corps, alors on peut quand même définir la loi de groupe qu'on va utiliser sur une courbe elliptique E sur A . Si A n'est pas un corps n est factorisable, les formules qui seront prises en défaut nous permettrons alors de trouver un facteur de n .

Proposition 5.3.2. Soit $n = pq$. L'application $E(\mathbb{Z}/n\mathbb{Z}) \rightarrow E(\mathbb{Z}/p\mathbb{Z}) \times E(\mathbb{Z}/q\mathbb{Z})$ est bijective.

Théorème 5.3.1 (Hasse). Soit p un nombre premier, E une courbe elliptique définie par $E(\mathbb{Z}/p\mathbb{Z})$. Alors on a :

- $\#E(\mathbb{Z}/p\mathbb{Z}) \in [p + 1 - \sqrt{2p}; p + 1 + \sqrt{2p}]$
- En choisissant a et b aléatoirement, le cardinal parcourt tout l'intervalle précédent.

5.3.2 Algorithme de Lenstra

Nous allons maintenant nous intéresser à l'algorithme de *Lenstra*, une amélioration de *Pollard* pour la décomposition en facteurs premiers (qui utilise les notions de courbe elliptique) Notons qu'il s'agit d'un algorithme probabiliste.

Soit n l'entier à factoriser. Contrairement à l'algorithme de *Pollard*, il ne s'agit pas ici d'augmenter la valeur de k de sorte à avoir $p - 1|k$.

Au lieu de cela, on choisit deux entiers a et b de sorte que le discriminant $4a^3 + 27b^2$ soit inversible dans l'anneau $\mathbb{Z}/n\mathbb{Z}$, puis on construit ainsi la courbe elliptique E ainsi qu'un multiple kP d'un point $P \in E$.

Si $\#E(\mathbb{F}_p) \mid k$ alors on a $kP = O \pmod{p}$ et on aura obtenu ainsi un facteur de n .

Si $\#E(\mathbb{F}_p) \nmid k$, au lieu d'augmenter la valeur de k , on choisit une courbe elliptique différente $E'(\mathbb{F}_p)$ de sorte qu'on ait $\#E'(\mathbb{F}_p) \neq \#E(\mathbb{F}_p)$.

Lorsque $E(\mathbb{F}_p) \mid k$, $kP = O \pmod{p}$. Ainsi, p divise le dénominateur d de kP .

Puisque $p \mid d$ et $p \mid n$, on a que $p \mid \text{pgcd}(d, n)$ donc $\text{pgcd}(d, n) > 1$ et d n'est pas inversible dans $\mathbb{Z}/n\mathbb{Z}$.

Remarque 5.3.3. Avant de commencer l'algorithme, il faut s'assurer que l'anneau sur lequel on va construire les différentes courbes elliptiques n'est pas de caractéristique 2 ou 3.

Algorithme 5.3.2.1. [LENSTRA]

Entrée : n un entier et B un entier

Sortie : d un diviseur de n

1. Vérifier que n n'est divisible ni par 2 ni par 3
2. "Vérifier que n n'est pas une puissance d'un nombre premier"
3. Choisir aléatoirement $a, x_P, y_P \in \llbracket 1, n \rrbracket$
4. On calcule $b = y_P^2 - x_P^3 - a \times x_P$
5. On calcule $\text{gcd} = \text{pgcd}(4a^3 + 27b^2, n)$
6. Si $\text{gcd} = n$
7. Retourner $\text{Lenstra}(n, B)$
8. Si $\text{gcd} \neq 1$
9. Retourner gcd
10. E la courbe elliptique d'équation $y^2 = x^3 + ax + b$
11. Pour ν allant de 2 à B faire :
12. Calculer $P = \nu \times P$ sur $\mathbb{Z}/n\mathbb{Z}$
13. Si le calcul n'aboutit pas :
14. Retourner $\text{PGCD}(\omega, n)$ avec ω l'entier non inversé
15. Retourner $\text{Lenstra}(n, B)$

La complexité de l'algorithme dépend de la taille du facteur à trouver ; elle peut être exprimée par $O(e^{(\sqrt{2}+o(1))\sqrt{\ln(p) \times \ln(\ln(p))}})$ où p est le plus petit facteur de n .

Chapitre 6

Analyse de Complexité

6.1 Complexité

La complexité est définie comme le nombre d'opérations élémentaires nécessaires pour résoudre un problème posé. Comme il a été énoncé dans la première partie, la notion de complexité est très importante car c'est bien cela qui va nous permettre de savoir si l'algorithme que l'on prévoit d'implémenter est utilisable ou non en fonction de son entrée. De plus, elle permet de comparer deux méthodes afin de définir laquelle est la plus efficace en terme de temps et de calculs.

6.1.1 La notation L

Définition 6.1.1. On définit, pour toute constante $c > 0$ et pour tout $\alpha \in [0, 1]$ la fonction suivante :

$$L_x(\alpha, c) = e^{((c+o(1))(\ln x)^\alpha (\ln \ln x)^{1-\alpha})}$$

Remarque 6.1.1. Voyons deux cas particuliers :

1. $\alpha = 0$: $L_x(0, c) = e^{((c+o(1))(\ln \ln x))} = (\ln x)^{(c+o(1))}$
2. $\alpha = 1$: $L_x(1, c) = e^{((c+o(1))(\ln x))} = x^{(c+o(1))}$

Dans le premier cas, on parle de complexité polynomiale, dans le second, de complexité exponentielle et pour $0 < \alpha < 1$ on parle de complexité sous-exponentielle. Ainsi, α permet de classer les algorithmes ayant une complexité en $L_x(\alpha, c)$.

6.1.2 L et la friabilité

Nous avons introduit et vu plus haut la notion de friabilité. Maintenant que nous avons défini la fonction $L_x(\alpha, c)$ voyons un corollaire intéressant dont nous nous servirons plus tard.

Théorème 6.1.1. *Un entier aléatoire de taille $L_x(\alpha, c)$ est $L_x(\beta, c')$ -friable avec une probabilité $L_x(\alpha - \beta, -\frac{c}{c'}(\alpha - \beta) + o(1))$ quand x tend vers l'infini.*

6.1.3 Le choix de B dans les courbes elliptique

Dans cette section, nous allons chercher à optimiser le choix de K pour minimiser le temps de calcul. Dans le pseudo code de Lenstra, B est la borne de friabilité.

Pour que l'algorithme fonctionne, il faut que le cardinal de la courbe elliptique (choisie aléatoirement) E soit B -friable modulo P (un diviseur premier de n) .

On pose $B = L_n(\beta, c')$ avec $\beta \in]0; 1[$.

Pour commencer, on pose P la probabilité de succès d'une étape.

On a alors :

$$P = L_n(\alpha - \beta, -\frac{c}{c'}(\alpha - \beta)) \text{ d'où } P = L_n(1 - \beta, -\frac{1}{2c'}(1 - \beta))$$

Le nombre moyen d'étapes nécessaires est donc :

$$P^{-1} = L_n(1 - \beta, \frac{1}{2c'}(1 - \beta)) = \exp\left(\frac{1}{2c'}(1 - \beta)(\ln(n))^{1-\beta}(\ln(\ln(n)))^\beta\right)$$

On fait B calculs sur la courbe elliptique car on multiplie successivement le point P jusqu'à obtenir $B! \times P$.

$$P \times 2P \times 3P \times \dots \times BP$$

De ce fait, le temps de calcul est :

$$B \times O(\tau)$$

avec τ le temps d'une étape. On notera que le temps d'une étape est négligeable devant B .

Le temps nécessaire total est alors :

$$\begin{aligned} T &= P^{-1} \times B \\ &= \exp\left(\frac{1}{2c'}(1 - \beta)(\ln(n))^{1-\beta}(\ln(\ln(n)))^\beta\right) \times \exp\left(c'(\ln(n))^\beta(\ln(\ln(n)))^{1-\beta}\right) \end{aligned}$$

Pour minimiser le temps de calcul T , on veut minimiser la valeur de P^{-1} et la valeur de B . Le terme $\ln(x)$ étant le terme dominant dans les deux cas, on veut minimiser $1 - \beta$ et β , donc minimiser $\max(1 - \beta, \beta)$. On prend donc $\beta = \frac{1}{2}$ et on obtient :

$$T = \exp\left(\left(\frac{1}{4c'} + c'\right)\sqrt{\ln(n)\ln(\ln(n))}\right)$$

Maintenant, minimisons T par rapport à c' : $\min(\frac{1}{4c'} + c') = 1$ qui est atteint lorsque $c' = \frac{1}{2}$.

On a donc :

$$- B = L_n(\frac{1}{2}, \frac{1}{2}) = \exp\left(\frac{1}{2} \times \sqrt{\ln(n)} \times \sqrt{\ln(\ln(n))}\right)$$

$$- T = \exp\left(\left(\frac{1}{4c'} + c'\right) \sqrt{\ln(n)} \times \ln(\ln(n))\right)$$

Chapitre 7

Implémentation

Le problème de la factorisation est un problème que l'on sait dans NP . De ce fait, il est difficile de trouver un algorithme efficace qui factorise des entiers. Cependant, même si l'algorithme n'est pas rapide, il reste néanmoins essentiel de savoir factoriser, notamment en cryptographie.

7.1 Ossature de l'algorithme et fonctions phares

L'idée pour notre algorithme de factorisation est la suivante :

D'où les fonctions phares suivantes :

- Une fonction qui enlève les petits facteurs premiers.
- Une fonction qui teste la primalité d'un entier.
- Une fonction qui teste si un nombre est une puissance..
- Deux fonctions qui trouvent un facteur et le renvoient.

La première question est la suivante : Comment peut-on, en C , gérer des grands nombres ?

7.2 Le problème des grands nombres

La première question est la suivante : Comment peut-on, en C , gérer des grands nombres ?

En effet, dans ce langage, les entiers sont stockés sur 8, 16, 32 et 64 bits. Les *"int"* sont codés sur 16 ou 32 bits, ce qui est beaucoup trop petit car le maximum que l'on peut manipuler sans altérer sa valeur est $32767 (= 2^{15} - 1)$ ou $2147483647 (= 2^{31} - 1)$. Les *"long int"* qui sont sur 32 bits avec une limitation à $2^{31} - 1$, cela reste trop petit. Les *"unsigned int"* sans le signe qui eux peuvent aller jusqu'à $2^{32} - 1$, c'est mieux mais toujours insuffisant. Les plus grands entiers sont les *"long long int"*, ils sont stockés sur 64 bits d'où une limite à $2^{64} - 1 (= 18446744073709551615)$, ceux-là sont grands mais restent petits à côté d'entiers de l'ordre de 10^{30} . Effectivement, l'objectif de l'algorithme est de factoriser des entiers de 60 chiffres, soit de l'ordre de 10^{60} .

A partir de ce postulat, deux options s'offrent à nous :

- Créer une structure qui stocke des entiers sur 512 bits et redéfinir toutes les opérations arithmétiques.
- Chercher une bibliothèque *"libc"* qui permet de faire ce que l'on souhaite.

Fort heureusement, une telle bibliothèque existe : *"GMP"*.

7.3 Une bibliothèque de cryptographie : GMP

"GMP" est le sigle pour *"GNU Multiprecision Library"*. Aussi connu sous le nom de *"GNU MP"* est une *"libc"* spécialisée dans le calcul de précision sur des nombres entiers, rationnels ainsi que flottants. Pour nous, seule la partie nombres entiers nous intéresse. La première version de *"GMP"* a été créée en 1992 par Torbjörn Granlund et est maintenue régulièrement à jour. Elle est utilisée dans les premières versions du projet *"SageMath"*

La bibliothèque est utilisée dans de nombreux domaines notamment la recherche, la cryptographie et les systèmes de calcul formel.

Grâce à elle, nous allons pouvoir stocker et manipuler des entiers aussi grands que nous le souhaitons (dans les limites de stockage de l'ordinateur).

Voilà une liste de quelques fonctions que nous avons utilisées venant de cette bibliothèque :

- *"mpz_init_set_str"* qui permet d'initialiser et d'allouer un espace mémoire pour stocker un entier de type *"mpz_t"*.
- *"mpz_out_str"* qui affiche un entier de type *"mpz_t"*.

- `"mpz_clear"` permet de libérer la mémoire pour éviter des fuites ou une surcharge.
- `"mpz_cmp_ui"` compare un entier de type `"mpz_t"` avec un entier de type `"int"`.
Une fonction équivalente permet de comparer deux entiers `"mpz_t"`.
- `"mpz_powm_sec"` fait l'exponentiation modulaire.
- `"mpz_urandomm"` renvoie un nombre aléatoire `"mpz_t"`.
- `"mpz_gcd"` qui calcule le *PGCD* de deux nombres `"mpz_t"`.

7.4 Éliminer les petits facteurs

Par trial division, il est assez rapide d'éliminer les petits facteurs premiers. Ainsi, on élimine, grâce à l'appel successif de cette fonction, les 100000 premiers nombres premier qui composeraient éventuellement notre entier.

Pour voir l'implémentation : ?? à la ligne 102 – 125.

7.5 Test de primalité

Nous pouvons utiliser la fonction qui est dans la bibliothèque `"GMP"` : `"mpz_probab_prime_p"`. Celle-ci retourne 2 si n est premier, retourne 1 si n est probablement premier, ou retourne 0 si n est non premier. Cette fonction effectue certaines divisions d'essai, un test de primalité probable Baillie-PSW, puis des tests de primalité probabilistes tels que Miller-Rabin.

La probabilité de se tromper avec nos paramètres est inférieure 4^{-50} .

Le plus gros problème avec ces méthodes c'est qu'elles sont probabilistes. Il est vrai qu'avec une probabilité d'erreur d'environ $7.8e - 31$, se tromper est fortement improbable.

Pour donner un ordre d'idées, la probabilité de gagner au loto est de $5.4e - 9$ et $8.8e - 16$ est la probabilité de faire 50 "piles" d'affiler au "pile ou face". Chacun de ces événements a une probabilité très faible de se produire. Il est donc raisonnable de penser que le cas problématique ne se produira pas.

Pour voir l'implémentation : ?? à la ligne 271 – 276.

7.6 Es-tu un carré ou plus ?

Cet algorithme va pouvoir vérifier si notre nombre est une puissance. Cela va permettre d'enlever certains grands carrés avant de tester avec l'algorithme de Pollard ou des courbes elliptiques.

Dans l'implémentation on retrouve ce test : `"mpz_cmp_ui(rem, 3) >= 0"` les calculs sur les entiers tronqués nous obligent à nous arrêter dès que `"rem"` est plus petit que 3 (strictement).

Pour voir l'implémentation : ?? à la ligne 281 – 309

7.7 Pollard

Voici une implémentation de $P - 1$ Pollard en C . Le but de cet algorithme est d'afficher un facteur de n . Une fois cela fait, on divise n par ce facteur pour pouvoir continuer la décomposition.

Nous avons fait le choix de 25 000 000 comme borne de friabilité car nous voulions un test qui prend au plus 90 secondes (cf : tests 7.10.1).

De plus, nous testons le *PGCD* plus souvent que la méthode standard ce qui nous permet de sortir plus tôt si la friabilité de N est plus petite que B . Mais nous ne le testons pas non plus à chaque fois car le calcul serait trop lourd. Ainsi, avec cette implémentation, nous le testons tous les 1 000 000 soit un total de 25 tests.

On a vraiment essayé d'optimiser $P - 1$ en trouvant un juste milieu entre faire des *PGCD* coûteux à chaque étape qui potentiellement permettent de s'arrêter plus tôt et ne faire le *PGCD* qu'à la fin ce qui n'encombre pas le calcul mais qui oblige à aller jusqu'au bout des multiplications.

Pour voir l'implémentation : ?? à la ligne 222 – 266.

7.8 Courbes elliptiques

La fonction qui retourne un facteur grâce aux courbes elliptiques est une fonction qui demande en premier l'implémentation de sous-fonctions.

7.8.1 Création et destruction de la courbe elliptique

Pour définir notre courbe elliptique, on crée une structure qui va se rappeler de tous les arguments nécessaires.

Pour voir l'implémentation : ?? à la ligne 24 – 63

7.8.2 Choix de la courbe elliptique

On a choisit une fonction récursive qui ne s'arrête que si elle a trouvé un facteur ou une courbe elliptique correcte.

Pour voir l'implémentation : ?? à la ligne 314 – 304.

7.8.3 Structure des points

Pour stocker la valeur des points sur la courbe, on définit une structure de points.

Pour voir l'implémentation : ?? à la ligne 67 – 98

7.8.4 Somme sur la courbe

Il suffit de suivre l'addition théorique en faisant attention à la non inversion possible.

Pour voir l'implémentation : ?? à la ligne 399 – 520.

7.8.5 Produit sur la courbe

Le produit sur la courbe n'est rien d'autre qu'une boucle sur la somme.

Pour voir l'implémentation : ?? à la ligne 524 – 541.

7.8.6 Le choix de B

Soit n l'entier a factoriser et n grand.

Pour B , il faut prendre $B = L_n(\frac{1}{2}, \frac{1}{2}) = \exp(\frac{1}{2} \times \sqrt{\ln(n)} \times \sqrt{\ln(\ln(n))})$ mais il n'existe aucune fonction \ln ni aucune fonction \exp dans GMP.

De ce fait, nous avons posé une condition sur l'entier qui va rentrer dans les courbes elliptique : il doit être plus petit que 10^{110} et nous avons instaurer des paliers grâce aux valeurs théoriques calculées en amont.

Pour voir l'implémentation : ?? à la ligne 551 – 724

7.8.7 Méthode de Lenstra

Une fois toutes les sous fonctions écrite, l’algorithme s’écrit en suivant le modèle théorique.

Pour voir l’implémentation : ?? à la ligne 729 – 775

7.9 Comment les assembler

Maintenant que toutes nos fonctions sont écrite, nous allons pouvoir les mettre ensemble pour crée notre fonction de factorisation.

Pour voir l’implémentation : ?? à la ligne 779 – 859

7.10 Tests et résultats

7.10.1 Tests $P - 1$ Pollard en C

Dans ces tests, l’algorithme de factorisation fonctionne uniquement avec ces fonctions :

- Une fonction qui enlève les petits facteurs premiers.
- Une fonction qui teste la primalité d’un entier.
- Une fonction qui teste si un nombre est une puissance.
- $P - 1$ Pollard.

Le protocole de test est le suivant : On tire un nombre aléatoire entre 2 et i (notre Entrée) puis on essaye de le factoriser (la procédure est chronométrée).

On appelle échec une tentative qui n’a pas abouti, une factorisation partielle compte comme un échec.

Le temps moyen est calculé avec la fonction `time`. On divise alors le résultat obtenu par le nombre de calcul effectués et on obtient un temps en seconde.

Avec nos paramètres de $p - 1$ notamment notre borne de friabilité à 25 000 000, nous obtenons les résultats suivants :

Avec 10^{20} en entrée sur 1500 tests :

Taux échec : 18.6%

Valeur de `time` : 7470 s

Temps moyen : 4.98 s

Avec 10^{30} en entrée sur 1500 tests :
Taux échec : 16.13%
Valeur de **time** : 17580 s
Temps moyen :11.72 s

Avec 10^{40} en entrée sur 1500 tests :
Taux échec : 21.46%
Valeur de **time** : 29044 s
Temps moyen :19.36 s

Avec 10^{50} en entrée sur 1500 tests :
Taux échec : 34.22%
Valeur de **time** : 66677 s
Temps moyen :44,45s

Avec 10^{60} en entrée sur 1500 tests :
Taux échec : 50.4%
Valeur de **time** : 111027 s
Temps moyen :1 min 14 s

Avec 10^{70} en entrée sur 1500 tests :
Taux échec : 60.46%
Valeur de **time** : 174373 s
Temps moyen : 1 min 56 s

Ainsi, avec ces valeurs, nous pouvons dresser un graphique qui représente la taille des entiers donnés en Entrée en fonction du temps d'exécution et du taux d'échec.

Ainsi, on obtient les variations suivantes :

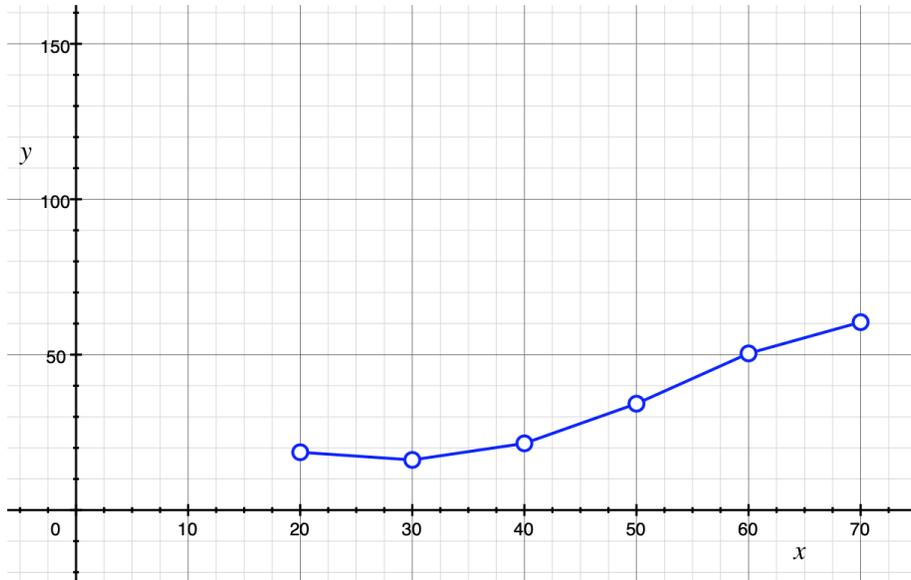


FIGURE 7.1 – Le nombre de chiffre dans l'entier a factoriser en fonction du taux d'échec en pourcent.

On remarque que le taux d'échec augmente plutôt doucement et suis une courbe presque polynomiale.

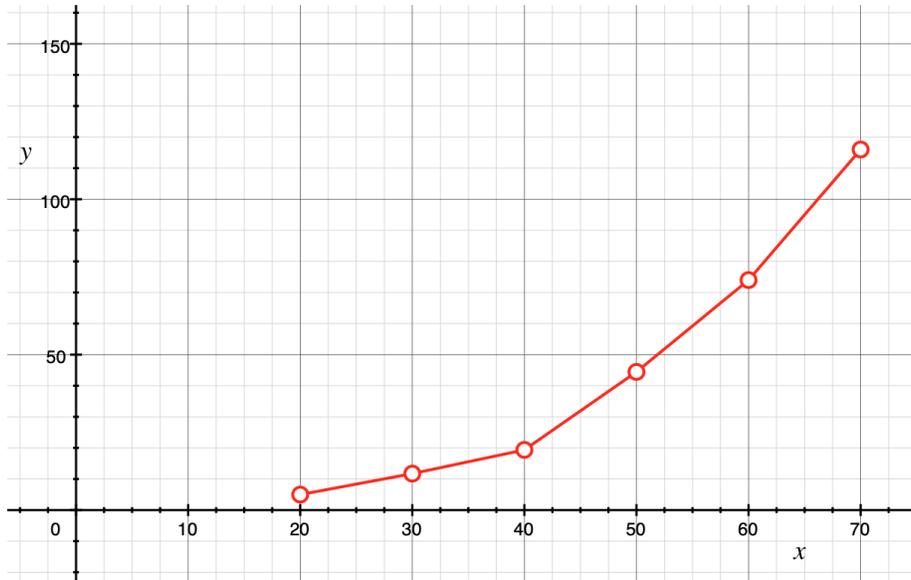


FIGURE 7.2 – Le nombre de chiffres dans l’entier à factoriser en fonction du temps de calcul en seconde.

Ici, on constate que le temps d’exécution est exponentiel par rapport à la taille. L’augmentation du temps s’explique simplement, si un entier ξ est plus grand qu’un autre Ψ , trois cas se présentent :

- L’entier ξ a plus de facteur que l’entier Ψ .
- L’entier ξ a des facteurs plus grands que ceux de l’entier Ψ .
- Les deux entiers vérifient les deux remarques précédentes.

Ainsi, pour le cas 1, on va exécuter l’algorithme plus de fois d’où le temps qui s’en trouve rallongé. Alors que pour le cas numéro 2 l’algorithme va s’exécuter plus longtemps en moyenne d’où une durée totale allongée. Pour le dernier cas, c’est simplement une conjonction des deux premiers (c’est le cas le plus fréquent).

Au vu des résultats, on constate que l’on a un algorithme efficace pour factoriser des entiers de l’ordre de 10^{50} , au-dessus, le taux d’échec est supérieur à 50% mais cela permet de dégrossir réduire les entiers que l’on veut factoriser. Le problème c’est que les grands entiers RSA ont de grandes chances de passer au travers de cet algorithme.

7.10.2 Test de l'algorithme complet en C

Notre algorithme fait environs 100 opérations sur une courbe elliptique par secondes. Sachant que sur chaque courbe, on fait B opérations et que l'on regarde au plus B courbes par facteur. Il est possible de n'en trouver aucun. Regardons le pire des cas, n 10^{110} non B -friable ce qui nous donne un temps de calcul de 374h/courbe donc un temps de calcul total de 5774043 années pour un résultat négatif.

Essayons de factoriser des entiers sur le même protocole que précédemment :

Avec 10^{10} en entrée sur 1500 tests :

Taux échec : 0%

Valeur de `time` : 1601 s

Temps moyen :1,06 s

Avec 10^{20} en entrée sur 70 tests :

Taux échec : 2.9%

Valeur de `time` : 1757 s

Temps moyen :25.1 s

Avec 10^{30} en entrée sur 70 tests :

Taux échec : 2.8%

Valeur de `time` : 65481 s

Temps moyen :15 m 35 s

Pour la suite des tests, le pire des cas est un calcul d'environ quatre-vingts jours. L'existence de ce cas rend des tests de ce genre impossible ou très coûteux en temps.

Cependant, on remarque, comme espéré, une diminution drastique du taux d'échec en échange d'un temps beaucoup plus grand. Le problème du temps peut être amélioré avec une machine plus performante. Malgré cela, même avec une machine qui va cent fois plus vite, il existe un palier au delà duquel, le pire cas est catastrophique en terme de temps. Pour la machine cent fois plus rapide, ce pallier serait 10^{55} pour environ soixante cinq jours de calculs.

Voici un tableau du temps de calcul théorique du "pire cas" avec nos observations :

Ordre de grandeur de l'entier	Temps de calcul théorique
10^{10}	5 min 50 s
10^{15}	15 min 40 s
10^{20}	1 h 47 min 48 s
10^{25}	12 h 2 min 55 s
10^{30}	3 j 2 h 35 min 20 s
$> 10^{35}$	> 17 j 1 h 20 min 10 s

FIGURE 7.3 – Temps de l'algorithme en C pour trouver un facteur dans le pire cas.

Pour les calculs ci-dessus, on a considéré que toutes les fonctions qui ne font pas parties des courbes elliptiques prennent au total 5 min à s'exécuter. Comme ça, on a une majoration du temps de calcul. Avec ces calculs, il est flagrant qu'il est déraisonnable de tenter de factoriser un entier plus grand que 10^{35} .

Pour être plus précis, il est déraisonnable que la partie courbe elliptique de notre programme travaille avec un entier plus grand que 10^{35} . Il est parfaitement possible de factoriser un entier plus grand rapidement s'il a plein de petits facteurs qui une fois enlevés donne un entier de taille raisonnable ($< 10^{30}$). L'autre possibilité, c'est qu'on soit "chanceux" et que le plus petit facteur moins un soit petit friable.

7.10.3 Tests complémentaires

Ici, nous avons décidé de tester notre algorithme sur des entiers qui sont produit de deux grand facteur premier.

Entier	Facteur 1	Facteur 2	C global
3937	31	127	1.012 s
1040257	127	8191	1.009 s
34370773	7559	4547	1.025 s
1920234803	38569	49787	1.084 s
1073602561	8191	131071	1.106 s
12993308117	105751	122867	1.196 s
68718821377	131071	524287	1.485 s
181206278419	327011	554129	1.689 s
836375980349731	13321823	62782397	9.059 s
1125897758834689	2147483647	524287	1.412 s
4951760154835678088235319297	2305843009213693951	2147483647	2h 11m 49.472 s
296275573855312647099835767421	599615272526051	494109452227871	2j 15h 10m 46.56 s

FIGURE 7.4 – Temps de l'algorithme en C pour factoriser des entiers spécifiques.

A titre de comparaison, voici les valeurs de sage avec un algorithme ECM :

Entier	Facteur 1	Facteur 2	ECM sage
3937	31	127	0,9 ms
1040257	127	8191	1,37 ms
34370773	7559	4547	4,25 ms
1920234803	38569	49787	18,21 ms
1073602561	8191	131071	13,53 ms
12993308117	105751	122867	36,34 ms
68718821377	131071	524287	62,63 ms
181206278419	327011	554129	91,85 ms
836375980349731	13321823	62782397	1,193 s
1125897758834689	2147483647	524287	0,693 s
4951760154835678088235319297	2305843009213693951	2147483647	6 min 43 s
296275573855312647099835767421	599615272526051	494109452227871	1h 25min 53s

FIGURE 7.5 – Temps de Lenstra en sage pour factoriser des entiers spécifiques.

Chapitre 8

Conclusion

La factorisation est un problème difficile. Nous avons montré qu'il existe plusieurs méthodes pour factoriser un entier mais aucune n'est réellement efficace. Pour casser des systèmes cryptographiques basés sur le problème de la factorisation, il faudra trouver des méthodes différentes et plus efficaces.

Comme nouvelles méthodes plus efficaces, nous pouvons citer par exemple l'algorithme basé sur le crible quadratique.

La dernière solution et la plus répandue est celle de l'ordinateur quantique qui sera un jour capable de résoudre ce type de problème en un temps record.

8.1 Bibliographie

- Cours de cryptographie par Gilles ZEMOR
- Cours de cryptologie par Jean Paul Cerri
- H. Williams, A $p + 1$ method of factoring, Mathematics of Computation
- Factorisation : algorithmes et implémentations (Introduction au domaine de recherche) - Cyril Bouvier [2011]
- <http://www.bibmath.net/crypto/index.php?action=affiche&quoi=complements/factoelliptique>
- <https://www.math.u-bordeaux.fr/~ajehanne/MasterCSI/ACF.pdf>
- <https://www.math.u-bordeaux.fr/~jcouveig/cours/factoring.pdf>
- <http://math.univ-lyon1.fr/~wagner/coursDelaunay.pdf>
- https://github.com/ashutosh1206/Crypton/tree/master/RSA-encryption/Factorisation-Pollard%27s_p-1
- https://fr.wikipedia.org/wiki/Algorithme_p-1_de_Pollard
- https://fr.wikipedia.org/wiki/Entier_friable
- <https://gmplib.org/gmp-man-6.0.0a.pdf>
- https://fr.wikibooks.org/wiki/Programmation_C/Types_de_base
- <https://www.math.u-bordeaux.fr/~jcouveig/cours/grenoble3.pdf>
- https://fr.wikipedia.org/wiki/GNU_MP
- <http://doc.sagemath.org>
- <https://www.math.u-bordeaux.fr/~jcouveig/cours/Z.pdf>
- http://docnum.univ-lorraine.fr/public/DDOC_T_2015_0065_JELJELI.pdf
- https://github.com/delta003/lenstra_algorithm/blob/master/Parker%20-%20Elliptic%20Curves%20and%20Lenstra's%20Factorization.pdf
- <https://gmplib.org/manual/Number-Theoretic-Functions.html>
- <http://webmath.univ-rennes1.fr/master/master2/textes/legeay.pdf>
- <https://www.apprendre-en-ligne.net/crypto/moderne/elliptique.html>
- <https://nitaj.users.lmno.cnrs.fr/Introdelliptic.pdf>
- https://en.wikipedia.org/wiki/Lenstra_elliptic-curve_factorization
- <https://cryptobourrin.wordpress.com/2018/10/22/factorisation-par-courbe-elliptique/>
- <https://members.loria.fr/pzimmermann/records/ecm/params.html>

Annexe A

Le code

A.1 Python

A.1.1 Pollard python

```
1 def pollard(n,B):
2     a=randint(2,n-1)
3     d=gcd(a,n)
4     if d!=1:
5         return d
6     for q in range(2,B+1):
7         a=pow(a,q,n)
8         d=gcd(a-1,n)
9         if d!=1 and d!=n:
10            return d
11    return gcd(a-1,n)
```

A.1.2 ECM python

```
1 def addPQ(P,Q,a,n):
2     xp,yp=P
3     xq,yq=Q
4     if P==(0,0):
5         #on a P=0
6         return Q
7     if Q==(0,0):
8         #on a Q=0
9         return P
10    if (xp==xq and yq==yp):
11        #on a P=Q
12        return (0,0)
13    if P==Q:
14        #on a P=Q
15        #on verifie si 2yp n'est pas inversible mod n
16        var=(2*yp)
17        if gcd(var,n)!=1:
18            return (-1,var)
19        #sinon on l'inverse
20        ivar=xgcd(var,n)[1]
21        #on calcule lambda
22        lam=((3*xp^2+a)*ivar)%n
23    else:
24        #on a P!=Q
25        #on verifie si (xq-xp) n'est pas inversible mod
26        var=(xq-xp)%n
27        if gcd(var,n)!=1:
28            return (-1,var)
29        #sinon on l'inverse
30        ivar=xgcd(var,n)[1]
31        #on calcule lambda
32        lam=((yq-yp)*ivar)%n
33    xr=(lam^2-xp-xq)%n
34    yr=(lam*(xp-xr)-yp)%n
35    return (xr,yr)
36
37 def Lenstra_machine(n,B,e,c):
38     a=randint(2,n-1)
39     xp=randint(2,n-1)
40     yp=randint(2,n-1)
41     b=(yp^2-xp^3-a*xp)
42     g=gcd(4*a^3+27*b^2,n)
43     if g==n:
44         return Lenstra_machine(n,B,e,c+1)
```

```

45     if g!=1:
46         return g,e,c
47     #boucle principale
48     (xr,yr)=(xp,yp)
49     (xnr,ynr)=(xp,yp)
50     for v in range(2,B):
51         for i in range(1,v):
52             (xnr,ynr)=addPQ((xr,yr),(xnr,ynr),a,n)
53             e=e+1
54             if xnr==-1:
55                 return gcd(ynr,n),e,c
56             (xr,yr)=(xnr,ynr)
57     return Lenstra_machine(n,B,e,c+1)
58
59 def Lenstra(n):
60     if is_prime(n):
61         return 'prime',0,0
62     if n%2==0:
63         return 2,0,0
64     if n%3==0:
65         return 3,0,0
66     B=exp((1/2)*sqrt(ln(n))*sqrt(ln(ln(n))))
67     #B=(ln(n)/ln(2))
68     #B=(ln(n)/ln(10))
69     B=int(B)
70     return Lenstra_machine(n,B,0,0)

```