Projet de Mathématiques :

Factorisation en nombre premiers

Réaliser par Yoann COUTURIER, Jeremy GALLAND, Prousoth KARTHIGESU et Tom TEA

Sommaire

Introduction

Dans le cadre du sixième semestre de notre formation postbac, correspondant au deuxième semestre de notre formation en cycle ingénieur à l'EFREI Paris et au cours du module de formation en mathématiques, nos enseignants nous ont proposé de réaliser, en projet de groupe, un programme dans le langage de notre choix, permettant de réaliser une décomposition en nombre premiers d'un nombre choisi le plus grand possible. Le programme devra ainsi permettre de comparer différentes méthodes choisies afin de déterminer quelle méthode est la plus efficace. Ce projet a pour but de conforter nos connaissances acquises lors des différents cours magistraux dispensés à l'EFREI par nos enseignants, mais également de permettre à un chercheur intéressé par comprendre les différents algorithmes de factorisation et leur efficacité, de pouvoir utiliser les résultats de nos travaux.

Ainsi, nous pouvons nous demander comment comparer différents algorithmes de décomposition en nombres premiers afin d'en évaluer l'efficacité.

Pour résoudre ce problème, nous verrons dans un premier temps quelques méthodes utiles pour différents algorithmes avant de nous intéresser dans une deuxième partie aux différents algorithmes de décomposition en nombres premiers et enfin à la comparaison des différents algorithmes dans une troisième et dernière partie.

Quelques fonctions utiles

Dans un premier temps, nous nous proposons de présenter quelques fonctions utiles à l'utilisation des algorithmes de décomposition en nombres premiers et qui seront communes à tous les algorithmes. En effet, ces fonctions sont particulièrement essentielles car elles permettent de déterminer si un nombre est premier ou non.

➢ Nombre premier

La première fonction que nous allons aborder afin de déterminer si un nombre est premier est relativement basique. Cette fonction permet de prendre en entrée un nombre quelconque et de retourner en sortie si ce nombre est premier ou non tel que la variable booléenne « True » signifie que le nombre est premier et la variable « False » signifie qu'il ne l'est pas. Ainsi, cet algorithme permet de tester des nombres compris entre 0 et 2 de manière très rapide, sans avoir à utiliser une méthode plus complexe et plus longue à exécuter.

Ainsi, nous pouvons présenter ci-dessous l'algorithme de cette fonction :

```
Entrée : p un entier
Sortie : retourne Vrai si le nombre est premier

Si p = 2 :
        retourner Vrai
Si p = 1  ou p mod 2 = 0 :
        retourner Faux

retourner Rabin Miller(p,itérations)
```

Dans le cas où aucune des conditions n'est vérifiée, alors l'algorithme va faire appel à une seconde fonction que nous allons décrire dans la seconde partie.

➢ Rabin Miller

La seconde fonction que nous allons aborder est le test de primalité de Rabin-Miller. Ce test permet de déterminer si un nombre est premier ou bien si ce nombre est composé de plusieurs nombres premiers, c'est-à-dire qu'il n'est pas premier. Ce test est effectué pour les nombres supérieurs ou égaux à 3 si la première fonction n'a pas donné de résultats.

Ainsi, nous pouvons présenter ci-dessous l'algorithme de cette fonction :

```
Entrée : p un entier impair > 3, iterations un entier
Sortie : Si le nombre p est premier ou non

r=0
s=p-1
Premier = Vrai

// On compte combien de fois s est divisible par 2
Tant que s mod 2 = 0 :
        r=r+1
        s= s/2

Pour i allant de 0 à itérations :
        a= entier aléatoire entre 2 et n-2
        x = a^s mod p

        Si x = 1 ou x = p-1 : continuer
        Pour j allant de 1 à r-2 :
                x = x² mod p

                Si x = p-1 :
                        STOP

        Sinon : Premier = Faux
retourner Premier
```

Nous savons à présent tester n'importe quel nombre quelconque afin de déterminer si ce nombre est premier ou non.

Nous allons donc voir dans la deuxième partie comment décomposer un nombre en nombres premiers grâce aux fonctions détaillées précédemment.

Algorithmes de décomposition

Dans cette seconde partie, nous allons maintenant nous intéresser à cinq algorithmes différents de décomposition en nombres premiers, afin de présenter les différentes fonctions utilisées, de justifier les choix que nous avons fait ainsi que de présenter les avantages et les inconvénients de chacun de ces algorithmes.

➢ Algorithme Naïf

Dans un premier temps, nous allons ainsi nous intéresser à un algorithme dit « naïf » de décomposition en nombres premiers. Cet algorithme peut se décliner sous deux formes :

- Une forme itérative ;
- Une forme récursive.

Après avoir développé l'algorithme sous ses deux formes, nous avons pu constater que la forme récursive était plus efficace que la forme itérative, c'est pourquoi nous avons décidé de garder cette forme plutôt que l'autre dans notre programme.

Cet algorithme prend en entrée un nombre à décomposer et permet de retourner en sortie la liste des facteurs premiers de ce nombre. Pour cela, l'algorithme divise le nombre en entrée par tous les nombres compris entre 2 et la racine carrée de ce nombre jusqu'à ce que le résultat obtenu soit un nombre entier, ce qui signifie que le nombre d'entré est composé du diviseur et du dividende. Le diviseur est alors ajouté à la liste des nombres premiers du nombre de départ et le résultat de la division devient le nombre d'entré de l'algorithme jusqu'à ce que le résultat ne soit divisible par aucun nombre compris entre 2 et la racine de ce résultat, ce qui signifie que le nombre

est premier. La liste de tous les diviseurs stockés est alors retournée et l'algorithme s'arrête.

Cette liste retournée constitue alors la liste des nombres premiers dont le nombre de départ est constitué.

Ainsi, nous pouvons présenter ci-dessous l'algorithme de cette méthode :

```
Entrée: n entier
Sortie : liste des facteurs premiers de n

Initialisation : liste naïf = liste vide

Si n est premier: on l'ajoute à la liste et retourne la liste_naïf
Sinon : Pour p allant de 2 à racine(n)
                Si n divisible par p : on l'ajoute dans la liste et retourner Algo Naïf(n/p)
        retourner liste_naïf
```

Enfin, nous pouvons présenter les différents avantages et inconvénients de cette méthode. Tout d'abord, l'algorithme naïf possède l'avantage d'être facile à appréhender et à exécuter. De plus, cet algorithme possède l'avantage de fonctionner pour tout type d'entier. Cependant, cet algorithme présente de gros inconvénients car il n'est absolument pas optimisé et nécessite un temps très long d'exécution pour de grands nombres (cet algorithme possède une grande complexité).

➢ Rho de Pollard

Dans un second temps, nous allons nous intéresser à l'algorithme de Rho de Pollard. Cet algorithme permet de décomposer des entiers naturels composés de petits facteurs (des nombres appelés « nombres friables »), en produits de facteurs premiers.

Cet algorithme prend en entrée un entier naturel à décomposer et permet de retourner en sortie la liste des facteurs premiers de ce nombre. Pour cela, l'algorithme est basé sur une fonction notée g tel que $g:R \rightarrow R$ où $g(x)=x^2+1$. Cette fonction est appliquée à deux variables modulo le nombre choisi en entrée. Cela permet de calculer le PGCD de la valeur absolue de la différence de ces deux nombres et du nombre choisi en entrée tant que la valeur obtenue est égale à 1. De cette manière, lorsque le PGCD n'est plus égal à 1, nous obtenons deux facteurs : d et n/d. Si l'un de ces deux facteurs est premier alors il est ajouté à la liste sinon, le deuxième facteur est renvoyé comme variable d'entrée de l'algorithme.

Dans le cas où le facteur d est premier, l'algorithme teste également le deuxième facteur afin de déterminer s'il est également premier. Si le deuxième facteur n'est pas premier, il est renvoyé comme variable d'entrée du programme sinon, le programme se termine et renvoie la liste des facteurs premiers du nombre initialement choisi.

Enfin le cas particulier où le facteur d est égal au nombre d'entrée est également traité.

Ainsi, nous pouvons présenter ci-dessous l'algorithme de cette méthode :

```
Entrée: n un entier
Sortie : liste des facteurs premiers de n

Si n = 1 : retourner 1

//Initialisation
liste fact = liste vide
x=2
y=2
d=1

Tant que d=1:
        x=g(x) %n
        y=g(g(x))%n
        d= pgcd( | x - y | ,n)
Fin Tant Que
```

```
Si d !=n :
        Si d premier : on l'ajoute dans la liste des facteurs premiers de n
        Sinon : (on essaye de trouver l'autre facteur en divisant n par d)
                Si n/d premier : on l'ajoute dans la liste
                Sinon : retourner Rho Pollard(n/d)
        Si d divise n (n%d=0) et n/d est premier : on ajoute n/d dans la liste
        Si d divise n et n/d non premier : retourner Rho Pollard(n/d)

        retourner liste fact


Sinon (d=n) :
        Si n est premier : on l'ajoute dans la liste
        Sinon : on a une erreur, on retourne la liste des facteurs
```

Enfin, nous pouvons présenter les différents avantages et inconvénients de l'algorithme rho de Pollard. Cette méthode offre l'avantage d'être très rapide lorsque le nombre choisi en entrée est composée de petits facteurs(jusqu'à 7 chiffres pour un facteur). Néanmoins, cet algorithme est par conséquent peu efficace lorsque les facteurs sont relativement grands et nécessite que le nombre en entrée de l'algorithme soit un entier naturel. De plus, il peut arriver que la suite soit bouclée, ce qui arrive lorsque le PGCD est égal au nombre d'entrée, ce qui entraine alors une erreur dans l'algorithme. Nous pouvons donc conclure que cet algorithme, bien que plus efficace et plus optimisé que l'algorithme naïf, reste tout de même un algorithme lent. La lenteur et le manque d'optimisation de cette méthode sont les raisons pour lesquelles nous nous proposons d'étudier une nouvelle méthode dans la partie suivante.

➢ P-1 Pollard

Dans un troisième temps, nous allons nous intéresser à l'algorithme de P-1 de Pollard. Cet algorithme permet de décomposer des entiers dont les facteurs ont une forme particulière, en produits de facteurs premiers. Pour cela, nous supposons que le nombre à décomposer, fourni en entrée de l'algorithme, possède un facteur premier p tel que p-1 ne possède que de petits facteurs premiers et ainsi que ce nombre soit dit « friable ».

Cet algorithme prend en entrée un entier naturel à décomposer ainsi qu'un entier correspondant au seuil de friabilité et permet de retourner en sortie la liste des facteurs premiers du nombre à décomposer si le seuil de friabilité est optimal. Pour cela, l'algorithme commence par vérifier que le seuil de friabilité n'est ni trop grand ni trop petit pour que l'algorithme puisse s'exécuter correctement.

Ensuite, afin d'utiliser les propriétés du petit théorème de Fermat, l'algorithme définit un nombre aléatoirement entre 1 et n puis pour chaque nombre premier compris entre 2 et le seuil de friabilité, ce nombre aléatoire est mis à la puissance le nombre premier, puissance lag du seuil de friabilité divisé par log du nombre premier, le tout modulo le nombre d'entrée. Ainsi, à la fin de cette opération, l'algorithme détermine un entier nécessairement inférieur ou égal au nombre d'entrée de l'algorithme.

Comme pour l'algorithme précédent, l'algorithme P-1 de Pollard détermine ensuite si l'un ou l'autre des facteurs est premier, recommence l'algorithme si le premier et le deuxième facteur ne sont pas premiers et retourne la liste des facteurs premiers si les deux facteurs sont premiers.

Ainsi, nous pouvons présenter ci-dessous l'algorithme de cette méthode :

Entrée : n un entier, B un entier
Sortie : si B est un seuil de friabilité optimal pour n alors renvoie liste des facteurs premiers de n sinon renvoie "erreur"

Initialisation :
liste facteur = liste vide

Si B > racine de n
    Retourner une erreur car le seuil est trop grand
Si B < 2
    Retourner une erreur car le seuil est trop petit

a ← entier aléatoire compris entre 1 et n

Pour i variant de 2 à B
    Si i est premier
        $M = \frac{Log(B)}{Log(i)}$

        $a = a^{i^M} \% n$
    Fin si
Fin Pour

g = PGCD(a - 1 , n)

Si 1 < g < n
    Si g est premier : on l'ajoute dans la liste
    Si n/g premier : on l'ajoute dans la liste (on cherche l'autre facteur)
    Sinon retourner p-1-Pollard(n/g, B)

Si n % g = 0
    Si n/g est premier : on l'ajoute dans la liste
    Sinon : retourner p-1-Pollard(n/g, B)
    Retourner la liste des facteurs

Si g = 1
    Si B < racine carrée de n (on peut encore augmenter le seuil de friabilité)
        Incrémentation de B
        Retourner p-1-Pollard(n, B)
    Sinon
        Retourner Erreur
Si g = n
    Si g premier :
        Ajouter g à la liste des facteurs
    Sinon
        Retourner Erreur

Enfin, nous pouvons citer quelques avantages et inconvénients de cette troisième méthode. Tout d'abord, l'algorithme P-1 Pollard possède les mêmes caractéristiques que son prédécesseur l'algorithme rho de Pollard. En effet, ce dernier ne fonctionne que pour des entiers possédant une forme particulière, c'est-à-dire des nombres possédant un facteur premier p avecp-1 friable, mais a surtout un temps d'exécution lent pour des grands nombres. Cet algorithme a donc l'avantage d'être rapide et efficace lorsque le nombre d'entrée est choisi pour rendre l'algorithme performant mais garde les mêmes inconvénients que l'algorithme rho de Pollard à savoir un manque de polyvalence et une lenteur lorsque le nombre d'entrée n'est pas optimal pour l'algorithme ou que celui-ci est trop grand. De plus, il est important de savoir que plus on augmente B le seuil de friabilité plus la probabilité d'obtenir un facteur est grande, mais on rallonge le temps de calcul et l'algorithme peut ainsi devenir plus lent que l'algorithme Naïf.

Afin de trouver un algorithme plus performant, nous allons nous intéresser à un nouvel algorithme dans la prochaine partie.

➢ Lenstra

Dans un quatrième temps, nous allons nous intéresser à l'algorithme de Lenstra. Cet algorithme qui reprend les principes de l'algorithme P-1 de Pollard, en est une amélioration car il permet de résoudre le problème rencontré dans l'algorithme P-1 de Pollard, qui empêchait de factoriser le nombre d'entrée si celui-ci ne possédait pas de facteurs premiers p avec p-1 friable. Pour cela, l'algorithme de Lenstra se base sur l'utilisation des courbes elliptiques ainsi que sur certaines fonctions telles que l'algorithme d'Euclide étendu, l'addition de points dans une courbe elliptique, la multiplication de points dans une courbe elliptique ou encore la fonction de crible d'Eratosthène, décrits en annexe 1.

Afin de réaliser la décomposition en facteurs premiers, l'algorithme de Lenstra prend en entrée un entier à factoriser ainsi qu'un entier correspondant au seuil

de friabilité et retourne en sortie la liste des facteurs premiers du nombre d'entrée choisi, tout comme l'algorithme P-1 de Pollard. Ensuite, contrairement à l'algorithme précédent, l'algorithme de Lenstra détermine non pas un mais trois nombres entiers aléatoires compris entre 1 et n, qui serviront de variables aux fonctions permettant de déterminer la première valeur du calcul du PGCD, la seconde étant comme pour l'algorithme précédent la valeur de l'entier à décomposer choisi en entrée. Enfin, là encore, la valeur du PGCD obtenue sert à déterminer quels choix effectuer afin d'ajouter l'un ou l'autre des facteurs à la liste des facteurs premiers du nombre de départ. L'algorithme recommence si le premier et le deuxième facteur ne sont pas premiers et retourne la liste des facteurs premiers si les deux facteurs sont premiers. Néanmoins, contrairement à P-1 de Pollard, si d n'est pas compris entre 1 et le nombre de départ choisi, l'algorithme va utiliser les courbes elliptiques afin de palier le problème rencontré dans l'algorithme précédent, qui empêchait de factoriser le nombre d'entrée si celui-ci ne possédait pas de facteurs premiers p avec p-1 friable.

Ainsi, nous pouvons présenter ci-dessous l'algorithme de cette méthode :

```
        Si 1 < d < n :
                Si d est premier :
                        On ajoute d à la liste
                        Si n/d est premier :
                                On ajoute n/d à la liste et on retourne la liste
                        Sinon
                                Retourner Lenstra(n/d, B)
                Sinon (d n'est pas premier) :
                        Si n/d est premier :
                                Ajouter n/d à la liste et retourner Lenstra(d, B)
                        Sinon
                                Retourner Lenstra(n/d, B)
Fin tant que

Si d = 1 :
        P = (x, y)
        k = PPCM(2, 3,...B) où tous les nombres sont premiers
        liste_premiers = Eratosthène(B)
        Pour p premier dans liste_premiers allant de 2 à B : k = PPCM(k, p)

        Multiplication Points(k, P, n) où k est le nombre de multiplications, P le point dans
        la courbe elliptique Fn et n le nombre premier tel que Z/nZ

        Si dans les coordonnées de P, P n'a pas d'inverse :
                facteur = PGCD(d, n)
                Si facteur < n :
                        Si facteur est premier : On l'ajoute à la liste
                                Si n/facteur est premier :
                                        On l'ajoute à la liste et on retourne liste facteur
                                Sinon : retourner Lenstra(n/facteur, B)
                        Sinon :
                                Si n/facteur est premier :
                                        On l'ajoute à la liste et on retourne Lenstra(facteur, B)
                                Sinon : retourner Lenstra(facteur, B)

                Si facteur = n : Réduire k
        Sinon : Retourner Lenstra(n, B)
```

Enfin, nous nous proposons de tirer les avantages et les inconvénients de cette quatrième méthode. Ainsi, l'algorithme de Lenstra offre de nombreux avantages tels que le fait que le temps d'exécution ne dépend pas du nombre choisi en entrée mais d'un facteur de ce nombre, ce qui signifie que la complexité de cet algorithme est exponentielle et dépend du plus petit facteur du nombre d'entrée choisi. De plus, l'amélioration proposée de la méthode de factorisation P-1 de

Pollard permet de ne pas avoir de cas d'erreurs. Concernant ses inconvénients, ils sont simplement liés à la vitesse d'exécution qui dépend de la taille d'un facteur p plutôt que du nombre n à factoriser et fait que sa vitesse est optimale lorsque la taille des diviseurs à extraire ne dépasse pas 20 chiffres, ce qui en fait le 3e algorithme de décomposition en facteurs premiers le plus rapide derrière l'algorithme du crible quadratique et du crible quadratique généralisé. Mais il faut savoir que les algorithmes de Lenstra, P-1 Pollard et Rho Pollard ont tous les trois un défaut commun. Pour de très petits nombres, inférieurs à 3 chiffres l'algorithme ne va jamais retourner du premier coup un facteur non trivial, il va devoir exécuter plusieurs fois avant de pouvoir le trouver.

Nous pouvons donc conclure que cet algorithme est très efficace, mais nous souhaitons obtenir encore mieux avec le prochain algorithme !

➢ Crible Quadratique

Dans un cinquième et dernier temps, nous allons nous intéresser à l'algorithme du crible quadratique. Ici, l'idée est de trouver x et y tels que x^2 congru y^2 modulo n, puis que $n \mid (x - y)(x + y)$ et enfin que PGCD(x − y, n) ou que PGCD(x + y, n) soit un facteur de n. Pour cela, l'algorithme prend en entrée le nombre à décomposer ainsi l'intervalle du crible souhaité et retourne en sortie la liste des facteurs premiers du nombre à décomposer choisi en entrée. L'algorithme peut ensuite être décomposé en quatre grandes parties, que sont :

- L'initialisation : cette étape permet de définir la valeur de trois variables que sont le seuil de friabilité dépendant du nombre à factoriser choisi en entrée, la base des facteurs qui est calculée grâce aux méthodes d'Eratosthène et de Legendre à partir du nombre à factoriser et du seuil de friabilité et enfin, l'initialisation permet de définir une variable τ correspondant à la taille du vecteur base des facteurs obtenu précédemment.

- Le criblage : cette étape permet de rechercher τ relations qui sont B-friables en définissant une fonction de criblage qui prend en entrée une variable n et une variable crible_entier où n nombre entier à factoriser et crible_entier représente l'intervalle du criblage. Cette fonction génère une liste d'équation x^2-N en partant de la racine de n et retourne la liste des nombres friables, ainsi que les coefficients a_1, ..., a_n et les indices

- L'algèbre linéaire : cette étape permet de générer une matrice composée de la liste des nombres friables et des nombres présents dans la base des facteurs. Ensuite, nous recherchons des nombres premiers dans la matrice de manière similaire aux algorithmes précédents puis, on applique l'Elimination de Gauss pour avoir les équations paramétrées et nous résolvons des équations paramétrées pour trouver le vecteur.

- La factorisation : cette étape permet de trouver des facteurs premiers à partir du vecteur calculé précédemment, de les ajouter à la liste et de retourner la liste lorsque tous les facteurs premiers ont été déterminés.

Ainsi, nous pouvons présenter ci-dessous l'algorithme de cette méthode :

Entrée : n l'entier à factoriser, Intervalle l'intervalle du crible
Sortie : la liste des facteurs premiers de n

Etapes de l'algorithme du Crible Quadratique :

Si n premier : on l'ajoute dans la liste et on retourne la liste
Si n est une racine : on l'ajoute dans la liste et on retourne la liste

1) Initialisation

a) Seuil de Friabilité B
    B = Seuil_Friabilite(n)
b) Base des facteurs
    base_facteur=Base_Facteurs(n,B)

Tau = taille du vecteur base des facteurs

2) Criblage

On définit la fonction Criblage

a. Définir la séquence de criblage liste_crible(n,crible_entier)

b. Une fois la liste des congruences générer, on la copie
liste_crible=liste_congruence.copy

c. i) Cas Particulier pour 2
Tant que les congruences sont divisibles par 2 on continue

ii) Cas Général (Shanks Tonelli)
On cherche à résoudre les équations du type $r^2 = N \bmod p$, on utilise donc l'algorithme de Shanks Tonelli pour trouver les racines

Pour premier allant de 2 à longueur(Base facteur) :
On trouve les racines
Solutions_résidues = Shanks_Tonelli(N,premier)

Pour r dans Solutions_résidues :
Pour k allant de r-racine(n) mod premier à longueur(liste_cribler), pas premier) :
Tant que premier | liste_crible[k] :
On divise liste_crible[k] par premier

d. Après toutes les divisions on a liste_crible composer que de 0 ou 1
On initialise liste_num_friables :
liste des nombres friables (liste des nombres tq x^2-N)
a_list : liste des nombres (liste des x)
index : liste des indice (liste des indices de x par rapport à la liste des congruences)

Pour i allant de 1 à liste crible :
Déclare T (facteur de tolérance augmente la probabilité de réussite)
Si taille de liste num friables > taille de base facteur :
Stop boucle
Si liste crible[i] =1 :
on ajoute liste_congruence[i] dans la liste num friables
on ajoute i + racine(n) dans a_list (liste des x)
on ajoute i dans index (liste des indices de x par rapport à la liste des congruences)

Retourner liste num friables, a_list, index

Si longueur(base facteur) > liste num friables :
Il faut augmenter l'Intervalle du criblage ou taille de base facteur

3) Algèbre Linéaire

```
Générer Matrice(liste_num_friables, base_facteur)

Si la matrice M est carrée :
        x = index de M_transpose dans liste_num_friables
        facteur = pgcd de a_liste[ x ] + racine(M_transpose) et n

        Si facteur est premier : on ajoute le facteur dans liste_facteur
                Si n / facteur premier : on ajoute n/facteur dans la liste_facteur et retourner
                        liste_facteur
                Sinon : retourner Crible_Quadratique(n/facteur, Intervalle)

        Sinon :
                Si n/facteur premier : ajouter n/facteur dans la liste_facteur
                        Retourner Crible_Quadratique(facteur, Intervalle)
                Sinon : retourner Crible_Quadratique(facteur, Intervalle)

eq_param, pivot_verif, M = Elimination_Gauss(M_transpose)
vect = Vect(eq_param, M, pivot_verif, 0)
```

4) Factorisation

```
facteur = Factorisation(vect, liste_num_friables, a_liste, n)
Pour i allant de 2 à longueur de eq_param :
        Si facteur = 1 ou facteur = n :
                vect= Vect(eq_param,M, pivot_verif, i)
                facteur = Factorisation(vect, liste_num_friables, a_liste, n)
        Sinon :
                Si facteur est premier : on ajoute le facteur dans liste_facteur
                        Si n / facteur premier : on ajoute n/facteur dans la liste_facteur
                                        Retourner liste_facteur
                        Sinon : retourner Crible_Quadratique(n/facteur, Intervalle)

                Sinon :
                        Si n/facteur premier : ajouter n/facteur dans la liste_facteur
                                Retourner Crible_Quadratique(facteur, Intervalle)
                        Sinon : retourner Crible_Quadratique(facteur, Intervalle)

// Si on ne trouve pas de facteur
Retourner liste_facteur, "Aucun facteur non triviaux "
```

Pour conclure l'étude de ce dernier algorithme, nous allons aborder ses nombreux avantages. En effet, cet algorithme est l'algorithme étant le plus rapide de ce que nous avons étudié mais est théoriquement devancé par le crible quadratique généralisé. Sa force vient du fait que son temps d'exécution dépend uniquement de la taille de l'entier à factoriser et non pas des propriétés de ce dernier. Cependant, théoriquement, la vitesse d'exécution du crible quadratique et de Lenstra est identique ; toutefois,

dans la pratique, l'algorithme du crible quadratique est plus rapide car il utilise des calculs de simple précision contrairement à l'algorithme de Lenstra qui utilise des calculs de multi précision. Finalement, cet algorithme est d'une grande rapidité et, de surcroît, fonctionne avec tout type d'entier.

Résultats obtenus

Dans cette dernière partie, nous nous proposons de présenter les résultats obtenus par les différents algorithmes présentés dans la deuxième partie et de les comparer afin de déterminer quel algorithme est le plus efficace.

➢ Méthode de comptage et de chronométrage

Dans un premier temps, nous souhaitons comparer de manière précise les différents algorithmes étudiés. Pour cela, nous avons décidés de nous appuyer sur deux critères :

- Le nombre d'opérations effectuées par l'algorithme : nous considérons les opérations comme étant soit des comparaison (=, >, <, etc.), soit des manipulations de vecteurs tels que des affectations ou des lectures de vecteurs. Nous considérons toutes les autres opérations négligeables.
- Le temps d'exécution de l'algorithme entre l'appel de la fonction et le retour du résultat.

Ainsi, pour le calcul du nombre d'opérations, nous avons déclaré dans chaque algorithme une variable de comptage nommée « count » utilisée en entrée de chaque fonction, incrémentée à chaque opération et renvoyée en sortie de chaque fonction. La valeur finale de count est ensuite stockée pour être utilisée pour tracer le graph représentant le nombre d'opérations nécessaires en fonction de l'algorithme. Lorsque toutes les valeurs de comptages sont obtenues, nous traçons le graphique à l'aide de la fonction plot de python.

Concernant le chronométrage de la durée d'exécution de chaque algorithme, nous avons décidé d'utiliser une fonction de chronométrage grâce à la bibliothèque « time » de python. Cette fonction renvoie en seconde le temps d'exécution de l'algorithme. Tout comme pour le comptage du nombre d' opérations, la valeur enregistrée par le chronomètre est stockée afin de réaliser

un graphique du temps d'exécution en fonction de l'algorithme utilisé lorsque toutes les valeurs sont obtenues.

Une fois ces deux méthodes implémentées pour chacun des algorithmes, nous pouvons nous intéresser à la présentation et l'interprétation des résultats.

➢ Présentation des résultats

Dans un deuxième temps, nous allons donc observer les résultats obtenus après exécution des différents algorithmes. Pour cela, nous avons décidé de faire varier n de 5 jusqu'à 1 000 005. Afin de permettre à nos ordinateurs peu puissants de réaliser les calculs ainsi que pour rendre les graphiques plus lisibles, nous avons décidé d'appliquer un pas de 10 000 entre chaque valeur. Nous avons également décidé de réaliser un graph différent pour chaque algorithme et non un seul graph pour toutes les méthodes afin d'améliorer la lisibilité des résultats.
Ainsi, nous pouvons observer les résultats obtenus comme suit :

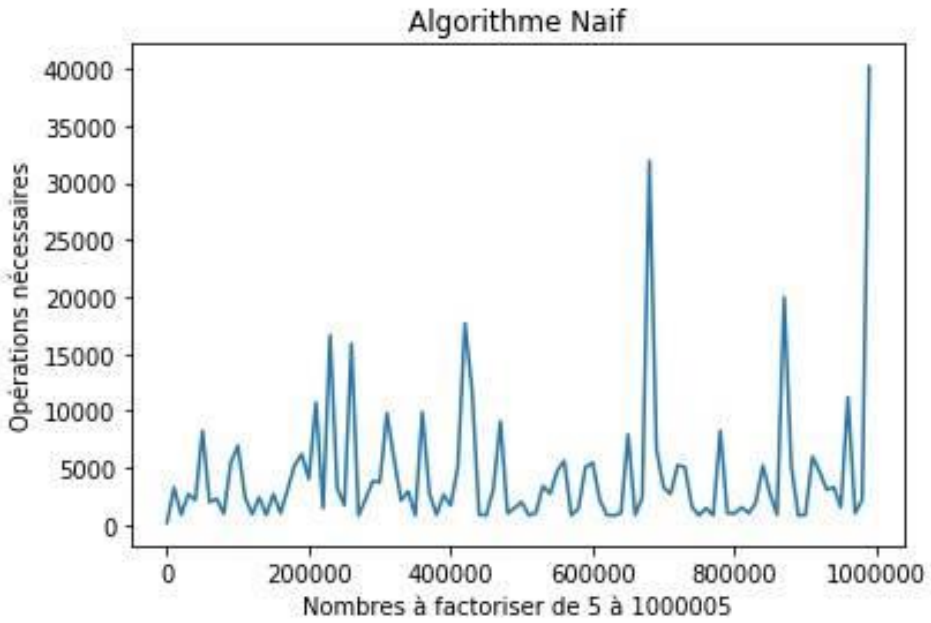

Figure 1 : Graphique de l'évolution du nombre d'opérations nécessaires en fonction du nombre à factoriser pour l'algorithme naïf

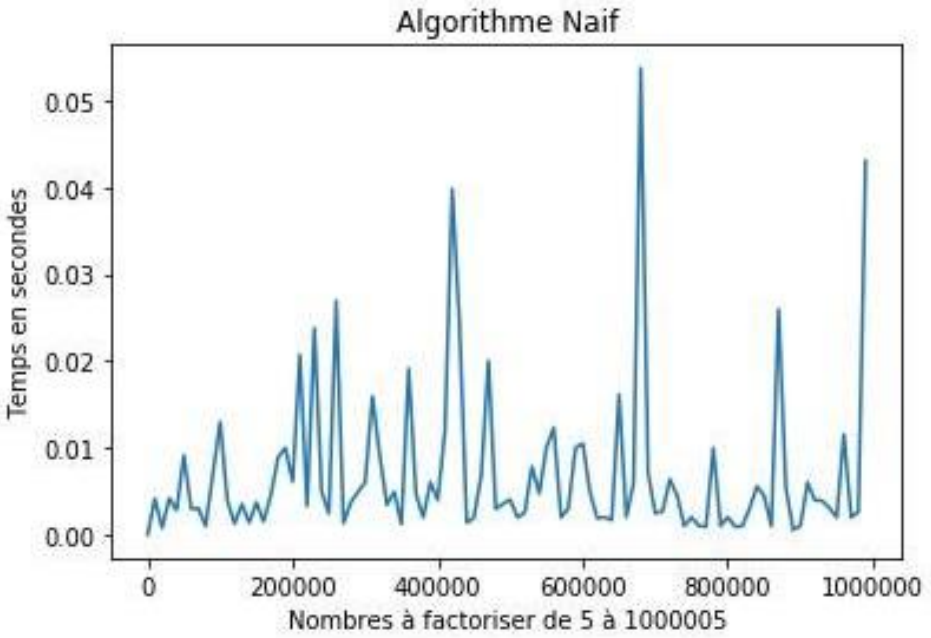

Figure 2 : Graphique de l'évolution du temps d'exécution nécessaire en fonction du nombre à factoriser pour l'algorithme naïf

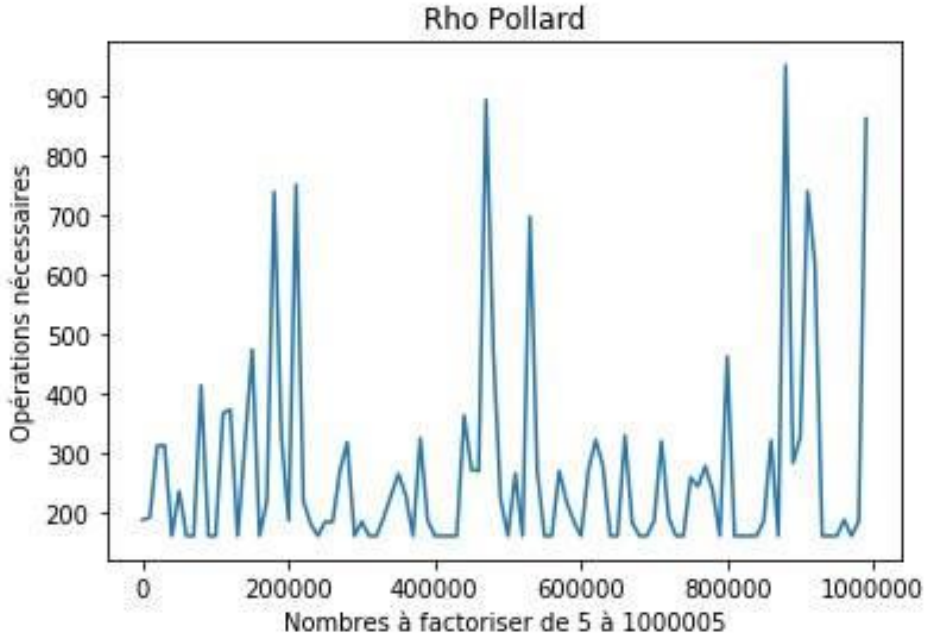

Figure 3 : Graphique de l'évolution du nombre d'opérations nécessaires en fonction du nombre à factoriser pour l'algorithme rho de Pollard

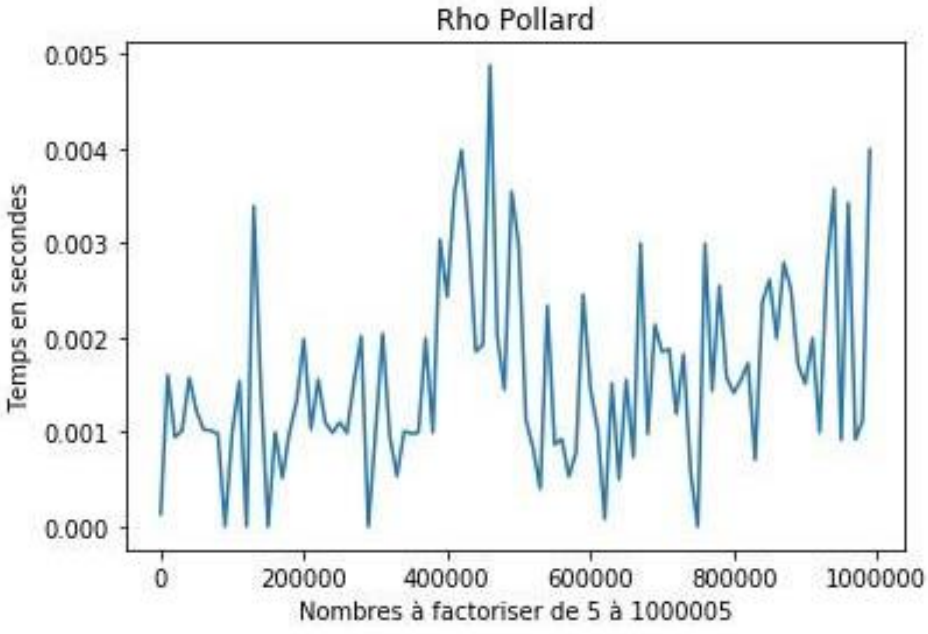

Figure 4 : Graphique de l'évolution du temps d'exécution nécessaire en fonction du nombre à factoriser pour l'algorithme rho de Pollard

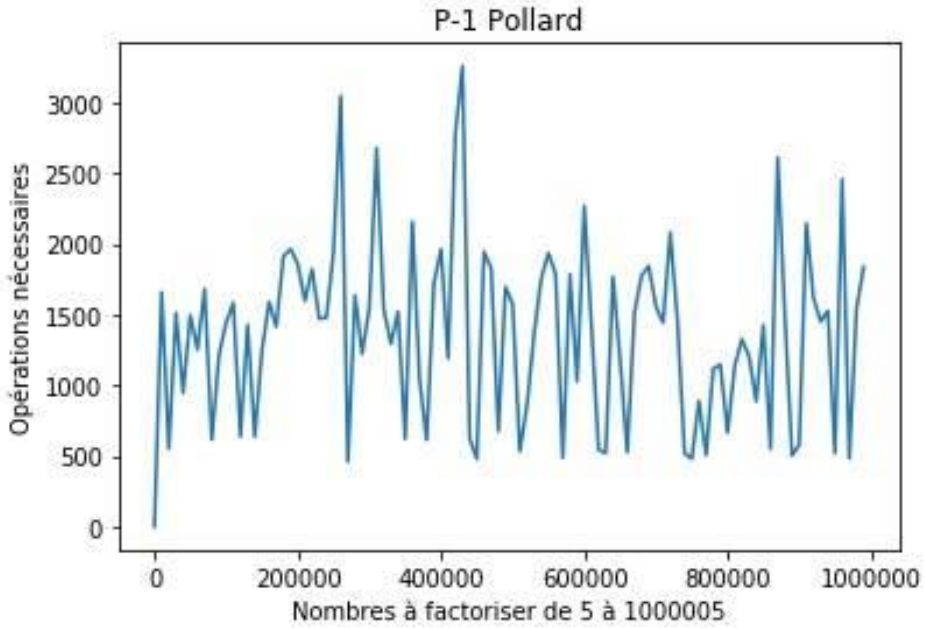

Figure 5 : Graphique de l'évolution du nombre d'opérations nécessaires en fonction du nombre à factoriser pour l'algorithme P-1 de Pollard

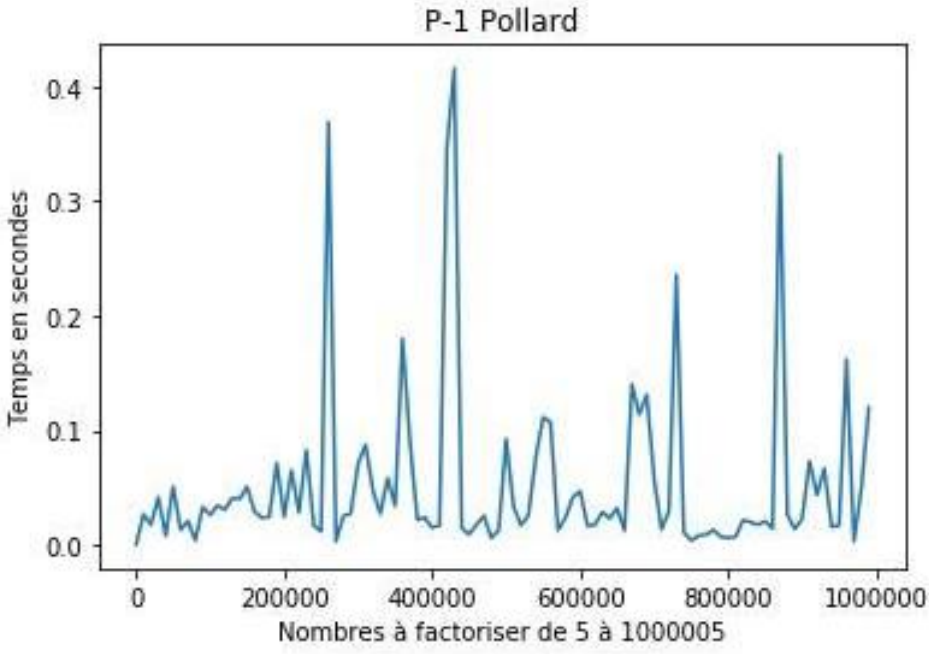

Figure 6 : Graphique de l'évolution du temps d'exécution nécessaire en fonction du nombre à factoriser pour l'algorithme P-1 de Pollard

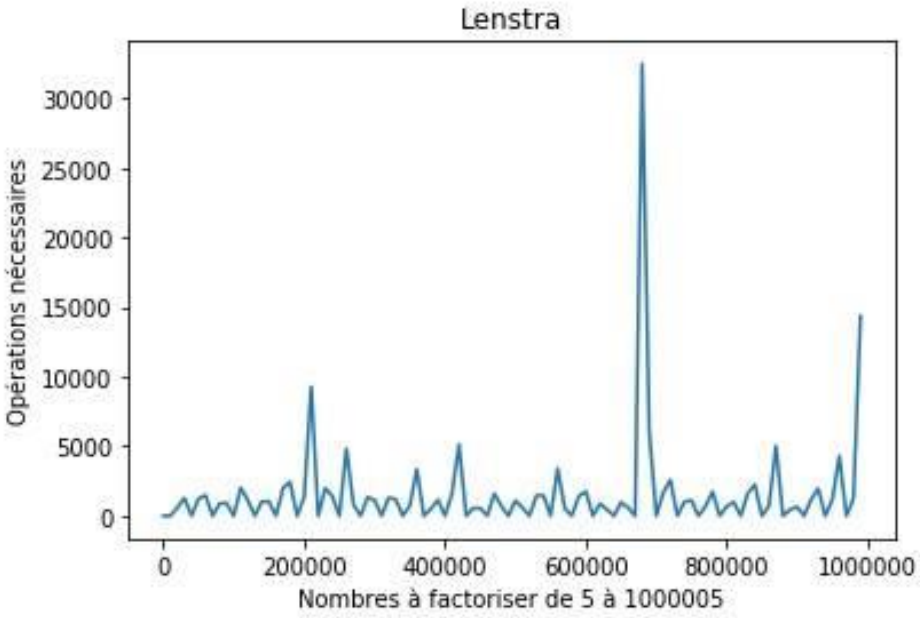

Figure 7 : Graphique de l'évolution du nombre d'opérations nécessaires en fonction du nombre à factoriser pour l'algorithme de Lenstra

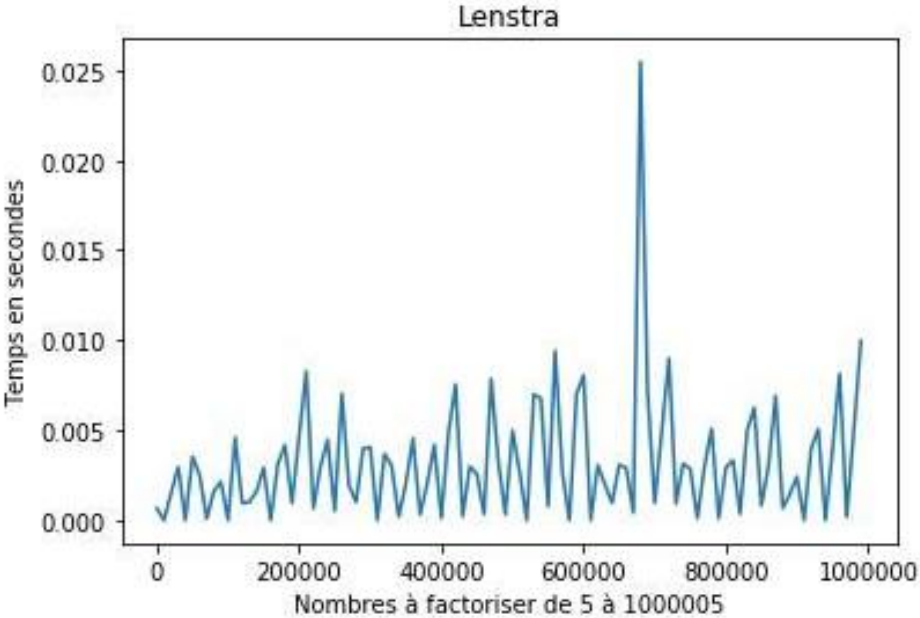

Figure 8 : Graphique de l'évolution du temps d'exécution nécessaire en fonction du nombre à factoriser pour l'algorithme de Lenstra

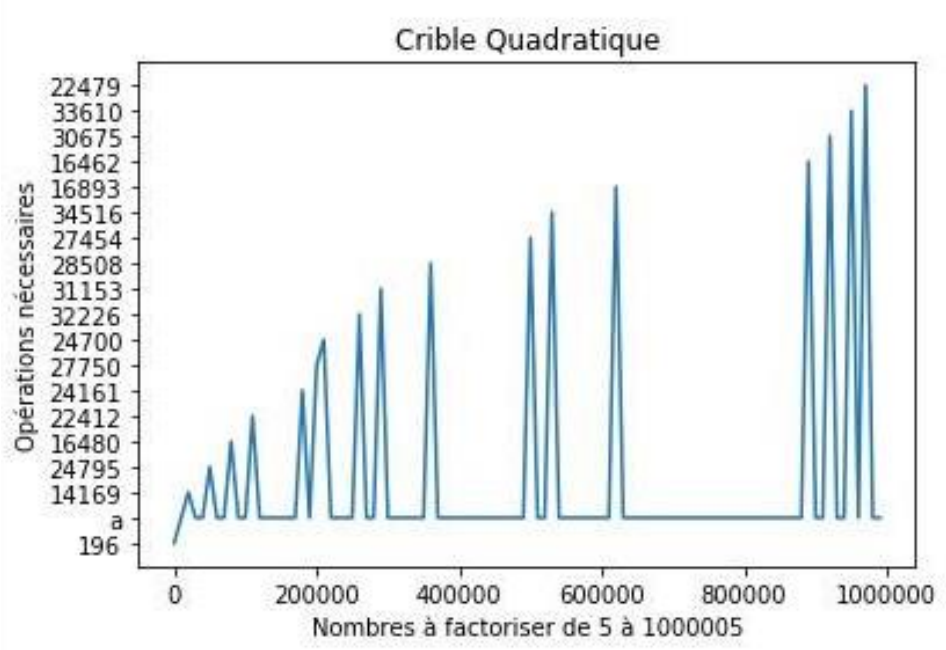

Figure 9 : Graphique de l'évolution du nombre d'opérations nécessaires en fonction du nombre à factoriser pour l'algorithme crible quadratique

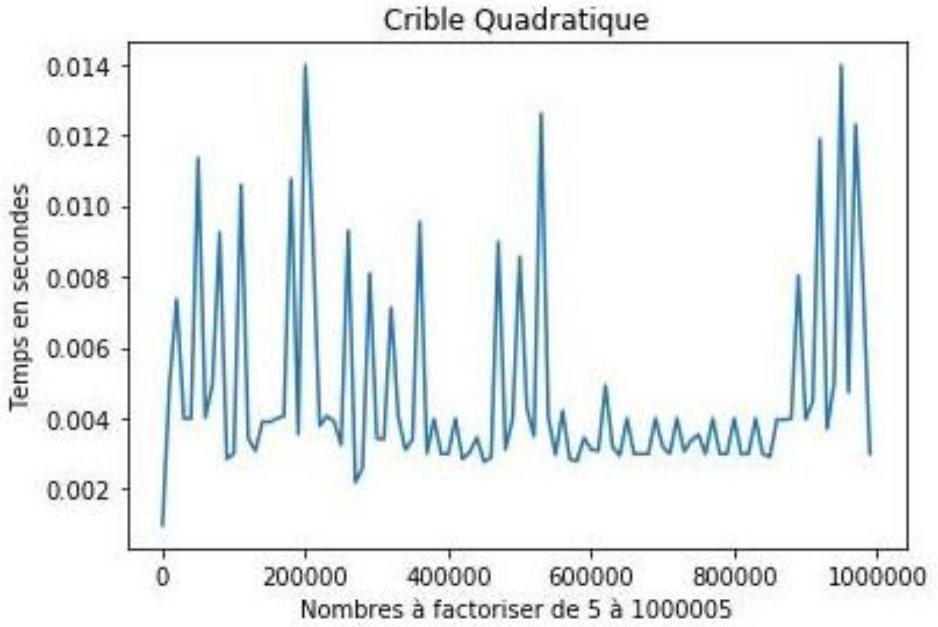

Figure 10 : Graphique de l'évolution du temps d'exécution nécessaire en fonction du nombre à factoriser pour l'algorithme crible quadratique

➢ Comparaison des résultats

Enfin, dans un dernier temps, nous nous proposons de comparer les résultats obtenus précédemment afin de tirer des conclusions concernant leur efficacité.

Ainsi, nous avons pu voir que d'après nos résultats, l'algorithme le plus rapide serait l'algorithme naïf avec un temps de factorisation maximum de 0,005s, ce qui est trois fois plus rapide que le résultat obtenu pour l'algorithme du crible quadratique. Cela peut s'expliquer par le fait que dans notre test, nous n'avons pas pu ajuster le B, c'est-à-dire le seuil de friabilité, car cela est beaucoup trop complexe et entraine donc un grand manque d'optimisation de l'algorithme.
En effet, en théorie, l'algorithme du crible quadratique est censé être beaucoup plus rapide que l'algorithme naïf car il s'agit du deuxième algorithme le plus rapide existant à ce jour après le crible quadratique généralisé.
Par ailleurs, nous pouvons constater sur le tracé bleu de la figure 11 ci-dessous que l'algorithme P-1 de Pollard se détache des autres algorithmes par sa grande lenteur, ce qui s'explique par le fait que cet algorithme n'est pas optimisé pour certaines valeurs.

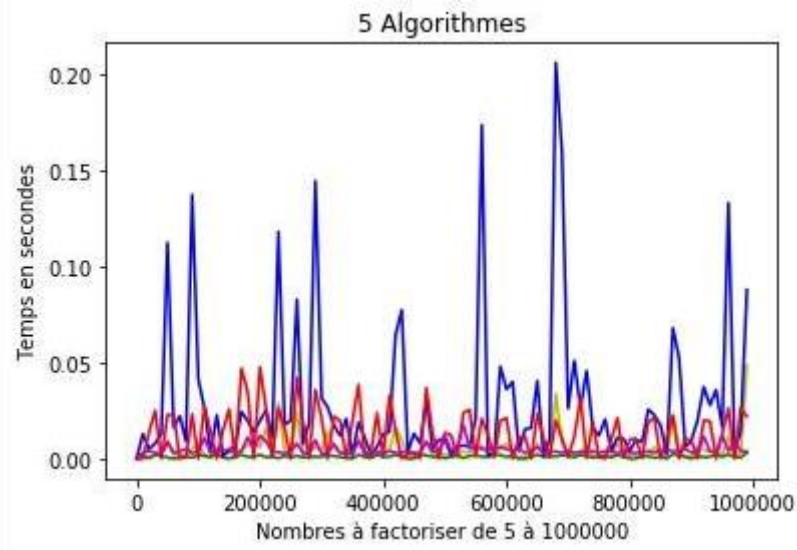

Figure 11 : Comparaison du temps d'exécution en fonction de n pour chacun des algorithmes étudiés

Concernant le nombre d'opérations effectuées, nous pouvons constater que l'algorithme le plus efficace est l'algorithme rho de Pollard avec un nombre maximum d'opérations de 900, ce qui est 25 fois plus efficace que l'algorithme du crible quadratique. Cela peut également s'expliquer par le fait que dans notre test, nous n'avons pas pu ajuster le B, mais également que l'algorithme n'est pas optimisé et que dans la récursivité, nous ne décrémentons pas le B et nous ne baissons pas l'intervalle. En effet, en théorie, l'algorithme du crible quadratique est également censé être beaucoup plus efficace que l'algorithme rho de Pollard. Par ailleurs, nous pouvons constater sur le tracé jaune de la figure 12 ci-dessous que l'algorithme naïf se détache des autres algorithmes par son manque d'efficacité, ce qui s'explique par le fait que cet algorithme n'est pas du tout optimisé.

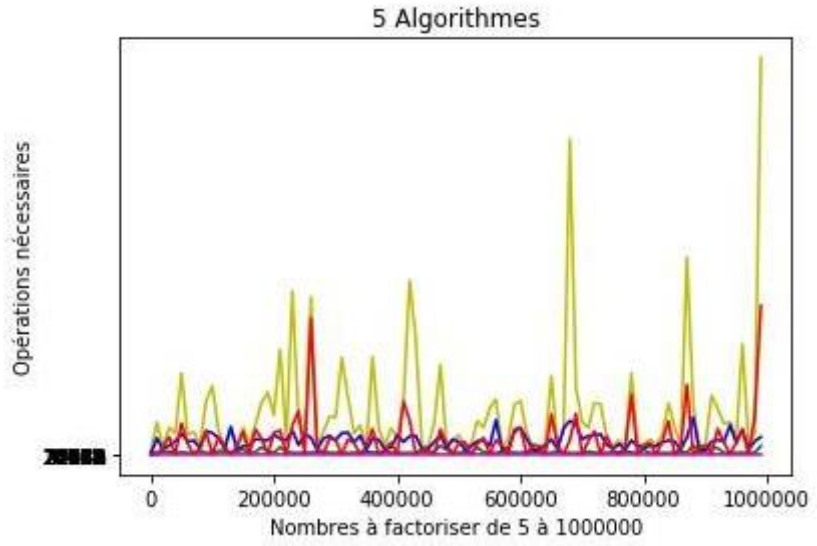

Figure 12 : Comparaison du nombre d'opérations effectuées en fonction du n pour chacun des algorithmes étudiés

Enfin, nous pouvons présenter ci-dessous un tableau présentant les différents temps d'exécutions observés sur chacun des algorithmes pour des valeurs à décomposer choisies de manière particulière.

Comparaison des algorithmes en termes de temps d'exécution					
Nombre à factorisé	**Algorithme Naif**	**Rho de Pollard**	**P-1 Pollard**	**Lenstra**	**Crible Quadratique**
70	0.00195	0.00049	B=10	B=10	B=48 / Intervalle=100
			0.00198	0.00236	0.00496
703253	0.09225	0.00098	B=13	B=25	B=101 / Intervalle=10000
			0.00297	0.00709	0.02975
146182562237	1.10257	0.00297	B=30	B=34	B=390 / Intervalle=50000
			11.87816	0.05706	0.32185
264839967043414254127	2.16904	0.02182	B=30	B=47	B=3000 / Intervalle=500000
			15.77478	0.19695	7.93362
290919706205487829572593	X	0.04067	X	X	B=4500 / Intervalle=1000000
					10.59218

Là encore, nous pouvons constater qu'en pratique, l'algorithme le plus rapide est l'algorithme rho de Pollard.

Ainsi, nous pouvons conclure de ces observations que l'algorithme le plus efficace de tous les algorithmes étudiés est l'algorithme rho de Pollard pour le temps d'exécution et le nombre d'opérations, mais qu'en théorie, l'algorithme le plus rapide et le plus efficace est l'algorithme du crible quadratique.

Conclusion

Au terme de ce projet, nous avons pu conforter nos connaissances au travers de la réalisation d'un programme de décomposition en nombres premiers en parvenant à implémenter l'ensemble des cinq algorithmes demandés ainsi qu'à en extraire les informations utiles pour comparer chacun de ces algorithmes et déterminer lequel était le plus performant.

Nous avons apprécié travailler en groupe sur cette mise en pratique car elle fût pour nous le moyen de découvrir de nouvelles méthodes mathématiques nous permettant de réaliser des décompositions en nombres premiers, mais également parce que ce projet nous a permis de renforcer nos compétences en travail d'équipe et ce, malgré la distance qui nous a séparé durant toute la période de réalisation de ce projet, dû à la pandémie de covid-19.

Enfin, les résultats obtenus à l'issu de ce projet nous ont permis de déterminer que l'algorithme de décomposition en nombres premiers le plus efficace était l'algorithme crible quadratique car contrairement aux autres algorithmes (hormis l'algorithme naïf), cet algorithme a l'avantage de fonctionner avec tous types d'entier. De plus, de tous les algorithmes étudiés au cours de ce projet, l'algorithme du crible quadratique est en moyenne le plus rapide et celui qui nécessite le moins d'opérations afin de retourner la liste des produits de facteurs premiers composant le nombre de départ. L'algorithme du crible quadratique se révèle donc être l'algorithme idéal.

Webographie

Rabin Miller Test :

- https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test

Algorithme Factorisation Naïf :

- https://fr.wikipedia.org/wiki/Algorithme_de_d%C3%A9composition_en_produit_de_facteurs_premiers

Crible D'Eratosthène :

- https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Algorithme Euclide Etendu :

- https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

Courbe Elliptiques :

- http://math.univ-lyon1.fr/~wagner/coursDelaunay.pdf

Shanks Tonelli pour les résidus quadratiques :

- https://en.wikipedia.org/wiki/Tonelli%E2%80%93Shanks_algorithm
- https://fr.wikipedia.org/wiki/Symbole_de_Legendre
- https://en.wikipedia.org/wiki/Quadratic_residue

Rho Pollard :

- https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm
- https://www.sciencedirect.com/science/article/pii/089054019190001I
- https://mathworld.wolfram.com/PollardRhoFactorizationMethod.html
- http://math.mit.edu/~goemans/18310S15/factoring-notes.pdf

P-1 Pollard :

- http://robin.pollak.io/wizard_factoring.pdf
- https://mathworld.wolfram.com/Pollardp-1FactorizationMethod.html
- https://frenchfries.net/paul/factoring/theory/pollard.p-1.html
- http://www.math.mcgill.ca/darmon/courses/05-06/usra/charest.pdf
- https://www.cryptologie.net/article/344/pollards-p-1-factorization-algorithm/

Lenstra :

- http://math.uchicago.edu/~may/REU2014/REUPapers/Parker.pdf
- https://en.wikipedia.org/wiki/Lenstra_elliptic-curve_factorization
- http://www.math.mcgill.ca/darmon/courses/05-06/usra/charest.pdf
- https://cdn.rawgit.com/andreacorbellini/ecc/920b29a/interactive/modk-add.html
- https://fr.qwe.wiki/wiki/Elliptic_curve_point_multiplication

Crible Quadratique :

- https://en.wikipedia.org/wiki/Quadratic_sieve
- https://www.math.u-bordeaux.fr/~jcouveig/cours/cribles.pdf
- http://math.univ-lyon1.fr/~roblot/resources/ens_partie_4.pdf
- http://pauillac.inria.fr/~maranget/X/IF/PI/maranget/sujet.html
- https://lipn.univ-paris13.fr/~banderier/Facto/index.html
- https://docplayer.fr/57725985-Mt10-mathematiques-pour-la-cryptographie-partie-4-factorisation-algorithme-du-crible-quadratique-1-walter-schon.html
- https://math.dartmouth.edu/~carlp/PDF/paper52.pdf
- https://www.cs.virginia.edu/crab/QFS_Simple.pdf
- https://www.researchgate.net/publication/320506019_Mathematical_Basics_Implementation_and_Performance_Benchmark_of_the_Multiple_Polynomial_Quadratic_Sieve_MPQS_Algorithm_for_Factorization
- https://libres.uncg.edu/ir/uncg/f/umi-uncg-1581.pdf

Annexe

➢ Annexe 1 : Méthodes pour l'algorithme de Lenstra

- Crible qui permet d'avoir la liste des nombres premiers inférieur à la limite n :

```
Eratosthène :
Entrée : n un entier
Sortie : liste de nombres premiers inférieurs à n

L = liste de 3 à n
i=2

Tant que i < racine(n) :
        Si i est dans L
                Pour j allant de 2*i à n, de pas i :
                        Si j dans L : Supprimer j de L
        Incrémenter i
Retourner L
```

- Algorithme d'Euclide étendu entre a et b :

```
extended gcd :
Entrée : a, b entiers naturels
Sortie : r = pgcd(a, b) avec r = a*u+b*v, u et v entiers relatifs
```

- Multiplication nP (Addition nP=P+P+..+P n fois) dans une courbe elliptique Fp :

```
Multiplication Point :
Entrée : n entier naturels, P point de la courbe elliptique, Fp l'ensemble Z/pZ
Sortie : Multiplication nP

Résultat = point infini
pow 2P = P

Tant que n != 0 :
        Si y1= "Pas d'inverse" :
                Si n mod 1 = 0 :
                        Résultat = Addition Point(pow 2P, résultat, Fp)
                        Si y1 = "Pas d'invers" : // (résultat[1] = y1)
                                Retourner résultat
                Résultat = Addition Point(pow 2P, pow 2P, Fp)
                pow 2P = résultat
                n = n /2
        Sinon : retourner pow 2P
Retourner résultat
```

- Addition de points dans une courbe elliptique :

```
Addition Point :
Entrée : P et Q points dans la courbe elliptique, Fp ensemble Z/pZ
Sortie : L'addition entre P et Q

Si P est le point infini : retourner Q
Si Q est le point infini : retourner P

Si x1=x2 et y1 = -y2 : retourner point infini
Si x1=x2 et y1 = y2 :
        a = nombre aléatoire entre 2 et Fp-1
        numérateur = 3*x1² + a mod Fp
        dénominateur = 2*y1
        g, x, y = extended gcd(dénominateur, Fp)  // On cherche l'inverse de dénominateur

        Si g != 1 : (Si le pgcd(dénominateur, Fp) != 1)
                Retourner dénominateur, "Pas d'inverse"
        dénom inv= x mod Fp
        m = numérateur * dénom inv
        x3 = m² - 2*x1 mod Fp

Sinon :
        numérateur = y2 - y1 mod Fp
        dénominateur = x2 - x1
        g, x, y= extended gcd(dénominateur, Fp)

        Si g!=1 :
                Retourner dénominateur, "Pas d'inverse"
        dénominateur inv = x mod Fp
        m= numérateur * dénominateur inv
        x3 = m²-x1-x2

Retourner [x3 mod Fp , m*(x1-x3) - x2 mod Fp]
```

➢ Annexe 2 : Méthodes pour l'algorithme du Crible Quadratique

- Fonctions pour L'Initialisation :

Seuil_Friabilite :
Entrée : n entier naturel
Sortie : B entier naturel, seuil de friabilité

Retourner exp[½* racine[log(n)*log(log(n))]]

Exponentiation_rapide :
Entrée : x réel, n entier naturel
Sortie : Exponentiation de x^n

Si n = 1 : retourner x
Si n mod 2 = 0 : retourner Exponentiation_rapide(x^2,n/2)
Si n mod 2 = 1 : retourner x*Exponentiation_rapide(x^2,(n-1)/2)

Legendre : (Symbole de Legendre, permet de vérifier les résidus quadratiques)
Entrée : a, p entiers
Sortie : Symbole de Legendre

Retourner Exponentiation_rapide(a,(p-1)/2) mod p

Base_Facteurs :
Entrée : n entier naturel, B seuil de friabilité, entier
Sortie : base des facteurs (liste nombre premiers < B avec restriction des résidus quadratiques)

```
B = Seuil_Friabilite(n)
liste_premier= Erasthothene(B)
base_facteur= liste vide
Ajouter 2 à base facteur

Pour i allant de 2 à longueur(liste_premier):
        Si Legendre(n,liste_premier[i]) = 1): // (Si c'est un résidu quadratique)
                Ajouter liste_premier[i] à base_facteur

Retourner base_facteur
```

- Fonctions pour le Criblage :

Shanks_Tonelli : Permet de trouver les racines des équations du type $r^2 = n \bmod p$
Entrée : n, p entier naturels
Sortie : r, p-r

```
q=s-1
s=0

Tant que q mod 2 = 0 :
        q= q/2
        s=s+1

Si s = 1 :
        r= n^(p+1)/4  mod p
        retourner [r, p-r]
Pour z allant de 3 à p:
        Si p-1 = Legendre(z,p) : // Si p-1 est un résidu quadratique
                Stop boucle
m=s
c = z^q mod p
t = n^q mod p
r = n^(q+1)/2 mod p
d = 0

Tant que (t-1 mod p != 0 ) :
        d= t² mod p
        Pour i allant de 2 à M-1
                Si  d-1 mod p = 0 : Stop boucle
                d = d² mod p
        b = c^(2*m-i-1)
        m = i
        c = b² mod p
        t = t *b² mod p
        r = r*b mod p
retourner [r,p-r]
```

liste_crible : Génère la liste des congruences du type x^2-n
Entrée : n, Intervalle entiers naturels
Sortie : liste d'équations du type $x^2 - n$ en partant de la racine de n

```
liste_congruence= liste vide
racine_N = racine(n)

Pour i allant de racine_N à (racine_N + Intervalle): ajouter i²-n à liste_congruence
retourner liste_congruence
```

Criblage : Recherche des congruences tq a^2 congru b^2 mod n
Entrée : base_facteur liste, n, Intervalle entiers naturels
Sortie : liste des nombres friables (x^2-N), liste des a_1,..a_n (les x en partant de la racine), index liste des indices de ces nombres

```
racine_N= racine(n)

liste_congruence=liste_crible(n,intervalle)
liste_cribler = faire une copie de liste_congruence

Si base facteur[1] = 2 : // (premier élément de base facteur)
        i=0
        Tant que liste_cribler[i] mod 2 != 0 : incrémenter i

        Pour j allant de i à la longueur de liste_cribler, pas 2 :
                Tant que liste_cribler[j] mod 2 = 0 :
                        liste_cribler[j] = liste_cribler[j] / 2

Pour premier dans base_facteur de 2 à la fin de la liste :
        solutions_residues = Shanks_Tonelli(N,premier)
        Pour r dans solutions_residues :
                Pour k allant de r-racine_N mod premier à longueur de liste_cribler, pas
                premier:
                        Tant que liste_cribler[k] mod premier = 0 :
                                liste_cribler[k] = liste_cribler[k] / premier

liste_num_friables = liste vide (liste des nombres friables x²-N)
a_list = liste vide (liste des x)
index = liste vide (liste des indices de ces nombres)

Pour i allant de 1 à longueur de liste_cribler :
        Si longueur de liste_num_friables >= longueur de base_facteur :
                Stop boucle

        Si liste_cribler[i] = 1 :
                ajouter  liste_congruence[i] dans liste_num_friables
                ajouter i+racine_N dans a_list
                ajouter i dans index

retourner liste_num_friables, a_list, index
```

- Fonctions pour L'Algèbre Linéaire :

decompo_facteur : Prend un nombre x et le décompose en facteurs premiers (où les facteurs premiers sont dans la base des facteurs)
Entrée : x entier relatif, base_facteur liste d'entiers
Sortie : liste des facteurs

```
liste_facteur = liste vide
Si x < 0 : ajouter -1 dans la liste des facteurs
Pour premier dans base_facteur :
        Si premier != 1 :
                Tant que x mod premier = 0 :
                        ajouter premier dans la liste facteur
                        x = x /premier
retourner liste_facteur
```

Matrice : Génère les vecteurs des exposants mod 2 de la liste des nombres friables par rapport à la base des facteurs sous forme de matrice
Entrée : liste_num_friables liste des nombres friables, base_facteur
Sortie : booléen si la matrice est carré, transposée de M liste de liste d'entiers (matrice)

```
M = liste vide
carre = Faux
Insérer -1 dans base_facteurs

Pour x allant de 1 à longueur de liste_num_friables :
        x_liste_decompo = decompo_facteur(liste_num_friables[x],base_facteur)
        liste_exposant = longueur de base_facteur

        Pour i allant de 1 à longueur de base_facteurs :
                Si base_facteurs[i] dans x_liste_decompo :
                        liste_exposant[i] = liste_exposant[i] + compter le nombre de facteurs
                        dans x_liste_decompo[base_facteur[i] mod 2

        Si 1 n'est pas dans liste_exposant : carre = Vrai et retourner carre, x

        Ajouter liste_exposant dans M
retourner carre, transpose de M
```

Elimination de Gauss :
Entrée : M liste de liste d'entiers (matrice de 0 et 1)
Sortie : eq_param liste des équations dépendantes, pivot_verif liste de booléens si la ligne admet un pivot ou non, M liste de liste d'entiers (matrice)

```
pivot_verif = [False] * longueur de M
Pour i allant de  1 à longueur de M :
        ligne = M[ i ]

        Pour m_i allant de 1 à longueur de ligne :
                Si ligne[m_i] = 1 :
                        indice = index de ligne[m_i] dans ligne
                        pivot_verif[indice = Vrai
                        Pour j allant de 0 à i puis de i+1 à longueur de M :
                                Si M[ j ][indice] = 1 :
                                        Pour i allant de 1 à longueur de M[ j ] :
                                                M[ j ][ i ] = M[ j ][ i ] + ligne[ i ] mod 2
                        STOP Boucle
M = transpose de M

eq_param = liste vide (représentera le système d'équations à résoudre pour avoir une solution)

Pour i allant de 1 à longueur pivot_verif :
        Si pivot_verif[ i ] = False :
                ligne_libre = [ M[ i ], i ]
                ajouter ligne_libre à eq_param

Si eq_param = liste vide : retourner pas de solutions
retourner eq_param, pivot_verif, M
```

Vect : résout et trouve le vecteur dans les équations paramétrées
Entrée : eq_param liste des équations paramétrés, M matrice, pivot_verif liste de booléen, K entier
Sortie : vect liste des solutions

```
vect = liste vide
indices = liste vide
ligne_libre = eq_param[0][0]

Pour i allant de 1 à longueur de ligne_libre :
        Si ligne_libre[ i ] = 1 :
                Ajouter i dans indices

Pour ligne allant de 1 à longueur de M :
        Pour index dans indices :
                Si M[ligne][index] = 1 et pivot_verif[ ligne] :
                        ajouter ligne dans vect
STOP BOUCLE

Ajouter vect dans eq_param[1][2]
Retourner vect
```

- Fonctions pour la Factorisation :

large_sqrt : retourne la racine d'un nombre x pour x très grand
Entrée : n entier naturel
Sortie : x entier racine de n

```
x = n
y = (x+1) / 2
Tant que x < y :
        x =  y
        y = (x + n / x ) / 2
retourner x
```

Factorisation :
Entrée : vect liste des solutions, liste_num_friables liste d'entiers, a_liste liste d'entier (x d'origine), n entier naturel
Sortie : facteur, entier naturel

```
nb_solution = liste vide
Pour i dans vect : ajouter liste_num_friables[ i ] dans nb_solution

nb_value = liste vide
Pour i dans vect : ajouter a_liste[ i ] dans nb_value

// Multiplication de tous les facteurs premiers
ysol = 1
Pour n dans nb_solution :  ysol = ysol * n

// Multiplication de tous les x
x = 1
Pour n dans nb_value : x = x^n

y = large_sqrt(ysol)

facteur = pgcd de x - y et n
retourner facteur
```