Rapport TER

Robin Joran - Slimani Arezki - Boudo Ibrahim

Projet M1 Informatique
Tests De Primalités

15/11/2019

Table des matières

Table des figures

Liste des Algorithmes

Liste des théorèmes et des définitions

1 Introduction

Ce document est le compte-rendu final de notre projet sur les tests de primalités qui s'inscrit dans le cadre du module *TER* du M1 informatique de l'*UVSQ*.

Les tests de primalités sont des algorithmes qui permettent de savoir si un nombre entier est premier. Ces tests sont indispensables pour la cryptographie à clé publique.

Il existe plusieurs algorithmes de tests de primalités, plus ou moins performants. L'efficacité de ces algorithmes est particulièrement liée à la taille des données entrées.

Notre travail consiste donc à implémenter ces différents tests et de comparer leurs performances.

Dans la première partie de ce document, on présentera l'architecture de notre application, illustrée par un organigramme.

Dans la seconde partie, on présentera les tests de primalités qu'on implémentera : algorithmes,compléxités et preuves.

Pour finir, on établira un comparatif entre ces algorithmes de tests de primalités.

2 Architecture de l'application

2.1 Organigramme et données échangées

L'application développée dans ce projet porte essentiellement sur l'implémentation des tests de primalités,avec quelques fonctionnalités supplémentaires. La manière dont l'application est structurée va permettre de faciliter son extension et sa réutilisation.

Cet organigramme représente la décomposition en modules de l'application ainsi que les informations qui circulent entre ces modules.

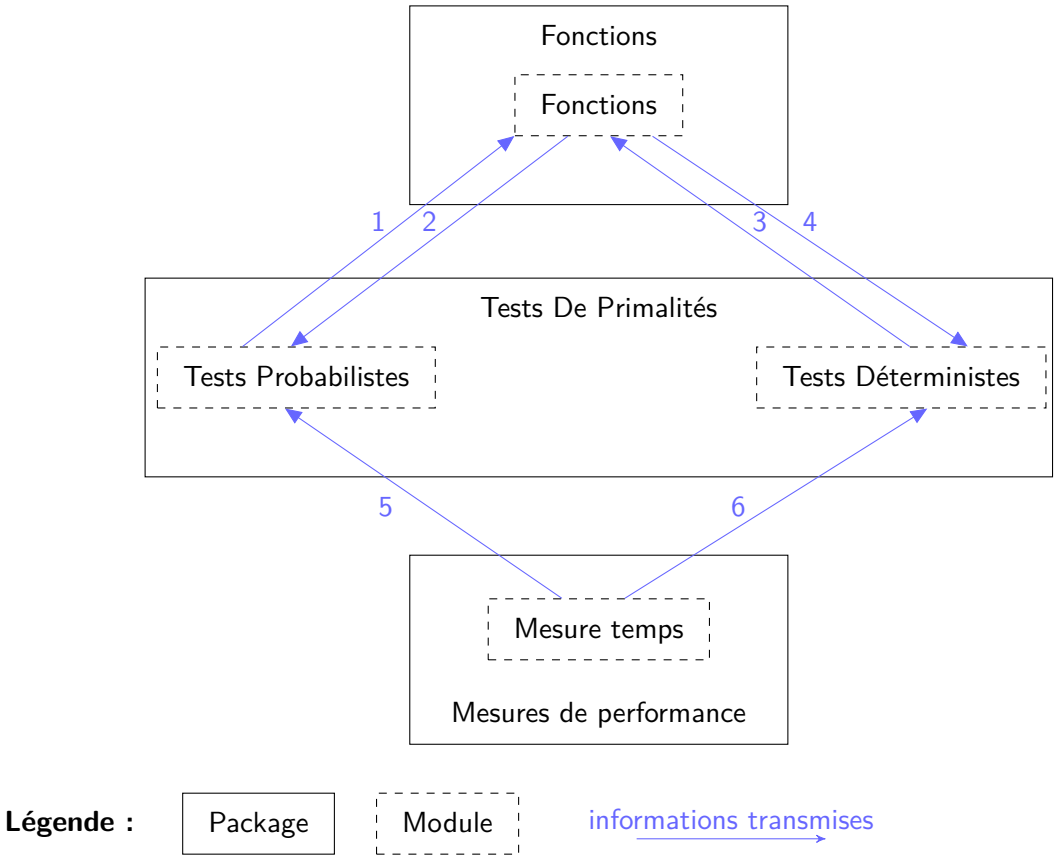

FIGURE 1 – Organigramme des différents modules de l'application

Notes :
(1) Transmet des nombres à tester dans les fonctions
(2) Renvoie le résultat des fonctions utilisées
(3) Transmet des nombres à tester dans les fonctions
(4) Renvoie le résultat des fonctions utilisées
(5) Transmet un nombre premier et écrit le résultat dans un fichier mesures.txt
(6) Transmet un nombre premier et écrit le résultat dans un fichier mesures.txt

2.2 Fonctionnalités des modules

2.2.1 Package Fonctions

1. Module Fonctions : implémentation de fonctions nécessaires pour le bon déroulement des algorithmes des Tests De Primalités :
   - PGCD
   - Exponentiation Rapide
   - Decomposition
   - Generation aléatoire

— Temoin De Miller
— Symbole De Jacobi
— Critère d'Euler
— Nombre De Fermat
— Calcul Sequence
— Mersen
— Suite De Fibonnacci
— Nombres De Lucas
— Nombre d'or
— Calcul Discriminant
— Polynôme De Fibonnacci
— Polynôme De Lucas
— Chaîne De Lucas

2.2.2 Package Tests De Primalités

1. Module Tests Probabilistes : implémentation de différents algorithmes de test probabilistes qui feront l'objet d'une étude comparative par la suite :
   — Test Fermat
   — Test de Miller-Rabin
   — Test de Solovay-Strassen
   — Test de Lucas-Frobenius
2. Module Tests Déterministes : implémentation de différents algorithmes de test déterministes qui feront l'objet d'une étude comparative par la suite :
   — Test Erastothène
   — Test de Pepin
   — Test de Lucas-Lehmer

2.2.3 Package Mesures de performance

1. Module Mesure : mesure du temps d'exécution de chaque test de primalités selon la taille en bits du nombre à tester.

2.3 Outils et langages de programmation

Notre application va être implémentée dans le langage `C`. Le langage `C` possède plusieurs types pour représenter des nombre entiers. Cependant, tous ces types ont une précision fixe et ne peuvent pas dépasser un certain nombre d'octets. Le type le plus grand est le `long long int` qui peut contenir des entiers d'une taille maximale de 64 bits. Or, tous ces types sont beaucoup trop courts pour les applications cryptographiques qui nécessitent la manipulation de données d'au moins 512 bits.

Nous allons donc utilisé `GNU MP` pour `GNU Multi Precision`, souvent appelée `GMP` qui est une bibliothèque `C/C++` de calcul multiprécision sur des nombres entiers, rationnels et à virgule flottante qui permet en particulier de manipuler de très grand nombres.

Finalement, le logiciel `Gnuplot` va être aussi utilisé pour faire des représentations graphiques à partir des résultats issus des mesures de performance de notre application.

3 Tests de primalités

Les tests de primalités sont des algorithmes qui permettent de savoir si un nombre entier est premier. Dans le cas où le nombre n'est pas premier, il est dit ***composé***. Dans cette partie, on va détailler différents algorithmes de tests de primalités.

Les tests de primalités peuvent être :

— **déterministes :** fournissent toujours la même réponse pour un nombre donné.

— **probabilistes :** peuvent fournir des réponses différentes pour un même nombre (utilisent des données tirées aléatoirement (des témoins)).

Voici la liste des différents algorithmes de tests de primalités qu'on va plus ou moins aborder :

Algorithme	Année	Type
Naïf (Crible d'Eratosthène)	-240	Déterministe
Fermat	1640	Probabiliste
Miller-Rabin	1976	Probabiliste
Solovay-Strassen	1977	Probabiliste
Pepin	XIX^{ème} siècle	Déterministe
Lucas-Lehmer	1878,1930	Déterministe
Lucas-Frobenius	XIX^{ème} siècle	Probabiliste

Certains tests seront énoncés rapidement du fait qu'il ne sont pas assez performants. Par contre, on s'intéressera plus en détail aux tests de *Fermat*, *Miller-Rabin*, *Solovay-Strassen*, *Lucas-Frobenius*.. Pour chacun de ces tests, on donnera un bref historique, son algorithme, sa complexité et sa preuve. L'application développée au cours de ce projet contient l'implémentation de tous ces tests.

3.1 Test Naïf - Crible d'Eratosthène

Le test Naïf représente l'idée la plus intuitive pour tester la primalité d'un nombre entier. Pour décider si un nombre n est premier ou composé, on teste si les entiers $2,3,...,n-1$ divisent n. Si un parmi ces entiers divise n alors on déduit que n est composé, sinon on conclut qu'il est premier. Ceci revient à factoriser le nombre en question.

Pour améliorer cet algorithme, on sait qu'un diviseur d'un entier n quelconque ne peut dépasser $n/2$. De plus, si n possède un diviseur plus grand que $\sqrt{n}$, alors il a forcement au moins un diviseur plus petit que $\sqrt{n}$. On peut donc accélérer la recherche en prenant en compte que des nombres premiers inférieurs à $\sqrt{n}$. Pour cela il suffit de pré-calculer et de stoker dans une table tous les nombres premiers $\leqslant \sqrt{n}$. Le ***crible d'Eratosthène*** par exemple peut être utilisé dans ce but.

Algorithme 1 : Test naïf

Données : un entier n
pour *tout nombre premier* $p \leqslant \sqrt{n}$ **faire**
    si p *divise* n **alors**
        retourner composé;
retourner premier;

3.1.1 Crible d'Eratosthène

Ce crible est un procédé établi par *Eratosthène*, un mathématicien grec du III[e] siècle av. J.-C., qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné N. Dans notre cas, cet entier donné est n, n étant le nombre dont on va tester la primalité.

L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers :

— retirer les multiples du plus petit entier premier restant (multiples de 2, puis de 3, etc.)
— on peut s'arrêter lorsque le carré de ce plus petit entier premier restant est supérieur au plus grand entier premier restant, car dans ce cas, tous les non-premiers ont déjà été retirés précédemment
— à la fin du processus, tous les entiers qui n'ont pas été rayés sont les nombres premiers inférieurs à N

L'algorithme du crible est le suivant :

Algorithme 2 : Crible d'Eratosthène

Données : un entier N qui correspond à $\sqrt{n}$
Créer une liste L de couples *(entier, primalité)*, pour les entiers allant de 2 jusqu'à N, avec une primalité initialisée à "premier" : ***L = {(2, premier), (3, premier), ..., (N, premier)}*** ;
borneSupPremier $= N$;
pour *tout nombre* p *marqué "premier" de la liste L (de manière croissante)* **faire**
    si $p^2 >$ ***borneSupPremier*** **alors**
        retourner L;
    m $= 2$;
    tant que $p * m < N$ **faire**
        Marquer "composé" l'entier à la position $p * m$;
        Mettre à jour ***borneSupPremier*** ;
        $m{+}{+}$;

Complexité

La complexité en temps de l'algorithme 1 (*Test Naïf*) dans le pire des cas est de $\pi(n) \approx \frac{2\sqrt{n}}{ln(n)}$ division, c'est-à-dire $O(\sqrt{n})$ opérations.

3.2 Test de Fermat

Le test de Fermat est un test de primalité probabiliste basé sur le *petit théorème de Fermat* :

> **Théorème 1** (Petit théorème de Fermat (énoncé 1)). *Si p est un nombre premier, alors pour tout nombre entier a premier avec p*
>
> $$a^{p-1} \equiv 1 \pmod{p}$$

Il existe un énoncé équivalent de ce théorème, qui est le suivant :

> **Théorème 2** (Petit théorème de Fermat (énoncé 2)). *Si p est un nombre premier, et a un nombre entier quelconque, alors*
>
> $$a^{p} \equiv a \pmod{p}$$

Ce théorème doit son nom à *Pierre de Fermat*, qui l'énonce la première fois en 1640.

3.2.1 Algorithme

Le premier énoncé du *théorème de Fermat* va être exploité pour construire l'algorithme de test de primalité. Ce théorème décrit une propriété commune à tous les nombres premiers qui peut être utilisée pour détecter si un nombre est premier ou bien composé.

En effet, pour un entier n dont on veut tester la primalité et un entier a quelconque tel que $1 < a < n-1$:

— Le fait de choisir $1 < a < n-1$ garantit que si n était premier, a sera forcément premier avec n (puisque $a < n-1$) et ainsi le test n'échouera pas.

— Si $\mathbf{a^{n-1} \not\equiv 1} \pmod{\mathbf{n}}$, alors n est surement composé.

Parmi les entiers a qui ne vérifient pas l'inégalité de Fermat, il y a évidement ceux qui ne sont pas premiers avec n. Si l'on trouve un tel entier a (qu'il soit premier ou non avec n), on dit que a est un ***témoin de non primalité*** de n issu de la divisibilité (*témoin de Fermat).*

> **Définition 1** (Témoin de Fermat). *Soit un entier $n \geqslant 2$. On appelle témoin de Fermat pour n, tout entier a, tel que*
>
> $$1 < a < n-1 \quad et \quad a^{n-1} \not\equiv 1 \pmod{n}$$

— Si $\mathbf{a^{n-1} \equiv 1} \pmod{\mathbf{n}}$, on ne peut pas conclure avec certitude que n est premier puisque la réciproque du *théorème de Fermat est fausse* (théorème 1).

Un nombre n vérifiant cette équation peut être premier, mais aussi composé, dans ce cas n est dit ***pseudo-premier*** *de base* a ou menteur.

> **Définition 2** (Nombre pseudo-premier)**.** *Un nombre pseudo-premier est un nombre premier probable (un entier naturel qui partage une propriété commune à tous les nombres premiers) qui n'est en fait pas premier. Un nombre pseudo-premier provenant du théorème de Fermat est appelé nombre pseudo-premier de Fermat.*

Si un nombre pseudo-premier n de base a est pseudo-premier pour toutes les valeurs de a qui sont premières avec n est appelé ***nombre de Carmichael***.

> **Définition 3** (Nombre de Carmichael)**.** *Un entier positif composé n est appelé nombre de Carmichael si pour tout entier a premier avec n,*
>
> $$a^{n-1} \equiv 1 \pmod{n}$$

L'entier $n = 561 = 3 \cdot 11 \cdot 17$ est le plus petit nombre de Carmichael puisque $a^{560} \equiv 1 \pmod{561}$ pour tout entier a premier avec 561. Les nombres de Carmichael sont très rares. Il existe par exemple seulement $246\ 683$ nombres de Carmichael inférieurs à 10^{16}. Le nombre de premiers inférieurs à 10^{16} est quant à lui égal à $279\ 238\ 341\ 033\ 925$. Donc la probabilité qu'un nombre premier inférieur à 10^{16} soit un nombre de Carmichael est plus petite que $1/10^9$.

Les nombres pseudo-premiers et les nombre de Carmichael sont relativement rares. On peut donc envisager d'adopter ce critère pour un test probabiliste de primalité, qui est le *test de Fermat*. En effet, le test va être répété k fois, et à chaque itération, on effectue le test avec une base a différente. Plus le nombre de répétitions est grand, plus la probabilité que le résultat du test soit correct augmente.

L'algorithme de test de primalité qu'on obtient finalement est le suivant :

Algorithme 3 : Test de Fermat

Données : un entier n et le nombre de répétitions k
pour $i = 1$ *jusqu'à* k **faire**
  Choisir aléatoirement a tel que $1 < a < n-1$;
  si $a^{n-1} \not\equiv 1 \pmod{n}$ **alors**
  retourner composé;
retourner probablement premier;

3.2.2 Complexité

La complexité de ce test va dépendre de l'exponentiation modulaire utilisée dans le corps de la boucle de k itérations.

Algorithme 4 : Square-and-Multiply (Left-to-right binary method)

Données : c, d, et n entiers : avec $d = \sum_{i=0}^{k-1} d_i \cdot 2^i$ et $d_{k-1} = 1$
Sorties : c^d modulo n
$x \leftarrow c$;
pour $i = k-2$ *jusqu'à* 0 **faire**
    $x \leftarrow x^2 \bmod n$;
    si $d_i = 1$ **alors**
    $x \leftarrow x \cdot c \bmod n$;
retourner x;

L'analyse de la complexité cet algorithme (dont on va admettre la preuve ici) nous donne un temps en $log(d)$ multiplications modulaires dépendantes de n.

Appliqué au contexte du *test de Fermat*, on a un exposant $n-1$ et des multiplications modulaires dépendantes de n.

Pour conclure, il nous reste à voir la complexité d'une multiplication modulaire. Étant donné que nous travaillons sur des entiers de grande taille, la multiplication ne sera pas en temps constant. Il existe un algorithme "naïf" et deux multiplications dites rapides que nous allons simplement citer :

— Une implémentation "naïve" consiste à effectuer les calculs comme pour une multiplication de primaire. À l'aide de deux boucles imbriquées, on multiplie chiffre par chiffre nos nombres.

  En appelant n notre nombre, sa taille est de l'ordre de $log(n)$, la complexité de cette multiplication est donc $log(n)^2$.

— La méthode FFT (*Fast Fourrier Transformation*) effectue la multiplication en complexité :

$$log(n) \cdot log(log(n)) \cdot log(log(log(n)))$$

— La méthode *Karatsuba* quant à elle calcule le résultat en complexité $log(n)^{log_2(3)}$.

On obtient une complexité finale pour le *Test de Fermat* de :

$$O(k \cdot log_2(n) \cdot C_{mult}(n))$$

où k est le nombre de répétitions dans le *Test de Fermat*, n l'entier testé et $C_{mult}(n)$ la complexité de la multiplication modulaire.

3.2.3 Preuve

Pour démontrer la correction du test de primalité, nous allons d'abord prouver le *petit théorème de Fermat*. Pour cela nous nous basons sur le fait que c'est un cas particulier du *théorème d'Euler* :

Théorème 3 (Théorème d'Euler)**.** *Soit n un naturel supérieur ou égal à 2, et a un entier premier avec n, alors*

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

où ϕ est la fonction indicatrice d'Euler :

$$\begin{aligned} \phi : \mathbb{N} &\to \mathbb{N} \\ n &\mapsto |\{k,\ 1 \leq k \leq n \text{ et } pgcd(k,n) = 1\}| \end{aligned}$$

La fonction ϕ évaluée sur un nombre premier p vaut $p-1$. Le *petit théorème de Fermat* est donc une application du théorème d'Euler en remplaçant n par un nombre premier p.

Preuve du *théorème d'Euler* :

En prouvant le *théorème d'Euler*, on aura prouvé *petit le théorème de Fermat*, on conclura ensuite avec le preuve de l'algorithme.

On effectuera les calculs dans le groupe multiplicatif $(\mathbb{Z}/n\mathbb{Z})^*$, l'ensemble des naturels inférieurs à n inversibles modulo n, ou de manière équivalente, l'ensemble des naturels inférieurs à n premiers avec n.

On considère l'application suivante, avec $\alpha \in (\mathbb{Z}/n\mathbb{Z})^*$:

$$\begin{aligned} \Gamma_\alpha : (\mathbb{Z}/n\mathbb{Z})^* &\to (\mathbb{Z}/n\mathbb{Z})^* \\ x &\mapsto \alpha \cdot x \end{aligned}$$

C'est une bijection. En effet son application inverse est $\Gamma_{\alpha^{-1}}$. $\alpha \in (\mathbb{Z}/n\mathbb{Z})^*$, donc il existe $\alpha^{-1} \in (\mathbb{Z}/n\mathbb{Z})^*$ inverse de α modulo n. C'est également une permutation (bijection d'un ensemble vers lui-même), on a donc :

$$\begin{aligned} \prod_{x \in (\mathbb{Z}/n\mathbb{Z})^*} x &= \prod_{x \in (\mathbb{Z}/n\mathbb{Z})^*} \Gamma_\alpha(x) \\ &= \alpha^{|(\mathbb{Z}/n\mathbb{Z})^*|} \cdot \prod_{x \in (\mathbb{Z}/n\mathbb{Z})^*} x \\ &= \alpha^{\phi(n)} \cdot \prod_{x \in (\mathbb{Z}/n\mathbb{Z})^*} x \end{aligned}$$

$\prod_{x \in (\mathbb{Z}/n\mathbb{Z})^*} x$ est inversible (produit d'éléments inversibles), donc on simplifie :

$$\alpha^{\phi(n)} \equiv 1 \text{ modulo } n$$

Preuve de l'algorithme : directement, en se basant sur la contraposée du *petit théorème de Fermat* on a :

Soit a un entier premier avec p, alors $a^{p-1} \not\equiv 1$ modulo $n \Rightarrow p$ non premier.

Remarque : on voit bien avec cette preuve qu'on peut affirmer avec certitude qu'un nombre qui ne passe pas le test est composé. Cela dit, si le test passe, ceci ne confirme pas forcément que notre nombre est premier.

3.3 Test de Miller-Rabin

Le test de Miller-Rabin est un autre test de primalité probabiliste basé sur le *petit théorème de Fermat* (théorème 1). Il exploite quant à lui quelques propriétés supplémentaires. La version originale de ce test, publiée par *Gary L. Miller* en 1976, est déterministe, mais ce déterminisme dépend d'une hypothèse non démontrée (hypothèse de Riemann généralisée). En 1980, *Michael Rabin* a modifié cette hypothèse pour obtenir un algorithme de test probabiliste inconditionnel.

3.3.1 Algorithme

Dans le cas du *test de Miller-Rabin*, la propriété en question est un raffinement du *petit théorème de Fermat* (théorème 1). De la même façon que le test de Fermat, le *test de Miller-Rabin* tire parti d'une propriété de l'entier n dont on va tester la primalité, qui dépend d'un entier auxiliaire, le témoin a, et qui est vraie dès que n est un nombre premier. Le principe de ce test probabiliste est donc de vérifier cette propriété pour suffisamment de témoins.

En effet, soit $n > 2$ un entier dont on va tester la primalité et a un entier quelconque tel que $1 < a < n$. Soient $s \in \mathbb{N}^*$ et $t \in \mathbb{N}$ ***impair***, on peut écrire $\mathbf{n-1 = 2^s t}$ (s est le nombre maximum de fois que l'on peut mettre 2 en facteur dans $n-1$).

Alors, dans le cas où n est premier, d'après le *petit théorème de Fermat* (théorème 1) on a :

$$\begin{aligned} a^{n-1} \equiv 1 \pmod{n} &\Leftrightarrow a^{2^s t} \equiv 1 \pmod{n} \\ &\Leftrightarrow a^{2^s t} - 1 \equiv 0 \pmod{n} \\ &\Leftrightarrow a^{2^s t} - 1 = (a^{2^{s-1} t})^2 - 1 = (a^{2^{s-1} t} + 1)(a^{2^{s-1} t} - 1) \equiv 0 \pmod{n} \quad (\text{puisque } s \geqslant 1) \end{aligned}$$

Si $s - 1 > 0$ alors le dernier terme est de nouveau une différence de carrés qui peut donc être factorisée. En continuant de la même manière, on obtient au final l'expression suivante :

$$(a^{2^{s-1} t} + 1)(a^{2^{s-2} t} + 1)...(a^t + 1)(a^t - 1) \equiv 0 \pmod{n} \qquad (*)$$

On sait que pour un nombre premier p, si $ab \equiv 0 \pmod{p}$ alors $a \equiv 0 \pmod{p}$ ou $b \equiv 0 \pmod{p}$. Par conséquent, si n est premier, alors l'équation (*) est vraie si et seulement si un des termes de sa partie gauche est $0 \pmod{n}$. Autrement dit, si n est premier alors

$$a^{2^j t} \equiv -1 \pmod{n} \quad \text{pour au moins un } j \in \{0, 1, ..., s-1\}$$

ou

$$a^t \equiv 1 \pmod{n}$$

Si pour un nombre entier n, une des équations ci-dessus est vérifiée, alors l'algorithme conclut que n est probablement premier et termine. Si aucune de ces équations n'est vérifiée, alors l'algorithme renvoie que n est composé avec certitude. On peut aussi apporter une amélioration à cette approche.

Si on trouve que

$$a^{2^j t} \equiv 1 \pmod{n} \quad \text{pour un } j \in \{0, 1, ..., s-1\},$$

on peut directement conclure que n est composé et terminer l'exécution de l'algorithme. Ceci est dû au fait que pour un nombre p premier, les seuls éléments pour lesquels un entier x vérifie $x^2 \equiv 1 \pmod{p}$ sont 1 et -1, qui sont les deux seules racines carrées de l'unité.

Compte tenu de tous les éléments développés ci-dessus, l'algorithme du test de primalité de Miller-Rabin est le suivant :

Algorithme 5 : Test de Miller-Rabin

Données : un entier n et le nombre de répétitions k
Décomposer $n-1 = 2^s t$, avec $s \in \mathbb{N}^*$ et $t \in \mathbb{N}$ impair ;
pour $i = 1$ *jusqu'à* k **faire**
  Choisir aléatoirement a tel que $1 < a < n$;
  $y \leftarrow a^t \pmod{n}$;
  si $y \not\equiv 1 \pmod{n}$ *et* $y \not\equiv -1 \pmod{n}$ **alors**
    pour $j = 1$ *jusqu'à* $s-1$ **faire**
      $y \leftarrow y^2 \pmod{n}$;
      si $y \equiv 1 \pmod{n}$ **alors**
        retourner composé;
      si $y \equiv -1 \pmod{n}$ **alors**
        Arrêter la boucle de j et continuer avec le i suivant (sans renvoyer composé);
      retourner composé;
retourner probablement premier;

Probabilité d'erreur et nombre d'itérations : si le *test de Miller-Rabin* renvoie composé, alors le nombre est effectivement composé. Il peut être démontré que si le test de Miller-Rabin dit que n est premier, le résultat est faux avec une probabilité inférieure à $1/4$. En effet, il existe des valeurs de a qui produiront de manière répétée des menteurs, qui indiqueront donc que n est premier alors qu'il est composé. On appelle un ***témoin*** fort pour n un entier a pour lequel

$$a^t \not\equiv 1 \pmod{n} \quad \text{et} \quad a^{2^s j} \not\equiv -1 \pmod{n} \quad \text{pour tout } j \in \{0, ..., s-1\}$$

Il peut être montré qu'il existe toujours un témoin fort pour n'importe quel composé impair n, et qu'au moins $3/4$ de ces valeurs pour a sont des témoins forts pour la composition de n. Si on répète ce test k fois, la probabilité que le résultat soit toujours faux décroit très rapidement. La probabilité que ce test renvoie premier à tort après k itérations est donc de $1/4^k$. Or probabilité du test de Fermat est de $1/2^k$ et aussi $1/2^k$ pour test de Solovay-Strassen.

3.3.2 Complexité

Le *test de Miller-Rabin* est similaire au test de Fermat. Pour prouver que nous effectuons bien de l'ordre de $log(n)$ multiplications modulaires il faut constater que nous effectuons dans Miller-Rabin $s + log_2(t)$ multiplications modulaires, c'est-à-dire $log_2(2^s \cdot t)$ ou bien $log_2(n-1)$ car $n-1 = 2^s \cdot t$

3.3.3 Preuve

La preuve de cet algorithme repose sur la preuve du *petit théorème de Fermat* de la partie précédente et des explications détaillés du principe de l'algorithme au début de cette partie.

3.4 Test de Solovay-Strassen

Le *test de Solovay-Strassen* est un test de primalité probabiliste, publié par *Robert Solovay* et *Volker Strassen* en 1977.

3.4.1 Algorithme

L'algorithme du *test de Solovay-Strassen* est essentiellement basé sur le ***critère d'Euler***, un théorème qui énonce que :

Théorème 4 (Critère d'Euler)**.** *Soient $p > 2$ un nombre premier et a un entier premier avec p*

- *Si a est un résidu quadratique modulo p, alors $a^{\frac{n-1}{2}} \equiv 1 \pmod{p}$.*
- *Si a n'est pas un résidu quadratique modulo p, alors $a^{\frac{n-1}{2}} \equiv -1 \pmod{p}$.*

Ceci se résume en utilisant le symbole de Legendre par :

$$a^{\frac{n-1}{2}} \equiv \left(\frac{a}{p}\right) \pmod{p}$$

Définition 4 (Résidu quadratique)**.** *On dit qu'un entier q est un résidu quadratique modulo p s'il existe un entier x tel que :*

$$x^2 \equiv q \pmod{p}$$

Autrement dit, un résidu quadratique modulo p est un nombre qui possède une racine carrée de module p. Dans le cas contraire, on dit que q est un non-résidu quadratique modulo p.

Le ***symbole de Legendre*** est utilisé pour résumer le *critère d'Euler*. Il est définit de la manière suivante :

Définition 5 (Symbole de Legendre)**.** *Le symbole de Legendre est une fonction de deux variables entières à valeurs dans $\{-1, 0, 1\}$. Si p est un nombre premier et a un entier, alors le symbole de Legendre $\left(\frac{a}{p}\right)$ vaut :*

$$\begin{cases} 0 & \textit{si } a \textit{ est divisible par } p \\ 1 & \textit{si } a \textit{ est un résidu quadratique modulo } p \textit{ mais pas divisible par } p \\ -1 & \textit{si } a \textit{ n'est pas un résidu quadratique modulo } p \end{cases}$$

Le cas particulier $p = 2$ est inclus dans cette définition mais est sans intérêt : $\left(\frac{a}{p}\right)$ vaut 0 si a pair et 1 sinon.

Pour pouvoir exploiter le *critère d'Euler* dans l'algorithme du test de primalité, on doit pouvoir calculer le *symbole de Legendre* pour tout entier n dont on veut tester la primalité. On introduit donc le ***symbole de Jacobi*** qui est une généralisation du *symbole de Legendre*, définit de la manière suivante :

Définition 6 (Symbole de Jacobi)**.** *Le symbole de Jacobi $\left(\frac{a}{n}\right)$ est défini, $\forall a \in \mathbb{Z}$ et $n \in \mathbb{N}$ impair, comme produit de symboles de Legendre, en faisant intervenir la décomposition en facteurs premiers de n. Si $n = p_1 * p_2 * ... * p_k$ pour $k \in \mathbb{N}$ telle que $p_1, p_2, ..., p_k$ sont des nombres premiers non nécessairement distincts, alors :*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{\prod_{i\in\{1,...,k\}} p_i}\right) = \prod_{i\in\{1,...,k\}} \left(\frac{a}{p_i}\right)$$

Compte tenu des notions décrites ci-dessus, on peut maintenant construire l'algorithme de test de primalité de Solovay-Strassen :

Algorithme 6 : Test de Solovay-Strassen

Données : un entier n impair et le nombre de répétitions k
pour $i = 1$ *jusqu'à* k **faire**
    Choisir aléatoirement a tel que $2 < a < n$;
    $x \leftarrow \left(\frac{a}{n}\right)$;
    si $x = 0$ *ou* $x \not\equiv a^{\frac{n-1}{2}} \pmod{n}$ **alors**
        retourner composé;
retourner probablement premier;

Cet algorithme exploite essentiellement le *critère d'Euler* (théorème 4). En effet, pour un entier n dont on veut tester la primalité et un entier a quelconque telle que $2 < a < n$:

— Si $\mathbf{a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod{n}}$, alors n est surement composé.

Parmi les entiers a qui ne vérifient pas le *critère d'Euler* (théorème 4), il y a évidement ceux qui ne sont pas premiers avec n. Si l'on trouve un tel entier a (qu'il soit premier ou non avec n), on dit que a est un ***témoin de non primalité*** de n (*témoin d'Euler*).

Définition 7 (Témoin d'Euler)**.** *Soit un entier $n > 2$. On appelle témoin d'Euler pour n, tout entier a, telle que*

$$2 < a < n \quad et \quad a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod{n}$$

— Si $\mathbf{a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n}}$, on ne peut pas conclure avec certitude que n est premier puisque la réciproque du *critère d'Euler* (théorème 4) est fausse.

Un nombre n vérifiant cette équation peut être premier, mais aussi composé, dans ce cas n est dit ***pseudo-premier d'Euler-Jacobi*** *de base* a ou menteur.

> **Définition 8** (Nombre pseudo-premier d'Euler-Jacobi)**.** *Un nombre pseudo-premier d'Euler-Jacobi de base a est un nombre composé impair n premier avec a et tel que la congruence suivante soit vérifiée :*
>
> $$a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n}$$

À la différence du test de primalité de Fermat, pour chaque entier composé n, au moins la moitié de tous les a sont des témoins d'Euler. Par conséquent, il n'y a aucune valeur de n pour laquelle tous les a sont des menteurs, alors que c'est le cas pour les nombres de *Carmichael* dans le test de Fermat.

3.4.2 Complexité

Pour étudier la complexité du *test Solovay-Strassen*, il faut étudier l'évaluation du *symbole de Jacobi*. En effet, dans le corps de la boucle, à la différence du *test de Fermat*, avant d'effectuer l'exponentiation modulaire nous allons évaluer un *symbole de Jacobi*. La complexité d'une itération sera de l'ordre du terme dominant entre le *symbole de Jacobi* et l'exponentiation modulaire.

Le *symbole de Jacobi* $\left(\frac{a}{b}\right)$ s'évalue en $O(log(a) \cdot log(b))$. Ainsi, dans le cadre de ce test, le *symbole de Jacobi* s'évalue en $O(log(n)^2)$.

L'évaluation du *symbole de Jacobi* reste dominée par l'exponentiation rapide même couplée d'une multiplication modulaire rapide.

3.4.3 Preuve

Durant cette démonstration, nous allons utiliser et admettre le *théorème de Lagrange*.

> **Théorème 5** (Théorème de Lagrange)**.** *Soient p un nombre premier et $f(X) \in \mathbb{Z}[X]$ un polynôme à coefficients entiers alors :*
>
> — *Soit tous les coefficients de f sont divisibles par p.*
>
> — *Soit $f(X) \equiv 0$ mod p admet au plus $degre(f)$ solutions non équivalentes.*

En admettant le *théorème de Lagrange*, il suffit de prouver le *critère d'Euler* pour avoir une preuve satisfaisante du théorème. En effet, comme énoncé précédemment, le *test de Solovay-Strassen* s'appuie directement sur la contraposée du *critère d'Euler*.

Nous partirons de plusieurs constatations pour prouver le *critère d'Euler* :

1. D'après le *théorème de Lagrange*, comme p est premier, $x^2 \equiv a \bmod p$ admet au plus deux solutions distinctes pour chaque a différent. Donc, hormis $x = 0$, comme chaque racine x peut être accompagnée d'une deuxième racine comme solution de l'équation, il y a au moins $(p-1)/2$ résidus quadratiques a différents.
2. $(\mathbb{Z}/p\mathbb{Z}, +, \cdot)$ est un corps (p premier).

Pour commencer on va partir du *théorème de Fermat* et le réécrire :

$$a^{p-1} \equiv 1 \bmod p \iff (a^{\frac{p-1}{2}} - 1) \cdot (a^{\frac{p-1}{2}} + 1) \equiv 0 \bmod p$$

Grâce à la constatation 2., on obtient que le produit est nul si et seulement si l'un au moins des facteurs est nul. Si a est un résidu quadratique, il existe x tel que $x^2 \equiv a \bmod p$, on a :

$$a^{\frac{p-1}{2}} \equiv (x^2)^{\frac{p-1}{2}} \equiv x^{p-1} \equiv 1 \bmod p$$

La dernière étape est obtenue à l'aide du *petit théorème de Fermat*.

On sait que d'après le *théorème de Lagrange*, $(a^{\frac{p-1}{2}} - 1) \equiv 0 \bmod p$ admet au plus $(p-1)/2$ solutions pour a. On sait également (constatation 1.) qu'il y a au moins $(p-1)/2$ résidus quadratiques modulo p. Donc, il y a exactement $(p-1)/2$ valeurs qui annulent le premier facteur : les résidus quadratiques. Et, les autres $(p-1)/2$ valeurs non-résidus annulent forcément le second terme pour satisfaire le petit *théorème de Fermat*.

En résumé :

— Si a est un résidu quadratique modulo p, alors $a^{\frac{n-1}{2}} \equiv 1 \pmod p$.

— Si a n'est pas un résidu quadratique modulo p, alors $a^{\frac{n-1}{2}} \equiv -1 \pmod p$.

3.5 Test de Pépin

Le *test de Pépin* est un test de primalité déterministe, utilisé pour déterminer si un nombre de Fermat est premier ou non. Ce test porte le nom du mathématicien français *Théophile Pépin*.

3.5.1 Enoncé

Soit $F_n = 2^{2^n} + 1$ qui est le n-ième nombre de Fermat. Le test de Pépin indique que, pour n > 0 :

F_n est premier si et seulement si $3^{(F_n-1)/2} \equiv -1 \pmod{F_n}$.

L'expression $3^{(F_n-1)/2}$ peut être évaluée modulo F_n par exponentiation rapide. Le test a donc une faible complexité en temps. Cependant, les nombres de Fermat se développent si rapidement que seule une poignée de nombres de Fermat peut être testée dans un laps de temps et d'espace raisonnable.D'autres bases peuvent être utilisées à la place de 3, par exemple 5, 6, 7 ou 10.

3.5.2 Algorithme

L'algorithme du test de Pépin est essentiellement basé sur le critère d'Euler(vu avec le Test de Solovay-Strassen) ainsi que sur l'exponentiation rapide pur calculer le nombre de fermat dont l'algorithme est le suivant :

Algorithme 7 : Exponentiation Rapide

Données : un entier x et l'exposant $expo$
si $expo == 0$ **alors**
  retourner 1;
$i = 1$;
$res = x$;
$expoEnBin =$ expo en base 2 ;
while $i < taille$ **do**
  $res = x * x$;
  si $expoEnBin[i] == 1$ **alors**
    $res = res * x$;
  $i++$;
$free(expoEnBin)$;
retourner res;

En comparant à la méthode ordinaire qui consiste à multiplier x par lui-même n – 1 fois, cet algorithme nécessite de l'ordre de O(log n) multiplications et ainsi accélère le calcul de x^n de façon spectaculaire pour les grands entiers.

On peut donc maintenant rentrer dans le vif du sujet avec l'algorithme du Test De Pépin :

Algorithme 8 : Test De Pépin

Données : un entier n qui représente l'indice du nombre de Fermat à tester
si $3^{(F_n-1)/2} \equiv -1 \pmod{F_n}$ **alors**
  retourner premier;
retourner composé;

3.5.3 Compléxité

On utilise deux fois l'exponentiation rapide pour calculer les nombres de Fermat et de la même manière on utilise encore une fois l'exponentiation rapide pour calculer $3^{(F_n-1)/2}$ donc on a une compléxité en O($(\log n)^2$).

3.5.4 Preuve

Supposons que la congruence $3^{(F_n-1)/2} \equiv -1 \pmod{F_n}$ tient.

Ensuite $3^{(F_n-1)} \equiv -1 \pmod{F_n}$ donc l'ordre multiplicatif de 3 modulo F_n divise $F_n - 1 = 2^{2^n}$, qui est une puissance de deux. D'autre part l'ordre ne divise pas $(F_n - 1)/2$, et doit être donc égal à $F_n - 1$. En particulier, il y a bien $F_n - 1$ nombre derrière F_n qui coprime F_n , et peut se produire seulement si F_n

est premier.

Condition nécessaire : supposons que F_n est premier.

$3^{\frac{F_n-1}{2}} \equiv \left(\frac{3}{F_n}\right) \pmod{F_n}$, où $\left(\frac{3}{F_n}\right)$ est le symbole de Legendre.
F_n est premier et $F_n \equiv 1 \pmod 4$, d'où $\left(\frac{3}{F_n}\right) \equiv \left(\frac{F_n}{3}\right)$.
$2^{2^n} \equiv 1 \pmod 3$, donc $F_n \equiv 2 \pmod 3$, et alors :$\left(\frac{F_n}{3}\right) = \left(\frac{2}{3}\right)$.
$\left(\frac{2}{3}\right) = -1$.
Ainsi, $\left(\frac{3}{F_n}\right) = -1$, ce qui implique $3^{(F_n-1)/2} \equiv -1 \pmod{F_n}$

3.5.5 Tests Historiques

En raison de la rareté des nombres de Fermat, le test Pépin n'a été utilisé que huit fois (sur des nombres de Fermat dont les statuts de primalité ne sont pas encore connus).Les chercheurs Mayer, Papadopoulos et Crandall pensent que, en fait, en raison de la taille des nombres de Fermat encore indéterminées, il faudra des décennies avant que la technologie permette d'exécuter plus de tests de Pépin. En 2016, le plus petit nombre de Fermat non testé sans facteur premier connu est F_{33} qui est composé de 2,585,827,973 chiffres.

Année	Chercheurs	Nombre de Fermat	Résultat Test	Facteur trouvé plus tard
1905	Morehead et Western	F_7	composé	Oui(1970)
1909	Morehead et Western	F_8	composé	Oui(1980)
1952	Robinson	F_{10}	composé	Oui(1953)
1960	Paxson	F_{13}	composé	Oui(1974)
1961	Selfridge et Hurwitz	F_{14}	composé	Oui(2010)
1987	Buell et Young	F_{20}	composé	Non
1993	Crandall, Doenias, Norrie et Young	F_{22}	composé	Oui(2010)
1999	Mayer, Papadopoulos et Crandall	F_{24}	composé	Non

3.6 Test de Lucas-Lehmer

Le test de *Lucas-Lehmer* est un test de primalité pour les nombres de Mersenne. Le test fut originellement développé par *Édouard Lucas* en 1878 et amélioré de façon notable par *Derrick Henry Lehmer* dans les années 1930, grâce à son étude des suites de Lucas.

les nombres de Mersenne sont les nombres de la forme : une puissance de 2 moins 1. Ils constituent la suite d'entiers :

$$M_n = 2^n - 1, \quad n \geq 1$$

Ces nombres doivent leur nom à un religieux érudit et mathématicien français du $XVII^{me}$ siècle,Marin Mersenne. Un nombre de Mersenne premier est un nombre qui est à la fois de Mersenne et premier. Pour que le n^ime nombre de Mersenne M_n soit premier, il est nécessaire,mais non suffisant, que son indice n le soit. Par exemple, M_4 n'est pas premier puisque 4 ne l'est pas $(2^4 - 1 = 15 = 3 * 5)$, et M_{11} n'est pas premier non plus bien que 11 le soit : $M_{11} = 2^{11} - 1 = 2047 = 23 * 89$.

3.6.1 Enoncé

L'algorithme de Lucas-Lehmer est basé sur le Théorème suivant :

> **Théorème 6** (Théorème de Lucas-Lehmer pour les nombres de Mersenne premiers)**.** *On considère la séquence V_k pour $k = 0, 1, \ldots$, définie récursivement par $V_0 = 4$ et $V_{k+1} = V_k^2 - 2$. Prenons p impair premier.Alors $M_P = 2^P - 1$ est premier si et seulement si $V_P - 2 \equiv 0 \pmod{M_p}$.*

3.6.2 Algorithme

Algorithme 9 : Test De Lucas-Lehmer

Données : un entier n qui représente l'indice du nombre de Mersenne à tester
$v = 4$;
pour $k = n - 2$ *jusqu'à* 1 **faire**
  $v = v^2 - 2 \pmod{2^n - 1}$;
si $v \equiv 0$ **alors**
  retourner premier;
retourner composé;

3.6.3 Compléxité

On utilise k fois l'exponentiation rapide pour calculer $v^2 - 2$ et pour calculer le nombres de Mersenne $2^n - 1$ donc on a une compléxité en O($(\log n)^2$).

3.6.4 Preuve

Pour la preuve,on se contente de vous renvoyer dans le livre *Prime Numbers : A Computational Perspective*[1] à la page 183 et à la page 184.

3.7 Test de Lucas-Frobenius

Le test de *Lucas-Frobenius* est un test de primalité probabiliste qui permet de tester la primalité d'un entier à l'aide de la suite de Lucas.

3.7.1 Algorithme

Tout d'abord,nous avons besoin de calculer la suite de Lucas.Pour cela il existe plusieurs manières.La première consiste simplement à calculer $V_n = V_{n-1} + V_{n-2}$ avec comme valeur d'initialisation $V_0 = 2$ et $V_1 = 1$ cependant cette manière de faire n'est pas optimale car beaucoup trop lente. Une autre méthode consiste à calculer la suite de Lucas à l'aide du nombre d'Or ce qui permet d'augmenter la rapidité de calcul mais cela devient trop lent pour de grands nombres. La manière la plus rapide consiste à calculer la suite de Lucas à l'aide de l'algorithme des Chaînes De Lucas.

Cet algorithme va prendre en paramètre un nombre premier et va le transformer en binaire.Ensuite il va parcourir la représentation binaire de cet entier en commençant par le bit de poids fort(celui situé

le plus à gauche).Selon si le bit est 1 ou 0, l'algorithme va exécuter des calculs en particulier en se basant sur les règles de calculs suivantes :

$$V_{2j} = V_j^2 \text{ et } V_{2j+1} = V_j \circ V_{j+1}$$

La fonction rond sera définie dans l'algorithme du Test De Lucas-Frobenius. Nous avons donc utiliser cette méthode :

Algorithme 10 : Chaîne De Lucas

Données : Deux entiers x_0 et x_1 qui représente les valeurs d'initialisation de la suite de Lucas

$u = x_0$;
$v = x_1$;
pour $j >= 0$ *jusqu'à* B **faire**
  si $n_j == 1$ **alors**
    $u = u \circ v$;
    $v = v * v$;
  sinon
    $u = u * u$;
    $v = u \circ v$;
retourner (u, v);

Pour l'algorithme principal du Test De Lucas-Frobenius,les régles de calcul de la chaîne de Lucas sont les suivantes :

$$V_{2j} = V_j^2 - 2 \pmod{n} \text{ et } V_{2j+1} = V_j * V_{j+1} - A \pmod{n}$$

Algorithme 11 : Test De Lucas-Frobenius

Données : Des entiers $n > 1$,a,b et Δ avec n qui représente le nombre à tester,a et b qui sont générés aléatoirement et $\Delta = a^2 - 4b$

$A = a^2 b^{-1} - 2 \pmod{n}$;
$m = (n - (\Delta/n))/2$;
ChaîneDeLucas$(2,A)$;
si $AV_m \equiv 2V_{m+1} \pmod{n}$ **alors**
  retourner n is a Lucas probable prime with parameters a, b ;
retourner n is composite;
OU
si $AV_m \neq 2V_{m+1} \pmod{n}$ **alors**
  retourner n is composite ;
$B = bn - 1/2 \pmod{n}$;
si $BV_m \equiv 2V_m \pmod{n}$ **alors**
  retourner n is a Frobenius probable prime with parameters a, b ;
retourner n is composite ;

3.7.2 Compléxité

La compléxité de l'algorithme des Chaînes De Lucas effectue des multiplications modulaires donc cet algorithme a une compléxité en O($(\log n)^2$).Comme l'algorithme du Test De Lucas-Frobenius utilise

celui des Chaînes De Lucas et que dans les conditions on effectue des tests avec des multiplications modulaires,on a donc une compléxité de O($(\log n)^2$) pour cet algorithme.

3.7.3 Preuve

Pour la preuve,on se contente de vous renvoyer dans le livre *Prime Numbers : A Computational Perspective*[1] de la page 144 à la page 149.

4 Mesures de performance et comparatifs

Dans la partie précédente, on s'est contenté de présenter différents tests de primalités sans conclure par rapport à leurs performances. Dans cette partie, on va mesurer ces performances et établir un comparatif entre les différents tests abordés.

4.1 Évolution des tests de primalités

Plusieurs algorithmes de tests de primalités se sont succédés au fil du temps. Des algorithmes de plus en plus efficaces apparaissent pour en remplacer d'autres.

4.1.1 Premiers tests déterministes

Les plus anciens algorithmes de tests de primalités sont les algorithmes de ***test naïf***, dont le ***Crible d'Erathostène*** qui date d'avant J-C. Ces algorithmes sont déterministes, c'est-à-dire qu'ils retournent toujours une réponse exacte. Ils constituent la façon la plus naturelle de tester la primalité d'un nombre mais leur complexité est trop élevée, surtout quand il s'agit de tester des grands nombres.

D'autres algorithmes déterministes basés sur des théorèmes plus récents sont aussi apparus, on peut citer le ***Test De Pepin***,qui teste la primalité sur les nombres de Fermat, ou encore le ***Test De Lucas-Lehmer*** qui permet de tester la primalité des nombres de Mersenne.

Par la suite, les avancées sur les algorithmes de tests probabilistes ont pris plus d'importance.

4.1.2 Tests probabilistes

Les algorithmes de tests probabilistes constituent une façon beaucoup plus efficace que les premiers algorithmes déterministes découverts pour tester la primalités d'un nombre. Ces algorithmes de type *Monte-Carlo* décident si un entier est premier ou pas avec une certaine probabilité P. La sortie d'un test de primalités probabiliste sur un entier n est soit :

— n est composé : toujours vrai

— n est premier : vrai avec une probabilité p

Dans le cas où la sortie du test est "premier", il y a toujours une petite probabilité que le nombre testé ne soit pas vraiment premier. Pour pallier à ce problème et diminuer cette probabilité, on a tendance à répéter le test probabiliste plusieurs fois.

Le premier algorithme probabiliste apparu est le ***test de Fermat***. Ce test qui repose sur un théorème énoncé en 1640 a longuement était utilisé en pratique, jusqu'à l'apparition du ***test de Solovay-Strassen*** en 1977.

À partir de 1980, le ***test de Solovay-Strassen*** a été lui aussi remplacé en pratique par le ***test de Miller-Rabin***, plus efficace, car reposant sur un critère analogue, mais ne donnant de faux positif qu'au plus une fois sur quatre lorsque le nombre testé n'est pas premier.

Un Test probabiliste qui possède une faible probabilité de ce trompé est celui de ***Lucas-Frobenius***.Ce test est donc meilleure que ***Miller-Rabin*** et aussi rapide.

4.2 Mesure du temps d'exécution

Pour comparer les performances des algorithmes de Test de primalités implémentés, on va mesurer le temps d'exécution de chaque algorithme, c'est-à-dire le temps de réponse, selon le nombre de bits de l'entier testé. Cette méthode va donc permettre de connaitre le test le plus performant pour un nombre de bits donné.

4.2.1 Algorithme de mesure

L'algorithme qui correspond à cette méthode de mesure du temps d'exécution est le suivant :

Algorithme 12 : Mesure temps exécution

Données : Un entier représentant le nombre de bits MAX et un entier représentant le nombre d'itérations et un nom de fichier dans lequel sera stocké les mesures

Sorties : Un Fichier .txt dans lequel il y aura les mesures de chaque Test

pour *pour un nombre de bits allant de 1 à 1024 ($j = 1$ jusqu'à 1024)* **faire**

- Générer un nombre premier p de j bits;
- **pour** *chaque Test De Primalité* **faire**
  - Stocker le temps que met chaque Test De Primalité pour vérifier p;

4.2.2 Analyse des mesures

Une fois que les mesures ont été effectuées, on va visualiser les résultats grâce à des représentations graphiques afin de les comparer et de conclure par rapport aux performances.

Le logiciel `Gnuplot` va être utilisé pour faire ces représentations graphiques. Pour cela, nous allons écrire les résultats dans des fichiers bien formatés dont voici un exemple :

```
1  Fermat    Miller  Strassen LucasFrob PFermat PMiller PStrassen PFrobinius
     Erastotene
2 1 0.000001 0.000001 0.000000  0.000001 0.000000 0.000000 0.000000 0.000000
3 2 0.001379 0.001035 0.001037  0.000001 0.000172 0.000016 0.000130 0.000000
     0.000002
```

```
3 0.001405 0.001063 0.001031  0.001134 0.000176 0.000017 0.000129 0.000000 0.000002
4 0.001516 0.001035 0.001055  0.001136 0.000190 0.000016 0.000132 0.000000 0.000003
5 0.001376 0.001066 0.001089  0.001142 0.000172 0.000017 0.000136 0.000000 0.000004
6 0.001394 0.001056 0.001036  0.001138 0.000174 0.000017 0.000129 0.000000 0.000003
7 0.001382 0.001091 0.001057  0.001237 0.000173 0.000017 0.000132 0.000000 0.000004
8 0.001375 0.001032 0.001038  0.001156 0.000172 0.000016 0.000130 0.000000 0.000005
9 0.001421 0.001068 0.001068  0.001192 0.000178 0.000017 0.000133 0.000000 0.000008
10 0.001818 0.001493 0.001223  0.001165 0.000227 0.000023 0.000153 0.000000 0.000023
11 0.001403 0.001063 0.001035  0.001228 0.000175 0.000017 0.000129 0.000000 0.000017
12 0.001427 0.001032 0.001033  0.001163 0.000178 0.000016 0.000129 0.000000 0.000013
13 0.001410 0.001126 0.001241  0.001225 0.000176 0.000018 0.000155 0.000000 0.000031
14 0.001377 0.001074 0.001041  0.001205 0.000172 0.000017 0.000130 0.000000 0.000045
15 0.001375 0.001029 0.001080  0.001162 0.000172 0.000016 0.000135 0.000000 0.000056
16 0.001422 0.001052 0.001027  0.001166 0.000178 0.000016 0.000128 0.000000 0.000069
17 0.001380 0.001034 0.001106  0.001162 0.000173 0.000016 0.000138 0.000000 0.000097
18 0.001409 0.001037 0.001042  0.001472 0.000176 0.000016 0.000130 0.000000 0.000131
19 0.001382 0.001039 0.001043  0.001185 0.000173 0.000016 0.000130 0.000000 0.000198
20 0.001381 0.001040 0.001043  0.001213 0.000173 0.000016 0.000130 0.000000 0.000276
21 0.001417 0.001098 0.001109  0.001210 0.000177 0.000017 0.000139 0.000000 0.000403
22 0.001385 0.001036 0.001070  0.001175 0.000173 0.000016 0.000134 0.000000 0.000604
23 0.001385 0.001043 0.001344  0.001212 0.000173 0.000016 0.000168 0.000000 0.000900
24 0.001414 0.001034 0.001047  0.001174 0.000177 0.000016 0.000131 0.000000 0.001365
25 0.001398 0.001051 0.001049  0.001178 0.000175 0.000016 0.000131 0.000000 0.002120
26 0.001384 0.001034 0.001048  0.001481 0.000173 0.000016 0.000131 0.000000 0.003232
27 0.001386 0.001039 0.001036  0.001182 0.000173 0.000016 0.000129 0.000000 0.005112
```

```
28 0.001698 0.001048 0.001050  0.001177 0.000212 0.000016 0.000131 0.000000
   0.007776
29 0.001380 0.001062 0.001053  0.001216 0.000173 0.000017 0.000132 0.000000
   0.012790
30 0.001393 0.001048 0.001070  0.001194 0.000174 0.000016 0.000134 0.000000
   0.022610
31 0.001383 0.001067 0.001047  0.001174 0.000173 0.000017 0.000131 0.000000
   0.033021
32 0.001409 0.001040 0.001048  0.001162 0.000176 0.000016 0.000131 0.000000
   0.054457
33 0.001389 0.001052 0.001054  0.001190 0.000174 0.000016 0.000132 0.000000
   0.086877
34 0.001393 0.001042 0.001044  0.001189 0.000174 0.000016 0.000131 0.000000
   0.141938
35 0.001413 0.001039 0.001060  0.001185 0.000177 0.000016 0.000133 0.000000
36 0.002249 0.001123 0.001162  0.001192 0.000281 0.000018 0.000145 0.000000
37 0.001532 0.001132 0.001088  0.001199 0.000192 0.000018 0.000136 0.000000
38 0.001415 0.001084 0.001085  0.001224 0.000177 0.000017 0.000136 0.000000
39 0.001382 0.001035 0.001081  0.001480 0.000173 0.000016 0.000135 0.000000
40 0.001548 0.001099 0.001045  0.001235 0.000194 0.000017 0.000131 0.000000
41 0.001399 0.001083 0.001043  0.001596 0.000175 0.000017 0.000130 0.000000
42 0.001500 0.001056 0.001068  0.001180 0.000188 0.000017 0.000133 0.000000
43 0.001385 0.001048 0.001090  0.001358 0.000173 0.000016 0.000136 0.000000
44 0.001902 0.001041 0.001082  0.001560 0.000238 0.000016 0.000135 0.000000
45 0.001391 0.001042 0.001048  0.001256 0.000174 0.000016 0.000131 0.000000
46 0.001396 0.001037 0.001047  0.001268 0.000175 0.000016 0.000131 0.000000
47 0.001411 0.001086 0.001424  0.001405 0.000176 0.000017 0.000178 0.000000
48 0.001411 0.001067 0.001053  0.001314 0.000176 0.000017 0.000132 0.000000
49 0.001424 0.001047 0.001079  0.001304 0.000178 0.000016 0.000135 0.000000
```

Listing 1 – mesure.txt (Tests De Primalités)

Remarque : Nous nous excusons pour la partie d'Erastohène dans le Fichier mais ce n'était pas possible d'ajuster sur le Rapport.Cependant dans le fichier .txt cela s'affiche bien à droite de Frobénius.

Nous disposons donc de 3 fichiers différents :

— mesure.txt

— deterministe.txt

— Pepin.txt

Chacun de ces fichiers contiennent le temps que les algorithmes ont mis pour vérifier la primalité du nombre de n bits passés en paramètre. En exploitant les résultats de ce fichier, le logiciel `Gnuplot` va permettre de construire les courbes représentatives du temps d'exécution de chaque test de primalités en fonction de la taille en bits du nombre premier générer préalablement (de 0 à 1024 bits) :

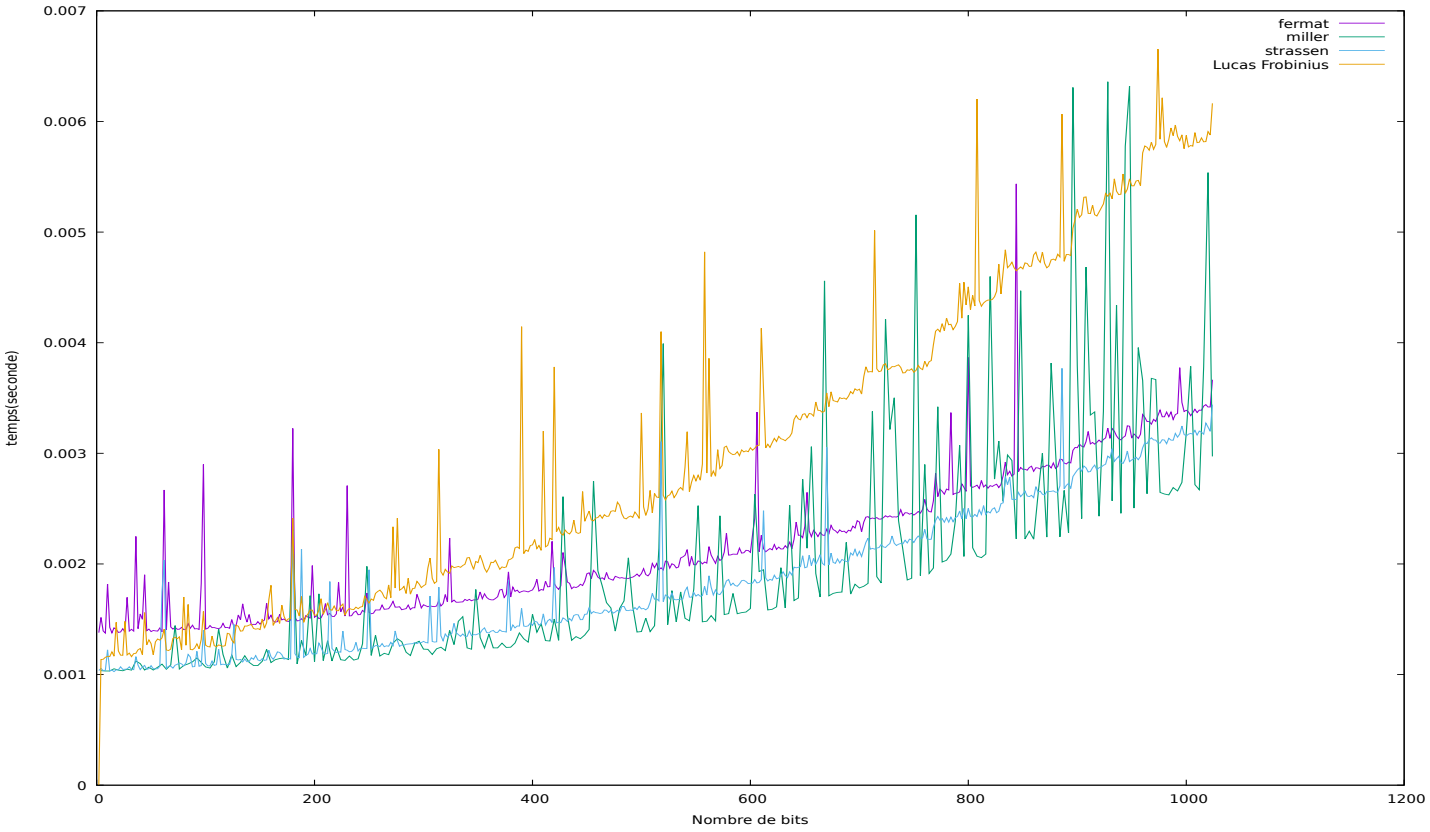

FIGURE 2 – Temps d'exécution des Tests Probabilistes en fonction de la taille en bits du nombre premier avec 3 itérations

Ce graphique, qui est la superposition des courbes du temps d'exécution des 4 Tests Probabilistes implémentés, va permettre d'observer les performances des différents algorithmes de test.

Ceux-ci ont un temps d'exécution rapide en moyenne.Le ***Test de Fermat***,***Test de Miller-Rabin***, ***Test de Solovay-Strassen*** et le ***Test de Lucas-Frobénius*** ont un temps d'exécution assez similaire avec peu de variations brusques. On remarquera surtout que le ***Test de Miller-Rabin*** inscrit un temps d'exécution faible en moyenne.

Pour le graphe suivant,On s'est demandé s'il était plus rentable de faire un algorithme en particulier plutôt qu'un autre concernant les Tests Probabilistes. Voici la courbe représentative :

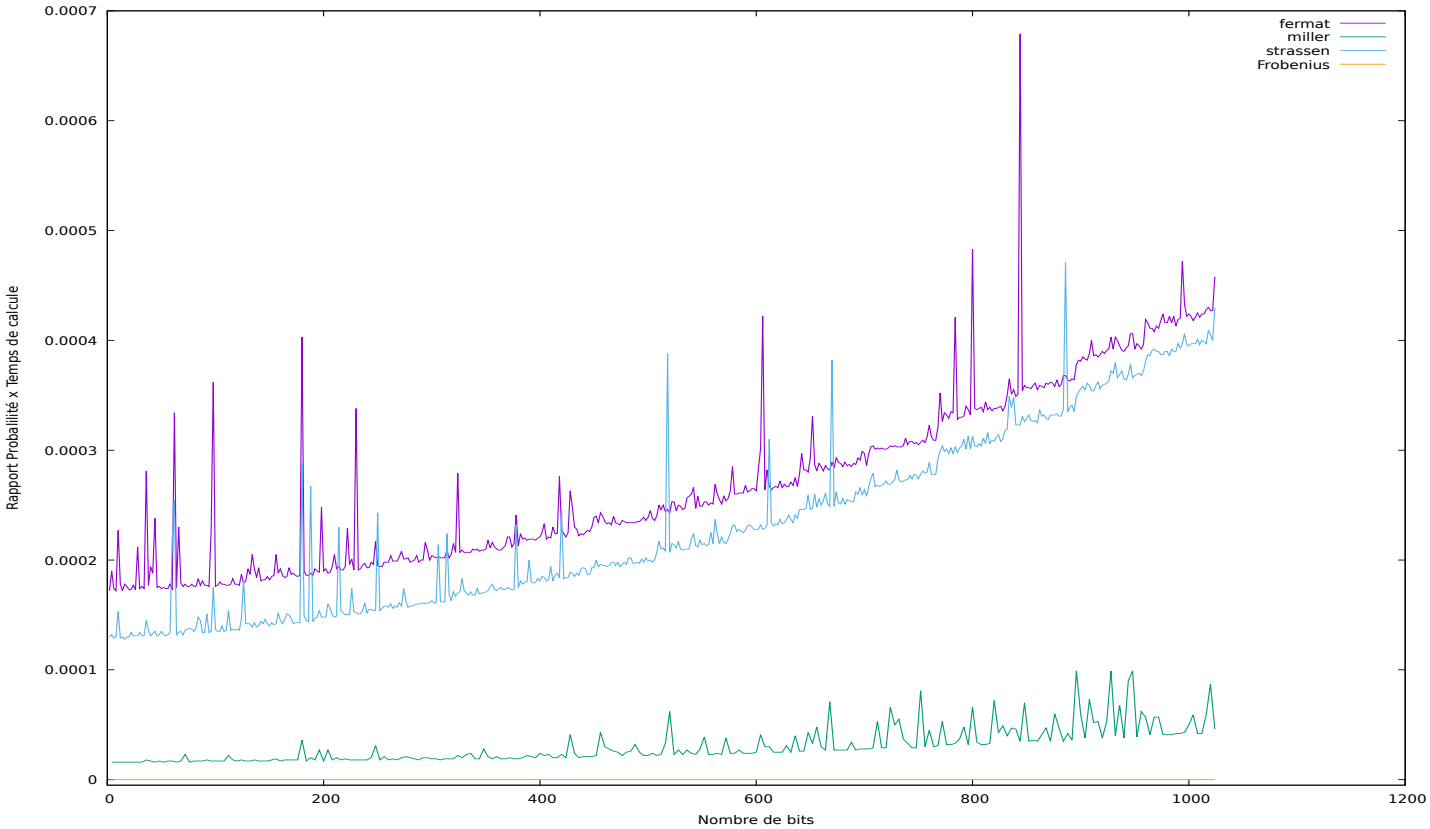

FIGURE 3 – Rentabilité en fonction de la probabilité d'échec des Tests Probabilistes avec 3 itérations

Ce graphe permet de voir la rentabilité vis à vis de la probabilité d'échec.Pour le Test ***de Lucas-Frobenius***,nous observons qu'il a une probabilité d'échec quasi nulle et qu'il est donc le plus performant.Pour le Test ***de Miller-Rabin***,la probabilité s'approche de 0 mais n'est pas nul.Et on observe que les Tests ***de Solovay-Strassen*** ainsi que celui de ***Fermat*** sont les moins rentables.

On a aussi observer qu'au bout de 20 itérations,tous ces Tests ont une probabilité quasiment nulle de faire une erreur.Si nous devions choisir un algorithme,nous ferions donc le choix du Test de ***de Lucas-Frobenius***.

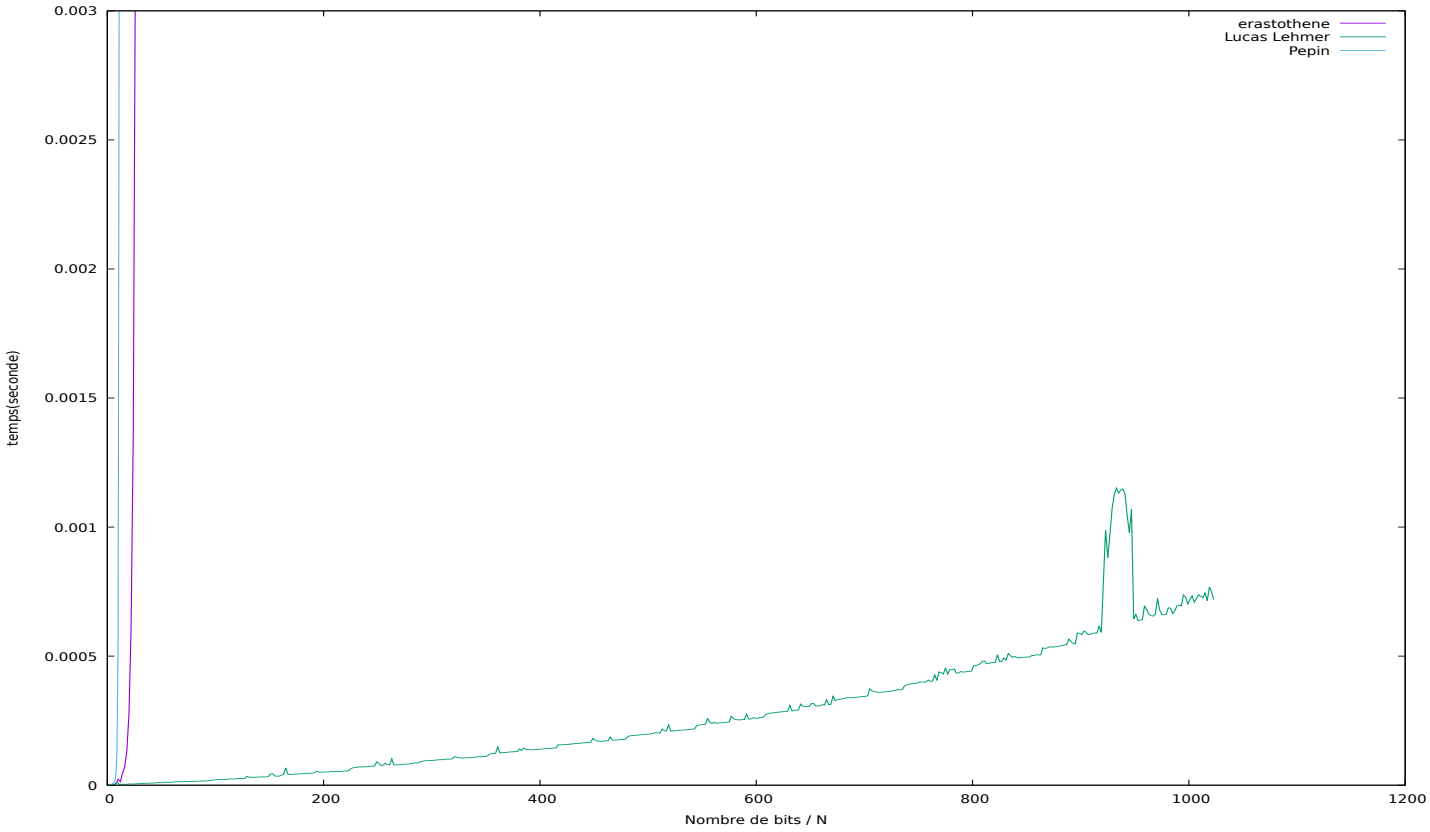

FIGURE 4 – Temps d'exécution des Tests Déterministes

Concernant les Tests Déterministes,le Test ***d'Eratosthène*** et de ***Pépin*** ont un temps de réponse rapide pour un nombre de bits très petit, mais ce temps de réponse va croître d'une manière exponentielle à partir d'un certain nombre de bits, ce qui en fait des tests très lents globalement. Cependant celui de ***Lucas-Lehmer*** est beaucoup plus rapide qu'eux.

En ce qui concerne le Test de ***Lucas-Lehmer***,celui-ci est très rapide car son algorithme repose sur le calcul d'une séquence obtenue en fonction du nombre n en entrée.

On va maintenant s'intéresser plus en détails au Test ***d'Eratosthène***. Voici la courbe représentative de ce Test

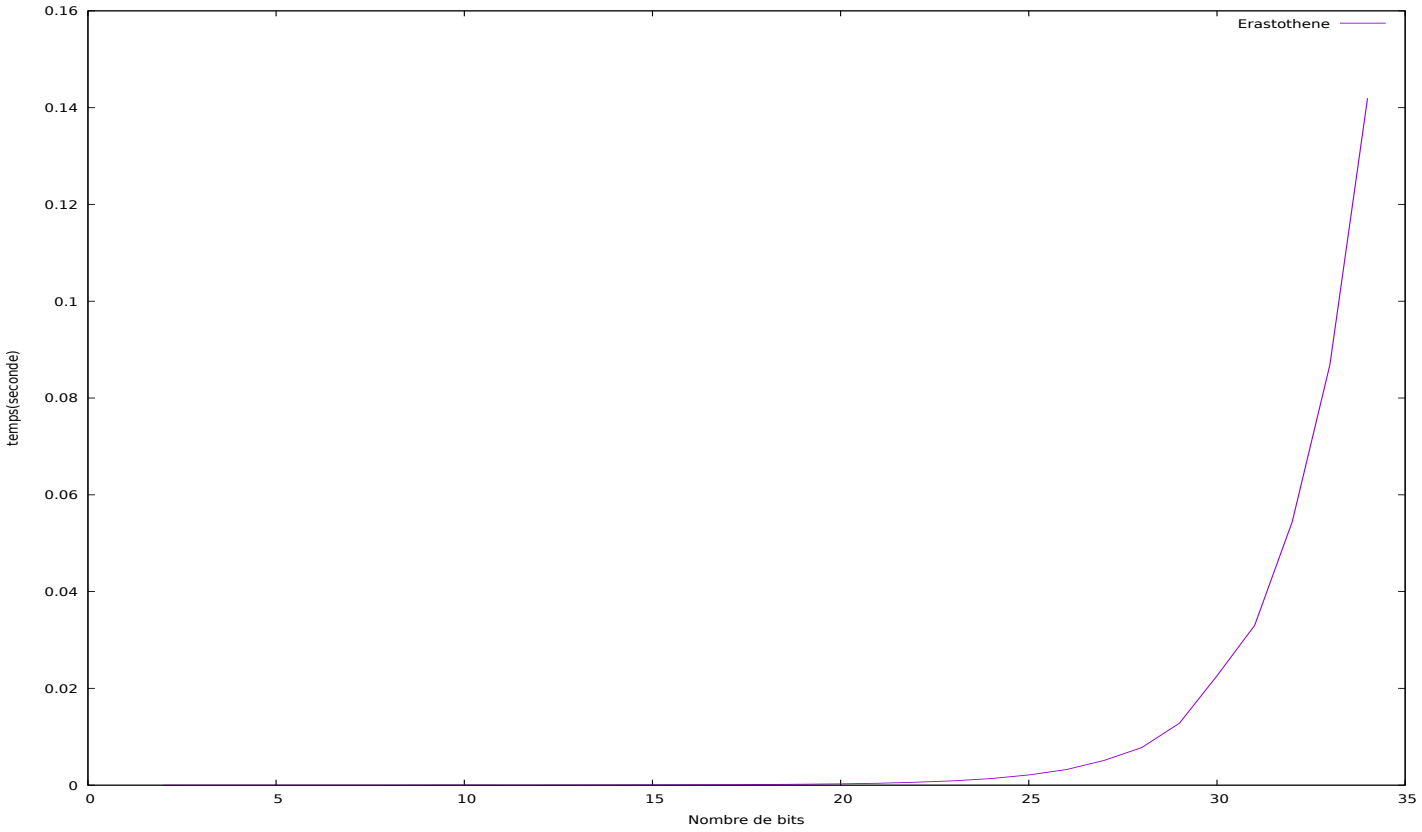

FIGURE 5 – Temps d'exécution du Test d'Erastothene

On voit bien qu'à partir de 28bits le temps d'exécution du Test commence à augmenter d'une manière exponentielle.On a donc décidé de stopper ce Test au bout de 34 bits car sinon cela devient impossible à calculer avec nos machines.

On va maintenant s'intéresser plus en détails au Test de ***Lucas-Lehmer***. Voici la courbe représentative de ce Test :

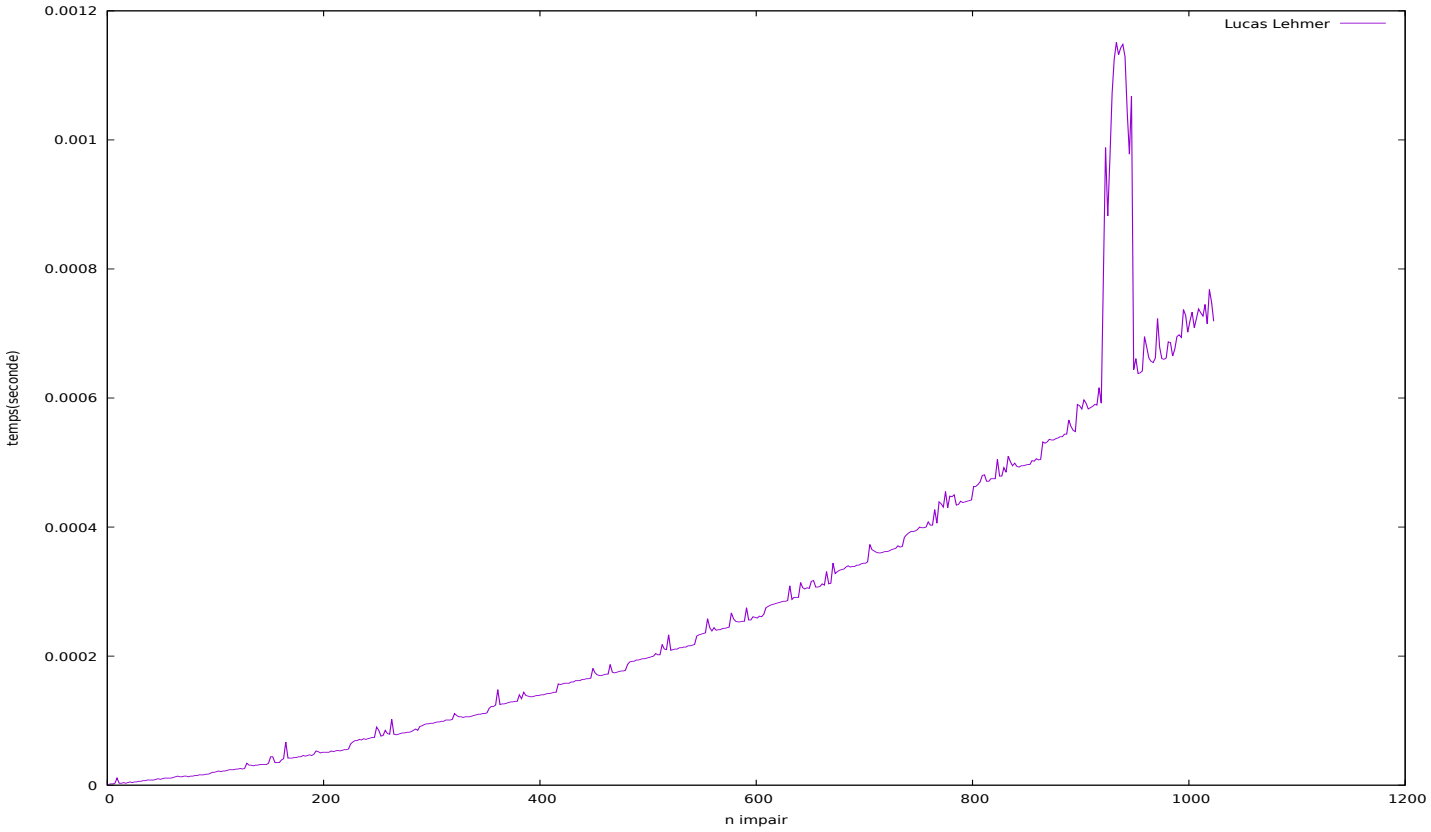

FIGURE 6 – Temps d'exécution du Test de Lucas-Lehmer

On voit bien que ce Test est rapide comparé aux autres Tests déterministes.Cela peut s'expliquer par le fait que ce Test utilise le calcul d'une séquence.

On va maintenant s'intéresser plus en détails au Test de ***Pepin***. Voici la courbe représentative de ce Test :

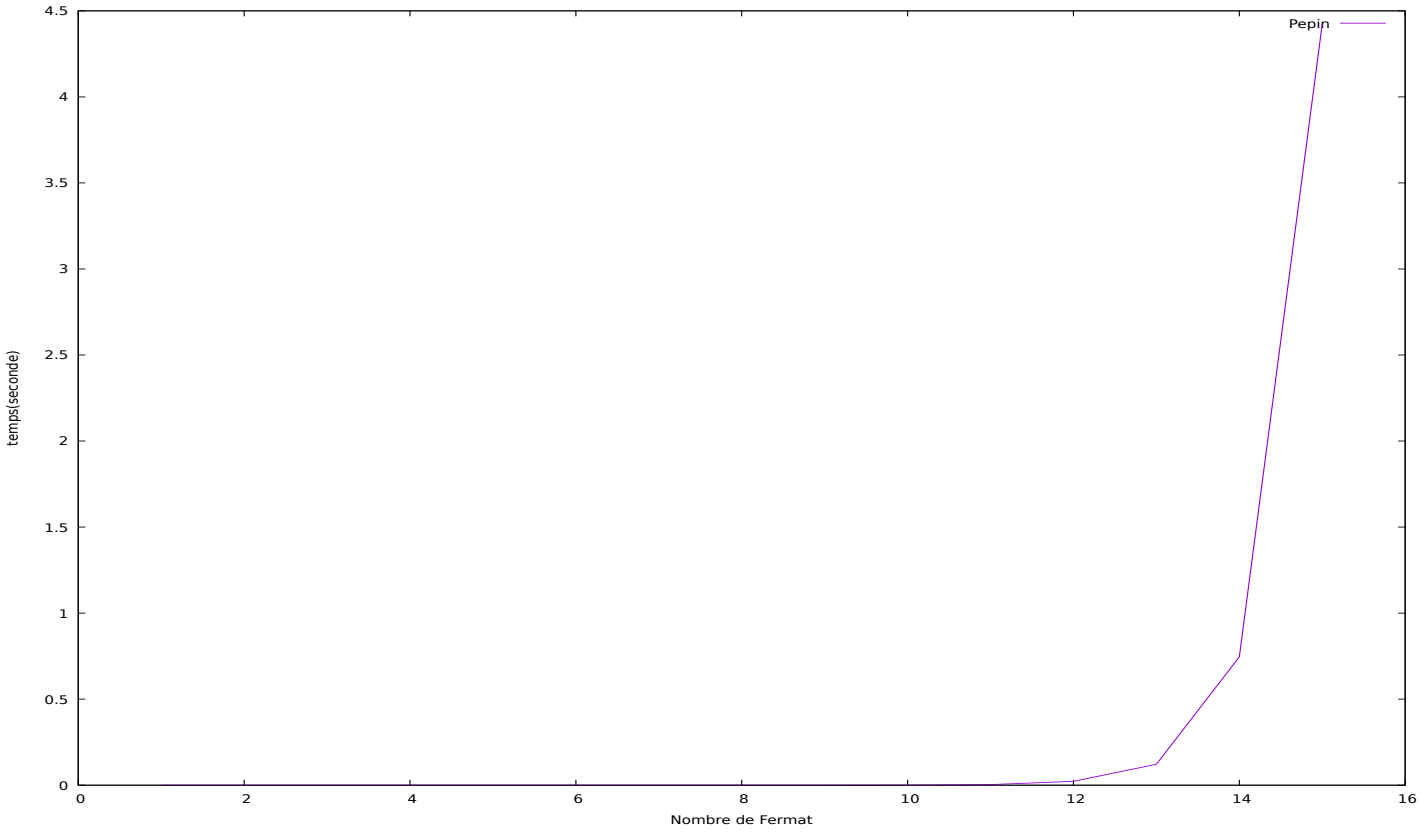

FIGURE 7 – Temps d'exécution du Test de Pepin

On voit bien qu'à partir de l'indice 14,le temps augmente énormément et donc pour cela nous avons décidé de stopper ce Test à partir de l'indice 15.Nous avons réussi à obtenir un résultat jusqu'à F_{18} mais nous ne l'avons pas affiché ici car cela avait pris environ 1H pour l'obtenir.

5 Conclusion

Pour conclure,notre travail a abouti à une implémentation de divers tests de primalités, une description de leur principe et de leur preuve, ainsi qu'une comparaison des performances de ces tests.

Nous avons rencontré des difficultés lors de l'établissement des complexités et des preuves des algorithmes implémentés. En ce qui concerne l'implémentation, le test de *Lucas-Frobenius* a été celui qui nous a demandé le plus de réflexion.

Certains points restent encore ouverts. On pourrait envisager d'appliquer d'autres méthodes de mesure de performance. Il faudrait aussi effectuer des mesures sur les tests probabilistes pour déterminer le nombre optimal de répétitions à effectuer. Cela reviendrait à voir plus en détail les probabilités d'erreur de ces tests.

Ce projet a eu la particularité de faire appel à des compétences en informatique et en mathématique de manière égale. Cette épreuve constitue donc une expérience enrichissante pour nous, sur le plan des

compétences, mais aussi collectivement.

Références

[1] Crandall Carl, Pomerance Richard. *Prime Numbers : A Computational Perspective*. Springer, 2001.