Rapport

Sonny Klotz - Idir Hamad - Younes Benyamna - Malek Zemni

Projet M1 Informatique
Primalité

20/05/2018

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
université PARIS-SACLAY

Module *TER*

Table des matières

Table des figures

Liste des Algorithmes

Liste des théorèmes et des définitions

Introduction

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

Les tests de primalité 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 primalité, plus ou moins performants. L'efficacité de ces algorithmes est particulièrement liée à la taille des données entrées.

Notre travail consiste donc à implémenter différents tests de primalité et de comparer leurs performances. Ces mesures de performance nous permettront de construire un algorithme qui génère un nombre premier de manière optimale. Cet algorithme sera le produit final de l'application développée dans ce projet.

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

Dans la deuxième partie, on évoquera les champs d'application principaux des tests de primalité, en l'occurrence, les cryptosystèmes à clé publique.

Dans la troisième partie, on détaillera le processus utilisé en pratique pour générer un nombre premier.

Ensuite, la quatrième partie traitera des tests de primalité qu'on implémentera : leur algorithme, leur complexité et leur preuve.

Finalement, on établira dans la dernière partie un comparatif entre ces algorithmes de tests de primalité ainsi qu'un processus de mesure de performance qui aboutira à une génération optimale d'un nombre premier.

1 Architecture de l'application

1.1 Organigramme et données échangées

L'application développée dans ce projet porte essentiellement sur l'implémentation des tests de primalité, 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.

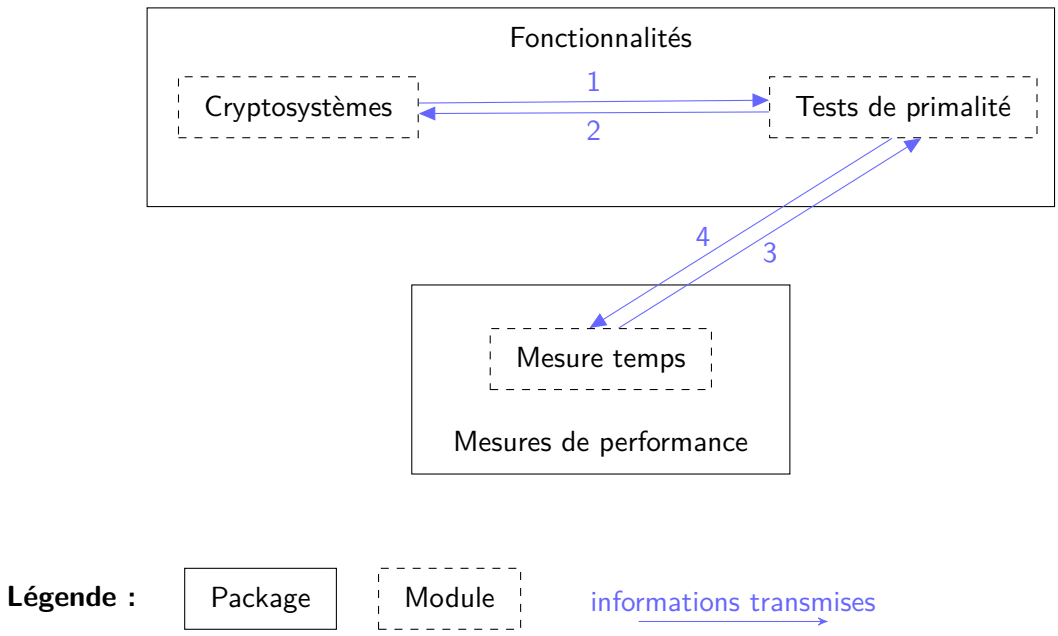

FIGURE 1 – Organigramme des différents modules de l'application

Notes :
(1) Nombre entier à tester (primalité)
(2) Réponse sur la primalité (0 composé, 1 probablement premier, 2 premier)
(3) Nombre entier à tester (primalité)
(4) Réponse sur la primalité (0 composé, 1 probablement premier, 2 premier)

1.2 Fonctionnalités des modules

Package Fonctionnalités

1. Module Cryptosystèmes : implémentation de cryptosystèmes ayant recours à des nombres premiers (***RSA***) et des générateurs aléatoires de nombres premiers (RPNG) :
   - Cryptosystème RSA
   - RPNG avec test de primalité déterministe
   - RPNG avec test de primalité probabiliste
   - RPNG optimal
2. Module Tests de primalité : implémentation de différents algorithmes de test de primalité qui feront l'objet d'une étude comparative par la suite :
   - Test Naïf
   - Test de Wilson
   - Test de Fermat
   - Test de Miller-Rabin
   - Test de Solovay-Strassen
   - Test AKS

Package Mesures de performance

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

1.3 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'un 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.

`GNU MP` pour `GNU Multi Precision`, souvent appelée `GMP` 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 utilisé pour faire des représentations graphiques à partir des résultats issus des mesures de performance de notre application.

2 Cryptosystèmes - RSA

Les tests de primalité sont des algorithmes indispensables pour la cryptographie à clé publique. Ces tests sont couramment utilisés par les cryptosystèmes ***RSA*** et ***ElGamal*** afin de générer de grands nombres premiers.

Pour *RSA*, les tests sont effectués lors la phase de génération de clés. Pour *ElGamal*, ils sont effectués lors de l'établissement d'un échange de clés.

Dans cette partie, on va détailler le cryptosystème *RSA* et exhiber rôle important des nombres premiers. On a choisi de s'intéresser qu'à un seul cryptosystème (RSA) puisque le choix du cryptosystème n'aura pas d'effet sur les performances des tests de primalité, objet principal de ce projet.

2.1 Description de RSA

Décrit en 1977 par Ronald Rivest, Adi Shamir et Leonard Adleman, RSA est un cryptosystème basé sur le problème de factorisation, qui utilise une paire de clés (publique, privée) permettant de chiffrer et de déchiffrer un message. Le fonctionnement de RSA peut être décrit en 3 phases :

1. **Génération des clés**
   - Choisir 2 grands ***nombres premiers*** distincts p et q.
   - Calculer $n = p * q$. n est le module RSA et fait 1024 bits au minimum en général.
   - Calculer $\Phi(n) = (p-1)(q-1)$.
   - Choisir $e \in \mathbb{Z}^*_{\Phi(n)}$ (e premier avec $\Phi(n)$).

— Calculer d telle que $d * e \equiv 1 mod \Phi(n)$ (d inverse de e pour la multiplication modulo $\Phi(n)$).

Les éléments échangés constituant la clé publique sont (n, e). Les éléments constituant la clé privé sont (p, q, d).

2. **Chiffrement**
   Pour chiffrer un message M en un chiffré C, on utilise les éléments de la clé publique (n, e) :

$$C \equiv M^e \pmod{n}$$

3. **Déchiffrement**
   Pour déchiffrer un chiffré C en un message clair M, on utilise les éléments de la clé privée (p, q, d) :

$$M \equiv C^d \pmod{n}$$

2.2 Rôle des nombres premiers

La première étape pour la mise en place d'un cryptosystème RSA est la génération de deux très grands nombres premiers p et q. Leur produit $n = p * q$ forme le module RSA. Pour cette raison, la taille de p et q en bits, doit être égale à la moitie de la taille en bits du module n. Par exemple, dans le cadre de RSA-1024, les deux nombres premiers doivent avoir une longueur de 512 bits.

En effet, un attaquant qui connait le module RSA n et la clé publique e doit connaitre la factorisation de n en nombres premiers pour trouver la clé privée d. Ainsi, l'entier n doit être très grand afin que sa factorisation ne soit pas possible avec les ressources de calcul actuelles. On voit donc l'intérêt crucial pour la sécurité de générer les deux grands nombres premiers p et q.

Parmi les algorithmes classiques de factorisation les plus efficaces, on retrouve ***GNFS*** (General Number Field Sieve) dont le temps d'exécution croît exponentiellement à la taille de n (complexité exponentielle). Avec les puissances de calcul actuelles, il est de plus en plus déconseillé d'utiliser un module RSA de taille 1024 bits. Il est estimé qu'un module de taille 2048 bits soit sécurisé (complexité factorisation supérieure à 2^{80}) jusqu'à l'année 2020. En 1994, l'*algorithme de Shor* appliqué sur des ordinateurs quantiques a permis d'effectuer un factorisation en un temps non exponentiel. Les applications des ordinateurs quantiques permettent théoriquement de casser RSA par la force brute, mais actuellement ces ordinateurs génèrent des erreurs aléatoires qui les rendent inefficaces.

3 Génération des nombres premiers

Les cryptosystèmes utilisent une approche commune pour la génération des nombres premiers. Cette approche générale consiste à utiliser un générateur de nombres aléatoires pour générer un entier, dont on testera ensuite la primalité. Ce processus est illustré par la figure ci-dessous :

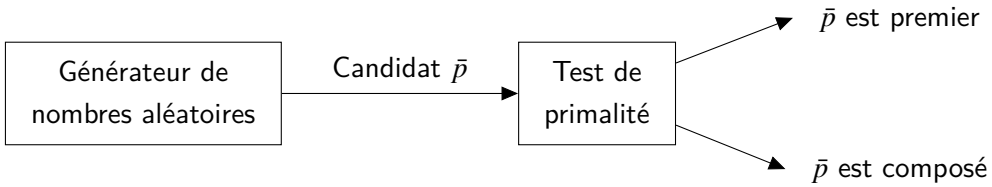

FIGURE 2 – Processus de génération des nombres premiers

Dans cette démarche, il est important d'utiliser un bon générateur de nombres aléatoires, qui ne doit dans aucun cas être prévisible. Si un attaquant réussit à deviner les nombres premiers qui composent le module RSA, alors le système est immédiatement cassé.

Fréquence des nombres premiers

Lors de la génération des nombres premiers à l'aide de ce processus, on voudrait savoir combien de nombres doit-on tester avant de trouver un nombre premier. La réponse à cette question est donnée par le *Théorème des Nombres Premiers*.

> **Théorème 1** (Théorème des Nombres Premiers)**.** *Soit $\pi(n)$ le nombre de premiers qui sont inférieurs à n, alors*
>
> $$\pi(n) \approx \frac{n}{ln(n)} \qquad (n \to +\infty)$$

Un graphique de la fonction $\pi(n)$ pour les 1000 premiers nombres premiers est donné dans la figure ci-dessous :

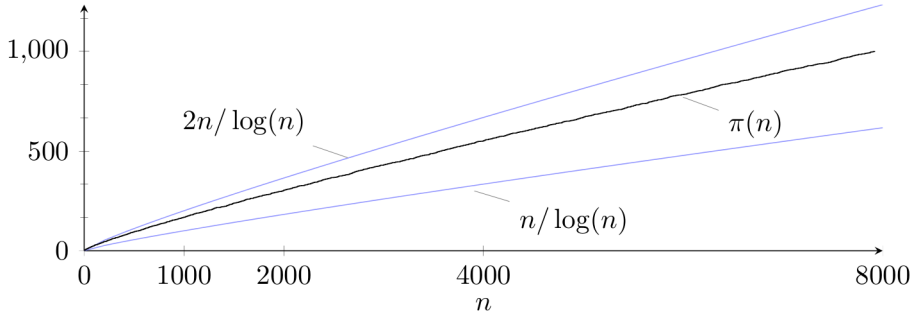

FIGURE 3 – La fonction $\pi(n)$ pour les 1000 premiers nombres premiers

Le tableau suivant contient l'approximation ainsi que le nombre exact de nombres premiers pour différentes valeurs de n. On y remarque que l'approximation est assez bonne.

n	$n/ln(n)$	$\pi(n)$
10^3	145	168
10^4	1086	1229
10^5	8686	9592
10^6	72382	78498
10^7	620420	664579

Probabilité de générer un nombre premier p de k bits : on sait que $2^{k-1} \leqslant p \leqslant 2^k - 1$. Le nombre de nombres premiers dans cet intervalle (c-à-d de k bits) peut être approximé par :

$$\pi(2^k) - \pi(2^{k-1}) \approx \frac{2^k}{ln(2^k)} - \frac{2^{k-1}}{ln(2^{k-1})} \approx \frac{2^{k-1}}{ln(2^{k-1})}$$

puisque $ln(2^k) = ln(2.2^{k-1}) = ln(2) + ln(2^{k-1})$, donc $ln(2^k) \approx ln(2^{k-1})$ pour k grand.

Donc, il y'a 2^{k-1} entiers $\in [2^{k-1}, 2^k - 1]$ (de k bits) dont approximativement $\frac{2^{k-1}}{ln(2^{k-1})}$ parmi eux qui sont premiers. Par conséquent, un nombre p de k bits sera premier avec une probabilité de :

$$\frac{1}{ln(2^{k-1})}$$

Cas de RSA-1024 : pour générer une des deux clé de RSA-1024 dont la taille en bits est 512, on a probabilité de $1/ln(2^{511}) \approx 1/355$ pour générer un nombre premier de 512 bits. Cette chance double si on se restreint sur les entiers impairs, c'est à dire qu'on doit générer à peu près 177 nombres avant de tomber sur un nombre premier.

4 Tests de primalité

Les tests de primalité interviennent dans la deuxième étape du processus de génération des nombres premiers. Ce 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 primalité.
Les tests de primalité 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)

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

Algorithme	Année	Type
Naïf (Crible d'Eratosthène)	-240	Déterministe
Fermat	1640	Probabiliste
Wilson	1770	Déterministe
Miller-Rabin	1976	Probabiliste
Solovay-Strassen	1977	Probabiliste
AKS	2002	Déterministe

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* et *AKS*. 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.

4.1 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 ne 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;

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 $\sqrt{n}$, 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)}*** ;

plusGrandPremier $= N$;

pour *tout nombre p marqué "premier" de la liste L (de manière croissante)* **faire**

  si $p^2 >$ ***plusGrandPremier*** **alors**

    retourner L;

  i $= 2$;

  tant que $p * i < N$ **faire**

    Marquer "composé" l'entier à la position $p * i$;

    Mettre à jour ***plusGrandPremier*** ;

    $i{+}{+}$;

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. Dans le cas de RSA-1024, la complexité de cette méthode avoisine les 2^{503} divisions.

4.2 Test de Wilson

Le test de primalité de Wilson est un test déterministe basé sur le théorème suivant :

> **Théorème 2** (Théorème de Wilson)**.** *Un entier $n > 1$ est un nombre premier si et seulement si*
>
> $$(n-1)! + 1 \equiv 0 \pmod{n}$$

Ce théorème fournit une caractérisation des nombres premiers assez anecdotique et ne constitue pas un test de primalité efficace. Son principal intérêt réside dans son histoire et dans la relative simplicité de son énoncé et de ses preuves.

En effet, ce théorème était connu à partir du XVII^{e} siècle en Europe. En 1770, *John Wilson* redécouvre une conjecture de ce théorème et la publie. Ensuite, les mathématiciens *Lagrange*, *Euler* et *Gauss* le démontrent chacun à son tour.

Algorithme

L'algorithme du test de primalité basé sur le *théorème de Wilson* est le suivant :

Algorithme 3 : Test de Wilson

Données : un entier n

si $(n-1)!+1 \equiv 0 \pmod{n}$ **alors**

  retourner premier;

sinon

  retourner composé;

Complexité

Ce test qui est basé sur une propriété très simple a cependant une complexité trop élevée. Il faut effectuer environ n multiplications modulaires, par conséquent la complexité est de $O(n)$.

4.3 Test de Fermat

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

> **Théorème 3** (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 4** (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.

4.3.1 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 telle que $1 < a < n$:

- Le fait de choisir $1 < a < n$ garantit que si n était premier, a sera forcément premier avec n (puisque $a < n$) et ainsi le test n'échouera pas.
- Si $\mathbf{a^{n-1} \not\equiv 1 \pmod{n}}$, 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, telle que*

$$1 < a < n \quad et \quad a^{n-1} \not\equiv 1 \pmod{n}$$

— 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 3).
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 $a^{560} \equiv 1 \pmod{5)61}$ 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, va être 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 4 : 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;
    si a^{n-1} ≢ 1 (mod n) alors
        retourner composé;
retourner probablement premier;
```

4.3.2 Complexité

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

Algorithme 5 : 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 $n-1$ notre exposant 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.

4.3.3 Preuve

Pour démontrer cet algorithme 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 5** (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 donc prouvé *petit le théorème de Fermat*, on concluera 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.

4.4 Test de Miller-Rabin

Le test de Miller-Rabin est un autre test de primalité probabiliste basé sur le *petit théorème de Fermat* (théorème 3). 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.

4.4.1 Algorithme

Dans le cas du *test de Miller-Rabin*, la propriété en question est un raffinement du *petit théorème de Fermat* (théorème 3). 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 telle 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 3) 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 6 : 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 le test renvoie premier à tort après k itérations est donc de $1/4^k$.

4.4.2 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 $n-1 = 2^s \cdot t$

4.4.3 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.

4.5 Test de Solovay-Strassen

Le *test de Solovay-Strassen* est un test de primalité probabiliste, publié par *Robert Solovay* et *Volker Strassen* en 1977. Ce test a une importance historique dans la démonstration de la faisabilité du cryptosystème RSA.

4.5.1 Algorithme

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

> **Théorème 6** (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 :*
>
> $$\begin{cases} 0 & \text{si } a \text{ est divisible par } p \\ 1 & \text{si } a \text{ est un résidu quadratique modulo } p \text{ mais pas divisible par } p \\ -1 & \text{si } a \text{ n'est pas un résidu quadratique modulo } p \end{cases}$$
>
> *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 7 : 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 6). 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 6), 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 6) 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.

4.5.2 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.

On rappelle qu'on avait obtenu des complexités en $O(log(n)^2 \cdot log(log(n)) \cdot log(log(log(n))))$ en utilisant la multiplication FFT et en $O(log(n)^{1+log_2(3)})$ (avec $1 + log_2(3) > 2$) avec la multiplication de *Karatsuba*.

4.5.3 Preuve

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

> **Théorème 7** (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, $x^2 \equiv a$ mod 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 :

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

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 $x^2 \equiv a$ mod p, on a :

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

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

On sait que d'après le *théorème de Lagrange*, $(a^{\frac{p-1}{2}}-1) \equiv 0 \mod p$ 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 $a^{\frac{n-1}{2}} \equiv -1 \pmod{p}$.

4.6 Test AKS

Le test AKS est un test de primalité déterministe publié en 2002 par trois scientifiques indiens *Manindra Agrawal*, *Neeraj Kayal* et *Nitin Saxena*, à qui il doit le nom. Ce test est le premier en mesure de déterminer la primalité d'un nombre dans un temps polynomial tout en s'appuyant sur des propriétés démontrées et non des hypothèses.

4.6.1 Algorithme et principe de la preuve

L'algorithme du test AKS est basé sur une généralisation du *petit théorème de Fermat* :

> **Théorème 8** (Petit théorème de Fermat généralisé)**.** *Pour tout entier $n \geqslant 2$ et $a \in \mathbb{Z}$ premiers entre eux,*
>
> $$n \text{ est premier} \quad \Leftrightarrow \quad (X+a)^n \equiv X^n + a \pmod{n}$$
>
> *Le symbole X représente un symbole formel.*

Ce théorème est démontrable à l'aide de la formule du *binôme de Newton* et de la propriété suivante du *coefficient binomial* :

$$n \text{ est premier} \quad \Leftrightarrow \quad \forall k \in 1,...,n-1, \binom{n}{k} \equiv 0 \pmod{n}$$

L'algorithme déterministe de test de primalité déduit à partir de ce théorème est très simple, mais il faut évaluer les n coefficients du polynôme $(X+a)^n$ à l'aide du *binôme de Newton* en plus de l'exponentiation modulaire, ce qui est bien trop long.

La solution proposée dans le test est de réduire le polynôme modulo $X^r - 1$ avec r bien choisi. Cependant avec cette modification, même si l'implication du théorème "$\Rightarrow$" est vérifiée, ce n'est plus le cas pour la réciproque.

Ce problème est géré dans les premières étapes du test. En effet, si l'équation est vérifiée pour r bien choisi et un nombre suffisant de a (obtenus en temps polynomiaux) alors n est une puissance de nombre premier, c'est-à-dire, $n = p^b$ avec p premier et $b \in \mathbb{N}^*$.

Ainsi, on obtient finalement un test de primalité déterministe polynomial en la taille de l'entrée n. On définit la notion d'*ordre multiplicatif* qu'on va utiliser dans l'algorithme :

> **Définition 9** (Ordre multiplicatif)**.** *Soient les entiers $n \in \mathbb{N}$ et $a \in \mathbb{Z}$ premiers entre eux, l'ordre multiplicatif de a modulo n, noté $Ord_n(a)$, est le plus petit entier $k > 0$ tel que $a^k \equiv 1 \pmod{n}$.*

L'algorithme du test AKS est composé des 6 étapes suivantes :

Algorithme 8 : Test AKS

Données : un entier $n > 1$

1 - **si** $n = a^b$ *pour des entiers* $a > 1$ *et* $b > 1$ **alors**
  retourner composé;

2 - Déterminer le plus petit entier r tel que $Ord_r(n) > log_2(n)^2$ dans $\mathbb{Z}_r$ (si r n'est pas premier avec n, on passe cet r);

3 - **pour** *tout* $a \leq r$ **faire**
  si $1 < pgcd(a,n) < n$ **alors**
    retourner composé;

4 - **si** $n \leq r$ **alors**
  retourner premier;

5 - **pour** $a = 1$ *jusqu'à* $\lfloor \sqrt{\varphi(r)} log_2(n) \rfloor$ **faire**
  si $(X+a)^n \not\equiv X^n + a$ *dans* $\frac{\mathbb{Z}/n\mathbb{Z}[X]}{(X^r-1)\mathbb{Z}/n\mathbb{Z}[X]}$ **alors**
    retourner composé;

6 - **retourner** premier;

4.6.2 Complexité

En supposant que les opérations d'addition, de multiplication et de division s'effectuent toutes en $log(n)$, l'ordre de grandeur de la complexité temporelle prouvée par les auteurs du test AKS est $O(log(n)^{15/2})$.

Une borne moins précise mais démontrable plus facilement est $O(log(n)^{21/2})$ c'est la première complexité prouvée et elle a donc servi à établir que trouver un nombre premier est un problème que l'on peut résoudre en temps polynomial.

La complexité repose sur l'étape 5 de l'algorithme. En effet, l'entier r trouvé à l'étape 2 est borné par $log(n)^5$, cela dit, sous certaines conjectures non prouvées (*Artin* et *Sophie-Germain*), cette borne est réduite à $log(n)^2$ ce qui améliore la complexité de l'algorithme en $O(log(n)^6)$. On retient cette complexité car les preuves des conjectures sont sur la bonne voie.

Finalement, si la conjecture d'*Agrawal* est vérifiée, ce test peut être amélioré en $O(log(n)^3)$ ce qui le rendrait assez comparable aux tests probabilistes plus largement utilisés en cryptographie.

5 Mesures de performance et comparatifs

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

5.1 Évolution des tests de primalité

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

Premiers tests déterministes

Les plus anciens algorithmes de tests de primalité 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 Wilson***. Il est énoncé pour la première fois par le mathématicien *Ibn al-Haytham* dans les années 1000, puis énoncé de nouveau au 18e siècle, pour enfin être prouvé par *Lagrange* une année plus tard. Cependant, la complexité reste trop élevée surtout pour de grands nombres.

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

Tests probabilistes

Les algorithmes de tests probabilistes constituent une façon beaucoup plus efficace que les premiers algorithmes déterministes découverts pour tester la primalité 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 primalité 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 logement 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.

Test déterministe rapide : AKS

Après plus de 20 ans de recherche, et par soucis de pallier au caractère probabiliste des tests de primalité utilisés jusqu'ici, un autre algorithme déterministe plus performant a été découvert. Il s'agit du test de primalité ***AKS*** qui a une complexité en temps polynomiale.

Cependant, la complexité de ce test n'est pas assez compétitive pour remplacer l'utilisation du test probabiliste de ***Miller-Rabin*** dans les cryptosystèmes. Cela dit, ce test déterministe est assez récent et comme décrit dans la partie dédiée au test ***AKS***, des travaux futurs envisagent de faire réduire cette complexité.

5.2 Mesure du temps d'exécution

Pour comparer les performances des algorithmes de test de primalité 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.

Algorithme de mesure

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

Algorithme 9 : Mesure temps exécution

Données : tableau tps[6][1025] à remplir, correspondant au temps d'exécution des 6 test pour un nombre de bits entre 0 et 1024
Sorties : tableau tps[6][1025] rempli
pour *chaque test* ($i = 0$ *jusqu'à* 5) **faire**
    pour *pour un nombre de bits allant de 0 à 1024* ($j = 0$ *jusqu'à* 1024) **faire**
        Générer un nombre premier p de j bits;
        tps[i][j] $\leftarrow$ temps pour que le test i vérifie p;

Cet algorithme est implémenté dans le fichier source mesureTemps.c.

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 conclue par rapport aux performances.

Le logiciel `Gnuplot` va être utilisé pour faire ces représentations graphiques. Pour cela, il faut écrire les résultats dans un fichier bien formaté dont voici des extraits :

```
# Test Fermat :
# BITS TEMPS
0  0.000000
1  0.000000
2  0.000384
3  0.000001
```

```
4  0.000385
5  0.000000
6  0.000375
7  0.000377
8  0.000368
9  0.000395
10  0.000381
11  0.000379
12  0.000367
13  0.000368
14  0.000368
15  0.000384
16  0.000384
17  0.000366
18  0.000379
19  0.000397
20  0.000365
21  0.000364
22  0.000366
23  0.000386
24  0.000381
25  0.000364
26  0.000363
27  0.000376
28  0.000371
29  0.000379
30  0.000366
31  0.000419
32  0.000373
```

Listing 1 – mesures.txt (Fermat)

```
# Test AKS :
# BITS TEMPS
0  0.000000
1  0.000000
2  0.000002
3  0.000055
4  0.000050
5  0.000115
6  0.000186
7  0.000660
8  0.000209
9  0.001069
10  0.001120
11  0.001270
12  0.000410
13  0.000775
14  0.002260
15  0.008880
16  0.001002
```

```
17  0.005463
18  0.001205
19  0.000742
20  0.001808
21  0.005990
22  0.006231
23  0.024235
24  0.006482
25  0.002028
26  0.002255
27  0.014572
28  0.012850
29  0.004259
30  0.003963
31  0.004276
32  0.013833
```

Listing 2 – mesures.txt (AKS)

Ce fichier contient 6 blocs de 1024 lignes, dont chaque bloc correspond à un test de primalité et dont les lignes correspondent au temps d'exécution mesuré pour le nombre de bit donné. En exploitant les résultat de ce fichier, le logiciel `Gnuplot` va permettre de construire les courbes représentatives du temps d'exécution de chaque test de primalité en fonction de la taille en bits du nombre premier à générer :

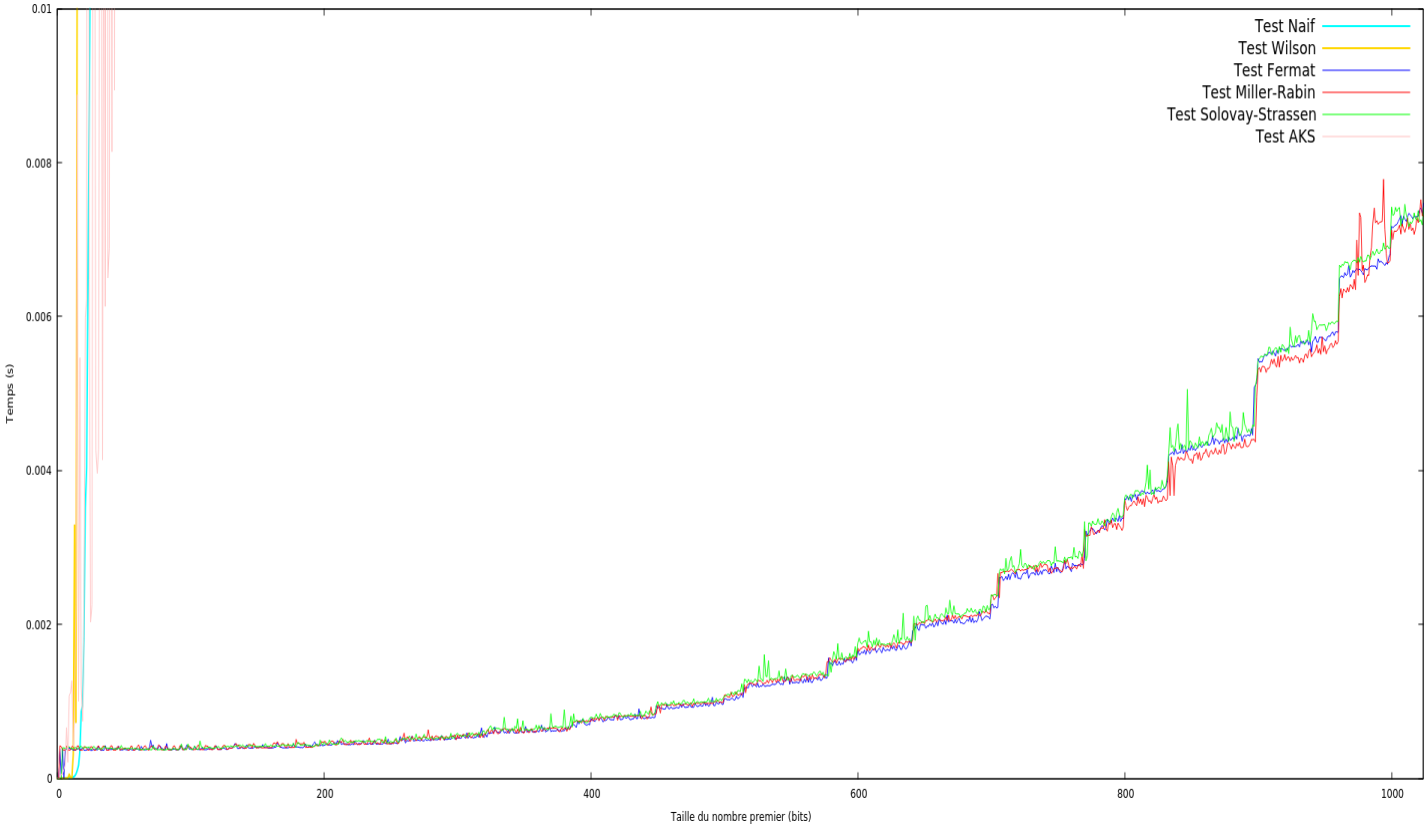

FIGURE 4 – Temps d'exécution des tests en fonction de la taille en bits du nombre premier

Ce graphique, qui est la superposition des courbes du temps d'exécution des 6 tests de primalité implémentés, va permettre d'observer les performances des différents algorithmes de test.

On remarque tout d'abord que les 3 tests déterministes implémentés, qui sont les tests ***Naif***, ***Wilson*** et ***AKS***, ont un comportement assez similaire.

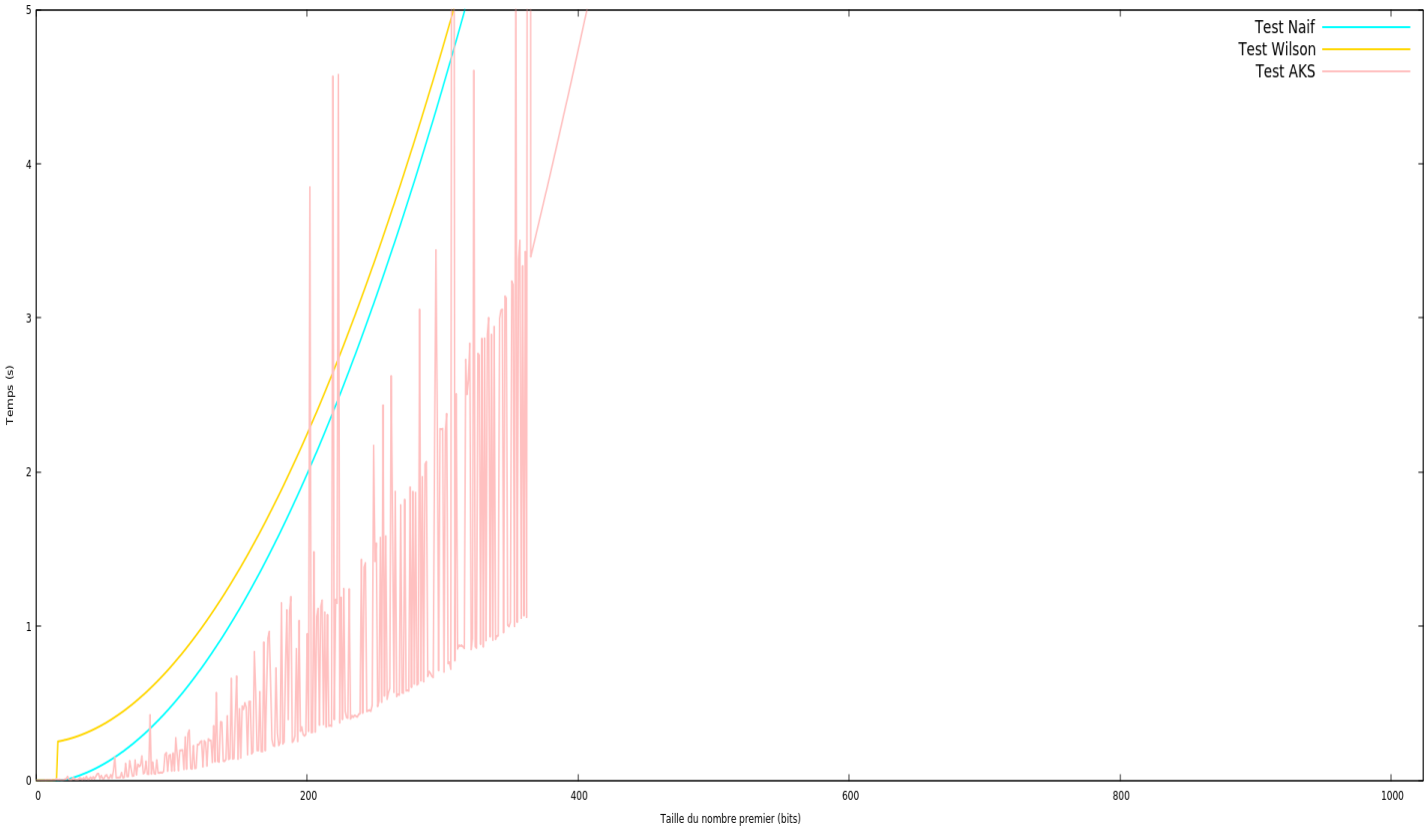

FIGURE 5 – Temps d'exécution des tests déterministes

En effet, ces 3 tests ont un temps de réponse rapide pour un nombre de bits petit, mais ce temps de réponse va croitre d'une manière exponentielle à partir de 16 bits pour le ***test Naif*** et le ***test de Wilson***, ce qui en fait des tests très lents globalement.

Le ***AKS*** a un temps assez variable, qui reste nettement plus rapide que le temps du ***test Naif*** et du ***test de Wilson*** jusqu'au nombres de taille 256 bits. Mais ce temps est relativement lent comparé aux tests probabilistes.

On va maintenant s'intéresser plus en détails au tests probabilistes. Voici la courbe représentative du temps d'exécution de ces tests :

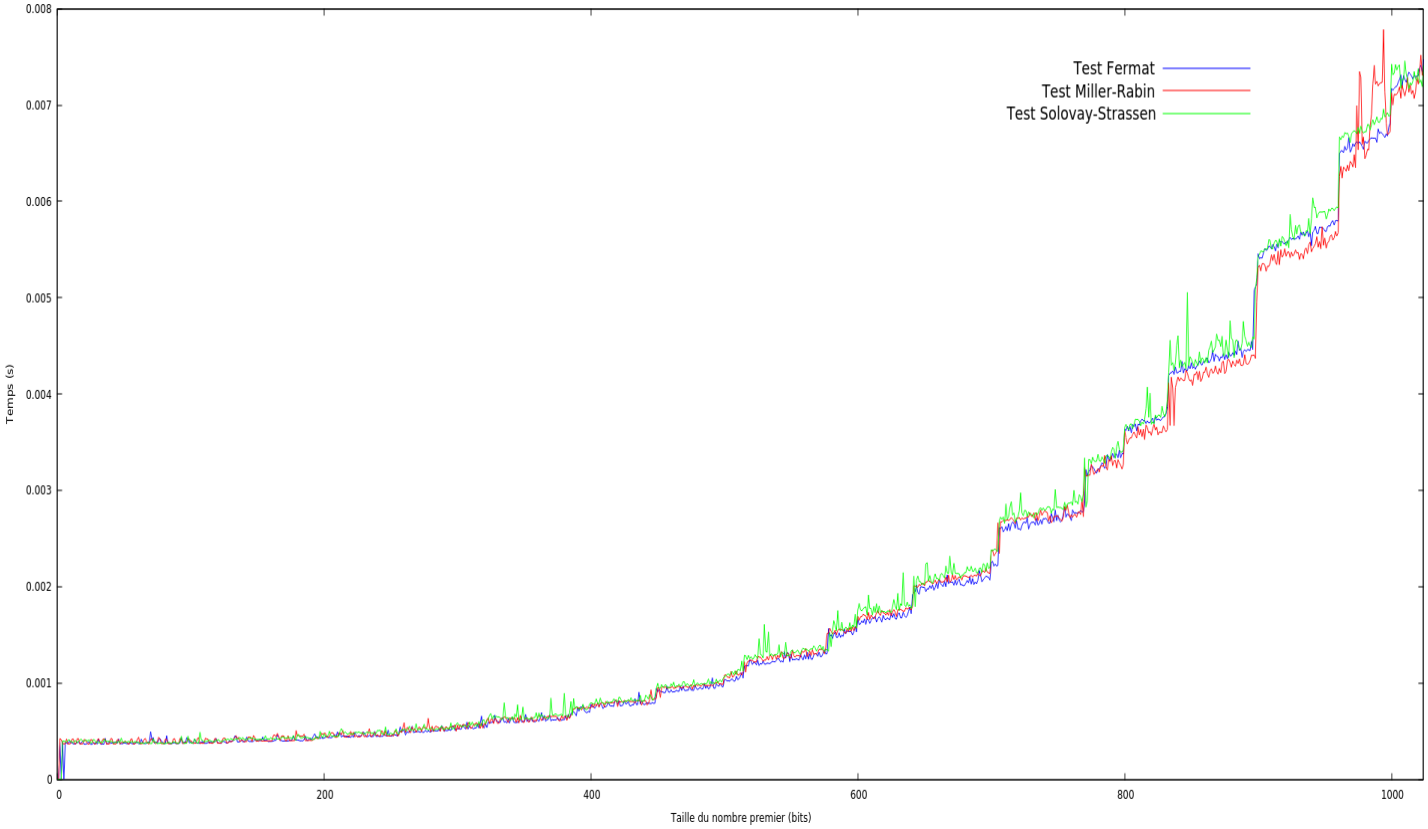

FIGURE 6 – Temps d'exécution des tests probabilistes

Concernant les tests probabilistes, ceux-ci ont un temps d'exécution rapide en moyenne. Les 3 tests 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 et qu'il est le plus performant quand le nombre de bits est très grand, à partir de 800 bits à peu près.

5.3 Génération optimale d'un nombre premier

Compte tenu des résultats des mesures effectuées sur le temps d'exécution de chaque test de primalité, il est maintenant possible, lors de la génération d'un nombre premier de t bits, de choisir l'algorithme de test de primalité qui présente le temps d'exécution minimal pour t bits. Pour cela, on aura juste à exploiter les résultats pré-calculés dans la structure `tps[6][1025]` de la manière suivante :

Algorithme 10 : RPNG Optimal

Données : la taille t en bits de l'entier premier à générer et le tableau tps[6][1025] des résultats de mesure du temps d'exécution des 6 test pour un nombre de bits entre 0 et 1024

Sorties : un nombre premier de t bits

Trouver le test de primalité d'indice i (i entre 0 et 5) telle que tps[i][t] est minimale (c'est-à-dire le test le plus rapide pour vérifier t bits);

Générer un nombre premier de t bits avec le test de primalité d'indice i;

Conclusion

Ce document est le rapport final de notre projet sur les tests de primalité. Ce travail a abouti par une implémentation de divers tests de primalité, une description de leur principe et de leur preuve, et une comparaison des performances de ces tests.

On peut revoir, avec du recul, la manière avec laquelle notre travail a été effectué. En effet, les majeures difficultés ont été principalement trouvées lors de l'établissement des complexité et surtout des preuve des algorithmes implémentés. En ce qui concerne l'implémentation, le *test AKS* a été le plus difficile, on a donc eu a réutiliser du code libre pour construire l'algorithme final.

Certains points restent encore ouverts. On pourrait envisager d'appliquer d'autres méthodes de mesure de performance pour établir non pas un générateur optimal, mais un test de primalité optimal. 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 compétence, mais aussi sur le plan de travail collectif, qui nous inspirera certainement pour de futurs projets.