Tests de primalité

Robert CONTIGNON Paulo PORTINHA

UNIVERSITÉ DU LUXEMBOURG -
Année académique 2016 - 2017 (Semestre d'été)

Contents

1 Introduction

Nous rencontrons les nombres premiers très tôt dans notre vie : À l'école primaire déjà, l'on apprend la notion de nombre premier (un nombre naturel est premier s'il n'a que 2 diviseurs : 1 et lui-même) et on dresse la liste des nombres premiers inférieurs à 100. Puis au lycée, on découvre qu'il existe plein d'autres nombres premiers que ceux rencontré á l'école primaire et l'on apprend ce que les mathématiciens appellent le théorème fondamental de l'algèbre (ici on considère une version simplifiée du théorème) : Tout nombre naturel se décompose de facon unique en produit fini de puissances naturelles de nombres premiers. Ainsi, toute personne qui finit la scolaritóbligatoire a au moins une fois de sa vie entendu parler des nombres premiers.

Mais beaucoup d'entre nous ont dû se demander à l'époque comment on peut trouver des nombres premiers qui sont "très grands" à partir des quelques notions que l'on a appris dans l'enseignement primaire et secondaire. Et bien, cette réponse a été fournie par plusieurs mathématiciens tout au fil des siècles. Néanmoins, la recherche d'algorithmes permettant de trouver des nombres premiers a connu une nette révolution avec l'apparition des ordinateurs. En effet, ces machines électroniques sont entre autres des super-calculatrices et sont capables d'effectuer une énorme quantité de calculs de en une seconde, ce qui a largement contribué à leur succès. C'est d'a'illeurs grâce aux ordinateurs que beaucoup d'algorithmes concernant la recherche de nombres premiers, que l'on appelle communément les tests de primalité, ont vu le jour et ont pu être testé facilement.

À ce jour, il existe plusieurs tests de primalité, mais dans le cadre de notre projet, nous allons nous limiter à l'étude de 3 différents tests de primalité: le test de Lucas-Lehmer, de Miller-Rabin et AKS.

2 Les tests de primalité

2.1 Notions utiles préparatoires

Dans cette sous-section, nous allons regrouper toutes les notions vues jusqu'à présent à l'université qui serviront par la suite pour expliquer les divers tests et ainsi créer nos programmes informatiques.

Définition 2.1. On appelle **nombre premier** un nombre naturel qui n'est divisible que par 1 et par lui-même.

Définition équivalente : On appelle **nombre premier** un nombre naturel dont aucun des nombres naturels strictement compris entre 1 et ce nombre naturel en est un diviseur.

En langage mathématique, cette définition est donnée ainsi :

Soit $p \in \mathbb{N}^*$. *On dit que* p *est premier si* $\forall i \in [2, 3, ..., p-1] : i \nmid p$.

Définition 2.2. On appelle **test de primalité** un algorithme mathématique (informatique) qui traite un ou plusieurs nombres naturels non nuls et qui dit, en retournant soit la valeur *True* soit la valeur *False*, si ce ou ces nombres sont premiers.

Théorème 2.1. *Il existe une infinité de nombres premiers.*

Définition 2.3. Soit n un entier naturel. Deux entiers relatifs a et b sont dits **congrus modulo n** si leur différence est divisible par n, c'est-à-dire si a est de la forme b + kn avec k entier.

Si n=0, alors la congruence entre a et b devient simplement une égalité.

Deux entiers a et b sont alors congrus modulo n si et seulement si le reste de la division euclidienne de a par n est égal à celui de la division de b par n.

On note la **congruence** à l'aide su symbole $\equiv$. Ainsi, a et b sont congrus modulo n peut s'écrire sous la forme suivante : $a \equiv b \ mod\ n$.

2.2 Les tests de primalité

2.2.1 Le test de Lucas-Lehmer

Le test de Lucas-Lehmer est l'un des premiers véritables tests de primalité. Il a été mis en oeuvre par le mathématicien francais Édouard Lucas en 1878 et a été modifié en une version plus forte par Derrick Lehmer, mathématicien américain important du XX^{me} siècle, environ 60 ans plus tard. Les 2 ayant travaillé sur ce même test malgré qu'ils n'étaient pas de la même époque, ce test porte jusqu'à ce jour le nom de test de Lucas-Lehmer. Ce test est un test que l'on dénomme non généraliste et déterministe, car il permet de trouver, et ce avec une certitude absolue, uniquement les nombres premiers de Mersenne dont l'explication sera fournie un peu plus loin dans ce paragraphe.

Avant d'expliquer le principe de ce test, nous allons d'abord brièvement traiter la version originale du test de Lucas. Le mathématicien francais a, en 1878, découvert un test de primalité basé sur le petit théorème de Fermat, mais l'a amélioré en 1891. Voici donc la vraie version du test de Lucas :

Théorème 2.2. *Soit* $p > 1$, $p \in \mathbb{N}$. *Supposons qu'il existe un* $a > 1$ $(a \in \mathbb{N})$ *tel que :*

1. $a^{N-1} \equiv 1 (mod\ p)$
2. $a^m \not\equiv 1 (mod\ p)\ \forall m < p, tel\ que\ m|(p-1)$

Alors p *est premier.*

Proof. Il suffit de montrer que tout entier m, $1 \leq m < p$, est premier à p, soit, $\varphi(p) = p - 1$. Pour cette tâche, il suffit de montrer qu'il existe a, $1 \leq a < p$, pgcd(a,p) = 1, tel que l'ordre de a mod p est p-1. C'est exactement ce qu'on déchiffre dans l'hypothèse.

□

Même si ce test a l'air efficace vu sa simplicité, le principal problème est que $\forall p$, il faut toujours connaître la décomposition en facteurs premiers de $p - 1$, ce qui devient rapidement problématique pour les grands nombres.

Pour remédier à ce problème, il a fallu attendre environ 40 ans avant que Derrick Lehmer, fameux mathématicien américain du XX^{me} siècle, poursuive le travail de Lucas jusqu'à donner une version plus puissante du test de Lucas. Mais avant d'en arriver à l'énoncé du test de Lucas-Lehmer, nous allons introduire la notion de nombre premier de Mersenne (mathématicien et religieux érudit francais du $XVII^{me}$ siècle), notion sur laquelle le test est essentiellement basé.

Définition 2.4.

- Un **nombre de Mersenne** est un terme de la suite $(M_n)_{n\in\mathbb{N}} = 2^n - 1$.
- Un **nombre premier de Mersenne** est un nombre $2^p - 1$ où p est premier. On le note M_p.

À présent, on peut énoncer le principe du test de Lucas-Lehmer :

Théorème 2.3. *(Théorème/Test de Lucas-Lehmer)*
Soit la suite d'entiers $(s_i)_{i\geq 0}$ définie par récurrence de la facon suivante :

$$\begin{cases} s_0 &= 4 \\ s_{i+1} &= (s_i)^2 - 2 \end{cases}$$

Soit p un nombre premier impair.
Alors le nombre de Mersenne $M_p = 2^p - 1$ est premier $\leftrightarrow M_p|(s_{p-2})$.
Le nombre (s_{p-2})mod M_p est appelé le <u>*résidu de Lucas-Lehmer*</u> *de p.*
La réciproque de ce théorème est aussi vraie.

Proof. Avant d'entamer la preuve du théorème, nous allons énoncer (sans démonstration) quelques assertions intermédiaires dont on aura besoin par la suite pour la preuve.

Lemme 2.1. *Soit $0, 1 \neq a \in \mathbb{Z}$ sans facteur carré.*

1. *L'application $\varphi : \mathbb{Z}[X] \to \mathbb{Z}[\sqrt{a}], f(X) \mapsto f(\sqrt{a})$ est un épimorphisme d'anneau avec $\ker\varphi = (X^2 - a)$.*

2. *Soit $M \in N \geq 1$ un entier positif et notons par I l'idéal principal $I = M \cdot \mathbb{Z}[\sqrt{a}] \leq \mathbb{Z}[\sqrt{a}]$.*

   Alors on a un isomorphisme d'anneau naturel $(\mathbb{Z}/M\mathbb{Z})[X]/(X^2 - a) \simeq \mathbb{Z}[X]/(M, X^2 - a) \overset{\overline{f(X) \mapsto f(\sqrt{a})}}{\Longrightarrow} \mathbb{Z}[\sqrt{a}]/I$, où a est la classe modulo M.

3. *Soit $M \geq 3$ un nombre premier tel que $(\frac{a}{m}) = -1$. Alors $\mathbb{Z}[\sqrt{a}]/I$ est un corps fini avec M^2 éléments.*

Théorème 2.4. *(Loi de réciprocité quadratique)*

Soit $p \neq q$ 2 nombres premiers impairs distincts.

1. $(\frac{-1}{p}) = (-1)^{\frac{p-1}{2}} = \{1$ *si* $p \equiv 1$ *mod* $4, -1$ *si* $p \equiv 3$ *mod* 4.

2. $(\frac{2}{p}) = (-1)^{\frac{p^2-1}{8}} = \{1$ *si* $p \equiv 1, 7$ *mod* $8, -1$ *si* $p \equiv 3, 5$ *mod* 8.

3. $(\frac{q}{p}) = (\frac{p}{q} \cdot (-1)^{\frac{p-1}{2} \cdot \frac{q-1}{2}}$ *. En particulier, si $p \equiv 1$ mod 4 ou $q \equiv 1$ mod 4, alors $(\frac{q}{p}) = (\frac{p}{q})$.*

Proposition 2.1. *(Euler)*

Soit $p > 2$ un nombre premier et $a \in \mathbb{Z}$. Alors on a la congruence $(\frac{a}{p}) \equiv a^{\frac{p-1}{2}} mod\ p$

À présent, nous pouvons démontrer le théorème de Lucas-Lehmer :

On travaille dans l'anneau $\mathbb{Z}[\sqrt{3}]$ et un anneau quotient quelconque. Écrivons $\omega = 2 + \sqrt{3}$,$\bar{\omega} = 2 - \sqrt{3} \in \mathbb{Z}[\sqrt{3}]$. Nous avons $\omega \cdot \bar{\omega} = 4 - 3 = 1$ (tel que ω est une unité) et $\omega = \frac{1}{2} \cdot (1 + \sqrt{3})^2$.

Montrons d'abord par récurrence : $S_k = \omega^{2^{k-1}} + \bar{\omega}^{2^{k-1}} \qquad (E_1)$

Le cas $k = 1$ est simplement l'égalité $S_1 = 4 = \omega + \bar{\omega}$.

Le cas général est le calcul suivant :

$S_{k+1} = (S_k)^2 - 2 = (\omega^{2^{k-1}} + \bar{\omega}^{2^{k-1}})^2 - 2 = \omega^{2^k} + \bar{\omega}^{2^k} + 2(\omega \cdot \bar{\omega})^{2^{k-1}} - 2 = \omega^{2^k} + \bar{\omega}^{2^k}$.

$\underline{\Leftarrow}$:

Supposons $M = M_p | S_{p-1}$, mais que M n'est pas un nombre premier. Soit alors $q|M$ un nombre premier tel que $q \leq \sqrt{M} = \sqrt{2p-1} < 2^{\frac{p}{2}}$.

La supposition $M|S_{p-1}$ cmbinée avec (E_1) implique $S_{p-1} = \omega^{2^{p-2}} + \bar{\omega}^{2^{p-2}} \equiv 0\ mod\ M \cdot \mathbb{Z}[\sqrt{3}]$, à partir duquel on conclue $\omega^{2^{p-2}} \equiv -\bar{\omega}^{2^{p-2}}\ mod\ M \cdot \mathbb{Z}[\sqrt{3}]$ et par conséquent $\omega^{2^{p-1}} \equiv -1\ mod\ M \cdot \mathbb{Z}[\sqrt{3}]$.

Ceci implique en particulier $\omega^{2^{p-1}} \equiv -1\ mod\ q \cdot \mathbb{Z}[\sqrt{3}]$, d'où l'image de ω est d'ordre 2^p dans $(\mathbb{Z}/q\mathbb{Z})[\sqrt{3}]$. Ainsi, $2^p < q^2 < (2^{\frac{p}{2}})^2 = 2^p$, ce qui est la contradiction recherchée.

$\underline{\rightarrow :}$ Nous supposons maintenant que $M = M_p$ est un nombre premier. On peut montrer que $(\frac{3}{2^{p-1}}) = \frac{3}{M} = -1$,d'où le lemme précédent montre que $R = \mathbb{Z}[\sqrt{3}]/M \cdot \mathbb{Z}[\sqrt{3}] \simeq \mathbb{F}_M[X]/(X^2 - \bar{3})$ est le corps avec M^2 éléments.

Montrons d'abord la congruence $\omega^{2^{p-1}} \equiv -1 \; mod \; M \cdot \mathbb{Z}[\sqrt{3}] \qquad (E_2)$.

La définition $M = 2^p - 1$ peut être reformulée ainsi : $2^{p-1} = \frac{M+1}{2}$. Par la proposition d'Euler et la loi de réciprocité quadratique, on a $(\frac{1}{2})^{\frac{M-1}{2}} \equiv (\frac{2}{M}) \equiv 1 \; mod \; M$, vu que $M \equiv -1 \; mod \; 8$.

En plus de cela, par le fait mathématique que l'on a utilisé auparavant au début de cette partie de preuve, on voit que $3^{\frac{M-1}{2}} \equiv (\frac{3}{M}) \equiv -1 \; mod \; M$.

On effectue le calcul suivant dans le corps R, qui est de charactéristique M (tel que $(a+b)^M = a^M + b^M$).

$\omega^{2^{p-1}} = (\frac{1}{2} \cdot (1+\sqrt{3})^2)^{2^{p-1}} = (\frac{1}{2})^{\frac{M+1}{2}} \cdot (1+\sqrt{3})^{M+1} = (\frac{1}{2})^{\frac{M-1}{2}} \cdot \frac{1}{2} \cdot (1+\sqrt{3}) \cdot (1+\sqrt{3})^M = \frac{1}{2} \cdot (1+\sqrt{3}) \cdot (1+\sqrt{3})^M = \frac{1}{2} \cdot (1+\sqrt{3}) \cdot (1+\sqrt{3}^M) = \frac{1}{2} \cdot (1+\sqrt{3}) \cdot (1+3^{\frac{M-1}{2}} \cdot \sqrt{3}) = \frac{1}{2} \cdot (1+\sqrt{3}) \cdot (1-\sqrt{3}) = -1$.

On utilise à présent (E_2) pour déduire que $-\bar{\omega}^{2^{p-2}} = \omega^{2^{p-1}} \cdot \bar{\omega}^{2^{p-2}} = \omega^{2^{p-2}} \cdot \omega^{2^{p-2}} \cdot \bar{\omega}^{2^{p-2}} = \omega^{2^{p-2}}$ et ainsi le tant recherché $S_p - 1 = \omega^{2^{p-2}} + \bar{\omega}^{2^{p-2}} = 0$ dans le corps R.

□

Le test de Lucas-Lehmer a pour particularité de fournir tous les nombres premiers de Mersenne vu que l'on teste en fait chaque nombre de Mersenne. Toutefois, ce test a l'inconvénient de ne fournir que ces nombres particuliers et omet ainsi plein d'autres nombre premiers. Mais en gros, ce test est pratique de part sa précision et son exactitude.

Grâce à ce théorème, on peut maintenant déduire un algorithme mathématique que l'on utilisera dans la prochaine section.
Décrivons-le en termes de phrases et par étape :

1. Choisir un nombre naturel p qui est à la fois impair et premier.
2. Calculer le nombre de Mersenne M_p correspondant.
3. Calculer s_{p-2} à l'aide de la boucle définie comme suit :
   (a) Initialiser la variable s_i en lui affectant la valeur 4, où i est une variable naturelle.
   (b) Créer la boucle de telle manière qu'elle parcout l'indice i de 0 à $p-2$.

   (c) Dans le corps de la boucle, l'on utilisera la relation de récurrence fournie par le théorème 2.3 afin d'effectuer les calculs des s_i jusqu'à avoir la valeur de s_{p-2}.

4. Calculer la division $\frac{s_{p-2}}{M_p}$ et évaluer son reste. S'il vaut 0, alors M_p est un nombre premier (de Mersenne), sinon c'en n'est pas un.

2.2.2 Le test de Miller-Rabin

Le test de Miller-Rabin est un test qui a été en grande partie développé par le mathématicien américain Gary Miller jusqu'en 1976 et dont certaines notions proviennent de celles employées par le mathématicien polonais d'origine israélienne Rabin, d'où le nom de ces 2 hommes pour ce test de primalité. Le test de Miller-Rabin est un test de primalité dit "probabiliste" en raison de l'utilisation de l'hypothèse de Riemann généralisée (par Rabin, car Miller avait à l'origine créé un test déterministe) qui n'a pas été vraiment démontrée et que l'on évoquera seulement ici pour comprendre le contexte dans lequel on se trouve. Puisque ce test est basé sur une hypothèse non-démontrée, les résultats provenant du test de Miller-Rabin sont vrais avec une certaine probabilité. Plus précisément, si un certain n naturel est premier, alors il l'est probablement. Sinon, il ne l'est pas avec une certitude absolue.

Le test de Miller-Rabin utilise plein de notions nouvelles que nous allons immédiatement définir en vue de comprendre le théorème qui décrit ce test.

Définition 2.5. 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 (probabilité d'être premier est comprise entre 0 et 1). Les nombres pseudo-premiers peuvent être classés selon la propriété qu'ils satisfont.

Citons par exemple la famille des nombres pseudo-premiers de Mersenne. La propriété satisfaite par ces nombres est qu'ils sont tous des termes de la suite de Mersenne.

Définition 2.6. Soit N un naturel,$N-1=2^s d$, avec $s \geq 0$,d impair. Soit 1 < a < N avec pgcd(a,N)=1. On dit alors que a est un **témoin de Miller** pour N si $a^d \not\equiv 1 (modN)$ et $a^{2^r d} \not\equiv 1 (modN) \forall r, 0 \leq r < s$.

Définition 2.7. Un **nombre composé** est un entier naturel différent de 0 qui possède un diviseur positif autre que 1 ou lui-même. Par définition, chaque entier plus grand que 1 est donc soit un nombre premier, soit un nombre composé, et les nombres 0 et 1 ne sont ni premiers ni composés.

Autre définition : Un nombre composé est le produit d'au moins deux nombres premiers (qu'ils soient distincts ou identiques).

Le théorème suivant établit les relations entre les 3 notions précdemment introduites. Il servira à mieux comprendre le test de Miller-Rabin.

Théorème 2.5. *Si a est un témoin de Miller, alors N est composé.*

Si N est composé, si 1 < a < N, si pgcd(a,N)=1 et si a n'est pas un témoin de Miller, alors N est un pseudopremier.

Réciproquement, si N est impair et N est un pseudopremier, alors a n'est pas un témoin de Miller pour N.

Nous présentons maintenant une proposition qui permet de déduire la méthode principale du test de Miller-Rabin pour tester la primalité d'un naturel non nul quelconque, à savoir trouver plusieurs et suffisamment de témoins de Miller pour montrer que le nombre concerné est composé respectivement pseudopremier.

Proposition 2.2. *Pour un nombre impair composé N,* $\frac{3}{4}$ *au moins des entiers a,* $1 < a < N$*, sont des témoins de Miller pour N.*

Nous pouvons à présent énoncer le théorème de Miller qui est à l'origine du test de Miller :

Théorème 2.6. *Soit N un naturel impair. S'il existe a tel que* $pgcd(a, N) = 1, 1 < a < 2(logN)^2$*, qui est un témoin de Miller pour N, alors N est composé. Sinon, N est premier.*

Étant donné que ce test est déterministe, il ne correspond pas encore tout à fait au test de Miller-Rabin, et ce pour la simple raison que Rabin a formulé un autre théorème qui a rendu le test de Miller probabiliste, ce qui a donné naissance au fameux test de Miller-Rabin.

Théorème 2.7. *Soit n un entier impair composé > 9, avec* $n - 1 = 2^s \cdot d$ *pour d impair. Alors il existe au plus* $\frac{\varphi(n)}{4}$ *menteurs forts a associés au pseudo-premier n (pgcd(a,n)=1), pour 1 < a < n, c'est-à-dire des entiers a dans cet intervalle vérifiant soit* $a^d \equiv 1 \ mod \ n$*, soit,*$a^{d2^r} \equiv -1 \ mod \ n$ *pour un certain r tel que* $0 \leq r < s$ *(*φ *est la fonction indicatrice d'Euler).*

La récirpoque de ce théorème est également vraie.

Proof. Comme dans la preuve du test de Lucas-Lehmer, nous allons ici d'abord énumérer quelques assertions ainsi qu'une définition utile dans le cadre de cette preuve, et en admettant toutes ces assertions vraies sans les démontrer :

Théorème 2.8. *(Théorème des restes chinois)*

Soient $n_1, ..., n_k$ *des entiers 2 à 2 premiers entre eux.*

Alors pour tout entier $a_1, ..., a_k$*, il existe un entier x, unique modulo* $n = \prod_{i=1}^{k} n_i$*, tel que* $x \equiv a_i \ mod \ n_i \ \forall i \in \{1, 2, ..., k\}$.

Une autre version existe lorsque l'on travaille dans l'anneau $\mathbb{Z}/N\mathbb{Z}$. La voici :

Si $n_1, ..., n_k$ sont 2 à 2 premiers entre eux, alors, en notant n le PPCM des n_i, c'est-à-dire dans le cas présent le produit des n_i, l'application (à valeurs dans l'anneau produit) :

$$\phi : \mathbb{Z} \to \mathbb{Z}/(n_1)\mathbb{Z} \times ... \times \mathbb{Z}/(n_k)\mathbb{Z}$$

$$\alpha[n] \mapsto (\alpha[n_1], ..., \alpha[n_k])$$

est un isomorphisme d'anneaux.

Théorème 2.9. *(Application du théorème des restes chinois (TRC))*

Soit $N = (p_1)^{e_1} \cdot ... \cdot (p_k)^{e_k}$ un entier impair dans sa décomposition en facteurs premiers. Alors dans $\mathbb{Z}/N\mathbb{Z}$ l'équation $X^2 - 1$ a 2^k solutions : à savoir, sous le TRC $\mathbb{Z}/N\mathbb{Z} \simeq \mathbb{Z}/(p_1)^{e_1}\mathbb{Z} \times ... \times \mathbb{Z}/(p_k)^{e_k}\mathbb{Z}$ les solutions sont précisément données par les éléments $(a_1, ..., a_k)$ avec $a_i \in 1, -1$ $\forall 1 \leq i \leq k$ (notons que dans les anneaux $\mathbb{Z}/(p_i)^{e_i}\mathbb{Z}$, l'équation $X^2 - 1 = 0$ a exactement 1 et -1 comme solutions vu que le groupe des unités est cyclique par le lemme introduit dans la preuve du théorème de Lucas-Lehmer.

Ainsi, on obtient le critère de primalité suivant :

N est premier $\leftrightarrow X^2 - 1 = 0$ admet seulement 1,-1 comme solutions dans $\mathbb{Z}/N\mathbb{Z}$.

Définition 2.8. Un nombre naturel $n \in \mathbb{N}$ est appelé **nombre de Carmichael** si$n$ n'est pas premier et $a^{n-1} \equiv 1 \ mod \ n \ \forall a \in \mathbb{Z}$ tel que $gcd(a, n) = 1$.

Proposition 2.3. *Soit $N \in \mathbb{N}_{\geq 2}$. Les assertions suivantes sont équivalentes:*

1. *N est premier ou un nombre de Carmichael.*
2. *L'exposant de $(\mathbb{Z}/N\mathbb{Z})^\times$ est un diviseur de $N - 1$.*
3. *Pour tout nombre premier p tel que p divise N :*
   - *$p^2 \not| N$ (nombres qui ne sont pas divisible par le carré d'un quelconque nombre premier sont appelés sans facteur carré)*
   - *$(p - 1)|(N - 1)$.*

Proposition 2.4. *Pour tout nombre de Carmichael est impair et a au moins 3 diviseurs premiers.*

Nous pouvons maintenant passer à la preuve du théorème de Miller-Rabin:

$\rightarrow$:

Supposons que N est un nombre premier et soit $a \in (\mathbb{Z}/N\mathbb{Z})^\times$. Écrivons $r = ord(a)$, c'est un diviseur de $N - 1$, qui est l'ordre de $(\mathbb{Z}/N\mathbb{Z})^\times$.

1^{er} cas : r est impair

Alors $r|m$ et par conséquent $a^m \equiv 1\ mod\ N$.

2^{nd} cas : r est pair

Alors $r = 2^s \cdot r'$ avec un nombre impair $r' \in \mathbb{N}$ et nécessairement $r'|m$. Puisque $(a^{2^{s-1} \cdot r'})^2 \equiv 1\ mod\ N$,il s'ensuit que $a^{2^{s-1} \cdot r'} \equiv -1\ mod\ N$, parce que dans un corps 1 et -1 sont les uniques éléments dont le carré est 1.

$\leftarrow$:

Supposons que N n'est pas premier. Dans tout les cas, on obtient en élevant au carré que $a^{N-1} \equiv 1\ mod\ N \forall a \in (\mathbb{Z}/N\mathbb{Z})^x$, d'où N est un nombre de Carmichael et ainsi sans facteur carré,$N = p_1 \cdot ... \cdot p_k$ avec au moins 3 nombres premiers impairs distincts $p_1, ..., p_k$ tels que $(p_i - 1)|(N - 1)\ \forall\ 1 \leq i \leq k$ par les 2 propositions précédentes.

On utilise à présent le théorème des restes chinois (TRC) en vue de construire un $a \in (\mathbb{Z}/N\mathbb{Z})^\times$, violant ainsi toutes les conditions. Par l'isomorphisme $\mathbb{Z}/N\mathbb{Z} \simeq \mathbb{Z}/p_1\mathbb{Z} \times ... \times \mathbb{Z}/p_k\mathbb{Z}$ à partir du TRC on définit a en tant que $(-1, 1, 1, ..., 1)$. Puisque m est impair, il s'ensuit que $a^m \neq 1$ dans $(\mathbb{Z}/N\mathbb{Z})^\times$ car la première composante reste égale à -1 ; d'où la première condition est violée. Il est également évident que la seconde condition n'est pas respectée (étant donné que -1 n'est pas une puissance de 1).

$\square$

Remarque : menteur fort = "opposé" de témoin de Miller

Grâce à ce théorème et aux assertions et définitions de cette section, on peut en déduire un algorithme mathématique qui nous pemettra dans la prochaine section de programmer le test de Miller-Rabin.

Décrivons-le, tout comme dans le test de Lucas-Lehmer, à l'aide de phrases et par étapes :

1. Choisir un nombre naturel p impair non nul quelconque (p ne doit pas forcément être composé et $>$9 à l'avance, car on ne sait pas toujours si c'est le cas, surtout pour les très grands nombres).

2. Résoudre l'équation de l'énoncé du théorème 2.6 d'inconnues s et d, sachant que $s > 0$ et d est impair :

$$p - 1 = 2^s \cdot d$$

3. $\forall 1 < a < p$, vérifier si a est témoin de Miller ou pas.

4. Si on en a suffisamment assez (minoré par la Proposition 2.1), alors d'après les théorèmes 2.4 et 2.5, p est composé.

5. Sinon, p est probablement premier, c'est-à-dire pseudo-premier.

2.2.3 Le test AKS

Le test AKS, dont les lettres correspondent au nom de famille des 3 mathématiciens indiens Agrawal,Kayal et Saxena ayant formulé ce test, est l'un des tests de primalitéles plus récents de l'histoire mathématique. En effet, il a vu le jour au début des années 2000. Ce test a pour énorme avantage que, contrairement au test de Miller-Rabin dont on a parlé dans la soussection précédente, c'est un test déterministe. En d'autres mots, le tests de primalité AKS dit avec certitude (probabilité égale à 1) si un nombre naturel quelconque est premier ou pas. En plus de cela, il fonctionne pour n'importe quel naturel non nul.

Tout comme le test de Miller-Rabin, le test AKS dit si un nombre naturel donné est premier ou composé.

En ce qui concerne le principe de ce test, il est surtout basé sur le théorème suivant (généralisation du petit théorème de Fermat) :

Théorème 2.10. *Soit* $n \geq 2, n \in \mathbb{N}$. *Soit* $a \in \mathbb{N}$ *tel que* $pgcd(a, n) = 1$ *(on dit alors que* a *est premier à* n*).*

Alors : n *est premier* $\leftrightarrow (X + a)^n \equiv (X^n + a) \; mod \; n$.

Ce théorème est certes très utile vu qu'on le qualifie de tel, mais n'empêche que si l'on utilise ce résultat dans l'algorithme, le programme ne sera pas efficace en raison du temps de calcul nécéssaire de tous les coefficients (il y en a n par $(X + a)^n$) pour comparer et ainsi savoir si n est premier ou composé. C'est donc pour cette raison que la généralisation du petit théorème de Fermat a été réétudiée de manière approfondie dans le but de déduire un nouveau théorème ou simplement une variante, mais qui allège nettement le temps de calcul dont on a parlé auparavant.

Ceci a été réalisé avec succès par 2 mathématiciens indiens, qui ne sont ni Agrawal ni Saxena et ni Kayal. Les 2 hommes ont modifié le théorème 2.10 de manière à obtenir une nouvelle formulation de ce théorème, mais avec la principale différence que le temps de calcul est clairement réduit par rapport à la version initiale du théorème en question.

La nouvelle formulation du théorème 2.10 est la suivante :

Théorème 2.11. *Soit* $n \geq 2, n \in \mathbb{N}$. *Soit* $a \in \mathbb{N}$ *tel que* $pgcd(a, n) = 1$.

Alors n *est premier* $\leftrightarrow \exists r, r \leq n$ *tel que* $(X + a)^n \equiv X^n + a \; (mod \; X^r - 1, n)$.

Cette version de la généralisation du petit théorème de Fermat a été à cette période une avancée décisive dans la réalisation du test de primalité AKS avec comme seul et sacré problème qu'elle n'a pas été demontrée. Ceci a été le travail de Kayal et Saxena qui l'ont prouvé peu de temps après la publication du résultat.

À partir de cette version de théorème, ces 2 mathématiciens indiens avaient à l'époque déduit une assertion particulièrement importante. La voici :

Proposition 2.5. *Soit* $n \in \mathbb{N}$,$r \in \mathbb{N}$ *tel que* $r \leq n$. *Si* $r \not| n$ *et si* $(X+1)^n \equiv X^n + 1 \ (mod\ X^r - 1, n)$, *alors* n *est soit premier soit on a* $n^2 \equiv 1 \ (mod\ r)$.

Grâce au théorème 2.11, à la dernière proposition et au cas spécial, on peut rédiger un algorithme pour le test de primalité AKS (l'étude détaillée se fera au cours de la prochaine section). Néanmoins, avant d'en arriver au pseudo-code source, on va d'abord éliminer un cas spécial pour lequel n est composé.

Le voici :

Si $n = m^k \quad (m \in \mathbb{N}, m \geq 2, k \in \mathbb{N}^* - \{1\})$, alors n est composé.

En plus de cela, il est primordial de savoir que le trio de mathématiciens indiens est également parvenu à prouver que si n est composé, alors $\exists r \leq (\log_2(n)^5), \exists a \leq \sqrt{\varphi_{Euler}(r)}\log_2(n)^2$ tels que l'équation dans le théorème 2.11 n'est pas vérifiée. La fonction $\varphi_{Euler}(r)$ est appelée <u>indicatrice d'Euler</u> et associe au nombre r le nombre d'entiers entre 1 et r inclus qui sont premiers à r. Ceci est une assertion qui découle entre autre des théorèmes 2.10 et 2.11.

Le test AKS donne, contrairement au test de Lucas-Lehmer, tous les nombres premiers possibles. Décrivons son algorithme en mots et phrases et par étapes :

1. Choisir un quelconque nombre naturel p supérieur ou égal à 2.

2. On définit une fonction qui effectue le test AKS. Décrivons-la en détail :

   (a) S'il existe un nombre naturel q supérieur ou égal à 2 et un $k \in \mathbb{N}^* - \{1\}$ tels que $p = q^k$, alors n est composé.

   (b) Sinon, on sort de cette instruction conditonnelle et on cherche le plus petit r tel que $p^i = 1 mod r$ avec $1 \leq i \leq B$,B est le plus grand entier inférieur ou égal à $(\log_2(p))^2$ et $r \geq 2$ majoré par le plus grand entier inférieur ou égal à $(\log_2(p))^5$.

   (c) Étudier ensuite un cas particulier où un nombre donné peut être composé ; pour chaque nombre,appelons-le a, entre 1 et r inclus, on regarde si le pgcd de a et p sont copremiers. Si oui, alors p est composé.

   (d) Étudier encore un cas spécial, à savoir si p est inférieur ou égal à r. Si c'est le cas, alors p est premier.

(e) Définir une borne sur la future variable a en lui affectant la valeur $\sqrt{\varphi_{Euler}(r)}\log_2(p)^2$,$r$ étant un entier supérieur ou égal à 2.

(f) Créer une boucle **for** de la manière suivante :

(g) Calculer $(x+a)^p \ mod \ (x^r - 1)$.

(h) Calculer $n \ mod \ r$.

(i) Si le premier résultat est différent du second résultat, alors p est composé.

(j) Sinon, on continue de parcourir la boucle. Si l'on a parcouru toute la boucle sans vérifier une seule fois la condition précédente, alors p est premier.

3. Appliquer la fonction prédéfinie au nombre p choisi au départ. La fonction retourne donc soit le message *premier* soit le message *composé*.

3 Étude informatique des tests de primalité

Comme le titre de cette section l'indique, nous allons maintenant nous intéresser aux programmes informatiques sur les 3 tests de primalité que nous avons créés nous-même.

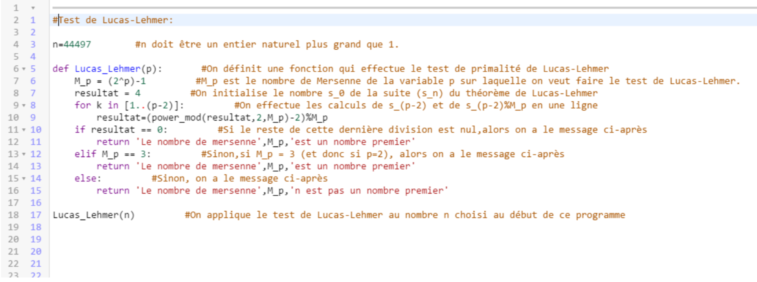
Pour chaque test de primalité, nous allons en premier lieu présenter le code source du programme sous forme de capture d'écran, le code source contenant également des explications des diverses étapes du programme. Ensuite, nous allons présenter 2 exemples en affichant le résultat final ; le premier exemple est celui d'un nombre qui est ou fournit un nombre premier tandis que le second est celui d'un nombre qui n'en est ou fournit pas un. Pour terminer l'étude, nous allons analyser la complexité informatique de chaque programme, plus clairement nous allons analyser la rapidité du programme en détaillant les endroits du programme où l'on perd du temps.

Enfin, nous préciserons au début de chaque sous-section quel est le plus grand nombre premier que l'on a trouvé.

3.1 Le test de Lucas-Lehmer

Le plus grand nombre premier de Mersenne que nous avons obtenu pendant un laps de temps correct (moins de 30 secondes) est $2^{44497} - 1$.

3.1.1 Code source



```
#Test de Lucas-Lehmer:

n=44497        #n doit être un entier naturel plus grand que 1.

def Lucas_Lehmer(p):       #On définit une fonction qui effectue le test de primalité de Lucas-Lehmer
    M_p = (2^p)-1         #M_p est le nombre de Mersenne de la variable p sur laquelle on veut faire le test de Lucas-Lehmer.
    resultat = 4         #On initialise le nombre s_0 de la suite (s_n) du théorème de Lucas-Lehmer
    for k in [1..(p-2)]:         #On effectue les calculs de s_(p-2) et de s_(p-2)%M_p en une ligne
        resultat=(power_mod(resultat,2,M_p)-2)%M_p
    if resultat == 0:         #Si le reste de cette dernière division est nul,alors on a le message ci-après
        return 'Le nombre de mersenne',M_p,'est un nombre premier'
    elif M_p == 3:         #Sinon,si M_p = 3 (et donc si p=2), alors on a le message ci-après
        return 'Le nombre de mersenne',M_p,'est un nombre premier'
    else:         #Sinon, on a le message ci-après
        return 'Le nombre de mersenne',M_p,'n est pas un nombre premier'

Lucas_Lehmer(n)         #On applique le test de Lucas-Lehmer au nombre n choisi au début de ce programme
```

Figure 1: Code source de l'algorithme de Lucas-Lehmer

3.1.2 Exemples

Cas où le nombre préalablement choisi fournit un nombre premier de Mersenne :

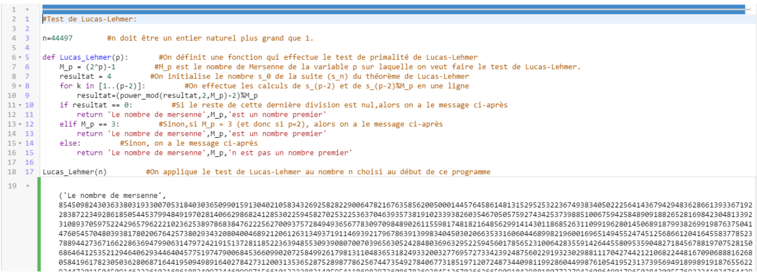


Figure 2: 1ère partie du programme

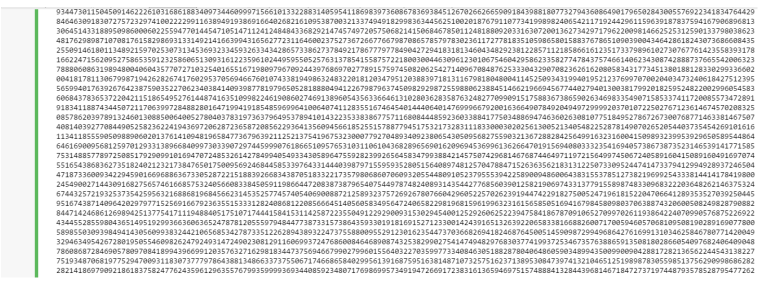


Figure 3: 2ème partie du programme

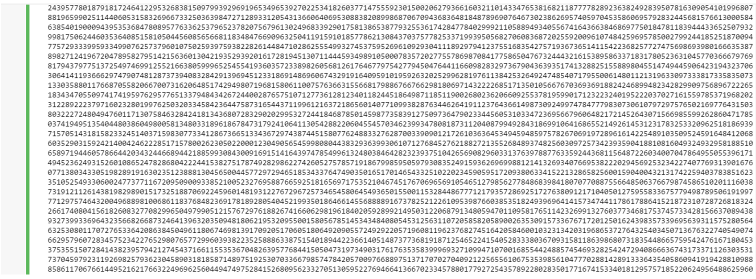
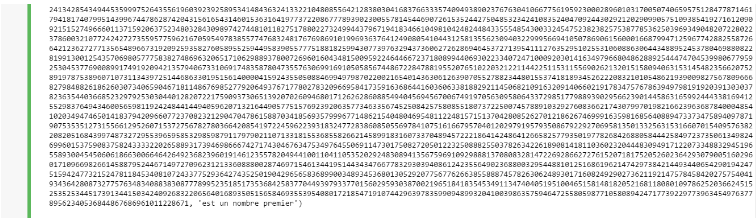


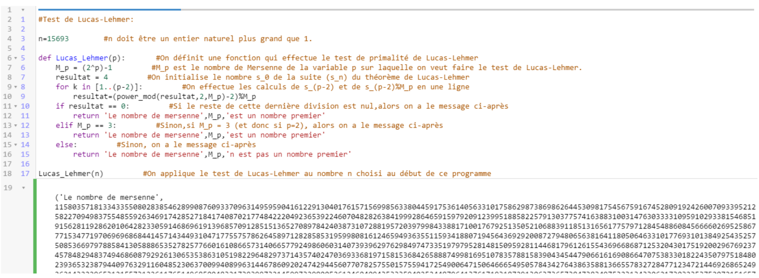
Figure 4: 3ème partie du programme



```
..., 'est un nombre premier')
```

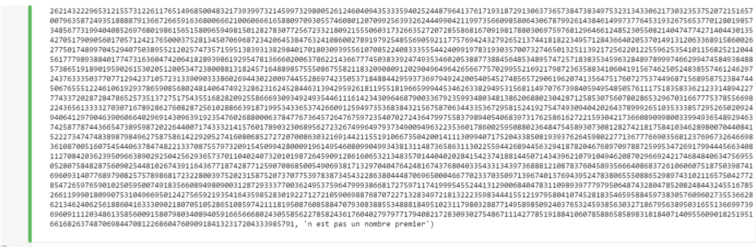
Figure 5: 4ème partie du programme

Cas où le nombre préalablement choisi ne fournit pas un nombre premier de Mersenne :



```
#Test de Lucas-Lehmer:

n=15693        #n doit être un entier naturel plus grand que 1.

def Lucas_Lehmer(p):       #On définit une fonction qui effectue le test de primalité de Lucas-Lehmer
    M_p = (2^p)-1         #M_p est le nombre de Mersenne de la variable p sur laquelle on veut faire le test de Lucas-Lehmer.
    resultat = 4         #On initialise le nombre s_0 de la suite (s_n) du théorème de Lucas-Lehmer
    for k in [1..(p-2)]:         #On effectue les calculs de s_(p-2) et de s_(p-2)%M_p en une ligne
        resultat=(power_mod(resultat,2,M_p)-2)%M_p
    if resultat == 0:         #Si le reste de cette dernière division est nul,alors on a le message ci-après
        return 'Le nombre de mersenne',M_p,'est un nombre premier'
    elif M_p == 3:         #Sinon,si M_p = 3 (et donc si p=2), alors on a le message ci-après
        return 'Le nombre de mersenne',M_p,'est un nombre premier'
    else:         #Sinon, on a le message ci-après
        return 'Le nombre de mersenne',M_p,'n est pas un nombre premier'

Lucas_Lehmer(n)         #On applique le test de Lucas-Lehmer au nombre n choisi au début de ce programme

('Le nombre de mersenne',
...
```

Figure 6: 1ère partie du programme



```
..., 'n est pas un nombre premier')
```

Figure 7: 2ème partie du programme

3.1.3 Analyse de la rapidité de l'algorithme

Dans cette sous-section tout comme dans les sous-sections du même nom dans les autres sections, nous allons en premier lieu présenter un tableau qui décrit en gros le nombre considéré et le temps de calcul mis par notre algorithme pour aficher le résultat. Nous allons ensuite expliquer et interpréter en détail le tableau. Enfin, nous fournirons encore d'autres tableaux qui donneront d'autres types de renseignements.

n	Mesure 1	Mesure 2	Moyenne
11213	0.85 s	0.34 s	0.595 s, <1s
21701	3.05 s	3.53 s	3.29 s
44497	16.34s	13.44 s	14.89 s
110503	2 min 23.31 s	2 min 25.88 s	2 min 24.595 s
216091			>15 min

Figure 8: 1er tableau expérimental sur le test de Lucas-Lehmer

Dans ce premier tableau, nous avons pris des nombres n qui donnent des nombres premiers de Mersenne. On voit bien que le temps d'exÃ©cution croît exponentiellement, car si on prend par exemple $n = 110503$, on voit que ceci est environ trois fois plus grand que $n = 44497$, mais le temps que l'ordinateur prend pour nous donner le résultat est environ 15 fois plus.

On a eu des difficultés à mesurer le temps pour $n = 11213$, car le temps d'exécution n'est même pas d'une seconde,ce qui engendre facilement des erreurs de mesure de l'ordre d'une seconde le temps que nous réagissions.

On voulait aussi faire l'expérience pour $n = 216091$, mais comme après 15 minutes l'ordinateur n'avait toujours pas trouvé la solution on a arrêté l'expérience.

n	Mesure 1	Mesure 2	Moyenne1
16457	1.91 s	1.8 s	1.855 s
33099	6.83 s	7.08 s	6.955 s
77500	51.01 s	51.33 s	51.17 s
163297	9 min 34.59 s	9 min 31.72 s	9 min 33.155 s

Figure 9: 2ème tableau expérimental sur le test de Lucas-Lehmer

Pour ce deuxième tableau nous avons pris des n qui se trouvent entre les n du premier tableau, car pour $n = 11213$, le temps d'exécution était trop court et pour $n = 216091$ le temps d'exécution était trop long.

On voit bien que si on prend un n du deuxième tableau qui se trouve entre deux n du premier tableau, le temps d'exÃ©cution va rester entre le temps pris par l'ordinateur pour les n du premier tableau. Mais ceci est facile à expliquer car si on regarde notre programme sur **SAGE**, on voit que la boucle va être toujours exécutée si le nombre n donne un nombre premier de Mersenne ou pas. La seule chose qui change dans l'exécution est la condition *if*. Mais comme le *if* ne contient pas de boucle, il prend un temps constant pour être exécuté.

On a arrêté à $n = 163297$ car l'ordinateur a pris $9min33.155\ s$ pour trouver la solution et en plus le nombre de Mersenne que nous avons trouvé n'était pas entièrement affiché. **SAGE** a laissé le message suivant :

WARNING: Output: 49216 truncated by MAX_STDOUT_SIZE to 40000

3.2 Le test de Miller-Rabin

Le plus grand nombre premier que nous avons trouvé à l'aide du test de Miller-Rabin dans un laps de temps correct est $2^{44497} - 1$.

3.2.1 Code source

```
#Test de Miller-Rabin:

from random import randint
n= 1000000016531         #n doit être un entier impair plus grand que 3.
k=1         #k doit être un entier plus grand ou égal à 1.

def temoin_de_miller(n,a):         #On définit une fonction qui vérifie si on a un témoin de Miller ou pas
    s=(n-1).valuation(2)
    d=(n-1) // 2^s
            #pour les deux premières lignes dans cette fonction, on calcule s et d de l'équation n-1=(2^s)*d
    x= power_mod(a,d,n)         #on prend x=a^d (mod n)
    if x==1 or x == n-1:         #Si x vaut 1 ou (n-1),alors on n'a pas un témoin de Miller
         return False
    while s>1:         #Aussi longtemps que s>1, on effectue le calcul ci-après
        x = power_mod(x,2,n)
        if x==n-1:         #Si x vaut (n-1),alors on n'a pas de témoin de Miller
            return False
        s = s-1
    return True         #Sinon, on a un témoin de Miller

def miller_rabin(n,k):
    for i in [1..k]:         #on veut repèter k fois
        a=randint(2,n-2)         #a est un entier aléatoire entre 2 et n-2.
        if temoin_de_miller(n,a)==True:         #Si a est un témoin de Miller pour n, alors n est composé
            return n,'est compose'
        else:
            for p in primes_first_n(k): #Sinon,on va chercher tous les nombres premiers plus petit que k
```

Figure 10: Code source de l'algorithme de Miller-Rabin (1ère partie)

```
            for p in primes_first_n(k): #Sinon,on va chercher tous les nombres premiers plus petit que k
                if Mod(n,p)==0:
                    if n==p:
                        return n,'est premier'
            return n,'est probablement premier si k est suffisamment grand'

miller_rabin(n,k)
```

Figure 11: Code source de l'algorithme de Miller-Rabin (2ème partie)

3.2.2 Exemples

Cas où le nombre préalablement choisi est probablement premier :

```
#Test de Miller-Rabin:

from random import randint
n= 1000000016531         #n doit être un entier impair plus grand que 3.
k=50000         #k doit être un entier plus grand ou égal à 1.

def temoin_de_miller(n,a):         #On définit une fonction qui vérifie si on a un témoin de Miller ou pas
    s=(n-1).valuation(2)
    d=(n-1) // 2^s
            #pour les deux premières lignes dans cette fonction, on calcule s et d de l'équation n-1=(2^s)*d
    x= power_mod(a,d,n)         #on prend x=a^d (mod n)
    if x==1 or x == n-1:         #Si x vaut 1 ou (n-1),alors on n'a pas un témoin de Miller
         return False
    while s>1:         #Aussi longtemps que s>1, on effectue le calcul ci-après
        x = power_mod(x,2,n)
        if x==n-1:         #Si x vaut (n-1),alors on n'a pas de témoin de Miller
            return False
        s = s-1
    return True         #Sinon, on a un témoin de Miller

def miller_rabin(n,k):
    for i in [1..k]:         #on veut repèter k fois
        a=randint(2,n-2)         #a est un entier aléatoire entre 2 et n-2.
        if temoin_de_miller(n,a)==True:         #Si a est un témoin de Miller pour n, alors n est composé
            return n,'est compose'
        else:
            for p in primes_first_n(k): #Sinon,on va chercher tous les nombres premiers plus petit que k
```

Figure 12: 1ère partie du programme

```
            for p in primes_first_n(k): #Sinon,on va chercher tous les nombres premiers plus petit que k
                if Mod(n,p)==0:
                    if n==p:
                        return n,'est premier'
            return n,'est probablement premier si k est suffisamment grand'

miller_rabin(n,k)

    (1000000016531, 'est probablement premier si k est suffisamment grand')
```

Figure 13: 2ème partie du programme

Cas où le nombre préalablement choisi est composé :

```
#Test de Miller-Rabin:

from random import randint
n= 57933         #n doit être un entier impair plus grand que 3.
k=100         #k doit être un entier plus grand ou égal à 1.

def temoin_de_miller(n,a):         #On définit une fonction qui vérifie si on a un témoin de Miller ou pas
    s=(n-1).valuation(2)
    d=(n-1) // 2^s
            #pour les deux premières lignes dans cette fonction, on calcule s et d de l'équation n-1=(2^s)*d
    x= power_mod(a,d,n)         #on prend x=a^d (mod n)
    if x==1 or x == n-1:         #Si x vaut 1 ou (n-1),alors on n'a pas un témoin de Miller
         return False
    while s>1:         #Aussi longtemps que s>1, on effectue le calcul ci-après
        x = power_mod(x,2,n)
        if x==n-1:         #Si x vaut (n-1),alors on n'a pas de témoin de Miller
            return False
        s = s-1
    return True         #Sinon, on a un témoin de Miller

def miller_rabin(n,k):
    for i in [1..k]:         #on veut repèter k fois
        a=randint(2,n-2)         #a est un entier aléatoire entre 2 et n-2.
        if temoin_de_miller(n,a)==True:         #Si a est un témoin de Miller pour n, alors n est composé
            return n,'est compose'
        else:
            for p in primes_first_n(k): #Sinon,on va chercher tous les nombres premiers plus petit que k
```

Figure 14: 1ère partie du programme

```
        for p in primes_first_n(k): #Sinon,on va chercher tous les nombres premiers plus petit que k
            if Mod(n,p)==0:
                if n==p:
                    return n,'est premier'
        return n,'est probablement premier si k est suffisamment grand'

miller_rabin(n,k)

(57933, 'est compose')
```

Figure 15: 2ème partie du programme

Nous allons montrer dans l'exemple qui suit l'importance et le rôle de la variable k dans notre programme.

```
#Test de Miller-Rabin:

from random import randint
n= 1117         #n doit être un entier impair plus grand que 3.
k=186         #k doit être un entier plus grand ou égal à 1.

def temoin_de_miller(n,a):         #On définit une fonction qui vérifie si on a un témoin de Miller ou pas
    s=(n-1).valuation(2)
    d=(n-1) // 2^s
            #pour les deux premières lignes dans cette fonction, on calcule s et d de l'équation n-1=(2^s)*d
    x= power_mod(a,d,n)         #on prend x=a^d (mod n)
    if x==1 or x == n-1:         #Si x vaut 1 ou (n-1),alors on n'a pas un témoin de Miller
         return False
    while s>1:         #Aussi longtemps que s>1, on effectue le calcul ci-après
        x = power_mod(x,2,n)
        if x==n-1:         #Si x vaut (n-1),alors on n'a pas de témoin de Miller
            return False
        s = s-1
    return True         #Sinon, on a un témoin de Miller

def miller_rabin(n,k):
    for i in [1..k]:         #on veut repèter k fois
        a=randint(2,n-2)         #a est un entier aléatoire entre 2 et n-2.
        if temoin_de_miller(n,a)==True:         #Si a est un témoin de Miller pour n, alors n est composé
            return n,'est compose'
        else:
```

Figure 16: 1117 est probablement premier si $k \leq 186$ (1ère partie)

```
            return n,'est compose'
        else:
            for p in primes_first_n(k): #Sinon,on va chercher tous les nombres premiers plus petit que k
                if Mod(n,p)==0:
                    if n==p:
                        return n,'est premier'
            return n,'est probablement premier si k est suffisamment grand'

miller_rabin(n,k)

(1117, 'est probablement premier si k est suffisamment grand')
```

Figure 17: 1117 est probablement premier si $k \leq 186$ (2ème partie)

```
#Test de Miller-Rabin:

from random import randint
n= 1117         #n doit être un entier impair plus grand que 3.
k=187         #k doit être un entier plus grand ou égal à 1.

def temoin_de_miller(n,a):         #On définit une fonction qui vérifie si on a un témoin de Miller ou pas
    s=(n-1).valuation(2)
    d=(n-1) // 2^s
            #pour les deux premières lignes dans cette fonction, on calcule s et d de l'équation n-1=(2^s)*d
    x= power_mod(a,d,n)         #on prend x=a^d (mod n)
    if x==1 or x == n-1:         #Si x vaut 1 ou (n-1),alors on n'a pas un témoin de Miller
         return False
    while s>1:         #Aussi longtemps que s>1, on effectue le calcul ci-après
        x = power_mod(x,2,n)
        if x==n-1:         #Si x vaut (n-1),alors on n'a pas de témoin de Miller
            return False
        s = s-1
    return True         #Sinon, on a un témoin de Miller

def miller_rabin(n,k):
    for i in [1..k]:         #on veut repèter k fois
        a=randint(2,n-2)         #a est un entier aléatoire entre 2 et n-2.
        if temoin_de_miller(n,a)==True:         #Si a est un témoin de Miller pour n, alors n est composé
            return n,'est compose'
        else:
```

Figure 18: 1117 est premier si $k \geq 187$ (1ère partie)

```
        for p in primes_first_n(k): #Sinon,on va chercher tous les nombres premiers plus petit que k
            if Mod(n,p)==0:
                if n==p:
                    return n,'est premier'
        return n,'est probablement premier si k est suffisamment grand'

miller_rabin(n,k)

    (1117, 'est premier')
```

Figure 19: 1117 est premier si $k \geq 187$ (2ème partie)

3.2.3 Analyse de la rapidité de l'algorithme

Comme nous l'avons dit dans l'introduction de la sous-section 3.1.3, nous allons procd́er ici de la même manière que dans la sous-section mentionée.

n	k	Mesure 1	Mesure 2	Moyenne
2^{21701}	100	4.87 s	4.91 s	4.89 s
2^{21701}	10^6	5.4 s	5.28 s	5.34 s
2^{44497}	100	26.58 s	26.63 s	26.605 s
2^{44497}	10^6	27.45 s	27.98 s	27.715 s

Figure 20: 1er tableau expérimental sur le test de Miller-Rabin

Dans ce premier tableau on voulait tester des nombres composés. On a essayé des nombres dans les $10^5 0$, mais le temps d'exécution était toujours inférieur à une seconde. Donc on a décidé de prendre des nombres de la forme des nombres de Mersenne augmentés de 1 (c'est-à-dire de la forme $n = 2^x$ où x est un entier naturel).

On a donc pris les nombres $n = 221701$ et 244497 et pour k on a aussi pris deux k différents,à savoir $k = 100$ et $k = 106$.

On a conclu que si on prend un plus grand nombre, l'ordinateur prend plus de temps à le calculer (ce qui est évident). Mais on a aussi vu que si on augmente la valeur de k, le temps d'exécution augmente aussi.

n	k	Mesure 1	Mesure 2	Moyenne
2^{11213}-1	1000	1.36 s	1.24 s	1.3 s
2^{21701}-1	1000	5.38 s	5.28 s	5.33 s
2^{44497}-1	1000	26.85 s	26.3 s	26.575 s
2^{110503}-1	1000	4 min 9.93 s	4 min 10.06 s	4 min 9.995 s

Figure 21: 2ème tableau expérimental sur le test de Miller-Rabin

Dans ce deuxième tableau on a pris les nombres de Mersenne que l'on a testé dans le test de Lucas-Lehmer avec $k = 1000$. Donc la réponse du programme était toujours que notre n est probablement premier si k est suffisamment grand.

Notre but dans ce tableau était de comparer le test de Lucas-Lehmer avec le test de Miller-Rabin. On a pu constater que le test de Lucas-Lehmer est beaucoup plus rapide et nous donne une information exacte tandis que le test de Miller-Rabin nous dit seulement que c'est probablement premier. On pourrait aussi augmenter la valeur de k pour obtenir une réponse exacte, mais si on faisait cela, le temps d'exécution augmenterait aussi. En guise de conclusion, nous avons que le test de Lucas-Lehmer est beaucoup plus efficace pour trouver les nombres premiers de Mersenne (et le nombres premiers en général) que le test de Miller-Rabin.

n	k	Mesure 1	Mesure 2	Moyenne
178439	**100000**	5.33 s	5.03 s	5.18 s
558179	**100000**	13.52 s	13.81 s	13.665 s
912727	**100000**	20.78 s	20.59 s	20.685 s
1297333	**100000**	28.54 s	28.56 s	28.55 s

Figure 22: 3ème tableau expérimental sur le test de Miller-Rabin

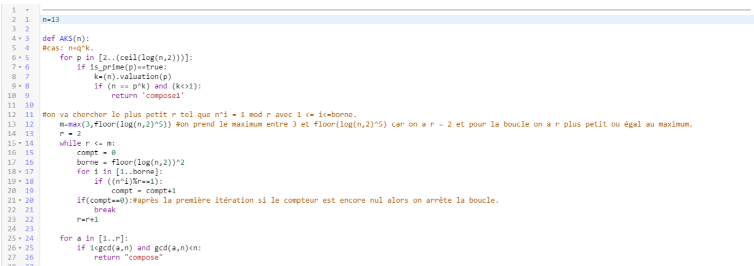
Dans ce dernier tableau, notre but était de mesurer le temps d'exécution nécessaire pour avoir un résultat précis. Donc pour chaque nombre on a pris le même k pour ne pas avoir d'erreur de mesure à cause du k. Dans chaque exécution on a eu la réponse : n *est premier*

La distance entre chaque nombre n est toujours d'environ 400000. Si on regarde le temps d'exécution pour chaque n, on voit que le temps augmente toujours de plus ou moins 8 s. Donc le temps semble être proportionnel au nombre n.

3.3 Le test AKS

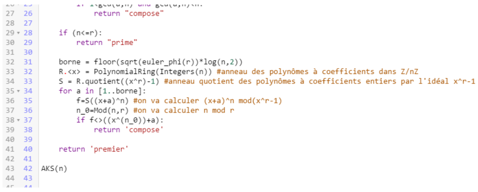
Le plus grand nombre premier que nous avons obtenu pendant un lap de temps correct est 178439 .

3.3.1 Code source



```

n=13

def AKS(n):
#cas: n=q^k.
    for p in [2..(ceil(log(n,2)))]:
        if is_prime(p)==true:
            k=(n).valuation(p)
            if (n == p^k) and (k<>1):
                return 'compose1'

#on va chercher le plus petit r tel que n^i = 1 mod r avec 1 <= i<=borne.
    m=max(3,floor(log(n,2)^5)) #on prend le maximum entre 3 et floor(log(n,2)^5) car on a r = 2 et pour la boucle on a r plus petit ou égal au maximum.
    r = 2
    while r <= m:
        compt = 0
        borne = floor(log(n,2))^2
        for i in [1..borne]:
            if ((n^i)%r==1):
                compt = compt+1
        if(compt==0):#après la première itération si le compteur est encore nul alors on arrête la boucle.
            break
        r=r+1

    for a in [1..r]:
        if 1<gcd(a,n) and gcd(a,n)<n:
            return "compose"
```

Figure 23: Code source de l'algorithme AKS (1ère partie)

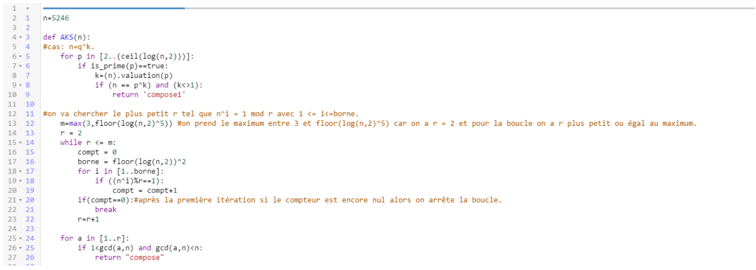


```
            return "compose"

    if (n<=r):
        return "prime"

    borne = floor(sqrt(euler_phi(r))*log(n,2))
    R.<x> = PolynomialRing(Integers(n)) #anneau des polynômes à coefficients dans Z/nZ
    S = R.quotient((x^r)-1) #anneau quotient des polynômes à coefficients entiers par l'idéal x^r-1
    for a in [1..borne]:
        f=S((x+a)^n) #on va calculer (x+a)^n mod(x^r-1)
        n_0=Mod(n,r) #on va calculer n mod r
        if f<>((x^(n_0))+a):
            return 'compose'

    return 'premier'

AKS(n)
```

Figure 24: Code source de l'algorithme AKS (2ème partie)

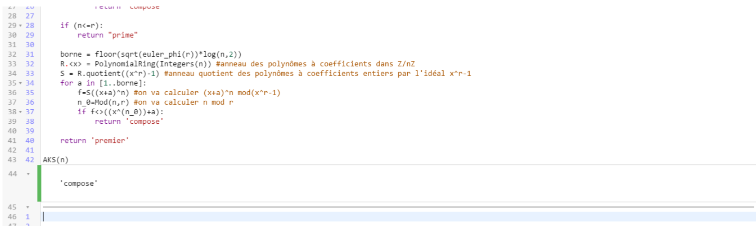
3.3.2 Exemples

Cas où le nombre choisi au départ est composé :



```
n=5246

def AKS(n):
#cas: n=q^k.
    for p in [2..(ceil(log(n,2)))]:
        if is_prime(p)==true:
            k=(n).valuation(p)
            if (n == p^k) and (k<>1):
                return 'compose1'

#on va chercher le plus petit r tel que n^i = 1 mod r avec 1 <= i<=borne.
    m=max(3,floor(log(n,2)^5)) #on prend le maximum entre 3 et floor(log(n,2)^5) car on a r = 2 et pour la boucle on a r plus petit ou égal au maximum.
    r = 2
    while r <= m:
        compt = 0
        borne = floor(log(n,2))^2
        for i in [1..borne]:
            if ((n^i)%r==1):
                compt = compt+1
        if(compt==0):#après la première itération si le compteur est encore nul alors on arrête la boucle.
            break
        r=r+1

    for a in [1..r]:
        if 1<gcd(a,n) and gcd(a,n)<n:
            return "compose"
```

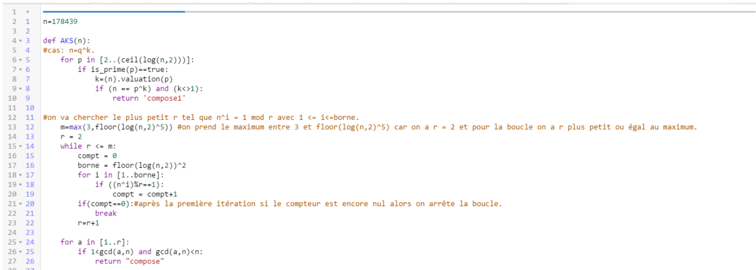
Figure 25: 1ère partie du programme



```
    if (n<=r):
        return "prime"

    borne = floor(sqrt(euler_phi(r))*log(n,2))
    R.<x> = PolynomialRing(Integers(n)) #anneau des polynômes à coefficients dans Z/nZ
    S = R.quotient((x^r)-1) #anneau quotient des polynômes à coefficients entiers par l'idéal x^r-1
    for a in [1..borne]:
        f=S((x+a)^n) #on va calculer (x+a)^n mod(x^r-1)
        n_0=Mod(n,r) #on va calculer n mod r
        if f<>((x^(n_0))+a):
            return 'compose'

    return 'premier'
AKS(n)

'compose'
```

Figure 26: 2ème partie du programme

Cas où le nombre préalablement choisi est premier :



```
n=178439

def AKS(n):
#cas: n=q^k.
    for p in [2..(ceil(log(n,2)))]:
        if is_prime(p)==true:
            k=(n).valuation(p)
            if (n == p^k) and (k<>1):
                return 'compose1'

#on va chercher le plus petit r tel que n^i = 1 mod r avec 1 <= i<=borne.
    m=max(3,floor(log(n,2)^5)) #on prend le maximum entre 3 et floor(log(n,2)^5) car on a r = 2 et pour la boucle on a r plus petit ou égal au maximum.
    r = 2
    while r <= m:
        compt = 0
        borne = floor(log(n,2))^2
        for i in [1..borne]:
            if ((n^i)%r==1):
                compt = compt+1
        if(compt==0):#après la première itération si le compteur est encore nul alors on arrête la boucle.
            break
        r=r+1

    for a in [1..r]:
        if 1<gcd(a,n) and gcd(a,n)<n:
            return "compose"
```

Figure 27: 1ère partie du programme

```
    if (n<=r):
        return "prime"

    borne = floor(sqrt(euler_phi(r))*log(n,2))
    R.<x> = PolynomialRing(Integers(n)) #anneau des polynômes à coefficients dans Z/nZ
    S = R.quotient((x^r)-1) #anneau quotient des polynômes à coefficients entiers par l'idéal x^r-1
    for a in [1..borne]:
        f=S((x+a)^n) #on va calculer (x+a)^n mod(x^r-1)
        n_0=Mod(n,r) #on va calculer n mod r
        if f<>((x^(n_0))+a):
            return 'compose'

    return 'premier'

AKS(n)

    'premier'
```

Figure 28: 2ème partie du programme

3.3.3 Analyse de la rapidité de l'algorithme

Comme nous l'avons dit dans l'introduction de la sous-section 3.1.3, nous allons procd́er ici de la même manière que dans la sous-section mentionée.

n	Mesure 1	Mesure 2	Moyenne
50021	3.28 s	2.98 s	3.13 s
178439	13.51 s	14.35 s	13.39 s
558179	1 min 9.24 s	1 min 8.71 s	1 min 8.975 s
912727	1 min 54.39 s	1 min 52.56 s	1 min 53.475 s
1297333	3 min 23.34 s	3 min 24.17s	3 min 23.755 s

Figure 29: 1er tableau expérimental sur le test AKS

Dans ce premier tableau, nous avons pris pour n les mêmes nombres que dans le test de Miller-Rabin dans le dernier tableau.

Si on compare les deux tableaux en question, on voit bien que notre programme AKS prend beaucoup plus de temps pour donner une solution que le test de Miller-Rabin. On pourrait aussi dire que c'est parce que le test de Miller-Rabin ne donne pas une solution exacte, mais dans notre cas on a pris le k suffisamment grand pour avoir une solution exacte. Donc on peut conclure que si on veut trouver un nombre premier, le test de Miller-Rabin est beaucoup plus efficace.

Mais par contre, si le nombre qu'on veut tester est composé, alors le test de primalité AKS s'av ère plus efficace. On a essayé de mesurer le temps d'exécution pour des nombres composé, et même pour les nombres que l'on a utilisé pour le premier tableau du test de Miller-Rabin, mais l'ordinateur nous donnait des solutions presque immédiatement.

Du coup, on a décidé de tester des nombres aléatoires de la forme suivante :

n	Mesure 1	Mesure 2	Moyenne
$\mathbf{2^{10000000}}$	5.01 s	4.09 s	4.55 s
$\mathbf{2^{12000000}}$	13.94 s	13.67 s	13.805 s
$\mathbf{2^{14000000}}$	20.04 s	20.75 s	20.395 s
$\mathbf{2^{16000000}}$	33.21 s	34.03 s	33.62 s
$\mathbf{2^{18000000}}$	45.75 s	42.31 s	44.03 s

Figure 30: 2ème tableau expérimental sur le test AKS

On voit bien que pour des nombres composés, le programme AKS est rapide, même plus vite que celui de Miller-Rabin.

On a aussi essayé avec des nombres n de la forme $n = p^k$, où p est un entier plus grand que 0 et k un entier plus grand que 1, mais les résultats étaient encore une fois presque immédiats.

4 Conclusion

Après avoir longuement étudié avec précision les 3 tests de primalité, nous pouvons dès à présent les comparer entre eux en énumérant leurs points forts et points faibles.

Tout d'abord, en ce qui concerne la rapidité de l'algorithme, le test de primalité de Lucas-Lehmer est de loin le plus rapide des 3. Vient ensuite le test de Miller-Rabin et enfin, loin derrière, le tes AKS.

Vient ensuite l'exactitude des résultats du programme ; les algorithmes de Lucas-Lehmer et AKS sont tous les 2 aussi efficace dans ce domaine étant donné qu'ils disent avec certitude si un nombre donné est premier ou pas. Par contre, le test de Miller-Rabin ne fournit en général qu'une probabilité à un nombre donné d'être premier tant que l'on pas choisi un k suffisament grand (pour la signification du k,référez vous au paragraphe 3.1.1), ce qui place ce test en dernière position parmi les 3.

Pour terminer les comparaisons, en ce qui concerne la quantité de nombres premiers trouvables à l'aide de ces 3 algorithmes, le test de Lucas-Lehmer est le plus désavantageux. En effet, là ou les tests de primalité de Miller-Rabin et AKS fournissent tous les nombres premiers possibles, le test de Lucas-Lehmer ne fournit que les nombres premiers de Mersenne qui sont les nombres d'une suite particulière.

À partir de ces comparaisons, l'on constate que suivants les critères que l'on prend en compte, certains tests de primalités sont plus à privilégier que d'autres. Ainsi par exemple, si une personne veut réaliser un test de primalité qui trouve très rapidement avec exactitude si un nombre donné, aussi grand soit-il, est premier, alors dans ce cas, le test de Lucas-Lehmer sera largement recommandé. C'est d'ailleurs avec ce test que des mathématiciens sont parvenus, en janvier 2016, à trouver le plus grand nombre premier connu à ce jour, à savoir $2^74207281 - 1$.

Ce qu'il faut donc retenir de tout cela, c'est qu'il y a des tests de primalité pour tous les goûts et qu'il est de nos jours encore difficile de recommander un test de primalité en particulier malgré que certains tests comme celui de Lucas-Lehmer soient extrêmement efficace. Tout dépendra simplement de l'objectif recherché de la personne par l'utilisations de ces tests de primalité.

5 Bibliographie

- Entretiens personnels avec Prof. Wiese du 8 et 22 mars,du 27 avril et du 18 mai 2017
- Paulo Ribenboim - The little book of bigger primes (Second edition of Springer edition)
- https://fr.wikipedia.org/wiki/Test_de_primalité_de_Lucas-Lehmer _pour_les_nombres_de_Mersenne
- https://fr.wikipedia.org/wiki/Édouard_Lucas
- https://fr.wikipedia.org/wiki/Derrick_Lehmer
- https://fr.wikipedia.org/wiki/Test_de_primalité_de_Lucas-Lehmer
- https://fr.wikipedia.org/wiki/Nombre_de_Mersenne_premier
- Notes de cours de "Théorie des nombres et applications à la cryptographie" du Prof. Wiese
- https://fr.wikipedia.org/wiki/Test_de_primalité_AKS
- https://fr.wikipedia.org/wiki/Gary_L._Miller
- https://fr.wikipedia.org/wiki/Michael_Rabin
- https://fr.wikipedia.org/wiki/Test_de_primalité_de_Miller-Rabin
- https://fr.wikipedia.org/wiki/Congruence_sur_les_entiers
- https://fr.wikipedia.org/wiki/Nombre_pseudo-premier
- https://fr.wikipedia.org/wiki/Nombre_composé
- http://www.trigofacile.com/maths/curiosite/primarite/aks/pdf/algorithme-aks.pdf
- https://de.wikipedia.org/wiki/AKS-Primzahltest
- https://en.wikipedia.org/wiki/Miller-Rabin_primality_test
- https://www.math.univ-toulouse.fr/~hallouin/Documents/Primalite.pdf
- https://fr.wikipedia.org/wiki/Entier_sans_facteur_carré
- https://fr.wikipedia.org/wiki/Loi_de_réciprocité_quadratique
- https://fr.wikipedia.org/wiki/Théorème_des_restes_chinois

- https://perso.univ-rennes1.fr/romain.basson/pdf/Crypto.pdf
- http://compoasso.free.fr/primelistweb/page/prime/liste_online.php
- http://www.bibmath.net/dossiers/index.php?action=affiche&quoi=dossier1/page4.html
- http://math.uni.lu/eml/projects/reports/Bachelor-Thesis-Notarnicola-Luca.pdf
- https://fr.wikipedia.org/wiki/Indicatrice_d'Euler