Rapport

Sonny Klotz - Idir Hamad - Younes Benyamna - Malek Zemni

Projet M1 Informatique
Primalité

20/04/2018

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. L'efficacité de ces algorithmes est particulièrement liée au cryptosystème utilisé.

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

Dans la première partie de ce document, on présentera l'architecture de notre application, illustrée par un organigramme.
Ensuite, on parlera des principaux cryptosystèmes faisant appel à des tests de primalité.
La troisième partie traitera des différents algorithmes de tests de primalité implémentés.
Les mesures de performance et le comparatif des tests de primalités seront détaillés dans la quatrième partie.
Finalement, on établira un bilan technique de notre projet, quant à l'application, à l'organisation interne au sein du groupe et aux coûts.

1 Architecture de l'application

1.1 Organigramme et données échangées

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

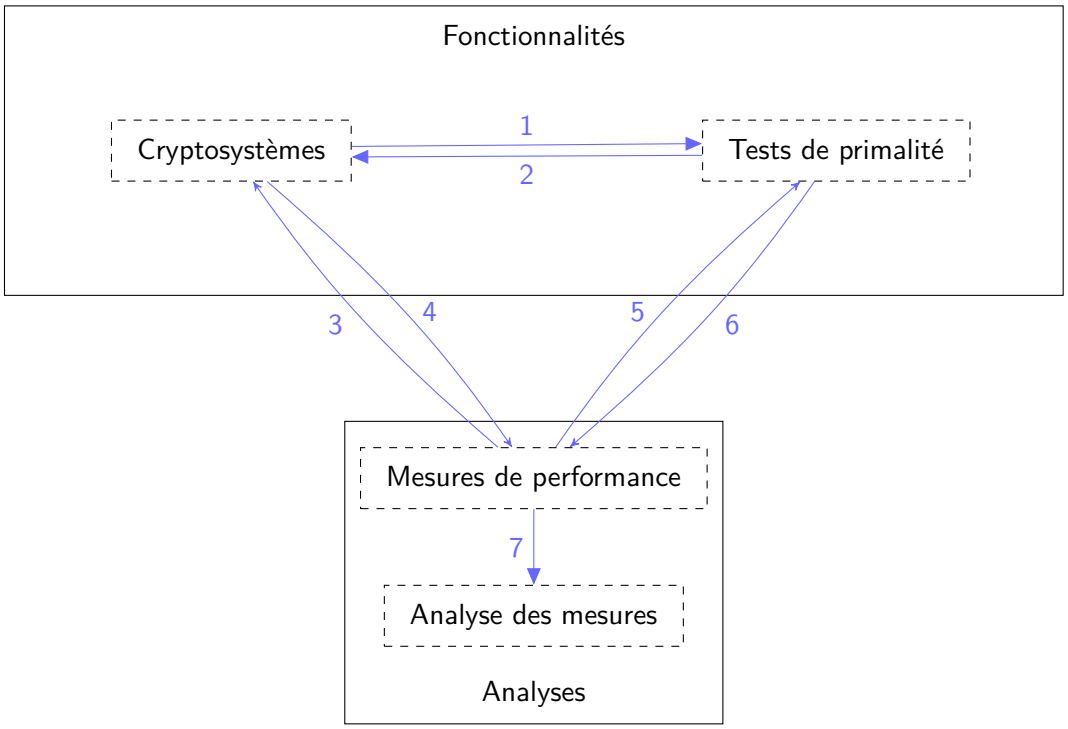

FIGURE 1 – Organigramme des différents modules de l'application

Notes :
(1) Nombre entier à tester (primalité)
(2) Réponse sur la primalité (0 composé, 1 premier, 2 pseudo-premier)
(3) Message à chiffrer
(4) Chiffré du message
(5) Nombre entier à tester (primalité)
(6) Réponse sur la primalité (0 composé, 1 premier, 2 pseudo-premier)
(7) Données collectées des différentes mesures de performance

1.2 Fonctionnalités des modules

Package Fonctionnalités

1. Module Cryptosystèmes : implémentation de cryptosystèmes ayant recours à des nombres premiers (***RSA***) et des générateurs de nombres premiers.
2. Module Test de primalité : implémentation de différents algorithmes de tests de primalité qui feront l'objet d'une étude comparative par la suite :
   — Test naïf

— Test de Fermat

— Test de Miller-Rabin

— Test de Solovay-Strassen

— AKS

3. Module Exemples : exemples d'utilisation des fonctionnalités implémentées (cryptosystèmes et tests de primalité), permet au package Fonctionnalités d'être utilisé comme une API indépendante de l'application finale.

Package Analyses

1. Module Mesures de performance : mesures des performances des différents tests de primalité implémentés selon les valeurs données en entrées et les cryptosystèmes qui les utilisent.
2. Module Analyse des mesures :

— Analyse comparative des différentes mesures calculées par le module Mesures de performance.

— Produit final de l'application.

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.

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 des 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 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 :

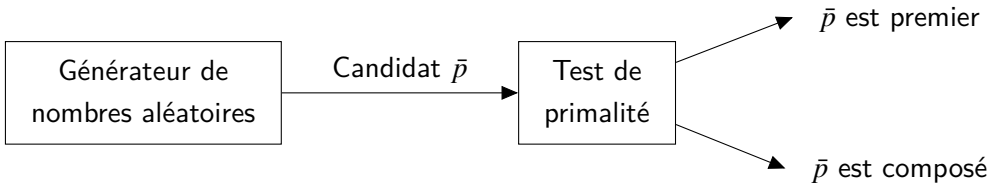

FIGURE 2 – Processus de génération des nombres premiers

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 :

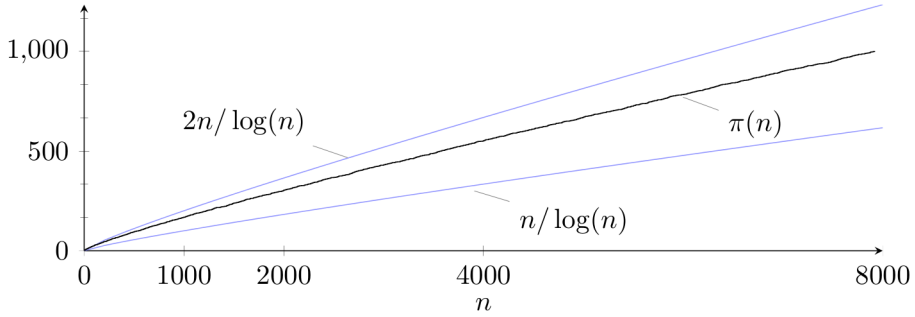

FIGURE 3 – La fonction $\pi(n)$ pour les 1000 premiers nombres premiers.

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
Crible d'Eratosthène	-240	Déterministe
Fermat	1640	Probabiliste
Wilson	1773	Déterministe
Miller-Rabin	1976	Probabiliste
Solovay-Strassen		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, sa preuve, ainsi que son implémentation.

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 Wilson*** est basé sur le théorème suivant :

Théorème 2 (Théorème de Wilson)**.** *un entier $n > 1$ est un nombre premier si et seulement si*

$$(n-1)! \equiv -1 \pmod{n}$$

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)**.** *si p est un nombre premier, alors pour tout nombre entier a premier avec p*

$$a^{p-1} \equiv 1 \pmod{p}$$

4.3.1 Algorithme et preuve

Le théorème de Fermat 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, si pour un entier a premier avec n :

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

— $a^{n-1} \equiv 1 \pmod{n}$, on ne peut pas conclure avec certitude que n est premier puisque la réciproque du théorème de Fermat est fausse. Un nombre n vérifiant cette équation peut être premier, mais aussi composé, dans ce cas n est dit ***pseudo-premier*** *de base* a.

Les nombres pseudo-premiers sont relativement rares. On peut donc envisager d'adopter ce critère pour un test probabiliste de primalité, qui est le test de Fermat.

Algorithme 3 : Test de Fermat

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

En effet, l'entier k correspond ici au nombre de fois que ce test sera répété. 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.

4.3.2 Complexité

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.

4.5 Test de Solovay-Strassen

4.6 AKS

5 Mesures de performance et comparatifs

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

5.1 Premiers tests déterministes

Algorithmes déterministes - Algorithmes probabilistes Les tests vus dans les parties précédentes sont des exemples d'algorithmes déterministes, dont la sortie est exacte avec probabilité de 1. Leur

complexité est cependant trop élevée. En pratique, on utilise des algorithmes beaucoup plus efficaces pour tester la primalité d'un nombre. Il s'agit des algorithmes probabilistes. Ces algorithmes 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 un certain nombre de fois.

5.2 Tests probabilistes

5.3 Tests déterministes rapides : AKS

5.4 Test générique

6 Bilan technique du projet

Notre produit final, c'est à dire l'application, se comporte comme prévu : l'application est fonctionnelle, la liaison entre ses différents modules réussit bien et les différentes fonctionnalités fournissent le résultat attendu.

6.1 Problèmes rencontrés

Problèmes résolus :

Lors de la réalisation de l'application, on a été confrontés à plusieurs problèmes et points délicats, principalement des verrous techniques, qui ont perturbé le bon déroulement de notre travail :

Problèmes non résolus :

Certains problèmes rencontrés n'ont pas été entièrement résolus. Ces problèmes ne sont pas déterminants pour l'acceptabilité de notre produit.

6.2 Organisation interne du groupe

Assignation des modules pour chaque membre du groupe : Cette répartition a été parfaitement respectée. Elle nous a permis de travailler efficacement et assez indépendamment, ce qui prouve que l'assignation des modules a été judicieusement faite. Nous sommes également restés en contact pendant toute la phase de développement pour s'entraider pour la prise en main des nouveaux outils.

6.3 Coûts

Ce tableau indique les coûts estimés et les coûts finaux, en nombre de lignes de code et pour chaque module :

Conclusion