Rapport

Jean-Didier Pailleux - Maxence Joulin - Damien Thenot - Romain Robert - Robin Feron

Test de primalité

https://github.com/CHPS-M1-PRIME-NUMBERS/Prime_numbers

10/01/2018

Projet de Programmation numérique

Table des matières

1 Introduction **1**

2 État de l'art **1**

2.1 Tests de primalité 1

2.2 Nombres Hautement composés 3

3 Implémentation **3**

3.1 Organigramme 3

3.2 Langages de programmation 4

3.3 Outils 4

4 Analyse des résultats **5**

4.1 Lancement d'un test depuis l'execution en ligne de commande 5

4.2 Lancement d'un test depuis le script SHELL 6

4.3 Matériel et logiciels utilisés 6

4.4 Résultats 6

4.4.1 Tests de primalité sur un échantillon 6

4.4.2 Tests de primalité sur une plage de données 9

4.4.3 Hautement Composé 12

5 Bilan Technique du Projet **13**

6 Organisation interne du groupe **14**

7 Conclusion **14**

1 Introduction

Ce document est le compte-rendu de notre travail qui s'inscrit dans le cadre du module *Projet de Programmation numérique* du master *Calcul Haute Performance Simulation* de l'***UVSQ***. Le sujet de ce projet a été proposé par l'encadrant Sébastien Gougeaud.

Ce projet découle d'un thème qui est le test de primalité, et consiste à tester si un nombre est bien premier ou non. Un nombre premier est un entier qui admet uniquement deux diviseurs distincts et positifs (1 et lui même). Les nombres premiers prennent une place importante dans le domaine des mathématiques et ont des propriétés très utiles particulièrement dans le domaine de la cryptographie. La recherche de très grands nombres premiers est devenue de plus en plus importante et de nombreux tests de primalité ont pu émerger, voir évoluer, devenir plus performants et de plus en plus rapides.

Il existe actuellement deux types de tests de primalité, les déterministes qui permettent d'établir avec certitude le résultat et les tests probabilistes qui émettent un résultat non fiable avec une certaine probabilité d'erreur mais sont plus rapides que les tests déterministes.

L'objectif de ce projet consiste à implémenter plusieurs de ces tests et ainsi comparer leur vitesse d'exécution. Ceci pour l'exécution des tests de primalité pour un nombre premier, et pour une exécution sur les tests avec n nombres premiers. De plus un test probabiliste sera programmé pour ainsi comparer la fiabilité du résultat et le temps que nécessite de faire cet algorithme.
Une partie bonus nous a été proposée lors de notre entretien avec Monsieur Gougeaud qui consiste à implémenter une fonctionnalité pour déterminer si un nombre est hautement composé (c'est à dire que le nombre de ses diviseurs est supérieur strictement à tout les nombres inférieur à lui).

Dans la première partie de ce document, on présentera l'état de l'art de notre projet. Après cela, on mentionnera les différents outils et langages de programmation utilisés au cours de la phase d'implémentation dans une partie implémentation. Puis dans une autre partie l'analyse des résultats établis lors des tests du projet accompagné d'un comparatif de la performance des tests. Finalement, dans les deux dernières parties, on établira un bilan quant à l'organisation interne au sein du groupe et un bilan technique du projet qui mettra en avant les limites de chaque test de primalité.

2 État de l'art

2.1 Tests de primalité

Au cours de ce projet il nous a été demandé d'implémenter plusieurs algorithmes pour déterminer si un nombre est premier ou non. Voici les différents tests utilisés pour ce travail :

— Méthode naïve le **Crible d'Ératosthène**[1](Déterministe) :
Cette méthode est utilisée pour le memory bound. Cet algorithme va procéder à une élimination

dans une table d'entiers de 2 à N de tous les multiples de chaque entier présent dans ce tableau. En supprimant tous les multiples à la fin, seuls les entiers qui ne sont multiples d'aucun autre resteront et par conséquent ces nombres seront donc premiers. Le memory bound va donc être une liste des entiers restant à la fin de l'application du crible.

— Méthode naïve **l'algorithme d'Euclide**[2](Déterministe) :
Les premiers algorithmes naïfs sont très simples mais coûteux en temps de calcul. Un nombre est en effet premier si aucun nombre qui lui est inférieur le divise. Cet algorithme va donc essayer de diviser le nombre n dont on souhaite tester la primalité par tout ces nombres. Si l'un d'entre eux donne pour reste de la division 0, alors n n'est pas premier. Afin de limiter le nombre de calculs nécessaires on peut se limiter à tester uniquement tout les nombres inférieurs à $\sqrt{n}$. En effet, si un nombre a $> \sqrt{n}$ divise n, alors a x b = n où b est forcement plus petit que $\sqrt{n}$. Donc n aura été prouvé non premier par b avant d'essayer a.
On peut également utiliser l'algorithme d'Euclide qui permet de trouver de manière itérative le plus grand diviseur commun de deux nombres (PGCD). Or on dit que a et b sont premiers entre eux lorsque leur PGCD est égal à 1. Il suffit donc de prouver que chaque nombre inférieur à $\sqrt{n}$ a pour PGCD 1 avec n pour dire que n est premier.
Ces deux méthodes sont très simples à implémenter mais deviennent très rapidement coûteuses en temps de calcul pour des nombres très grands, s'approchant de 2^{64} par exemple.

— **Pocklington**[3](Déterministe) :
Le test de primalité de Pocklington-Lehmer utilise le théorème de Pocklington pour vérifier la primalité d'un nombre. L'algorithme consiste à utiliser une factorisation partielle du nombre N-1, où N est le nombre dont on souhaite tester la primalité. Cette factorisation partielle nous permet d'obtenir un nombre que l'on nommera A et qui doit remplir certaines conditions pour qu'on puisse appliquer le théorème. Nous avons déjà la factorisation de A qui nous servira pour la suite de l'exécution de l'algorithme. La suite consiste à tester les conditions requises sur tous les facteurs premiers de A, ces conditions sont de trouver un nombre a qui mis à la puissance N-1 modulo N donne 1 et que le plus grand diviseur commun entre $\frac{a^{(N-1)}}{p}$ et N soit égal à 1, avec p le facteur premier de A tester en ce moment. Si on a trouvé un a remplissant les conditions pour chaque p alors on peut dire que le nombre N est premier.

— **AKS** Version 2002[4][5](Déterministe) :
L'algorithme AKS est l'acronyme de ses créateurs (Agrawal-Kayal-Saxena). AKS est un algorithme déterministe, inconditionnel (il fonctionne peut importe l'entrée) et polynomiale c'est l'un des rares algorithmes à vérifier toutes ces propriétés. Cet Algorithme se base sur une généralisation du petit théorème de Fermat[4] qui est : "Pour tout entier $n \geq 2$ et tout entier a premier avec n, n est un nombre premier si et seulement si : $(X+a)^n \equiv X^n + a \ mod \ n$".

En ce qui concerne l'algorithme il est effectuée en 5 étapes[6]. Pour la version réalisée (2002),

la complexité est de l'ordre de $\log(n)^{12}$ cependant il existe beaucoup de variantes qui améliorent l'algorithme (exemple en 2005 une nouvelle variante d'AKS s'exécute en $\log(n)^6$ ce qui représente une belle optimisation comparée à la version initiale). La plupart des optimisations concernent l'étape 5 qui occupe une grande partie de l'exécution de l'algorithme.

— **Miller-Rabin**[7](Probabiliste) :
Miller-Rabin, basé sur le petit théorème de Fermat
$a^p - 1 \equiv 1 \pmod{p}$, consiste à tirer parti d'une équation ou d'un système d'équations qui soient vraies pour des valeurs premières et à regarder si elles sont toujours vraies pour un nombre dont nous voulons tester la primalité. L'algorithme Miller-Rabin prend en entrée un entier n dont on souhaite vérifier la primalité et un entier k qui représente le nombre d'itération du test de Miller-Rabin, plus k est élevé et plus l'analyse est précise (moins de chance d'erreurs). Il fait partie des tests probabiliste et est donc plus rapide que les tests déterministe.

2.2 Nombres Hautement composés

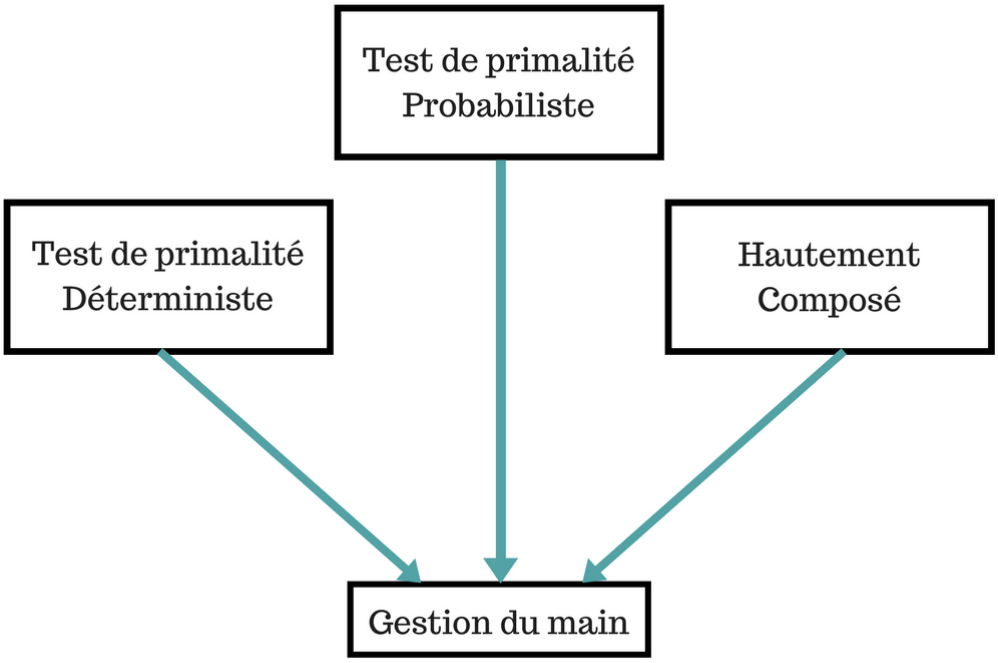
Deux algorithmes ont été étudiées pour les nombres hautement composés[8]. Le premier est une méthode naïve qui consiste à calculer le nombre de diviseur d'un nombre n et de comparer ce nombre avec le nombre de diviseurs pour tout entier k compris entre 1 et $n-1$.
Le deuxième algorithme utilise la propriété sur la forme qu'un nombre hautement composé doit avoir. Un nombre hautement composé possède des facteurs premiers qui sont les plus petits possible. Si l'on doit prendre en considération que la décomposition d'un entier $n > 1$ en produit de facteurs premiers comme suit $n = p_1^{c_1} * p_2^{c_2} * ... * p_k^{c_k}$ où $p_1 = 2 < p_2 = 3 < \ldots < p_k$ sont les k plus petits nombres premiers, avec c_k le dernier exposant non nul.

En conséquence, pour que n soit hautement composé, il faut que $c_1 \geq c_2 \geq ... \geq c_k$. En cas de non respect de cette règle si nous échangeons deux exposants on diminue le nombre n tout en conservant exactement le même nombre de diviseurs. Un exemple pour illustrer : $18 = 2^1 * 3^2$ peut être remplacé par $12 = 2^2 * 3^1$, ces deux nombres ont tout les deux 6 diviseurs. De plus il a été montré que le dernier exposant $c_k = 1$, sauf dans deux cas particuliers $n = 4$ et $n = 36$.

3 Implémentation

3.1 Organigramme

L'organigramme suivant fait part de la décomposition que va adopter le projet. Le projet se décompose en trois catégories de tests pour les nombres hautement composé, les tests de primalité déterministe et les tests de primalité probabiliste.



3.2 Langages de programmation

Pour la réalisation du projet, nous avons convenu avec notre encadrant les langages que nous allions utiliser. Premièrement nous avons choisi le langage C++, langage adapté pour la programmation procédurale et pour la programmation orientée objet. De plus, il dispose de nombreuses bibliothèques permettant de manipuler des grands entiers supérieur à 2^{64}, de chronométrer l'exécution d'une portion du programme, et autre.
Deuxièmement le langage Shell. Celui-ci faisant partie du système d'exploitation UNIX, va nous permettre d'établir un script *test.sh* avec l'enchaînement de plusieurs lignes de commandes pour l'exécution de tests.

3.3 Outils

Pour ainsi produire le projet certains outils ont été utilisés pour obtenir le résultat présenté. En premier lieu nous avons installé la bibliothèque *GMP*[9]. Cette dernière est une bibliothèque nécessaire pour utiliser NTL, de plus elle pourrait nous être utile dans le cas où l'on voudrait générer de très grands nombre car les types de base proposés par le C++ nous limite à l'utilisation de nombres de 64 bits maximum.

La deuxième bibliothèque utilisé est donc la bibliothèque *NTL*[10], qui propose certaines fonctionnalités pour le calcul du PolyModulo[6] qui nous est indispensable pour l'implémentation du test de primalité AKS. N'ayant aucune connaissances nécessaire dans le domaine des mathématiques sur

l'algèbre modulaire, l'utilisation de cette bibliothèque nous permet d'obtenir des fonctions pour ces calculs et de façon optimisée.

Nous avons également utilisé un service d'hébergement de gestion de développement utilisant le système de gestion de version *Git* qui est **GitHub**. Ceci nous a permis de travailler en groupe sur un dépôt de façon très facile tout en étant indépendant lorsque chacun devait travailler de son côté sur une fonctionnalité sans pour autant perturber le travail du reste du groupe. GitHub nous permet également de stocker les différentes versions de notre travail.

Enfin le dernier outil utilisé est *CMake*[11]. *CMake* est une famille d'outils multi-plateformes et open source permettant de construire, tester et emballer des logiciels. Nous l'utilisons pour la compilation du projet, la liaison avec plusieurs bibliothèques utilisées et pour la génération d'un makefile.

4 Analyse des résultats

4.1 Lancement d'un test depuis l'execution en ligne de commande

Pour lancer un test depuis la console il suffit d'entrer la commande suivante dans le terminal situé au niveau du dossier build du projet. (Dossier qui doit être construit pour l'utilisation de cmake et de make).

```
./prime_numbers -options nb_iterations nbs_a_tester
```

Les options permettent de connaître le ou les tests que l'on veux exécuter. Pour cela nous avons à disposition les arguments suivants :

- a : Effectue chaque test çi-dessous
- k : AKS
- e : Euclide (Computation Bound)
- o : Modulo (Computation Bound)
- m : Crible d'Eratosthène
- p : Pocklington
- i : Miller-Rabin
- h : Nombre hautement composé naïve
- H : Nombre hautement composé définition.

Cette commande est également composée d'un argument qui permet de préciser le nombre d'itérations d'un même test de nombre premier afin d'obtenir une moyenne du temps d'exécution de celui-çi. Enfin le dernier argument est composé d'autant de nombre dont on souhaite la primalité séparés par un espace.
Le programme va ainsi afficher dans le terminal les moyennes en temps d'exécution pour chaque test

et nombre. Le programme va générer trois fichiers. Le fichier "*data.txt*" contient les résultats pour le temps d'exécution de l'échantillon de nombres donnée. Le fichier "*result.txt*" quant à lui contient les retours des tests pour le dernier nombre de l'échantillon. Le dernier fichier "*memory.txt*" contiendra la liste modifié par la fonctionnalité de *memory_bound*.

4.2 Lancement d'un test depuis le script SHELL

Pour lancer un (voir une série de) test on peut aussi utiliser le script "*test.sh*".
Le programme va ensuite demander différents paramètres afin de définir avec précision les tests que l'utilisateur souhaite effectuer.

4.3 Matériel et logiciels utilisés

Pour l'exécution des tests, nous avons utilisé une machine doté de la configuration suivante :

— Windows 10 Professionnel.

— Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz 3.50GHz.

— 16.0 Go de RAM.

— Système d'exploitation 64 bits, processeurx64

Le programme est exécuté sur une machine virtuelle présente sur la machine qui est oracle VM VirtualBox version 5.2.4 et l'OS debian64bits_ufr_sciences_2017 fournis par l'UVSQ. La VM est lancé avec 4 CPUs et 10 240 Mo de RAM.
Concernant les logiciels, pour la compilation du code C++ nous avons utilisé le compilateur g++ avec la version 7.2.0. Pour les librairies utilisées, la version de NTL est 10.5.0 et la version de GMP utilisé est 6.1.2. Enfin la version utilisé pour *CMake* est 3.9.4.

4.4 Résultats

4.4.1 Tests de primalité sur un échantillon

Tout d'abord nous avons réalisé une première batterie de tests sur un échantillon de nombres premiers avec 10 itérations pour établir une moyenne du temps d'exécution pour chaque membre de cet échantillon. Cet échantillon de nombres premiers est le suivant : '3', '97', '1039', '50023', '102013', '1300837', '4301623', '9990887', '15487253', '25003151', '33083221', '40003387', '110003329', '520002179', '720001787' et '1073741857'. L'Étude demandé pour ce projet nous impose de faire une comparaison des résultats obtenus pour chaque test de primalité.

Dans les graphes 1, 2, 3 et 4 on peux observer l'évolution du temps d'exécution en fonction du nombre. Pour chaque courbe, la marge d'erreur sur le temps d'exécution d'un nombre est indiqué par une barre. Celle-ci nous permet d'observer la stabilité du temps d'exécution d'une exécution à une autre.

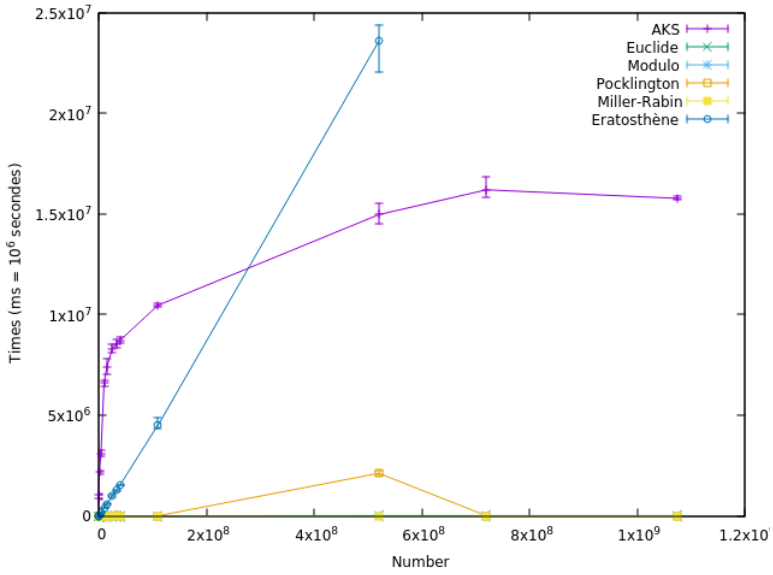


FIGURE 1 – Évolution du temps d'exécution pour les 6 tests de primalité (AKS, Pocklington, Miller-Rabin, Euclide, Ératosthène et Modulo).

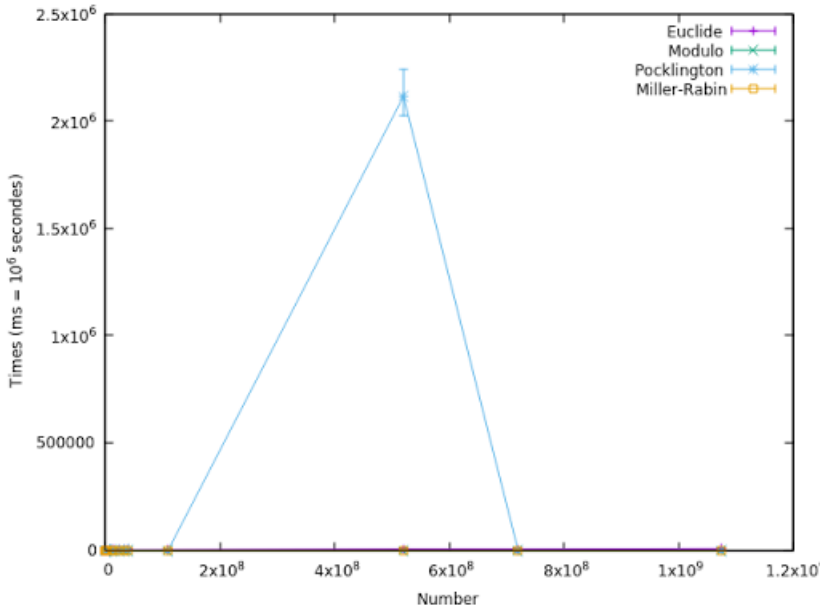


FIGURE 2 – Zoom de la figure 1 sur (Pocklington, Miller-Rabin, Euclide et Modulo).

Pour commencer nous pouvons observer que les courbes d'évolution du temps d'exécution pour les méthodes naïves d'Euclide et du Modulo ont des temps d'exécution très faibles pour de petits nombres. Mais les nombres utilisés ici sont de taille raisonnable, mais pour des nombres de l'ordre de 2^{50} leur temps d'exécution augmenteront de façon considérable à cause de leur complexité. Pour

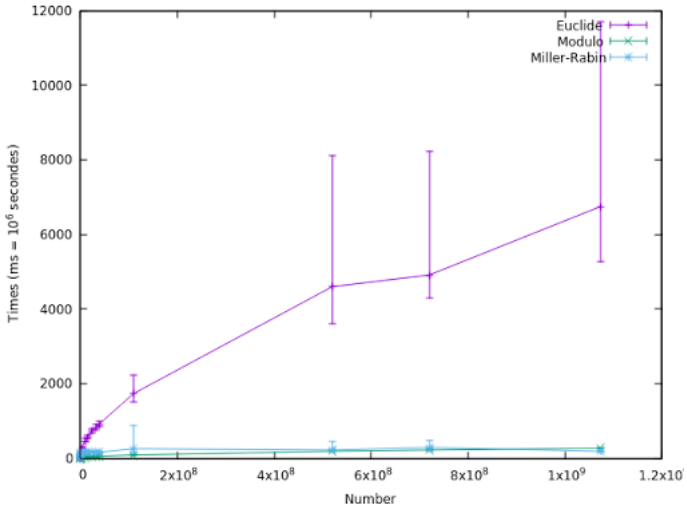


FIGURE 3 – Zoom de la figure 1 sur (Miller-Rabin, Euclide et Modulo).

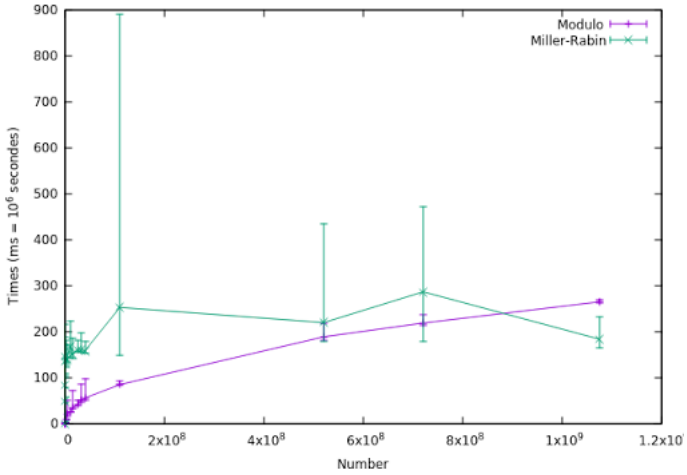


FIGURE 4 – Zoom de la figure 1 sur (Miller-Rabin et Modulo).

l'algorithme de memory_bound/Ératosthène l'utilisation de la mémoire devient beaucoup trop importante à cause du tableau de taile N+1 créer par l'algorithme de crible (Exemple pour 97 on utilise 97 bytes, mais pour 2^{50} on aura donc 2^{50} bytes utilisés en mémoire). Nous pouvons donc ainsi observer que la courbe pour cet algorithme s'accroît de manière très rapide. De plus la courbe d'évolution d'Ératosthène se fini avant par rapport aux autres tests de primalité, cela est causé par une limitation de nos machines au niveau de la RAM nous empêchant ainsi d'obtenir des résultats pour des nombres supérieurs. Pour les autres tests nous nous sommes arrêtés à 1073741857 également limité par la RAM disponible sur la machine utilisée.

En observant les résultats de l'algorithme de Miller Rabin, on observe que le temps d'exécution

pour les petits nombres est bien au dessus des algorithmes plus naïfs mais qu'il devient très rapidement plus efficace que tous les autres algorithmes utilisés.

Concernant le test de Pocklington, la courbe ne nous permet pas de définir une évolution précise du temps d'exécution en fonction de la taille du nombre. En effet, la factorisation est la partie théoriquement la plus lourde de cet algorithme, elle est plus ou moins longue en fonction du N en entrée, il est en effet difficile de déduire le temps d'exécution à partir de la taille de N mais le temps d'exécution a quand même tendance a augmenter avec la taille du nombre. Des optimisations de cette partie lourde existent tel que le crible quadratique ou encore le crible algébrique.
En ce qui concerne le test AKS, conformément à nos attentes, nous pouvons voir que le temps d'exécution est long pour de petits nombres. La courbe croit de façon rapide au début et commence à se rapprocher de $1.5x10^7 \mu s$ (compte tenu de la complexité de l'algorithme (O $\log(n)^{12}$)). Seulement Ératosthène devient plus lent que AKS à partir de nombres de taille un peu inférieure à $3x10^8$.
Les résultats nous laissent penser que pour des nombres très grands (supérieur à 2^{64}) le temps d'exécution de tous les algorithmes devrait passer au dessus d'AKS.

4.4.2 Tests de primalité sur une plage de données

Pour continuer, nous avons effectué de nouveaux tests sur une plage de données. Les figures 5, 6, 7 et 8 représentent les variations du temps d'exécution sur la plage [1 :10 000].

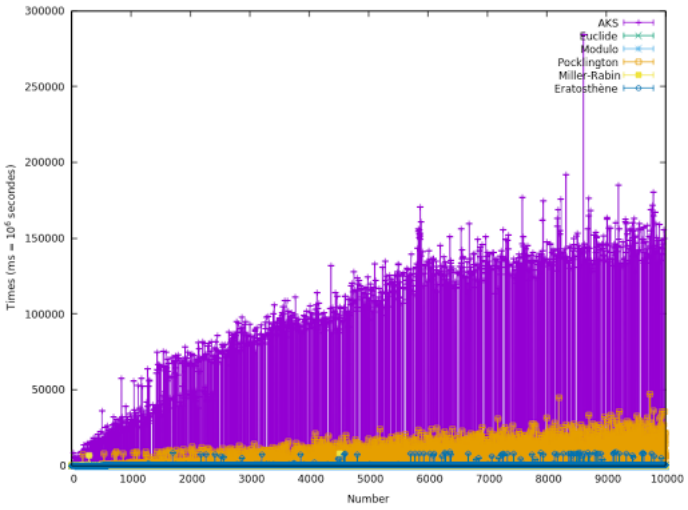


FIGURE 5 – Évolution du temps d'exécution pour les 6 tests de primalité sur une plage de données(AKS, Pocklington, Miller-Rabin, Euclide, Ératosthène et Modulo).

Après avoir exécuté les tests sur la plage de données, nous avons pu établir une moyenne du temps d'exécution global de la plage de données. Ces temps sont indiqués ci-dessous :

— L'algorithme AKS met en moyenne 118064433 μs soit environ 118,06s

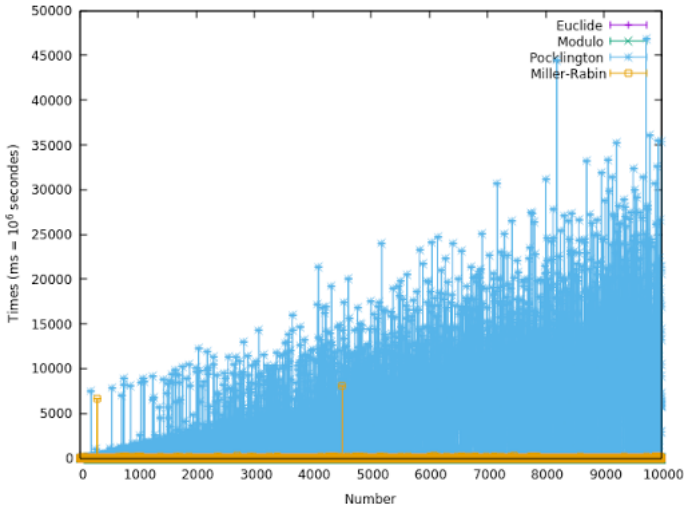


FIGURE 6 – Zoom de la figure 5 sur (Pocklington, Miller-Rabin, Euclide et Modulo).

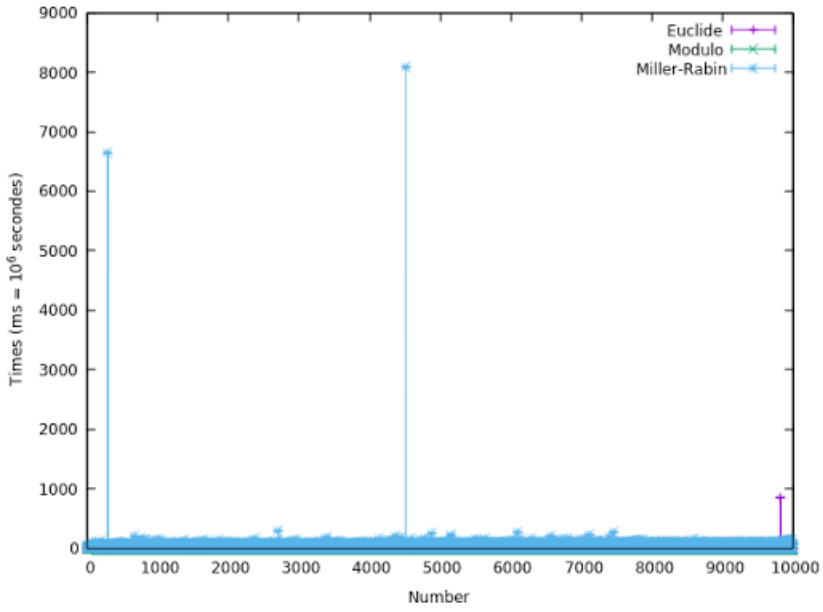


FIGURE 7 – Zoom de la figure 5 sur (Miller-Rabin, Euclide et Modulo).

— L'algorithme d'Euclide met en moyenne 11937 μs soit environ 0,012s

— L'algorithme appliquant le modulo met en moyenne 389 μs soit environ 0,0004s

— L'algorithme de Pocklington met en moyenne 41178556 μs soit environ 41,17s

— L'algorithme d'Eratosthène met en moyenne 2674942 μs soit environ 2,67s

— L'algorithme de Miller-Rabin met en moyenne 163611 μs soit environ 0,163s

Une étude statistique sur le test de Miller-Rabin a pu être fait pour cette plage de données. On a

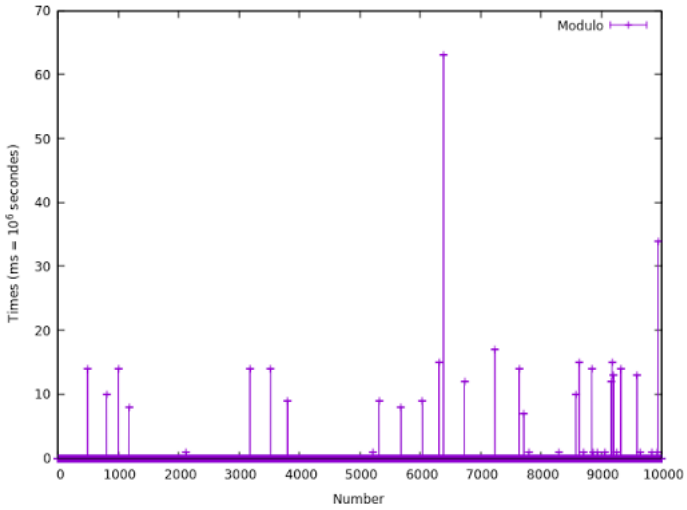


FIGURE 8 – Zoom de la figure 5 sur (Miller-Rabin et Modulo).

pu voir que pour les 100 000 premiers nombres (entre 2^{16} et 2^{17}) l'algorithme a un taux de réussite de 100% même avec un faible nombre d'itérations (ici testé avec 5). Les tests ont été poussés jusqu'à 2^{27} sans trouver d'erreurs avec une plage de données de 100 000 nombres, le temps d'exécution commence alors à atteindre plus d'une dizaine de minutes, ici encore pour un faible nombre d'itérations.

Le graphe 9 est la réalisation du test sur la plage [10 000 :100 000] pour obtenir une meilleure vision de l'évolution de la moyenne du temps d'exécution global sur une plus grande plage de données. On peut remarquer que le temps d'exécution de AKS reste relativement haut mais Pocklington

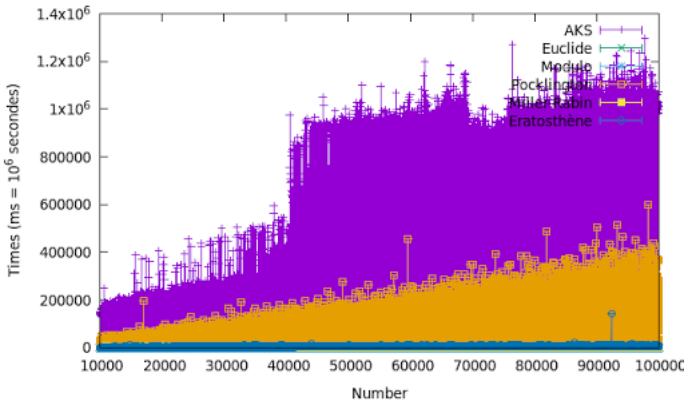


FIGURE 9 – Évolution du temps d'exécution pour les 6 tests de primalité sur [10 000 :100 000].

s'approche plus rapidement de AKS.

4.4.3 Hautement Composé

Pour les deux fonctionnalités sur les nombres hautement composés, l'échantillon suivant : ['2' '120' '240' '360' '720' '840' '1260' '1680' '2520'] a été utilisé pour comparer les deux procédures et observer l'évolution du temps d'exécution dans les figures 10 et 11.

Nous pouvons observer que la méthode naïve prend énormément de temps pour s'exécuter en

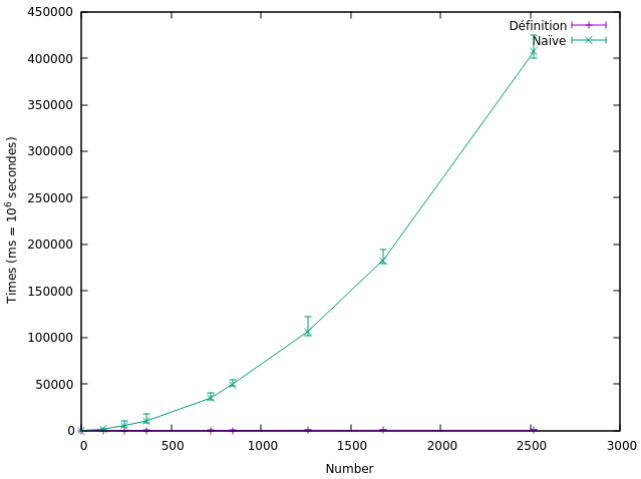


FIGURE 10 – Évolution du temps d'exécution pour les nombres Hautement Composés.

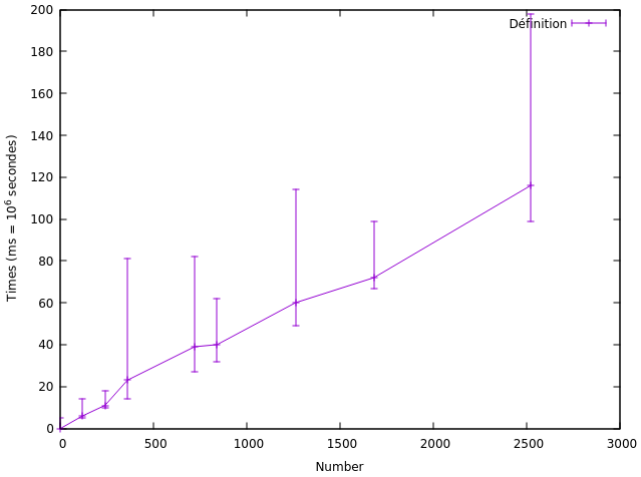


FIGURE 11 – Zoom sur la courbe pour la méthode appliquant la définition.

comparaison à la méthode appliquant la définition de ce type de nombre. En effet, pour un nombre N donné, la fonction va devoir calculer le nombre de diviseurs pour les N-1 nombres inférieur à N et effectuer N comparaisons ce qui engendre beaucoup de latence. Et le temps pour le nombre 1680 est déjà assez élevé pour un nombre supérieur à 2^{10}. On peut donc imaginer la courbe accroître plus rapidement pour de très grands nombres. Mais étant limités par notre matériel, nous nous sommes

contentés de ce type de nombres. Le décrochage que l'on peut observer peut-être justifié par le temps que l'algorithme a mis pour trouver la factorisation de ce nombre pour le calcul du nombre de diviseur. En effet, pour certains nombres il est possible de trouver plus rapidement la factorisation en nombre premier, ce qui peut justifier le fait que 2520 s'exécute plus rapidement que 1680. On peut noter également une forte instabilité au niveau du temps d'exécution d'après les barres d'erreur présentent sur les courbes.

5 Bilan Technique du Projet

Après l'analyse des résultats nous pouvons constater que les différents tests appliqués possèdent chacun des limites. Tout d'abord, la principale limite du crible d'Eratosthène concerne la taille de son tableau. En effet, pour un nombre N très très grand, le crible va créer un tableau de booléen de taille N+1, ce qui va prendre énormément de place en mémoire, et avec les machines que nous avons à disposition nous sommes très vite limités à cause de la RAM. Puis pour la création de la liste contenant les nombres premiers de 2 jusqu'à N, la fonctionnalité de memory_bound va devoir parcourir toutes les cases du tableau du crible pour remplir sa liste, ce qui engendre une complexité de N pour cette opération de remplissage.

La méthode naïve effectue au pire $\sqrt{N}$ divisions euclidiennes. Or cette division euclidienne n'est pas à temps constant. De plus pour un nombre N écrit en base 2, sa longueur vaut log_2(N). Notre algorithme effectuera donc $\sqrt{2^{log_2(n)}}$ divisions euclidiennes. Nos méthodes naïves s'exécuteront donc en temps exponentiel. C'est pourquoi ces méthodes très simples et rapides pour de petits nombres deviennent très rapidement non applicables pour de très grands nombres. Le test d'un nombre aux alentours de 2^{64} prendra plusieurs années à s'exécuter même sur nos machines actuelles.

Ensuite, pour le test de Pocklington la limite principale est le besoin de factoriser le voisin d'en dessous du nombre dont on cherche à valider la primalité, c'est-à-dire N-1 si N est le nombre dont on met la primalité en question. La factorisation des grands nombres est une lourde tâche de calcul de difficulté exponentielle, ainsi même les programmes les plus efficaces demandent énormément de temps, des années voir plus, pour factoriser certains nombres très grands. En réalité, nous avons la possibilité de factoriser N-1 qu'en partie, il suffit de factoriser N-1 jusqu'à trouver un A, créé à partir des facteurs premiers de N-1, qui respecte les conditions nécessaires pour appliquer le théorème.

Pour Miller-Rabin, les limites concernent le retour de l'algorithme auquel ils nous faut s'assurer que ce dernier ne renvoie pas "premier" pour un nombre composé ou inversement (lors de l'étude de très grand nombres). Pour cela il faut faire plusieurs itérations de l'algorithme, et ce nombre d'itérations est difficile à déterminer et si on veut être sûr que le résultat soit correct le nombre d'itérations peut être très élevé ce qui augmente le temps d'exécution nécessaire pour avoir des données correctes.

Concernant l'implémentation de AKS, ce dernier pourrait être sûrement amélioré avec une plus

grande connaissance de la bibliothèque NTL utilisée pour l'arithmétique modulaire ou même avec l'utilisation d'autres bibliothèques. De plus une plus grande connaissance en mathématiques aurait permis d'offrir une optimisation pour certaines lignes de code, notamment dans l'étape 5 de l'algorithme. Il aurait aussi été possible d'implémenter une variante de l'algorithme AKS de base qui est plus récente et donc plus optimisée pour un gain de performance certain. Il est également possible que la bibliothèque NTL fausse les résultats de temps pour l'algorithme AKS. En effet, la bibliothèque effectue certaines vérifications superflues qui ralentiraient le temps d'exécution du test.

Enfin, les tests d'un nombre hautement composé effectuent N calculs du nombre de diviseurs d'un nombre en partant de sa factorisation en nombre premier. C'est pourquoi tout comme les méthodes naïves pour les tests de primalité, en utilisant la méthode naïve d'un nombre aux alentours de 2^{64} prendra un temps considérable à s'exécuter même sur nos machines actuelles.

6 Organisation interne du groupe

Pour débuter la programmation de ce projet, il nous fallait en premier lieu établir la répartition du travail de groupe pour que le projet puisse avancer de façon efficace et de manière rapide. Le tableau ci-dessous va ainsi indiquer pour chaque membre du groupe la ou les fonctionnalités pour laquelle il a pu contribuer à l'élaboration :

Tâches	Jean-Didier	Maxence	Romain	Robin	Damien
Eratosthène/Memory Bound	x				
Euclide/Computation Bound		x			
AKS			x		
Pocklington					x
Miller-Rabin				x	
Highly Composite	x				
Cmake	x	x			
Script/main	x	x			

Concernant le rapport et les slides de présentation, nous avons tous contribué à l'élaboration du contenu de ces derniers et principalement Jean-Didier pour tout ce qui concerne la mise en page.

7 Conclusion

Ce document résume tout ce qui a été établi avant, pendant et après la réalisation de notre projet. Ce projet est donc une application de plusieurs algorithmes pour tester la primalité d'un nombre ou si il est hautement composé. Après une étude des tests sur un échantillon de valeurs on a pu faire plusieurs comparaisons entre ces méthodes et on a pu observer que les méthodes naïves

sont très utiles pour les petits nombres mais deviennent très lentes pour les très grands nombres. De plus, Miller-Rabin est un test probabiliste qui pour nous serait intéressant. En effet ce dernier a une exécution rapide mais avec une probabilité très très faible d'obtenir un résultat non conforme.

Dans l'état actuel, on peut revoir avec du recul, la manière avec laquelle notre projet a été réalisé. En effet, nous aurions sûrement revue notre répartition des tâches concernant certains algorithmes comme AKS et Pocklington. Ces derniers ont posés des problèmes pour certains membres du groupe car ces algorithmes ne sont pas faciles à appréhender et demande certaines connaissances notamment en mathématique pour les Polynômes Modulaire pour AKS. Des groupes de 2 pour leur implémentation aurait été préférable.

Ce travail a été réalisé par un groupe de 5 personnes. Le sujet a été très bien appréhendé par les membre du groupe et l'entente a été excellente. Grâce aux informations préalablement établies lors de notre entretien avec Monsieur Gougeaud, le travail au sein du groupe a pu être mené de façon assez indépendante, sans générer de conflits ni de problèmes d'organisation.

Pour finir, après observation de nos résultats, nous sommes rentrés en phase de réflexion concernant la deuxième étape du projet qui aura lieu pendant le semestre 2. Cette étape consiste à paralléliser l'implémentation faite au cours de ce semestre. Mais ici tout les tests de primalité ne sont pas forcément parallélisables. Nous pensons actuellement que seul le crible d'Ératosthène est parallélisable et donc il sera de notre ressort de penser à une manière de paralléliser l'implémentation faite ou alors paralléliser les tests comme ceux réalisés sur des grands ensembles de nombres pour accélérer l'exécution et de réussir les enjeux posés par l'équilibre des charges sur les différents processeurs.

Références

[1] JP. Zanotti. zanotti.univ-tln.fr/ALGO/I51/Crible.html. 16-03-2016.

[2] https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide.

[3] https://en.wikipedia.org/wiki/Pocklington_primality_test.

[4] https://fr.wikipedia.org/wiki/Test_de_primalit%C3%A9_AKS.

[5] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004.

[6] https://en.wikipedia.org/wiki/AKS_primality_test.

[7] https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test.

[8] http://mathworld.wolfram.com/HighlyCompositeNumber.html.

[9] https://gmplib.org/.

[10] http://www.shoup.net/ntl/.

[11] https://cmake.org/.