Projet Tutoré : Licence Maths-Info

Génération aléatoire de grands nombres premiers

Pierre GRABER, Elias DEBEYSSAC, Toky RANDRIAMALALA

Année 2019-2020

Introduction

Les nombres premiers sont des nombres mystérieux que les mathématiciens étudient depuis des siècles tant pour leurs propriétés algébriques intéressantes que pour le caractère aléatoire de leur répartition dans l'ensemble ordonné des entiers naturels $\mathbb{N}$. En effet de nos jours les nombres premiers sont largement utilisés en cryptographie, car leurs propriétés permettent de garantir la sécurité de systèmes cryptographiques exploitant des problèmes mathématiquement difficiles à résoudre algorithmiquement, qui nécessiteraient des années de calcul par les ordinateurs actuels.
La sécurité de ces systèmes de chiffrement repose par exemple sur la difficulté de retrouver la factorisation de très grands nombres en produit de "grands" facteurs premiers. Les ordinateurs, téléphones, cartes à puces utilisent une quantité industrielle de nombres premiers afin d'assurer la fiabilité de leurs méthodes de chiffrement. La génération de grand nombres premiers est donc indispensable pour la sécurité des systèmes informatiques. Ce projet tutoré a pour but d'implémenter et d'expérimenter des algorithmes efficaces de génération de grands nombres premiers, et utilisables à des fins cryptographiques. Dans un premier temps les algorithmes seront implémentés dans un langage proche de celui de Python, grâce au logiciel de calculs mathématiques SageMath, tous les algorithmes seront tirés du livre "Handbook of Applied Cryptography".

Table des matières

1 L'aléatoire, les tests de primalité

1.1 Informatique et aléatoire

Tous nos algorithmes utilisent la libraire *random* de python afin de générer des nombres aléatoires. Il parait donc judicieux de se pencher sur la façon dont les langages de programmation tels que python produisent ces nombres et sur le caractère vraiment aléatoire de ces nombres. En effet si nos algorithmes ont pour but de pouvoir être utilisés à des fins cryptographiques, il est important qu'ils soient surs, et pour cela il faut que les nombres tirés aléatoirement ne suivent aucune régularité et ne puissent pas être devinés par un attaquant quelconque. Si python utilisait un algorithme spécial pour calculer ces nombres alors les systèmes que nous tentons de mettre en place n'auraient aucune valeur car un attaquant aurait potentiellement des informations quant à la façon dont nous construisons nos nombres premiers aléatoires.

—- voir Yarrow
—- voir Fortuna

—- voir la méthode des carrés médians
—- voir source :

https ://openclassrooms.com/fr/courses/1389636-a-la-decouverte-de-laleatoire-et-des-probabilites/1389794-fabriquez-votre-propre-fonction-rand

—-Python *rand* == Pseudo-aléatoire

1.2 Les tests de Primalité

1.2.1 Les divisions successives

Lors de la génération d'un nombre aléatoire n, afin de savoir si celui ci est un nombre premier il parait raisonnable d'essayer de trouver des candidats pour sa factorisation par divisions successives avant d'effectuer un "réel" test de primalité. En effet comme tout nombre peut se décomposer en facteurs de nombres premiers, il suffit d'effectuer les divisions euclidiennes de ce n par une liste de premiers inférieurs à sa racine carrée afin de savoir si celui est composé ou non. Si après avoir testé tous les nombres premiers $p \leq \sqrt{n}$, on ne trouve pas de p tel que :

$$n = 0 \pmod{p}$$

alors on peut conclure que n est premier. Cependant sur des nombres à plusieurs centaines de chiffres, codés par exemples sur 1024 bits, cet algorithme ne peut s'avérer efficace car il demanderait environ 2^{512} divisions ce qui est évidemment beaucoup trop couteux en temps pour être efficace. Pour tester si un nombre codé sur 1024 bits (environ 300 chiffres décimaux) est premier on peut néanmoins utiliser ces divisions successives jusqu'à un certains rang que

l'on appellera B déterminé de manière expérimentale, avant de passer à un test de primalité comme celui expliquer dans le paragraphe suivant. Le premier objectif de ce projet tutoré a donc été de fixer expérimentalement ce rang B afin de savoir combien de divisions successives il est intéressant d'effectuer avant d'effectuer le test de Miller-Rabin.

1.2.2 Le Test de Miller-Rabin

On ne connait pas de formule donnant la totalité des nombres premiers ou permettant de calculer le "n-ième" terme de la suite des nombres premiers. Une première idée est donc d'utiliser des tests de primalité afin de déterminer si un nombre généré aléatoirement est premier ou non. La répartition des nombres premiers nous assure qu'en effectuant de manière répétitive un tel algorithme nous finirons par tomber sur un candidat probablement premier.
L'algorithme de test de primalité le plus utilisé à des fins cryptographiques de par son efficacité est l'algorithme de Miller-Rabin (et ses variantes). Ce test prend en entrée un entier N et nous retourne soit "non" : dans ce cas N est composé de façon certaine, soit "oui" : dans ce cas N est probablement premier. Le test de Miller Rabin repose sur 3 théorèmes principaux.

— Tout dabord le petit théorème de Fermat qui nous indique que pour p premier, quelque soit a premier avec p, $a^{p-1} \equiv 1 \pmod{p}$.

— Un autre théorème nous indique que soit N un nombre impair avec $N - 1 = 2^s t$ où t est impair. S'il existe un entier a premier avec N tel que $a^t \not\equiv 1 \pmod{N}$ et $a^{(2^i)t} \not\equiv -1 \pmod{N}$ pour i = 0,1,...s-1 alors N est composé. Ce théorème nous donne un critère supplémentaire qui nous permet d'obtenir des témoins de non-primalité pour les nombres de Carmichael qui posent problème au petit théorème de Fermat.

— Enfin le dernier théorème nous permet d'affirmer que pour $N > 9$ un nombre composé impair composé avec $N - 1 = 2^s t$, où t impair. Alors $Card\, a \in \left\{(\mathbb{Z}/N\mathbb{Z})^*, a^t \equiv 1 \pmod{N}\ ou\ a^{2^i t} \equiv -1 \pmod{N}\ pour\ un\ 0 <= i <= s-1\right\} <= \frac{phi(N)}{4}$. En itérant donc k fois l'algorithme de Miller-Rabin, on obtient donc une probabilité $<= 1/4^k$ qu'un nombre composé soit déclaré probablement premier ce qui devient négligeable avec quelques dizaines d'itérations.

Cependant il existe d'autres tests de primalité permettant de fournir une preuve de leur résultat tels que AKS, APRCL (corps cyclotomiques) ou ECPP (courbes elliptiques) contrairement à l'algorithme de Miller-Rabin. Cependant ces algorithmes sont bien plus lents et principalment utilisé à des fins théoriques.

Dans le cadre ce projet tutoré nous utiliserons donc le test de primalité de Miller-Rabin pour nos algorithmes. Nous comparerons l'efficacité de nos résultats (en temps) avec le test utilisé par Sage lors de l'appel à la fonction

"is pseudoprime", i.e le test de Baillie-PSW, qui est une combinaison de test de Miller-rabin et de Lucas.

2 Algorithmes de Génération de grands nombres premiers

2.1 Génération "naïve"

Le premier algorithme de génération dit "naif" est simple, il consiste en deux étapes : - Tirer un nombre n aléatoirement jusqu'à tomber sur un impair - Tester si cet impair est premier ou non Si ce nombre "n" est composé on incrémente de 2 jusqu'à tomber sur un premier.

2.2 Algorithme de recherche aléatoire

2.3 Génération de nombres premiers forts : Algorithme de Gordon

2.4 Méthode NIST

2.5 Premiers prouvables : Algorithme de Maurer

3 Résultats expérimentaux

Sources