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 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"

—¿ pas aléatoire

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.

Cette méthode présente un inconvénient majeur, en effet cette génération ne peut pas être considérée comme aléatoire.

2.2 Algorithme de recherche aléatoire

On sait qu'il existe un nombre infini de nombres premiers (Euclide l'a démontré), et même si leur répartition est très irréguliere et qu'ils se rarifient dès qu'ils grandissent, on sait que leur répartition vérifie plus ou moins la distribution suivante : la proportion de premiers $p \leq x$ est approximativement égale à $1/ln(x)$.

On peut donc penser qu'en tirant un nombre au hasard un certains nombre de fois dans un ensemble donné on tombera forcément à un moment ou un autre sur un nombre premier. De plus comme dit dans la partie concernant l'aléatoire, le tirage aléatoire d'un nombre par python est très rapide, il prend environ 4×10^{-6} seconde. En répétant donc cette opération un certain nombre de fois on arrive a trouver des nombres premiers grands en un temps raisonnable, dépendamment du test de primalité utilisé.

L'algorithme de recherche aléatoire se décompose donc de la manière suivante :

soit le code suivant en SageMath (python) :

```
def RS(k):
    " Algo random_search avec boucle while =/= récursif "
    b = False
    while not b:
        a=Integer(randint(0,2**k))
        if TrialDivision(a):
            if a.is_pseudoprime():
                b=True
    return a
```

2.3 Génération de nombres premiers forts : Algorithme de Gordon

Un deuxieme algrithme proposé par le livre "Handbook of Appplied cryptography" permet de générer des nombres premiers mais cette fois ci avec des caractéristiques supplémentaires, qui permettent donc d'être utilisés de manière plus sécurisées notamment pur des systèmes de type RSA. Ces nombres premiers sont appelés des *strongprime* en anglais, ils remplissent les conditions suivantes :

p est un "strong prime" si il existe des entiers r,s,t tels que :
— $p-1$ a un "grand" facteur premier noté r ;
— $p+1$ a un "grand" facteur premier noté s ;
— $r-1$ a un "grand" facteur premier noté t

Algorithme 1 Algorithme de Gordon pour la génération de premiers forts
Entrée: Un entier "k".
Sortie: Un premier codé sur k-bits.

1 - Générer deux premiers s et t de même taille.
2 - Choisir un entier aléatoire i
3 - **tant que** $2it+1$ n'est pas premier :
        i=i+1
4 - Calculer $r = 2it + 1$
5 - Calculer $p_0 = 2(s^{r-2} \mod r)s - 1$
6 - Tirer un entier aléatoire j de même taille que i
7 - **tant que** $p_0 + 2jrs$ n'est pas premier :
        $p_0 = p_0 + 1$
8 - **retourne** $p = p_0 + 2 * j_0$

La génération de tels nombres premiers nécessite donc un peu plus de temps que l'algorithme précedent mais elle fournit des nombres premiers qui sont mieux "protégés" des algorithmes de factorisation de $p-1$ et $p+1$. En effet l'intêret de l'algorithme de Gordon repose sur les propriétés de $p-1$ et

$p+1$.
Pour un attaque de Pollard par exemple, la raison pour laquelle $p-1$ est spécial est que $p-1$ est l'ordre du groupe multiplicatif $(Z/nZ)*$. Si on cherche par exemple à factoriser un module RSA $n = p * q$, on choisit un nombre aléatoire $a \in Z/nZ$, on calcule ensuite $d = pgcd(a, n)$ et si $d > 1$ alors on renvoie d. Sinon, si on a supposé que p était B-friable alors on trouve un entier m diviseur de $p-1$, et on peut alors calculer $x = a^{mn} mod\ n$, si $d = pgcd(x-1, n) > 1$ alors d est un diviseur non-trivial de n.

2.4 Méthode proposée par le NIST : génération de premiers pour DSA

Le NIST est l'Institut national des normes et de la technologie américain. Cet institut a proposé un algorithme en 1991 basé sur l'exponentiation modulaire et le problème du logarithme discret permettant de fournir une signature numérique, outil essentiel pour garantir l'intégrité d'un document ainsi que l'authentification de son auteur.
Le mécanisme proposé par le NIST se décompose en 3 étapes :

— Génération de clés.
— Signature du document.
— Vérification du document.

Dans la cadre de notre projet nous nous intéresserons à la premiere étape de ce processus c'est à dire la génération des clés. La génération des clés doit donner un couple de premiers (q, p) qui doivent satisfaire les conditions suivantes :

Soient L et N des longueurs, avec N divisible par 64.

— Le premier p doit être longueur L.
— Le premier q doit être longueur N
— $(p-1)$ doit être divisible par q.

2.5 Premiers prouvables : Algorithme de Maurer

Le prochain algorithme est spécial dans le sens ou il est le seul de notre liste d'algorithmes capable de fournir des nombres premiers surs, prouvables. Son temps d'execution est légérement plus élevé que le temps d'exécution d'un algorithme générant des pseudo-premiers, mais il est néanmoins plutôt efficace pour les tailles dont nous avons besoin. Cet algorithme est un raffinement du théorème de Pocklingon.

Algorithme 2 Algorithme de Maurer pour la génération de premiers prouvable

Nom de l'algorithme pour l'appel récursif :
$PremierProuvable(k)$

Entrée: Un entier "k".
Sortie: Un premier codé sur k-bits.

1 - *Si k est petit (inférieur à 20) on renvoie un nombre premier aléatoire testé grâce à la méthode des divisions successives*
2 - Initialiser : $c = 0.1$ et $m = 20$
3 - Calculer : $B = c * k^2$
4 - **tant que** $(k - rk) > m$:
si $k > 2m$:
- Tirer une réel s aléatoire tel que : $s \in [0, 1]$
- Calculer $r = 2^{s-1}$

sinon :
- Initialiser : $r = 0.5$

5 - Calculer : $q = PremierProuvable(k)$
6 - Initialiser : I = $\lfloor 2^{n-1}/2q \rfloor$

7 - Initialiser : success = 0
8 - **tant que** success = 0
- Tirer un entier aléatoire $R \in [I + 1, 2I]$
- Calculer $n = 2Rq + 1$
- Tester la primalité de n par l'algorithme de divisions successives jusqu'à B.

si n n'a pas de facteurs premiers inférieurs à B :
- Tirer un entier aléatoire $a \in [2, n - 2]$
- Calculer $b = a^{n-1} mod n$
- **si** $b = 1$:
  - Calculer $b = a^{2R}$ mod n et $d = pgcd(b - 1, n)$
  - **si** d = 1 :
    success = 1

9 - **retourne** n

3 Résultats expérimentaux

Dans cette partie nous expliciterons les résultats obtenus avec nos algorithmes. Nous comparerons l'efficacité en temps obtenue en utilisant les tests de primalité fournis par Sage et nos tests pour chacun des générateurs cités précedemment.

3.1 Détermination du nombre de Division successives

La première étape de nos tests expérimentaux a été la détermination du paramètre appelé "B" dans le "Handbook of applied cryptography". Ce paramètre définit le nombre de divisions successives qu'il est judicieux d'effectuer avant de passer à un test de primalité du type Miller-Rabin. Cette étape permet d'éliminer un certain nombre de candidats de manière plus rapide qu'un "réel" test de primalité et donc de gagner du temps sur la génération d'un premier.
Afin de définir ce paramètre nous avons pré-calculé une liste de premiers inférieurs ou égaux à 2^{32}. Ensuite nous utilisons notre générateur afin de connaitre le temps moyen pour générer 1000 chiffres premiers avec l'algorithme de notre choix. Le parametre B est alors la valeur jusqu'à laquelle nous allons effectuer les divisions euclidiennes. Plus précisemment nos algorithmes de divisions successives effectue les divisions jusqu'à la racine de B.

3.2 Temps Moyen de génération

Les tableaux ci dessous donnent un aperçu du cout en temps de chacun des algorithmes cités précedemment.

3.2.1 Avec un simple test de Miller-Rabin

Temps moyen en secondes sur 1000 générations avec Miller-Rabin

	Random-Search	**Gordon**	**Maurer**
256 bits	0.00648	0.00211	0.016218
512 bits	0.024274	0.00825	0.052886
1024 bits	0.13308	0.07741	0.204052
2048 bits	0.41064	0.82159	1.55760

3.2.2 Avec le test de pseudo-primalité de Sage

Temps moyen en secondes sur 1000 générations

	Random-Search	**Gordon**	**Maurer**
256 bits	0.00328	0.00211	0.016218
512 bits	0.00642	0.00825	0.052886
1024 bits	0.03954	0.07741	0.204052
2048 bits	0.41064	0.82159	1.55760

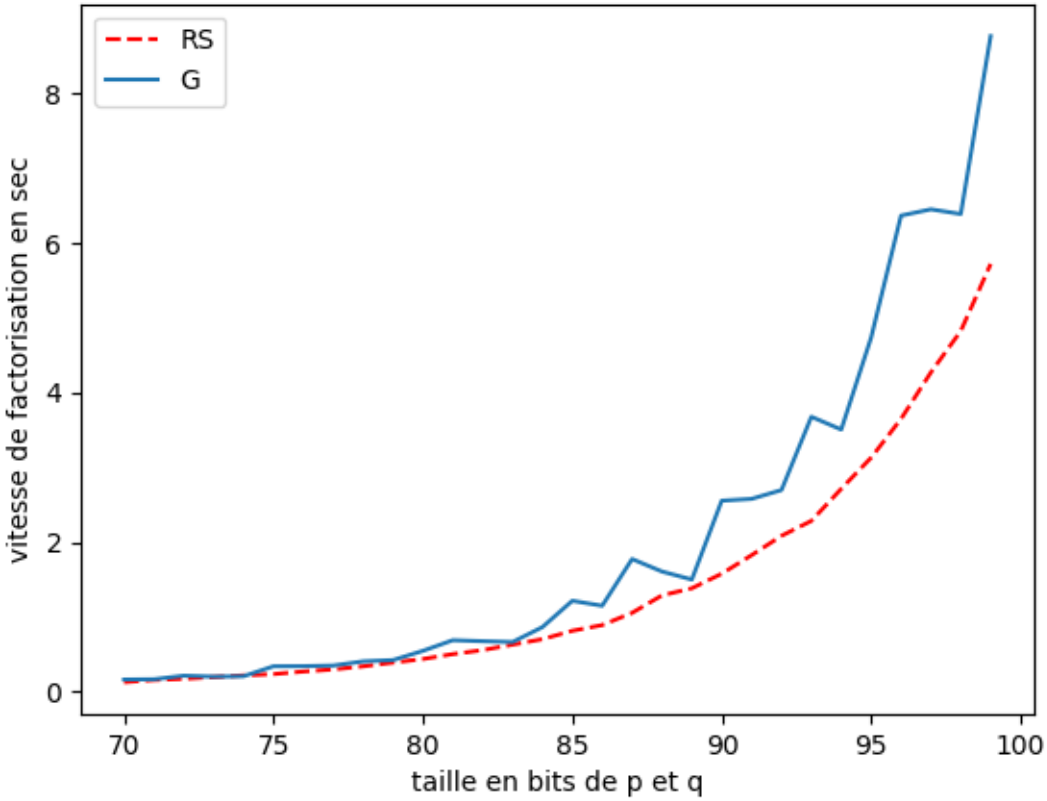
3.3 Factorisation

Danc cette partie nous comparons les temps de factorisation de modules RSA (de tailles raisonnables) générés avec deux de nos algorithmes : random-search et gordon. Cette partie tente donc de montrer l'utilité de l'algorithme

de gordon qui permet de générer des premiers "forts", présentant donc des avantages quant à la sécurité du module. Sage fournit une fonction permettant de factoriser un entier si sa taille est raisonnable (exemple : la fonction prend quelques secondes (1/10s) pour des entiers codés sur 70/80bits).

Nous avons donc ecrit une fonction permettant de calculer le temps de factorisation d'un module RSA, donc d'un entier n tel que : $n = p * q$ avec p et q deux premiers générés soit par l'algorithme random-search, soit par celui de gordon.

Les temps de factorisation pour des modules de petites tailles ($< 70bits$) sont sensiblement identiques.



3.4 Distribution aléatoire

Sources