

Génération aléatoire de grands nombres premiers

Pierre GRABER, Elias DEBEYSSAC

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 grands 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. Les algorithmes sont 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	4
1.1	Informatique et aléatoire	4
1.2	Les tests de Primalité	5
1.2.1	Les tests implémentés :	5
1.2.2	Les tests fournis par Sage	6
2	Algorithmes de Génération de grands nombres premiers	7
2.1	Génération "naïve"	7
2.2	Algorithme de recherche aléatoire	7
2.3	Génération de nombres premiers forts : Algorithme de Gordon	8
2.4	Méthode proposée par le NIST : génération de premiers pour DSA	9
2.5	Premiers prouvables : Algorithme de Maurer	10
3	Résultats expérimentaux	12
3.1	Détermination du nombre de Division successives	13
3.2	Temps Moyen de génération	13
3.2.1	Avec un test de Miller-Rabin	13
3.2.2	Avec le test de pseudo-primalité de Sage	14
3.3	Factorisation	14
3.4	Distribution aléatoire	16
4	Conclusion	18

1 L'aléatoire, les tests de primalité

1.1 Informatique et aléatoire

Dans un premier temps nos algorithmes utilisaient la bibliothèque *random* de python afin de générer des nombres aléatoires, il paraissait donc judicieux de se pencher sur la façon dont cette bibliothèque produit des nombres et sur le caractère vraiment aléatoire de leur distribution. 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 ne suivent aucune régularité et ne puissent pas être devinés ou prédits par un attaquant quelconque.

La bibliothèque *random* de Python est basée sur un générateur nommée Mersenne Twister, qui utilise le nombre premier de Mersenne $2^{19937} - 1$. Ce générateur pseudo-aléatoire présente de nombreux avantages tels que sa vitesse de génération et son comportement aléatoire lorsqu'il est utilisé sur des nombres de tailles raisonnables. Ce module est donc régulièrement utilisé en probabilités et statistiques afin de simuler le comportement de variables aléatoires, ou en modélisation. Cependant son utilisation est largement déconseillée à des fins cryptographiques car fortement inappropriée. En effet si un attaquant arrive à recevoir un échantillon suffisamment grand de nombres générés par Mersenne Twister il a alors une chance de prédire les futures générations. Ce qui représente une violation des propriétés que doivent suivre les générateurs aléatoires sécurisés utilisés en cryptographie. Un simple programme Python disponible via le lien suivant sur github montre à quel point l'utilisation de ce générateur est dangereuse à des fins cryptographiques :

- <https://github.com/kmyk/mersenne-twister-predictor>

L'exemple suivant montre comment en fournissant 624 nombres codés sur 32 bits, générés avec la bibliothèque *random* (donc avec un générateur dit "Mersenne Twister") on arrive à prédire les tirages suivants.

predict.png

Même s'il ne s'agit ici que d'une prédiction sur des petits nombres aléatoires de taille 32 bits, l'exemple ci dessus montre la dangerosité d'utiliser un tel générateur aléatoire à des fins de sécurité. Heureusement, python fournit d'autres bibliothèques permettant de générer de l'aléa. Notamment le module *secrets* qui est un Générateur de nombres pseudo-aléatoires Sécurisés au sens Cryptographique. Ce module bien qu'un peu plus lent pour la génération que le module *random*, offre des garanties de sécurité que ne peut atteindre ce dernier. D'après la documentation python, *secrets* est un module permettant de générer des nombres aléatoires forts, adaptés à la gestion des secrets tels que l'authentification des comptes ainsi que la gestion des mots de passe et des jetons de sécurité. Le principe de cette

librairie repose sur l'aléa généré par le système d'exploitation de la machine sur laquelle on travaille, plus particulièrement sur la fonction *os.urandom* qui renvoie une suite d'octet via une source d'aléa spécifique à chacun des ces OS. Toujours d'après la documentation Python, les sorties de cette fonction sont imprédictibles contrairement à celles des fonctions du module *random*, donc plus adaptées pour la crypto. Cependant la bibliothèque *secrets* n'est pas disponible avec le logiciel SageMath lorsque ce dernier utilise Python 2, le module n'est accessible qu'à partir de la version 9 de SageMath qui utilise Python 3. Nous avons donc implémenté notre propre générateur d'aléa, en nous inspirant de la fonction *randbelow*, et donc en utilisant la classe *SystemRandom* qui elle même appelle la fonction *os.urandom()* afin de construire des nombres aléatoires surs.

1.2 Les tests de Primalité

1.2.1 Les tests implémentés :

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 certain rang que l'on appellera B déterminé de manière expérimentale, avant de passer à un test de primalité comme celui expliqué 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.

Le Test de Miller-Rabin On ne connaît 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 d'abord 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 $i = 0, 1, \dots, s-1$ 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 $\text{Card } a \in \left\{ (\mathbb{Z}/N\mathbb{Z})^*, a^t \equiv 1 \pmod{N} \text{ ou } a^{2^i t} \equiv -1 \pmod{N} \text{ pour un } 0 \leq i \leq s-1 \right\} \leq \frac{\phi(N)}{4}$. En itérant donc k fois l'algorithme de Miller-Rabin, on obtient donc une probabilité $\leq 1/4^k$ 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 principalement utilisés à des fins théoriques.

1.2.2 Les tests fournis par Sage

La bibliothèque d'arithmétique de Sage fournit deux types de test de primalité. Le premier test s'utilise via la méthode `is_prime()`, il permet de prouver la primalité d'un entier n , il n'est cependant pas assez performant pour être utilisé pour le genre d'algorithme que l'on souhaite utiliser pour ce projet. En effet si l'on souhaite générer des premiers surs, on se servira de l'algorithme de Maurer, permettant de fournir des premiers prouvables, en

un temps beaucoup plus intéressant que si on utilisait ce test sur un grand nombre aléatoire.

Le second test est un test de pseudo-primalité, qui retourne avec une très forte probabilité la primalité ou non d'un nombre de manière très efficace. Ce test est le test de Baillie-PSW, une combinaison du test de Miller-Rabin, de Divisions Successives et du test de Lucas-Lehmer. Ce test étant le plus efficace que nous ayons pu avoir, il est donc utilisé par défaut sur chacun des algorithmes que l'on a implémentés.

Dans le cadre ce projet tutoré nous utiliserons donc soit le test de primalité de Miller-Rabin, soit le test de pseudo-primalité de Sage pour nos algorithmes. Nous comparerons l'efficacité de nos résultats (en temps) entre ces deux tests.

2 Algorithmes de Génération de grands nombres premiers

2.1 Génération "naïve"

Le premier algorithme de génération dit "naïf" 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 comme nous le montrerons dans la partie 3 de ce rapport, cette génération ne peut pas être considérée comme aléatoire. De plus le nombre de tests de primalité que l'on doit effectuer augmente exponentiellement par rapport à la taille des nombres que l'on veut générer.

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 irrégulière et qu'ils se raréfient dès qu'ils grandissent, on sait que leur répartition vérifie plus ou moins une distribution continue.

On peut donc penser qu'en tirant un nombre au hasard un certain 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) :

[scale=0.75]RS.png

2.3 Génération de nombres premiers forts : Algorithme de Gordon

Un deuxième algorithme proposé par le livre "Handbook of Applied cryptography" permet de générer des nombres premiers mais cette fois ci avec des caractéristiques supplémentaires, qui permettraient donc d'être utilisés de manière plus sécurisées notamment pour 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

Algorithm 1 Algorithme de Gordon pour la génération de premiers forts

Require: Un entier "k".

Ensure: Un premier codé sur k-bits.

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

La génération de tels nombres premiers nécessite donc un peu plus de temps que l'algorithme précédent mais elle fournit des nombres premiers qui sont mieux "protégés" des algorithmes de factorisation de $p - 1$ et $p + 1$. En effet l'intérêt 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 $d = \text{pgcd}(a, n)$ 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 $x = a^m \bmod n$, si $d = \text{pgcd}(x - 1, n) > 1$ alors d est un diviseur non-trivial de n . Nous reviendrons plus tard lors de ce rapport sur la question du réel intérêt de travailler sur de tels premiers.

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 première 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 .

Algorithm 2 Algorithme proposé par le NIST pour la génération de clés DSA

Require: Un entier l , $0 \leq l \leq 8$

Ensure: Un premier q de 160-bits, et un premier p de longueur L où $L = 512 + 64l$

1 - Calculer $L = 512 + 64l$ en utilisant les divisions longues de $(L - 1)$ par 160, trouver n, b tels que $L - 1 = 160n + b$

2 - **while** q n'est pas premier

- Tirer une graine aléatoire s de longueur g_l 160
- Calculer $U = H(s) \oplus H((s+1) \bmod 2^g)$
- Former q à partir de U , en initialisant à 1 le bit le plus fort ainsi que le bit le moins fort.
- Tester la primalité de q

3 - Initialiser : $i = 0, j = 2$

4 - **while** $i < 4096$:

- **for** k allant de 0 à n :
 - $V_k = H((s + j + k) \bmod 2^g)$
 - $W = V_0 + V_1^{160} + V_2^{320} + \dots + V_{n-1}^{160(n-1)} + (V_n \bmod 2^b) 2^{160n}$
- Calculer $c = X \bmod 2q$ et initialiser p à : $p = X - (c - 1)$
- **if** $p \geq 2^{L-1}$:
 - **if** p est premier :
 - return** (p, q)

5 - Retourner à l'étape 2.

2.5 Premiers prouvables : Algorithme de Maurer

Le prochain algorithme est spécial dans le sens où il est le seul de notre liste d'algorithmes capable de fournir des nombres premiers surs, prouvables. Son temps d'exécution est légèrement 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 repose sur un théorème qui est un raffinement du théorème de Pocklington et dont voici les bases mathématiques.

Les maths de l'algorithme de Maurer (provable primes) : On souhaite dans cette sous-partie prouver la correction de l'algorithme de Maurer, c'est à dire qu'il nous renvoie bien des premiers prouvables. Nous devons premièrement donner quelques résultats utiles afin de le démontrer.

Définition 1 : L'ordre d'un élément g d'un groupe G est le plus petit m tel que $g^m = e$, où e est l'élément neutre du groupe.

Nous travaillons dans le groupe $(\mathbb{Z}/n\mathbb{Z})^*$, il en résulte donc que $g^m = 1 \pmod{n}$

Théorème 2 : Si G est un groupe multiplicatif d'ordre n , et $g \in G$, alors l'ordre de g appelé m divise n . (Théorème de Lagrange).

Corollaire : Si $b \in (\mathbb{Z}/n\mathbb{Z})^*$, alors $b^{\phi(n)} = 1 \pmod{n}$

Théorème 3 : (Théorème de Lucas) : Soit $n > 1$, s'il existe un facteur premier q de $n - 1$ tel que :

$$\begin{aligned} - a^{n-1} &\equiv 1 \pmod{n} \\ - a^{(n-1)/q} &\not\equiv 1 \pmod{n} \end{aligned}$$

Alors n est premier.

Théorème 4 : (Théorème de Pocklington) :

Soit $n - 1 = q^k * R$ où q est un nombre premier tel que q ne divise pas R . S'il existe un entier a tel que :

$$\begin{aligned} 1) \quad &a^{n-1} \equiv 1 \pmod{n} \\ 2) \quad &\text{pgcd}(a^{(n-1)/q} - 1, n) = 1 \end{aligned}$$

Alors chaque facteur premier p de n est de la forme $p = q^k * r + 1$ où r est un entier.

Les Théorèmes 3 et 4 peuvent être prouvés assez rapidement en utilisant la Définition 1, ainsi que le Théorème 2 et le Corollaire.

Le Théorème 4 limite nos possibilités étant donné que seul les composés de la forme $q^k * R$ peuvent être prouvés. C'est pourquoi on doit étendre ce résultat au cas général où $n - 1 = F * R$, avec $F > R$ et $\text{pgcd}(R, F) = 1$.

Théorème 5 : Soit $n - 1 = F * R$, où $F > R$ (R est donc $< \sqrt{n}$), $\text{pgcd}(F, R) = 1$ et la factorisation p_1, p_2, \dots, p_m de F est connue. Si pour tous les p_i , il existe un $a > 1$ tel que :

$$\begin{aligned} - a^{n-1} &\equiv 1 \pmod{n} \\ - \text{pgcd}(a^{(n-1)/p_i} - 1, n) &= 1 \end{aligned}$$

alors n est premier.

La preuve de ce théorème résulte directement de celle des théorèmes 3 et 4 en se rappelant que q ne divise pas R , il suffit maintenant de le montrer pour tous les facteurs premiers de F .

Le théorème de Maurer se découpe donc en 2 phases, une qui utilise les divisions successives (pour $k < 20$ pour k le nombre de bits du premier souhaité), l'autre qui utilise le Théorème 5.

Ainsi on obtient le certificat du nombre n généré par l'algorithme de Maurer : le triplet (R, F, a) ainsi que la factorisation de F où $n = 2RF + 1$.

Ci dessous l'algorithme en pseudo-code ?

Algorithm 3 Algorithme de Maurer pour la génération de premiers prouvable

Nom de l'algorithme pour l'appel récursif :

PremierProuvable(k)

Require: Un entier "k".

Ensure: 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 - **while** $(k - rk) > m$:

if $k > 2m$:

 - Tirer un réel s aléatoire tel que : $s \in [0, 1]$

 - Calculer $r = 2^{s-1}$

else :

 - Initialiser : $r = 0.5$

5 - Calculer : $q = \text{PremierProuvable}(k)$

6 - Initialiser : $I = \lfloor 2^{n-1}/2q \rfloor$

7 - Initialiser : $\text{success} = 0$

8 - **while** $\text{success} = 0$

 - 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 .

if n n'a pas de facteurs premiers inférieurs à B :

 - Tirer un entier aléatoire $a \in [2, n - 2]$

 - Calculer $b = a^{n-1} \bmod n$

 - **if** $b = 1$:

 - Calculer $b = a^{2R} \bmod n$ et $d = \text{pgcd}(b - 1, n)$

 - **if** $d = 1$:

$\text{success} = 1$


9 - **return** 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 sur certains algorithmes cités précédemment.

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 à 2^{32} . Ensuite nous utilisons notre générateur (la fonction *GenerationAverageTime* du fichier *générateur.py*) afin de connaître le temps moyen pour générer 1000 chiffres premiers avec l'algorithme de notre choix en faisant varier le paramètre B . Nous traçons les résultats et obtenons la courbe suivante :  [totalheight=0.3]DS.png Après de nombreux tests, on trouve qu'effectuer environ 2^{12} soit 4096 divisions successives permet d'optimiser le temps de retour des tests de primalité, et donc d'optimiser les temps de génération de premiers. Ces expérimentations nous amènent donc à prendre comme valeur 2^{12} pour B . Ce paramètre est donc initialisé par défaut à 2^{12} sur les fonctions qui nécessitent d'utiliser l'algorithme des divisions successives (les paramètres des fonctions par défaut sont faits pour générer des nombres de 1024bits).

3.2 Temps Moyen de génération

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

3.2.1 Avec un test de Miller-Rabin

Temps moyen en secondes sur 1000 générations avec Miller-Rabin

	Random-Search	Gordon	Maurer
256 bits	0.00448	0.003017	0.02229
512 bits	0.013274	0.016414	0.08730
1024 bits	0.13308	0.13318	0.411052
2048 bits	0.694861	1.50748	2.38364

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	0.0122	0.00825	0.052886
1024 bits	0.07620	0.08341	0.254052
2048 bits	0.41064	1.079873	1.55760

Comme en atteste les tableaux ci contre ainsi que l'exemple de la courbe ci dessous, les temps de génération sont meilleurs avec le test de pseudo-primalité fourni par Sage, ce qui nous permet de dire que le test de Baillie-PSW est très efficace. D'après la documentation, ce test mélange test de Miller-Rabin de niveau 2 et le test de lucas-lehmer, la probabilité que les générateurs retournent un nombre composé en utilisant ce test de pseudo-primalité est négligeable, toujours d'après la documentation il n'existe pas d'exemple de ce type d'entier composé ayant passé le test.

[totalheight=0.3]IPPRS.png

Exemple de comparaison des temps de génération avec Miller-Rabin et is-pseudoprime de Sage

3.3 Factorisation

Avec la fonction `factor()` de Sage : Dans 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 à priori 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 à 10s) pour des entiers codés sur 70/80bits).

Nous avons donc écrit 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.

Comme en atteste la courbe ci dessous les temps de factorisation pour des modules de petites tailles (avec p et $q < 70bits$) sont sensiblement identiques. Par contre dès lors que l'on génère des modules avec p et $q > 2^{80}$ on observe une différence non négligeable entre les temps de factorisation des différents modules. Il semble en effet que les modules construits avec gordon soit plus lents à factoriser que les modules faits avec random-search. Cet écart peut s'expliquer par la méthode de factorisation utilisée par Sage lors de l'appel à la fonction `factor()`. D'après la documentation, cette fonction implémente une combinaison de plusieurs méthodes de factorisation,

telles que la méthode Rho de Pollard, la méthode des formes quadratiques de Shanks, celle des courbes elliptiques de Lenstra ainsi qu'une recherche des puissances pures, elle renvoie ensuite des facteurs qui ont passé le test de pseudo-primalité de BPSW *is - pseudoprime* cité plus haut. La méthode des courbes elliptiques ainsi que la méthode rho de Pollard sont très efficaces si soit p soit q sont de petites tailles, générer un module RSA avec p et q assez grands garantit donc la sécurité du module via une attaque par une de ces factorisations, le fait que $(p-1)$ et $(p+1)$ aient des facteurs premiers n'intervient donc pas ici. La vérification de la primalité du facteur trouvé peut être un élément de réponse quant à la question du temps de factorisation des modules 'gordon'. En effet pour que le facteur trouvé noté p passe le test de BPSW, et notamment le test de Lucas, il faut trouver un entier a tel que :

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

et prouver que pour tout diviseur premier q de $p - 1$ on ait :

$$a^{(p-1)/q} \not\equiv 1 \pmod{p}$$

De telles vérifications semblent donc plus coûteuses lorsque $p - 1$ n'a pas de petit facteur premier, ce qui expliquerait la perte de temps constatée.

factr.png

Avec la fonction qsieve : Sage permet en plus de sa fonction *factor()*, l'utilisation de la méthode de factorisation du crible quadratique grâce à sa fonction *qsieve()*. Cette méthode du crible quadratique est reconnue comme étant la méthode la plus rapide pour factoriser $n = p * q$ lorsque p et q sont de tailles semblables, après le crible sur corps de nombres. L'algorithme du crible quadratique est un raffinement de la l'algorithme de factorisation de Dixon, lui même étant basé sur l'algorithme de Fermat.

Pour tester la factorisation d'un entier n , les tests de Fermat et de Dixon cherchent des entiers x et y tels que :

$$x^2 \equiv y^2 \pmod{n} \text{ et } x \not\equiv y \pmod{n}$$

on aurait alors :

$$x^2 - y^2 \equiv 0 \pmod{n},$$

donc :

$$(x - y)(x + y) \equiv 0 \pmod{n}.$$

Et donc n aurait comme facteurs non triviaux $f1 = \text{pgcd}(n, x + y)$ et $f2 = \text{pgcd}(n, x - y)$.

L'utilisation de telles techniques de factorisation ne font pas intervenir de bornes de friabilité de $p - 1$, on ne voit donc pas vraiment l'avantage de fabriquer des nombres premiers forts face à de telles factorisations, même si en moyenne sur nos tests expérimentaux, on trouve que la fonction *qsieve* est légèrement moins efficace pour factoriser des modules faits avec des premiers forts. Cette perte d'efficacité peut peut-être s'expliquer, comme pour

la fonction *factor* par le test de pseudo-primalité effectué par *qsieve* avant de retourner les facteurs pseudo-premiers.

Conclusion sur l'utilité des premiers forts de Gordon : Même si à première vue il semble que la méthode *factor* de Sage soit plus performante sur des modules générés avec un simple générateur aléatoire qu'avec l'algorithme de Gordon (avec p et q de tailles avoisinant les 100bits), il est impossible d'affirmer que ces premiers "forts" soient réellement utiles à des fins de sécurité contre les attaques par factorisation. Certains algorithmes tels que le crible sur corps de nombres, ou même le crible quadratique permettent de factoriser avec la même efficacité les deux types de modules cités précédemment. Pour garantir la sécurité du système RSA, il semblerait donc que rien ne soit plus efficace que de choisir p et q suffisamment grands afin de rendre le nombre de calculs beaucoup trop grand par rapport à ce que peuvent réaliser les ordinateurs utilisés de nos jours. Le "handbook" datant de prêt de 20 ans, il semblerait qu'à l'époque les attaques de factorisation de $(p-1)$ étaient les méthodes les plus en vues, ce qui expliquerait que l'on y préconise l'utilisation de premiers forts. Il se pourrait aussi qu'en essayant de protéger les modules RSA contre certaines attaques telles que les attaques par $(p-1)$, on les affaiblisse contre de nouvelles plus efficaces, qui pourraient exploiter les propriétés connues des premiers forts.

3.4 Distribution aléatoire

Dans cette sous-partie nous comparons les distributions obtenues avec l'utilisation d'un générateur naïf (Algorithme de la partie 2.1) et le générateur random-search. Le but étant de montrer que la distribution du générateur naïf n'est pas aléatoire et que certaines valeurs ont plus de chances de sortir que d'autres, impliquant donc une perte de qualité quant à la sécurité des nombres générés.

Distribution de la Génération naïve : De manière intuitive, on se rend bien compte que la génération naïve ne peut être avoir une distribution aléatoire, car l'écart entre deux nombres premiers influe sur la probabilité de générer un de ces premiers. En effet si deux nombres premiers p_1 et p_2 sont très proches l'un de l'autre, par exemple si ils sont jumeaux alors on a $p_1 = p_2 + 2$ en utilisant l'algorithme naïf on a alors que très peu de chances de tomber sur p_2 , la seule possibilité pour que l'algorithme nous retourne p_2 est que l'on soit tombé sur p_2 à la première étape de cet algorithme, i.e que la fonction *randombelow* tombe directement sur p_2 .

Si on note $l=[p_1, p_2, \dots, p_n]$ la liste ordonnée des premiers jusqu'au n -ième premier p_n . Alors la différence entre p_k et p_{k-1} (avec $k \in N$) influe sur la probabilité de tomber sur p_k . Pour illustrer cela nous avons généré 100 000

000 de premiers $p < 2^{18}$ tels que $p \in [25000, 30000]$ On obtient alors la distribution suivante :

[totalheight=0.3]distrib-naiveGen.png
générateur naïf

Distribution du

Le pic que l'on aperçoit représente le premier 31469 qui a été généré plus de 54 000 fois. Cet exemple illustre bien le caractère non aléatoire d'un tel générateur, car en effet le premier "avant" 31469 est 31397, l'écart entre ces deux premiers est l'écart le plus important entre deux premiers de notre échantillon. A titre comparatif, 31513 a été généré seulement 1592 fois, car il a un premier jumeau (31511) qui le précède. Il a donc été atteint environ 36 fois moins souvent que le pic maximum. La probabilité d'atteindre un premier avec l'algorithme naïf est donc proportionnelle à l'écart qui existe entre ce premier est celui qui le précède.

Distribution de la Génération avec Random-Search Cet algorithme est censé être complètement aléatoire, étant donné qu'il ne contient qu'une boucle qui génère un entier, qui teste sa primalité et qui s'arrête et renvoie cet entier si le test est positif. Contrairement à la génération naïve nous n'avons pas pu générer 10 000 000 de nombres $< 2^{18}$ car c'était beaucoup trop coûteux en stockage et en recherche sur les dictionnaires Python que nous utilisons afin de tracer des histogrammes. En effet quand un premier est généré avant de l'ajouter à notre structure de données qui est le dictionnaire, on vérifie si notre élément est présent, donc lorsque des mêmes premiers sont générés les tests sont rapides, et lorsque ces premiers sont tous différents les uns des autres les tests sont beaucoup plus lents. Nous avons donc effectué le test sur un échantillon de 1 000 000 de premiers. Nous obtenons alors une distribution qui semble aléatoire, dont l'histogramme entre [25000, 35000] est le suivant :

[totalheight=0.3]RS-distrib.png

Distribution de Random search

Distribution de la Génération des premiers forts de Gordon Alors que sur un million d'essais de génération de premiers de taille 18-bits avec random-search on obtenait une liste de pratiquement 1000 premiers entre 25000 et 35000, avec l'algorithme de gordon la liste obtenue est dix fois moins longue. Même si la génération des nombres premiers avec l'algorithme de Gordon semble aléatoire, le peu de nombres que peut retourner cet algorithme semble être une faiblesse. En effet si un attaquant sait que l'algorithme de Gordon est appliqué pour la génération des clés RSA, qu'il a la taille du module, il peut alors dresser une liste de premiers intéressant à tester par leur propriétés de premiers forts. Encore une fois la sécurité est

accrue dès lors que les tailles augmentent. S'ils existent 100 premiers forts entre 25 000 et 35 000 on peut se douter qu'il en existe une quantité suffisamment importante entre 2^{2040} et 2^{2050} (exemple d'intervalle qui correspond à peu près à des tailles de modules RSA avec p et q de 1024 – *bits*) pour qu'un ordinateur ne puisse les stocker, et donc qu'un attaquant n'exploite cette faiblesse.

4 Conclusion

Nous avons pu lors de ce Projet tutoré étudier le comportement des algorithmes de générations de premiers aléatoires fournis par le "Handbook of Applied Cryptography" en fonction des tests de primalité utilisés. Les résultats que nous obtenons avec nos propres tests sont cohérents et les temps de génération sont satisfaisants. Nous avons axé nos recherches et nos tests expérimentaux autour des modules RSA, afin de faire un lien entre ce Projet et le cours d'Arithmétique et Cryptologie que nous avons suivi ce semestre. Les tests expérimentaux que nous avons pu faire sont évidemment limités par la capacité des ordinateurs que nous utilisons (ordinateurs personnels, de bureau), c'est pourquoi les résultats obtenus ainsi que les conclusions que nous en tirons sont à relativiser. Les conclusions notamment quant aux temps de factorisation de modules RSA sont tirées de tests expérimentaux sur des tailles petites (120 bits pour p et q au maximum), on ne peut donc pas vraiment tirer de conclusions quant au comportement et à l'efficacité de certains de ces algorithmes sur des tailles plus importantes. L'utilisation des fichiers sur notre Github dont le lien est disponible ci-dessous donne des fonctions permettant la génération de nombres premiers sûrs, en des temps honorables, qu'ils utilisent le test de Sage ou le test de Miller-Rabin que nous avons implémenté, bien que celui ci reste légèrement moins performant.

<https://github.com/Pierre-Graber/Projet-Tutor-/>

Sources

Menezes A, Oorschot PV, Vanstone S (1997) Handbook of applied cryptography. CRC, Boca Raton

Ueli M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. J. Cryptology, 8(3) :123–155, 1995. *Ueli M. Maurer.* Fast Generation of Secure RSA-Moduli with Almost Maximal Diversity. In Advances in Cryptology - EUROCRYPT '89, pages 636–647

Ronald L Rivest Robert D Silverman Are Strong Primes Needed for RSA, 1999

<http://doc.sagemath.org/>

<https://pari.math.u-bordeaux.fr/dohtml/html/>