Introduction à la cryptographie
Devoir Maison – Test de Miller-Rabin

À rendre pour le 4 Mars 2019
M1, Université de Lorraine, 2018/2019
Marine Minier, `marine.minier@loria.fr`

1 Introduction

Ce devoir est à réaliser par groupe de deux étudiant-e-s. Au plus, un seul trinôme sera accepté.

Le rendu se fera sur ARCHE et est en deux parties :

- Un rapport dactylographié au format PDF contenant les réponses aux questions posées dans le sujet. Il vous est demandé de rédiger correctement ces réponses, de détailler votre raisonnement et de justifier vos affirmations.
- Les programmes que vous avez développés. Le `main` de votre programme devra suivre le format de celui décrit en Annexe B. La sortie de votre programme devra être assemblée dans un unique fichier appelé `test.txt`.
- Vos programmes devront compiler une machine de salle de TP (2 points).
- Vous devez fournir un `README` de compilation de vos codes et de vos différents fichiers ainsi qu'un `MAKEFILE` (2 points).

L'ensemble des fonctions sera développé dans le langage de votre choix.

Les objectifs de ce DM sont les suivants :

- Comprendre le fonctionnement du test de Miller-Rabin.
- Implémenter ce test.
- Tester votre implémentation.

2 Introduction générale sur le test de Miller-Rabin

Nous avons vu que de nombreux protocoles cryptographiques nécessitent de grands nombres premiers pour être mis en place. Mais comment fait-on pour produire des nombres premiers ? L'objet de ce DM est le test de primalité de Miller-Rabin, qui permet de déterminer très efficacement si un entier est composé ou "probablement premier". Notons cependant que ce test est probabiliste.

Remarques :

- Dans tous les calculs faits, il faut effectuer la réduction modulo p après chaque opération élémentaire. En particulier, lors de l'exponentiation modulaire, il faut réduire modulo p après chaque étape de l'exponentiation binaire (voir la decription de cette méthode à l'Annexe A).

2.1 Test de Fermat

Le test de Fermat est basé sur le petit théorème de Fermat qui dit, rappelons-le, que pour tout nombre premier p et pour tout $a \not\equiv 0 \mod p$,

$$a^{p-1} \equiv 1(\mod p).$$

Cette propriété est caractéristique des nombres premiers, et il y a peu de chances en général qu'elle soit vérifiée par d'autres nombres. Supposons qu'on ait un nombre n dont on ne sait pas s'il est premier ou composé. Pour le vérifier on peut choisir un entier a au hasard et vérifier si $a^{n-1} \not\equiv 1(\mod n)$.

Si tel est le cas, on conclut que n n'est pas premier ; sinon on ne peut rien dire, mais on affirme que n a une chance d'être premier.

Par exemple, voici la valeur de $a^{14} \bmod 15$ pour a allant de 1 jusqu'à 14 :

$$1, 4, 9, 1, 10, 6, 4, 4, 6, 10, 1, 9, 4, 1.$$

Il n'y a donc que les valeurs $a = 1, 4, 11, 14$ qui risquent de nous induire en erreur.

Si un entier a est tel que $a^{n-1} \not\equiv 1 \mod n$, on dit que a est un "témoin de composition" pour n. Si n n'est pas premier et a est tel que $a^{n-1} \not\equiv 1 \mod n$, on dit que a est un "menteur" pour n. Remarquez que 1 et $n-1$ sont des mauvais choix : le premier est toujours un menteur, le deuxième est un menteur pour tout n impair.

Quelques répétitions du test de Fermat sont en général suffisantes à décider correctement de la primalité d'un nombre. Il existe cependant des nombres, dits de Carmichael, pour lesquels tout entier est un menteur. Le plus petit nombre de Carmichael est $561 = 3 \cdot 11 \cdot 17$.

2.2 Test de Miller-Rabin

Le test de Miller-Rabin est un raffinement du test de Fermat. En plus d'exploiter le petit théorème de Fermat, il exploite une propriété supplémentaire des nombres premiers. Pour tout entier n, premier ou non, il est toujours vrai que $1^2 \equiv -1^2 \equiv 1(\mod n)$. Ce qui, par contre, est vrai seulement pour les nombres premiers p c'est que -1 et 1 sont les deux seules racines carrées de 1. Par exemple on a $1^2 \equiv -1^2 \equiv 4^2 \equiv 11^2 \equiv 1(\mod 15)$.

Étant donné un entier n à tester, le test de Miller-Rabin procède comme suit :

1. On écrit $n - 1 = 2^s d$ avec d impair ;
2. On choisit $1 < a < n - 1$ au hasard ;
3. On calcule $a^d \bmod n$: si c'est égal à 1 ou -1 on ne peut rien dire et on arrête ;
4. Pour i allant de 1 jusqu'à s on calcule $a^{d2^i} \bmod n$:
   — si $a^{d2^i} \equiv -1 \mod n$ on ne peut rien dire et on arrête ;
   — si $a^{d2^i} \equiv 1 \mod n$ on a trouvé une racine carrée de l'unité différente de 1 et -1, on conclut que n est composé et on arrête ;
5. Si on est arrivés à la fin de la boucle et que $a^{d2^s} \not\equiv 1(\mod n)$, on conclut que n est composé, comme d'après le test de Fermat.

Le test de Miller-Rabin est très efficace : on peut montrer que pour tout n la probabilité qu'un a pris au hasard soit un menteur est au plus $1/4$ (et souvent en pratique beaucoup plus faible). Si on répète k fois le test de Miller-Rabin (avec des a différents au hasard) la probabilité d'identifier erronément un nombre composé comme étant premier est bornée par $1/4^k$.

Dans les applications pratiques on estime souvent qu'une vingtaine de répétitions du test de Miller-Rabin suffisent à affirmer avec confiance qu'un nombre est premier. Notez, tout de même, qu'il existe d'autres algorithmes qui permettent de certifier qu'un nombre est premier !

3 Avant d'implémenter le test de Miller-Rabin

Le but de ce devoir maison est donc d'implémenter le test de Miller-Rabin.

3.1 Implémentation de fonctions élémentaires

Nous allons y aller pas à pas. Il faut que vous commenciez par choisir un langage de programmation. Une fois ce langage choisi, comme nous allons travailler avec des nombres de très grande taille, bien plus grande que les longueurs standards utilisées (64 ou 128 bits), il faut trouver la bibliothèque de grands nombres adaptée au langage choisi. Par exemple, si vous décidez de programmer en C, la bibliothèque GMP [1] permet de travailler en multi-précision sur des nombres entiers de grande taille.

Question 1. Quel langage de programmation avez-vous choisi ? Quelle bibliothèque permettant de gérer des nombres entiers de grande taille allez-vous utiliser ? Quelles sont les opérations implémentées dans cette bibliothèque (multiplication, addition, etc.) ?

De plus, pour générer des valeurs de n au hasard, il faut être capable de générer des nombres aléatoires de très bonne qualité.

Question 2. En vous aidant d'Internet, donnez la définition d'un nombre aléatoire. Selon le langage de programmation choisi, donnez le nom de la bibliothèque qui va vous permettre de générer ces nombres aléatoires.

Il est à noter que nous aurons besoin de nombres de plusieurs milliers de bits, il faut donc s'assurer que le générateur de nombres aléatoires utilisé produit une chaîne binaire aléatoire suffisamment longue. Soit il le fait par défaut, soit vous avez besoin d'implémenter une fonction intermédiaire `BigRandom` qui prend en entrée plusieurs petits nombres aléatoires et produit en sortie un nombre aléatoire de la très grande taille demandée.

3.2 Décomposition de $(n-1)$

Question 3. Implémentez la fonction `Decomp()` qui prend en entrée n et renvoie s et d tel que $n-1 = 2^s d$ avec d impair. Testez-la sur 10000 valeurs différentes.

3.3 Implémentation de l'exponentiation binaire

Cette méthode rapide d'exponentiation modulaire est décrite dans l'Annexe A. Il s'agit donc d'implémenter ici cette méthode sous la forme d'une fonction `ExpMod()` qui prend en entrée n, a et t et qui

1. voir : `https://gmplib.org/`.

renvoie en sortie $a^t \bmod n$.

Question 4. Implémentez la fonction `ExpMod()`. Testez-la sur 10000 valeurs différentes.

4 Implémentation et tests du test de Miller-Rabin

Question 5. Implémentez le test complet de Miller-Rabin `MillerRabin()` qui prend en entrée n et un nombre d'itérations du test *cpt*. Cette fonction répète le test décrit à la Section 2.2 *cpt* fois et renvoie ensuite '0' si le nombre est composé et '1' si le nombre est probablement premier.

Question 6. Utilisez votre fonction pour tester les 3 nombres suivants (écrits en hexadécimal). Dites si chaque nombre est pseudo-premier ou composé avec $cpt = 20$.

```
n1 (768 bits) =
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A63A3620 FFFFFFFF FFFFFFFF
```

```
n2 (768 bits) =
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FEC4FFFF
FDAF0000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 0002D9AB
```

```
n3 (1024 bits) =
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE65381
FFFFFFFF FFFFFFFF
```

Question 7. Ecrivez une fonction `Eval()` qui prend en entrée une taille en bits b, le compteur *cpt* et donne en sortie le nombre d'itérations qu'il a fallu répéter avant de trouver un nombre probablement premier.

```
Entrée: b, cpt
Compteur = 0
Tirez un n au hasard de b bits
Tant que (MillerRabin(n, cpt) = composé)
        Compteur=Compteur+1
        tirez un nouvel n au hasard de b bits
Fin Tant que
Retournez Compteur
```

Question 8. Pour tester votre fonction `Eval()`, répétez 100 fois la fonction pour les valeurs de b suivantes : $128, 256, 512, 1024, 2048$ et 4096. Faites un graphique avec en abscisse la taille en bits du nombre et en ordonnée la moyenne sur les 100 tests de la valeur de Compteur.

Question 9. Que constatez-vous ?

Question 10. Le test de Miller-Rabin est un test probabiliste, c'est-à-dire qu'il donne la réponse "pseudo-premier" avec une probabilité de se tromper. Cependant il existe un test qui permet de garantir si un nombre est premier ou non. En vous aidant d'Internet, pouvez-vous donner le nom de ce test et sa complexité ? Je ne demande pas de comprendre ce que fait ce test.

A Calculer une exponentiation modulaire à l'aide de la méthode dite d'exponentiation binaire

Lorsque l'on cherche à calculer a^t, la méthode naïve consiste à multiplier a par lui-même t fois. Ce n'est pourtant pas la méthode la plus rapide. La meilleure méthode consiste à décomposer t en écriture binaire $t = \sum_{i=0}^{r-1} t_i 2^i$ où r est la taille en bits de t et où $t_i \in \{0, 1\}$. On a alors : $a^t = a^{t_0}(a^2)^{t_1} \cdots (a^{2^{r-1}})^{t_{r-1}}$. Il faut ainsi $r-1$ opérations pour calculer les a^{2^i} et $r-1$ opérations pour former le produit des $(a^{2^i})^{t_i}$, soit, au total, $2 \cdot r - 2$ opérations. Ainsi, l'algorithme le plus rapide pour calculer une exponentiation est le suivant dans sa version récursive qui calcule a^t pour t strictement positif :

$$\texttt{puissance}(a,t) = \begin{cases} a, & \text{si } t{=}1 \\ \texttt{puissance}(a^2, t/2), & \text{si } t \text{ est pair} \\ a \times \texttt{puissance}(a^2, (t-1)/2), & \text{si } t > 2 \text{ est impair} \end{cases}$$

Attention : Cet algorithme correspond à une élévation à la puissance classique. Comme nous souhaitons ici travailler modulo n pour calculer $a^t \bmod n$, tous les calculs de l'algorithme récursif doivent être faits modulo n.