Introduction à la cryptographie
Devoir Maison – C'est quoi l'aléa ?

A rendre pour le 8 Avril 2022
L3, Université de Lorraine, 2021/2022
Marine Minier, `marine.minier@loria.fr`

1 Introduction

1.1 Indications Générales

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 A. La sortie de votre programme devra être assemblées dans un unique fichier appelé `test.txt`.
- un fichier `README.txt` expliquant comment compiler vos codes. Vos codes doivent compiler sur une machine de TP.

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

1.2 But du TP

Les objectifs de ce TP sont les suivants :

- Faire passer des tests statistiques à des générateurs pseudo-aléatoires (via trois tests de qualité de séquences) afin de comparer la qualité des séquences produites. Vous aurez à programmer vous-même deux générateurs simples.
- Transformer la distribution uniforme sur $[0, 1]$ en une distribution plus complexe.

2 Tests de générateurs pseudo-aléatoires

Le but de ce TP est de tester la qualité des séquences aléatoires produites par différents générateurs aléatoires. Nous étudierons les 4 générateurs suivants :

1. le générateur de Von Neumann,

2. un générateur à congruence linéaire usuel, dit Standard Minimal,
3. un générateur à congruence linéaire avec un choix différent des paramètres, RANDU,
4. le générateur par défaut fourni par une librairie du langage que vous utilisez.

Le principe de ces générateurs est décrit ci-après.

Afin de comparer ces générateurs, vous aurez ensuite à programmer des tests classiques de probabilité (voir Section 2.2) permettant de tester la qualité des suites produites. La qualité d'une suite sera mesurée par la probabilité que les valeurs obtenues suivent bien une loi uniforme comme cela est souhaité.

2.1 Définition d'un générateur aléatoire

Dans beaucoup d'applications informatiques (la simulation, les jeux vidéos, la cryptographie, etc.), il est nécessaire de tirer des nombres au hasard pour initialiser différents algorithmes. Pour cela, on utilise ce qu'on appelle des nombres **pseudo-aléatoires** pour souligner leur différence par rapport aux véritables suites de variables aléatoires indépendantes identiquement distribuées. Nous nous intéressons ici à quelques méthodes classiquement utilisées pour générer des nombres pseudo-aléatoires (pour plus de détails, se référer par exemple à [Ber05a]). Plus précisément, il existe deux types de générateurs :

- les générateurs à sorties imprédictibles : ils génèrent des nombres pseudo-aléatoires dont on ne peut prévoir la valeur,
- les générateurs à sorties prédictibles : dans ces générateurs, on initialise (on dit nourrir) l'algorithme à partir d'un nombre connu appelé graine et le générateur produira toujours la même suite s'il est initialisé avec la même valeur.

2.1.1 Définition

Un générateur pseudo-aléatoire est une structure $G = (S, \mu, f, U, g)$ avec :

- S un (grand) ensemble fini d'états,
- μ est une distribution de probabilité sur S (le plus souvent la loi uniforme),
- $f : S \to S$ est la fonction de transition utilisée pour passer d'un état S_i au suivant S_{i+1}.
- U est l'ensemble image de la fonction $g : S \to U$ qui fait correspondre à chaque état S_i un échantillon de U. Très classiquement, on va avoir $U = [0, 1]$ pour une loi uniforme sur [0,1].

Le générateur fonctionne donc en itérant la fonction f à partir d'un état initial S_0 appelé graine, choisi par l'utilisateur. En notant $(X_n)_{n\geq 1}$ la liste des valeurs successives produites par le générateur, on a la relation $X_n = g(f \circ \cdots \circ f(S_0))$ où f est appliquée n fois.

En pratique, le générateur ne garde en mémoire que l'état courant $S_n \in S$, initialisé à S_0 et mis à jour lors de chaque appel de f : $S_n = f(S_{n-1})$, la valeur renvoyée étant égale à $X_n = g(S_n)$.

Le choix de la graine S_0 détermine donc entièrement la suite de nombres pseudo-aléatoires produite par le générateur car une fois cette graine choisie, le comportement de la suite est complétement déterministe.

2.1.2 Les générateurs pseudo-aléatoires étudiés

Les différents générateurs étudiés sont basés sur ce principe. Ainsi dans le générateur de Von Neumann proposé en 1946, la fonction f consiste à élever le nombre S_n au carré et à ôter les premiers et les derniers chiffres, de manière symétrique, de sorte à ce que le nombre obtenu soit compris entre 0 et 9999. Par exemple si $S_n = 1315$ alors $1315^2 = 1729225$ donc $S_{n+1} = 292$.

Introduits en 1948, les générateurs à congruence linéaire ont ensuite eu beaucoup de succès. L'idée est d'appliquer une transformation linéaire suivie d'une opération de congruence :

$$S = U = \{0, \cdots, m-1\}, \quad f(S_n) = a * S_{n-1} + b \mod m, \quad X_n = g(S_n) = s/m$$

où / est la division entière. D'autre types de congruence peuvent être considérés. Par exemple, la fonction `rand` de `C` utilise par défaut une congruence de type polynomiale.

Nous étudierons dans ce DM deux générateurs à congruence linéaire. Le premier est Standard Minimal. Il s'agit du générateur défini ci-dessus avec les paramètres

$$a = 16807,\ b = 0,\ m = 2^{31} - 1.$$

Vous générerez également le générateur à congruence linéaire dit RANDU, qui prend pour paramètres

$$a = 65539,\ b = 0,\ m = 2^{31}.$$

Cette méthode ayant toutefois montré ses limites, le générateur pseudo-aléatoire par défaut de `Matlab` est actuellement Mersenne-Twister. C'est aussi le générateur par défaut du logiciel `R`. Il engendre des séquences pseudo-aléatoires de qualité satisfaisante, bien qu'il ne soit pas cryptographiquement sûr. Ce générateur ne sera pas décrit ici. Nous renvoyons à [Wikb] pour une description détaillée.

2.1.3 Graine des générateurs

Il existe beaucoup de méthodes pour générer des nombres pseudo-aléatoires autres que la congruence linéaire sur laquelle repose la fonction `rand` du langage que vous avez choisi. On peut citer les générateurs à récursivité multiple (GRM), l'algorithme Blum-Blum-Shub, etc. Pour plus de détails, le lecteur peut se référer à [Ber05a] ou à [Wika].

Si vous utilisez la fonction `rand` directement dans vos programmes sans changer la graine vous allez retomber systématiquement sur la même suite de nombres et ainsi produire une suite prédictible. Pour choisir la graine, on emploie une fonction `rng` que l'on "nourrit" à l'aide d'un nombre.

Si vous souhaitez faire une expérience reproductible, par exemple afin de comparer deux algorithmes sur deux jeux de données simulées identiques, en ce cas, il faut initialiser la graine avec un nombre fixé. Mais si le but est de générer une suite difficilement prévisible et qui varie rapidement, il faut l'initialiser avec un nombre qui varie rapidement. Par exemple, vous pouvez prendre le nombre de cycles utilisés par votre processeur depuis son démarrage. C'est par exemple le but de la fonction en C, `time(NULL)` qui renvoie le temps courant.

Question 1. Selon le langage que vous avez choisi, indiquez comment générer une suite pseudo-aléatoire, en d'autres termes, comment est défini la fonction `rand` et quel générateur implémente-t'elle ?

Question 2. Toujours selon le langage que vous avez choisi, comment définir une bonne fonction `rng` et qu'utilise cette fonction `rng` ?

2.1.4 Génération de séquence avec les 4 générateurs étudiés

La majorité des techniques de simulations de lois de probabilité génère des entiers sur un intervalle $\{0, \ldots, m\}$. Cependant en pratique il est plus souvent utile de faire appel à une loi uniforme sur l'intervalle [0, 1]. Cette loi est aussi fort utile dans la génération d'autres lois de probabilités. On pourra remarquer que si U suit une loi uniforme sur l'intervalle $[0, N]$ alors U/N suit une loi uniforme sur l'intervalle [0, 1].

On générera uniquement les valeurs sur $\{0, \ldots, 2^{31} - 1\}$ dans cette partie.

Question 3. Implémentez une fonction `VonNeuman` qui prendra comme paramètre la graine associée au générateur et retournera la valeur courante obtenue par l'algorithme de Von Neumann (*i.e.* retournant le carré de la graine auquel on a ôté le premier et le dernier chiffre, puis qu'on a renormalisé).

Question 4. Implémentez également la fonction de congruence linéaire Standard Minimal. On appelera la fonction `STM`, et elle prendra pour paramètre la graine du générateur.

Question 5. Implémentez également la fonction de congruence linéaire RANDU. On appellera la fonction `RANDU`, et elle prendra pour paramètre la graine du générateur.

Vous aurez besoin pour ces deux générateurs de donner une graine qui doit être une variable globale. Introduisez trois variables globales `sVN`, `sSTM` et `sRANDU`, qui prendront des valeurs comprises entre 0 et $2^{31} - 1$.

2.2 Qualité de la séquence produite par un générateur pseudo-aléatoire

Il existe beaucoup de tests permettant de s'assurer de la qualité des nombres aléatoires produits (voir les suites de tests complètes du NIST [oST] ou de DIEHARD [Mar95]). Nous ne les programmerons pas tous, nous allons nous focaliser sur trois tests particuliers que nous appliquerons ensuite sur les générateurs décrits précédemment. Le cours de statistique de quatrième année vous donnera plus de détails sur le principe d'un test statistique. On pourra aussi se référer à [Ber05b] ou [Can06] pour voir les nombreux tests statistiques qui existent.

On vous demande donc d'implémenter les tests présentés dans la suite de ce document. Plus précisément, il s'agira de tester tous les générateurs (Von Neumann, Standard Minimal, RANDU et le `rand` du langage que vous avez choisi) :

- pour le premier test, sur une séquence de $k = 1000$ valeurs.
- pour les deux derniers tests sur une séquence de $n = 1000$ et pour 100 initialisations différentes.

On rappelle qu'on demande à générer les valeurs sur les intervalles $\{0, \ldots, 2^{31} - 1\}$.

2.2.1 Test Visuel

Question 6. Tracez, pour chacun des générateurs, le graphique des sorties observées pour une suite de $k = 1000$ valeurs. Que constatez-vous ? Expliquez. Nous vous conseillons d'utiliser un historigramme sous libre office calc ou excel par exemple.

2.2.2 Test de fréquence monobit

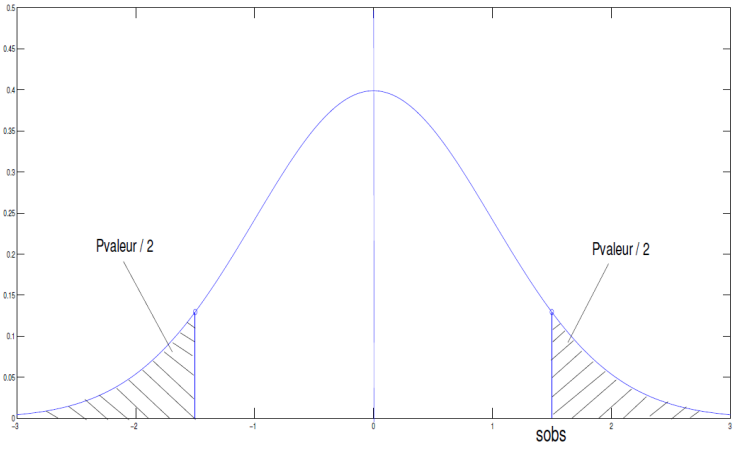
Principe du test : Le but de ce test est de s'intéresser à la proportion de zéros et de uns dans les bits d'une séquence entière : on regarde ici tous les bits des 1000 réalisations de la séquence. On teste donc si le nombre de uns et de zéros d'une séquence sont approximativement les mêmes comme attendu dans une séquence vraiment aléatoire.

Définition de la fonction à implémenter : la fonction à implémenter devra s'écrire : `Frequency(`x`,` nb`)` où x est le vecteur des nombres observés et nb le nombre de bits à considérer pour chacun de ces nombres. Elle effectue les opérations suivantes sur la séquence de bits $(\epsilon) = \epsilon_1 \cdots \epsilon_n$:

- Conversion en +1 ou -1 : les zéros et uns de la séquence d'entrée (ϵ) sont convertis en -1 pour 0 et 1 pour 1. Cela revient à réaliser l'opération $X_i = 2\epsilon_i - 1$. Ces valeurs sont ensuite additionnées : $S_n = X_1 + \cdots + X_n$. Par exemple, la séquence $\epsilon = 1011010101$ avec $n = 10$ devient $S_n = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 = 2$.

- Calcul de s_{obs} : $s_{obs} = \frac{|S_n|}{\sqrt{n}}$. Pour l'exemple précédent, on obtient : $s_{obs} = \frac{|2|}{\sqrt{10}} = 0.6325$.

  Si nous avons bien indépendance dans la séquence de bits, alors le théorème de la limite centrale assure que pour n grand, $\frac{|S_n|}{\sqrt{n}}$ est une variable aléatoire qui suit approximativement une loi normale $\mathcal{N}(0,1)$. L'idée est alors de regarder la valeur de la fonction de répartition de la loi $\mathcal{N}(0,1)$ pour la valeur s_{obs} observée. Cette valeur, appelée P_{valeur} donne la probabilité d'avoir bien observé s_{obs} lorsqu'on a la loi $\mathcal{N}(0,1)$. Ainsi, si cette P_{valeur} est petite, cela signifie qu'il était improbable d'avoir obtenu s_{obs}, donc qu'a priori le théorème de la limite centrale ne peut pas s'appliquer. Ceci signifie que l'indépendance dans la suite de bits n'est pas vérifiée.



  Le calcul de la P_{valeur} sous Matlab peut se faire ainsi : $P_{valeur} = 2 * (1 - \text{cdf('Normal'}, s_{obs}))$. Pour notre exemple, on obtient $P_{valeur} = 0.5271$.

L'appel à la fonction `Frequency(`x`)` devra retourner la P_{valeur} du test de fréquence monobit sur x.

Règle de décision à 1% : Au vu du principe décrit ci-dessus, plus la P_{valeur} est petite plus on peut rejetter de manière sûre le fait que le séquence est aléatoire. En pratique, si la P_{valeur}

calculée est inférieure à 0.01 alors la séquence n'est pas aléatoire. Sinon, on ne peut pas conclure pour autant qu'elle l'est, mais rien n'infirme cette hypothèse, au sens de ce test. Dans l'exemple précédent, comme $P_{valeur} = 0.527089$, on peut valider que la séquence est aléatoire au sens de ce test. Il est recommandé que chaque séquence testée fasse au minimum 100 bits (i.e. $n \geq 100$) afin que l'application du théorème de la limite centrale ait un sens.

Question 7. Selon le langage de programmation que vous avez choisi, quelle fonction déjà implémentée (et dans quelle librairie) permet de calculer la P_{valeur} ?

Question 8. Implémentez la fonction `Frequency(`x`,` nb`)` décrite et testez à l'aide de cette fonction la qualité des générateurs aléatoires étudiés.

2.3 Test des runs

Principe du test : Le but de ce test est de s'intéresser à la longueur des suites successives de zéros et de uns dans la séquence observée. Il teste donc la longueur moyenne de ce qu'on appelle les "runs", *i.e.* les suites consécutives de 0 ou de 1. Il s'appuie sur la propriété suivante :

Propriété 1 *Soit $s = (s_i)_{i\in\mathbb{N}}$ une suite de période T sur l'alphabet $\mathcal{A}$. On appelle run de longueur k un mot de longueur k constitué de symboles identiques, qui n'est pas contenu dans un mot de longueur $k+1$ constitué de symboles identiques, i.e. $(s_i, \cdots, s_{i+k-1})$ est un run de longueur k si : $(s_i = \cdots = s_{i+k-1})$ et $(s_{i-1} \neq s_i)$ et $(s_{i+k} \neq s_{i+k-1})$. La séquence s possède la propriété des runs si le nombre $N(k)$ de runs de longueur k sur une période T vérifie : $\left\lfloor \frac{T\cdot(|A|-1)^2}{|A|^{k+1}} \right\rfloor \leq N(k) \leq \left\lceil \frac{T\cdot(|A|-1)^2}{|A|^{k+1}} \right\rceil$.*

Définition de la fonction à implémenter : La fonction à implémenter devra s'écrire : `Runs(`x`,`nb`)` où x est le vecteur des nombres observés et nb le nombre de bits à considérer pour chacun de ces nombres. Elle retournera la P_{valeur} du test, obtenue en effectuant les opérations suivantes sur la séquence de bits $(\epsilon) = \epsilon_1 \cdots \epsilon_n$:

- Pre-test : Calculer la proportion de 1 dans la séquence observée : $\pi = \frac{\sum_{j=1}^{n} \epsilon_j}{n}$. Par exemple, si $\epsilon = 1001101011$, alors $n = 10$ et $\pi = 6/10 = 3/5$.

- Déterminer si ce pre-test est passé ou non en vérifiant si $|\pi - 1/2| \geq \tau$ avec $\tau = \frac{2}{\sqrt{n}}$. Si oui, arrêter le test ici (dans ce cas, on renvoie $P_{valeur} = 0.0$). Sinon, continuer à l'étape suivante. Avec l'exemple précédent, on obtient : $\tau = 2/\sqrt{10} \approx 0.6346$ et $|\pi - 1/2| = 0.6 - 0.5 = 0.1 < \tau$, donc on continue à appliquer le test.

- Calculer la statistique $V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1$ avec $r(k) = 0$ si $\epsilon_k = \epsilon_{k+1}$ et $r(k) = 1$ sinon. En reprenant l'exemple précédent, on obtient : $V_{10}(obs) = (1+0+1+0+1+1+1+1+0)+1 = 7$.

- On calcule alors la P_{valeur} comme étant :

$$P_{valeur} = 2 * (1 - \text{cdf('Normal', } \frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{n}\pi(1-\pi)}).$$

  Si on reprend l'exemple précédent, on obtient : $P_{value} = 2*(1-\text{cdf('Normal', } \frac{|7-2\cdot 10\cdot \frac{3}{5}(1-\frac{3}{5})|}{2\sqrt{10}\cdot(3/5)\cdot(1-3/5)}) \approx 0.187632$.

Règle de décision à 1% : Si la P_{valeur} calculée est inférieure à 0.01 alors on peut conclure que la séquence n'est pas aléatoire. Dans l'exemple précédent avec une P_{valeur} égale à 0.19, on ne peut donc conclure que la séquence est de mauvaise qualité. Il est recommandé que chaque séquence testée fasse au minimum 100 bits (i.e., $n \geq 100$).

Question 9. Implémentez la fonction décrite et testez à l'aide de cette fonction la qualité des générateurs aléatoires étudiés.

3 Simulation d'une loi de probabilité exponentielle

Nous souhaitons ici simuler une loi exponentielle. Celle-ci repose sur la simulation d'observations issues d'une loi uniforme sur [0, 1].

3.1 Loi exponentielle

Supposons que l'on veuille simuler une loi de probabilité de fonction de répartition F. Les propriétés des fonctions de répartition assurent que F est inversible. Notons F^{-1} son inverse. Soit U suivant une loi uniforme sur $[0, 1]$. Alors nous pouvons montrer que $X = F^{-1}(U)$ a pour fonction de répartition F.

Dans le cas d'une loi exponentielle de paramètre λ, la fonction de répartition vaut

$$F(x) = 1 - \exp(-\lambda x).$$

Par conséquent nous avons

$$F^{-1}(u) = -\ln(1-u)/\lambda.$$

Définition de la fonction à implémenter : la fonction à implémenter devra s'écrire : `Exponentielle(lambda)` où `lambda` est le paramètre de la loi exponentielle considérée. Cette fonction doit retourner une valeur issue de la réalisation d'une loi exponentielle de paramètre `lambda`.

Question 10. Implémenter la fonction simulant une réalisation de loi exponentielle décrite ci-dessus. Comparer l'histogramme des valeurs obtenues à une loi exponentielle.

References

[Ber05a] J. Berard. *Fiche 1 - Nombres pseudo-aléatoires.* ISTIL, 2005. disponible à `http://math.univ-lyon1.fr/~jberard/genunif-www.pdf`.

[Ber05b] J. Berard. *Fiche 2 - Génération de variables pseduo-aléatoires.* ISTIL, 2005. disponible à `http://math.univ-lyon1.fr/~jberard/genloi-www.pdf`.

[Can06] Anne Canteaut. Présentation des principaux tests statistiques de générateurs aléatoires, 2006. disponible sur `http://www.picsi.org/parcours_63.html`.

[Mar95] George Marsaglia. Diehard battery of tests of randomness, 1995. disponible à `http://i.cs.hku.hk/~diehard/cdrom/`.

[oST] National Institute of Standards and N.I.S.T Technology. Random number generation. disponible à `http://csrc.nist.gov/groups/ST/toolkit/rng/index.html`.

[Wika] Wikipédia. Générateur de nombres pseudo-aléatoires. disponible à `http://fr.wikipedia.org/wiki/Generateur_de_nombres_pseudo-aleatoires`.

[Wikb] Wikipédia. Mersenne Twister. disponible à `http://fr.wikipedia.org/wiki/Mersenne_Twister`.

A Description du `main()` attendu et des sorties dans `test.txt`

Le `main()` attendu implémentera les 5 premières occurences des tests demandés dans les différentes questions et les résultats de ces tests seront affichés dans le fichier `test.txt`.