Analyse d’Algorithmes et Génération Aléatoire (AAGA)

Organisation

Équipe Pédagogique :

- Maximilien Danisch (Max)
- Bùi Xuân Bình Minh (Bin)

Emploi du temps :

- Séances 1, puis séances 5 à 7 avec Max
- Séances 2 à 4, puis séances 8 à 14 avec Bin

Évaluation :

- Deux examens répartis : 25% (V1GA) et 20% (V2final).
- Deux projets: 20% (V1AA) et 25% (V2démo).
- Un devoir sur table: 10% (V2CC).

1- Génération d’entiers : Introduction

“*Le vrai hasard étant hasardeux*, *contentons-nous* d’un *pseudo-hasard et adaptons-le à nos besoins*” (Jean-Paul Delahaye dans Pour la Science 1998).

Depuis des années, on tente d’enlever tout hasard au sein d’un système informatique :

- Aujourd'hui un calcul composé de milliards d'opérations peut être lancé plusieurs fois et donnera à chaque fois le même résultat.
- Algorithme de correction d'erreurs (Transmission de données et stockage de données).

Pourtant maintenant que l'on gère l'élimination du hasard, on a besoin de le faire réapparaître. Trois grands domaines :

1. La simulation (Par exemple pour modéliser un feu de forêt)
2. L'algorithmique (Algorithmes probabilistes : par exemple l'Algorithme de Kager pour MIN-CUT. **Calculer la probabilité d'erreur ?**)
3. La cryptographie (par exemple : Masque jetable et Chiffrement RSA)

Pour ces trois applications, ce n'est pas le même hasard qui est nécessité ! Dans la plupart des langages de programmation, il y a un générateur de hasard appelé "pseudo random generator".

Attention : Python depuis la version 2.3 Mersenne Twister :

> "The pseudo-random denerators of this module should not be used for security purposes."

Vous, en tant qu'experts en informatique, vous n'avez pas besoin de connaître toutes les raisons qui peuvent poser problème. Mais vous devez réagir correctement face à ce type de message.

LIVE DEMO !!! : *(Exemple à partir de Java 6)*

- Avec deux entiers consécutifs générés, je peux deviner tous les suivants en moins de 10 lignes de code python.
- Pire : avec deux entiers consécutifs, je retrouve tous les précédents !

Deux types de générateurs d'entiers :

1. True Random Number Generators
2. Pseudo Random Number Generators

True Random Number Generators

- Usage en cryptographie
- Obligation : non prévisible !
- Basé la plupart du temps sur des processus physiques :
  - Un opérateur fait des piles ou face avec une pièces…
  - Calcul de la décroissance radioactive
  - Utilisation de la température du processeur
  - Utilisation de l'horloge

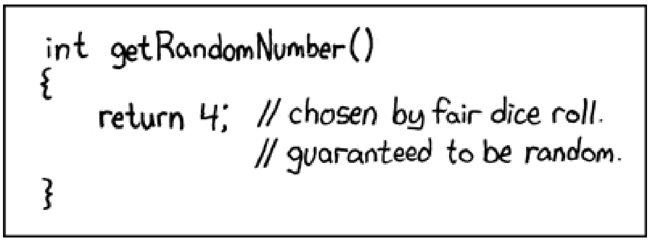
Il peut y avoir un biais (face 55%, pile 44.9% et tranche 0.1%) : on peut faire un post-traitement si nécessaire.

Problèmes :

- coûte cher
- vitesse lente de génération

- dé-bug de programmes

- [random.org]()
- [Générateur de nombres aléatoires matériel]()
- [lavarand]()
- [comsire]()
- [ORION]()



Pseudo Random Number Generators

- Ils sont déterministes : pas réellement d'aléatoire !
- Très rapide
- Portable
- On peut reproduire des séquences d'entiers aléatoires (grâce à une graine)

Problème : trouver un bon Pseudo Random Number Generators !

Les générateurs de nombres pseudo-aléatoires sont "tous" construits avec une idée d'itération :

- Soit E l'ensemble des entiers dans lequel on souhaite générer.
- On choisit $f : E \to E$.
- On définit "aléatoirement" $X_0 \in E$, nommé la graine.
- Puis on fait $X_{n+1} = f(X_n)$.

Propriété :

> - $\exists \lambda \in \mathbb{N}^*$ appelé période du générateur (on a $\lambda \leq |E|$).
> - $\exists n_0 \leq \lambda, \forall n \geq n_0, X_{n+\lambda} = X_n$.

Remarques :

> - Une fois que n_0 est atteint, on boucle indéfiniment dans la même suite d'entiers avec une période λ.
> - La période est définie par les entiers i, j les plus proches possible telle que $X_i = X_j$.
> - La longueur de la période est cruciale pour qu'un générateur soit de qualité.

Preuve de l'existence de λ ?

Exemple :

> $E = 1, 2, \ldots, 8$, $f(x) = 2x \mod (7) + 1$ et $X_0 = 8$. **Que valent λ et n_0 ?**

Paradoxe des anniversaires :

> On veut construire aléatoirement un générateur aléatoire d'entier dans $[\![1, n]\!]$. On choisit une suite d'entiers dans $[\![1, n]\!]$ de façon indépendante et uniforme. **Quelle est la probabilité qu'une suite de j entiers contienne au moins deux entiers identiques ?**

Conclusion ?

2- Exemples de PRNG

Il existe une infinité de PRNG !

Question : Proposer quelques PRNG ?

Générateurs congruentiels linéaires

Schéma introduit par Lehmer en 1949. Jusque dans les années 90 c'était la méthode la plus utilisée. Même les générateurs actuels ne sont pas complètement différents.

Principe :

> On a besoin de 4 entiers :
>
> - m : le modulo ($m > 0$)
> - a : le multiplicateur ($0 \leq a < m$)
> - c : l'incrément ($0 \leq c < m$)
> - X_0 : la valeur initiale, appelée graine ($0 \leq X_0 < m$)
>
> On construit une suite de nombre dans $[\![0, m-1]\!]$ de la manière suivante : $X_{n+1} = (aX_n + c) \mod m$. (X_n) est appelée suite congruentielle (ou modulaire) linéaire

---

Exemple :

> $X_{n+1} = (2X_n + 3) \mod 10$

Exercice :

> Exprimer X_{n+k} en fonction de X_n, a, c, m et k. On supposera $a \geq 2$.

Propriété :

> Si (X_n) est une suite congruentielle linéaire, alors les suites extraites $(X_{n_0+i_0 n})$ avec n_0 et i_0 fixés sont des suites congruentielles linéaires.

Preuve ?

Multiplicateur = a^{i_0} Incrément = $\frac{a^{i_0}-1}{a-1}c$ Graine = X_{n_0}

Donner une formule directe pour calculer X_n (en fonction de X_0, a, c, m, et n) ?

$X_n = a^n X_0 + \frac{a^n-1}{a-1}c \mod m$

- Choix du module m ?
  - Pour générer des bits, ne pas prendre $m = 2$, si non 010101 dans le meilleur cas
  - Utiliser $Y_n = X_n \mod d$
  - $m = 2^{64}$ rapide (**Pourquoi ?**), mais attention aux bits de poids faibles : si d divise m, alors $Y_{n+1} = (aY_n + c) \mod d$. Prendre les bits de poids fort dans ce cas.
  - Prendre $m =$ grand nombre premier
- Choix du multiplicateur a et de l'incrément c ?
  - On veut la période la plus longue possible, mais pas seulement $(a = c = 1)$!

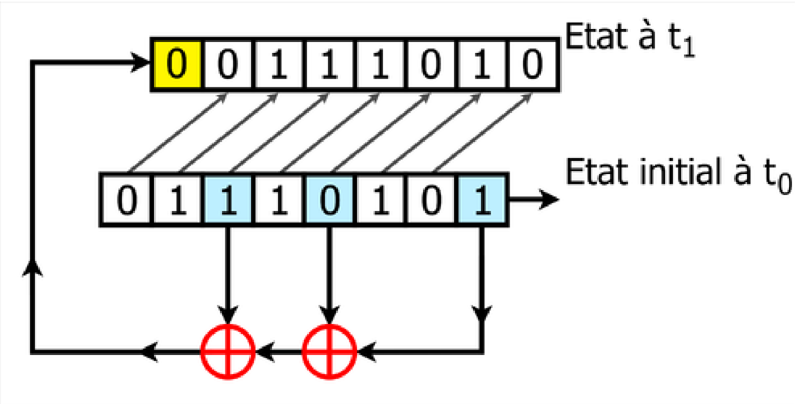
Générateur Blum Blum Shub (BBS)

Soit n un produit de deux entiers premiers, chacun de la forme $4m + 3$. On prend comme graine un entier x sans facteur commun avec n. On définit ensuite $x_{i+1} = x_i^2 \mod n$. La parité de $x(i)$ donne la suite pseudo-aléatoire de bits proposée par BBS.

Si la graine est choisie aléatoirement, et si "trouver les racines carrées modulo n est un problème difficile", alors aucun algorithme rapide ne fait mieux que le hasard pour prédire le m-ième digit à partir des précédents.

https://fr.wikipedia.org/wiki/Blum_Blum_Shub

Registre à décalage



https://en.wikipedia.org/wiki/Linear-feedback_shift_register

Le Mersenne Twister

- C'est un générateur de nombre pseudo-aléatoire particulièrement réputé pour sa qualité.
- Développé par M. Matsumoto et T. Nishimura en 1997.
- Basé sur un type particulier de registre à décalage à rétroaction et tient son nom d'un nombre premier de Mersenne.
- C'est le générateur par défaut de Python, PHP, Maple, Matlab, R, GNU Multiple Precision Arithmetic Library…

Avantages :

> - Sa période : $2^{19937}-1$ (c'est un nombre premier de Mersenne)
> - k-distribué avec une précision de 32 bits
> - Passe la grande majorité des tests statistiques (notamment Die Hard tests)

Inconvénients :

> Espace d'état trop grand : une période de l'ordre de 2^{512} devrait être suffisante (aussi bon et plus rapide) pour n'importe quelle application.

Remarques :

> - La période est aussi grande car l'algorithme est basé sur un ensemble de 624 entiers (32 bits) indépendants.
> - Il faut pouvoir fournir une graine (relativement aléatoire) de 624 entiers : en général, on fait appel à un autre générateur aléatoire pour construire la graine.
> - Il existe une version simplifiée datant de 2006 avec une période de $2^{607}-1$.

3- Suite aléatoire : qu'est-ce que c'est ?

Paradoxalement, la théorie des probabilité n'est pas la mieux adaptée pour définir ce qu'est une suite aléatoire. Considérons les deux suites suivantes :

- 01010101010101010101
- 11010001101001011001

Le bon sens dirait que la deuxième est plus aléatoire que la première et pourtant on peut prouver que chacune a la même probabilité d'apparaître que l'autre (si on prend une pièce parfaite et qu'on fait pile ou face).

Complexité de Kolmogorov

En informatique, la notion de suite aléatoire a convergé vers la notion de suite complexe, ou incompressible.

Définition : *Complexité de Kolmogorov :*

> Soit $a = a_1 a_2 \ldots a_n$ une suite binaire. La complexité de Kolmogorov de la suite a est la longueur, en nombre de bits, du plus petit programme permettant à un ordinateur donné de générer a.

Pour nous, un ordinateur correspond à une [machine de Turing universelle](). La complexité de Kolmogorov est définie à une constante additive près, car elle dépend de l'ordinateur choisi.

Propriété : *in-calculabilité :*

> La complexité de Kolmogorov n'est pas [calculable](). En d'autre termes, il n'existe pas de programme informatique qui prenne en entrée s et renvoie $K(s)$.

Preuve ?

Par l'absurde, on suppose que Kolmo() existe: Kolmo prend en entrée une suite de caractères s et retourne $K(s)$, cad la taille du plus petit programme générant la suite de caractères s. On note k la complexité de Kolomogorov de Kolmo().

Considérons le programme suivant:

```
n := 1
Tant que Kolmo(n) < k + 1000 faire:
    n := n + 1
Fin du Tant que
écrire n
```

Cet algorithme écrit le plus petit nombre à avoir une complexité de Kolmogorov supérieur à $k + 1000$ (ce nombre existe car il n'y a qu'un nombre fini de programmes de taille plus petite que $k + 1000$ et il y a une infinité de nombres entiers naturels)

Mais l'algorithme ci-dessus s'écrit justement avec moins de $k + 1000$ caractères: il est donc de complexité inférieure à $k + 1000$, or il écrit justement un nombre de complexité supérieur à $k + 1000$, ce qui est absurde. Donc il n'existe pas de fonction qui calcul la complexité de Kolmogorov.

Définition : *Indépendance de l'ordinateur :*

> Une suite infinie $a = a_1 a_2 \dots a_n \dots$ est dite aléatoire si $\lim_{n\to\infty} \frac{K(a)}{n} = 1$.

Propriété :

> La plupart des suites de longueur n a une complexité de Kolmogorov au moins d'ordre n.

Preuve ?

$$K(a) > m = (1 - \epsilon) * n$$

Soit n un entier naturel. Il existe 2^n suites binaires de taille n. Soit $\epsilon \in [0, 1]$ et soit $m = (1 - \epsilon)n$
Il existe $\sum_{i=0}^{m} 2^i \leq 2^{m+1}$ suites binaires de taille inférieure ou égale à m.

Ainsi, seule une proportion inférieure à $\frac{2^{m+1}}{2^n}$ de suites binaires de taille n peuvent avoir une complexité de Kolmogorov inférieure à m.

Or, $\frac{2^{m+1}}{2^n} = 2^{(1-\epsilon)n+1-n} = 2^{-\epsilon * n+1} = \frac{2}{2^{\epsilon * n}}$

(tend vers 0 quand n tend vers l'infini). CQFD!

Propriété :

> Une suite binaire aléatoire $a = a_1 a_2 \dots a_n$ vérifie la loi des grands nombres : $\lim_{n\to\infty} \frac{\sum_{i=1}^{n} a_i}{n} = \frac{1}{2}$.

Idée de preuve:

> Shannon source coding theorem: taille compressée arbitrairement proche de $H * n$, avec $H = -p_1 \log_2(p_1) - (1 - p_1) \log_2(1 - p_1)$ et $H < 1$ pour $p_1 \neq 0.5$
> https://en.wikipedia.org/wiki/Arithmetic_coding

Remarques :

> - Plusieurs autres propriétés statistiques sont valides sur les suites aléatoires.
> - Suite des décimales de π non aléatoire, mais suit les bonnes propriétés de statistiques : $pi = 3.14159265358979323846264338327950\ldots$
> - Fabriqué de manière déterministe une suite aléatoire de bits (ou de n'importe quel objet) est contradictoire.

But : construire une suite pseudo-aléatoire, c'est-à-dire une suite qui s'approche d'une suite aléatoire.

Remarque : (Informellement)

> On a besoin de s'approcher de l'imprévisible. Une suite a est pseudo-aléatoire si elle est produite par un algorithme et qu'il est algorithmiquement difficile de prévoir avec une probabilité $> \frac{1}{2}$ le bit a_{n+1} en connaissant tous les bits $a_1 a_2 \ldots a_n$ précédents (Formalisé par Yao en 1982).

Remarque :

> On ne sait pas prouver qu'il existe une suite pseudo-aléatoire et le prouver permettrait de conclure $P \neq NP$. Pour le moment, on a une approche plutôt expérimentale pour construire des générateurs. Après construction d'un générateur, on vérifie que les propriétés mathématiques nécessaires (notamment statistiques) sont vérifiées.

Fonction à sens unique

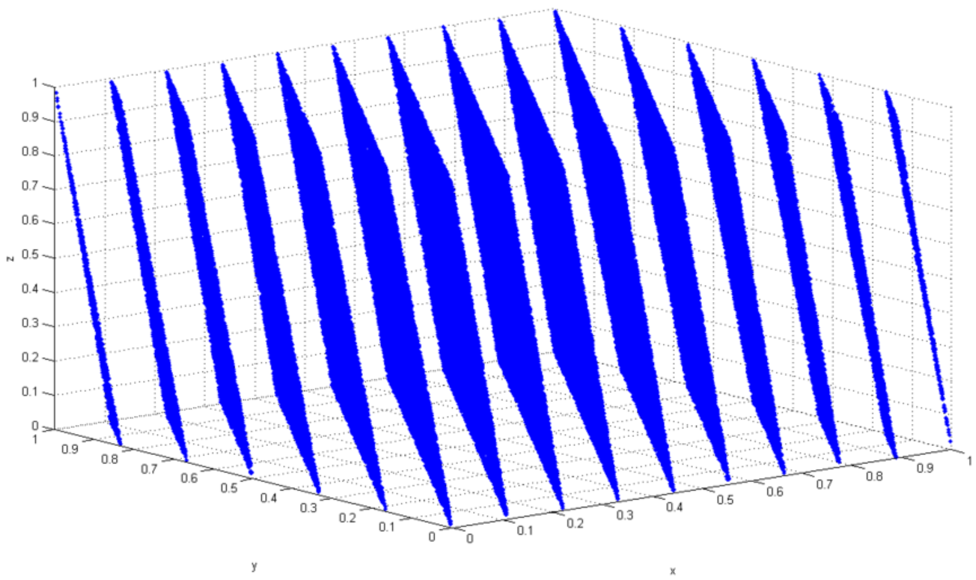
Tests statistiques

- Difficile pour un humain de décider manuellement si une suite est aléatoire ou pas (tendance à éviter les choses qui "semblent" non-aléatoires, e.g. mêmes nombres consécutifs).
- $\pi = 3.14159$\textcolorred2653589\textcolorblue79323846\textcolorred2643\textcolorblue38327950
- Utiliser des tests mécaniques non-biaisés !

Proposer des tests statistiques.

Test spectral

Représentation en 3D de 100 000 valeurs générées par RANDU. Chaque point est détermine par trois tirages pseudo-aléatoires consécutifs. On voit ainsi que les points se trouvent sur un ensemble de 15 plans de l’espace.



Diehard test and BigCrush test

- **Diehard test :** https://en.wikipedia.org/wiki/Diehard_tests ; https://webhome.phy.duke.edu/~rgb/General/dieharder.php
- **BigCrush test :** http://simul.iro.umontreal.ca/testu01/tu01.html

Trois degrés de hasard

- **Hasard faible :** Satisfaction des tests statistiques. Peut être produit par des algorithmes rapides. Utile en simulation.
- **Hasard moyen :** Imprévisibilité pour un observateur ne disposant que de moyens de calcul réalistes. Peut être produit (vraisemblablement) par algorithmes moyennement rapides. Utile en cryptographie.
- **Hasard fort :** Imprévisibilité totale, incompressibilité, contenu maximum en information. Ne peut jamais être produit par algorithme, mais vraisemblablement par des moyens physiques. Utile en cryptographie et en théorie de l’information.

4- Génération de Structures Arborescentes

Les arbres sont omniprésents en informatique. Savoir les générer permet par exemple de tester des programmes manipulant ces objets.

Si a priori, il n’y a pas de contrainte structurelle sur les objets à tester, la méthode idéale consiste à la génération uniforme, c’est-à-dire que deux objets de même taille sont générés avec la même probabilité.

Proposition : *Génération optimale (en nombre de bits aléatoires)*

> Soit E un ensemble d'objets (pas forcément de même taille). Le nombre minimum de bits aléatoires pour générer uniformément un objet de E est $\lceil\log_2(|E|)\rceil$.

Preuve ?

Remarques :

> Souvent on veut choisir uniformément un objet de E sans les avoir tous construits.

Définition : *(Arbres binaires)*

> Un arbre binaire est soit (i) une feuille, soit (ii) un nœud interne et deux fils qui sont des arbres binaires.

Proposition :

> Il y a $C_n = \frac{1}{n+1}\binom{2n}{n}$ (dit nombres de Catalan) arbres binaire avec n nœuds internes (et donc $n+1$ feuilles).

Preuve ?

Algorithme de Rémy (1985)

But : Générer un arbre binaire à n nœuds internes.

- Point de départ : une feuille numérotée 1
- Supposons que l'on ait construit un arbre binaire de taille k (k nœuds internes et $k+1$ feuilles étiquetées de 1 à $k+1$).
  - On choisit uniformément un nœud, soit F l'arbre enraciné en ce noeud et A l'arbre global.
  - On ajoute un nœud interne et on tire pile/face pour savoir si F est le fils gauche ou le fils droit de ce nœud. L'autre fils est une feuille étiquetée $k+2$.
  - On remplace dans A, F par le nouveau sous-arbre.

Correction et Complexité

Théorème : *(Correction)*

> Après avoir effacé les étiquettes des feuilles de l'arbre généré, on obtient un arbre binaire de taille n uniforme parmi tous les arbres binaires de taille n.

Preuve ?

> Chaque arbre binaire étiqueté a une et une seule possibilité d'être construit (on peut le voir en le déconstruisant : en revenant en arrière dans la construction).

Remarques : *(Complexité en nombre de bits aléatoires)*

- On ne peut pas faire mieux que $\lceil\log_2(C_n)\rceil \in \Theta(n)$.
- Avec l'algorithme de Rémy $\Theta(n\log(n))$ bits sont nécessaires.

Preuve ?
