Algorithmique Probabiliste

Philippe Duchon

LaBRI - ENSEIRB-Matméca - Université de Bordeaux

2014-15

Sur la génération d'aléa sur les ordinateurs

- Accumulateurs d'entropie

Sur la génération d'aléa sur les ordinateurs

- Accumulateurs d'entropie
- Générateurs pseudo-aléatoires

Sur la génération d'aléa sur les ordinateurs

- Accumulateurs d'entropie
- Générateurs pseudo-aléatoires
- Quelques tests auxquels on peut les soumettre

Le paradoxe du tirage de nombres aléatoires

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. (J. von Neumann, 1951)

- Toute la construction d'un appareil électronique "classique" est organisée dans le but de rendre *déterministe* (prévisible, répétable) son comportement en réponse aux entrées

Le paradoxe du tirage de nombres aléatoires

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. (J. von Neumann, 1951)

- Toute la construction d'un appareil électronique "classique" est organisée dans le but de rendre *déterministe* (prévisible, répétable) son comportement en réponse aux entrées
- Le comportement d'un microprocesseur n'est pas une exception

Le paradoxe du tirage de nombres aléatoires

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. (J. von Neumann, 1951)

- Toute la construction d'un appareil électronique "classique" est organisée dans le but de rendre *déterministe* (prévisible, répétable) son comportement en réponse aux entrées
- Le comportement d'un microprocesseur n'est pas une exception
- Et pourtant, on a pléthore d'applications informatiques qui ont "besoin" de "nombres aléatoires"

Le paradoxe du tirage de nombres aléatoires

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. (J. von Neumann, 1951)

- Toute la construction d'un appareil électronique "classique" est organisée dans le but de rendre *déterministe* (prévisible, répétable) son comportement en réponse aux entrées
- Le comportement d'un microprocesseur n'est pas une exception
- Et pourtant, on a pléthore d'applications informatiques qui ont "besoin" de "nombres aléatoires"
  - Test de programmes via des données "aléatoires"

Le paradoxe du tirage de nombres aléatoires

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. (J. von Neumann, 1951)

- Toute la construction d'un appareil électronique "classique" est organisée dans le but de rendre *déterministe* (prévisible, répétable) son comportement en réponse aux entrées
- Le comportement d'un microprocesseur n'est pas une exception
- Et pourtant, on a pléthore d'applications informatiques qui ont "besoin" de "nombres aléatoires"
  - Test de programmes via des données "aléatoires"
  - Simulations de phénomènes modélisés avec de l'aléatoire

Le paradoxe du tirage de nombres aléatoires

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. (J. von Neumann, 1951)

- Toute la construction d'un appareil électronique "classique" est organisée dans le but de rendre *déterministe* (prévisible, répétable) son comportement en réponse aux entrées
- Le comportement d'un microprocesseur n'est pas une exception
- Et pourtant, on a pléthore d'applications informatiques qui ont "besoin" de "nombres aléatoires"
  - Test de programmes via des données "aléatoires"
  - Simulations de phénomènes modélisés avec de l'aléatoire
  - Et bien sûr, tous les algorithmes explicitement probabilistes, qu'ils soient d'intérêt théorique ou pratique

Début de classification : aléa vs pseudo-aléa

On a typiquement deux types de sources d'aléa accessibles dans les ordinateurs :

- certaines sont en réalité parfaitement déterministes et reproductibles, mais ressemblent à des sources aléatoires (générateurs pseudo-aléatoires) ;

Début de classification : aléa vs pseudo-aléa

On a typiquement deux types de sources d'aléa accessibles dans les ordinateurs :

- certaines sont en réalité parfaitement déterministes et reproductibles, mais ressemblent à des sources aléatoires (générateurs pseudo-aléatoires) ;
- d'autres sont *réellement* imprévisibles et "aléatoires" ("accumulateurs d'entropie")

Ça existe, le hasard ?

Au niveau de la “réalité physique” :

- La physique classique (mécanique newtonienne ou même relativiste) est purement déterministe : en théorie du moins, si un système est décrit totalement dans les moindres détails, son évolution dans le temps est entièrement déterminée.

Ça existe, le hasard ?

Au niveau de la “réalité physique” :

- La physique classique (mécanique newtonienne ou même relativiste) est purement déterministe : en théorie du moins, si un système est décrit totalement dans les moindres détails, son évolution dans le temps est entièrement déterminée.
- La physique quantique est intrinsèquement probabiliste : elle se contente de décrire des probabilités d’observer tel ou tel événement ; principe de Heisenberg : borne inférieure absolue sur le produit des incertitudes sur la position et la vitesse d’une particule.

Ça existe, le hasard ?

Au niveau de la “réalité physique” :

- La physique classique (mécanique newtonienne ou même relativiste) est purement déterministe : en théorie du moins, si un système est décrit totalement dans les moindres détails, son évolution dans le temps est entièrement déterminée.
- La physique quantique est intrinsèquement probabiliste : elle se contente de décrire des probabilités d'observer tel ou tel événement ; principe de Heisenberg : borne inférieure absolue sur le produit des incertitudes sur la position et la vitesse d'une particule.
- (à échelle microscopique, il semblerait que le monde soit intrinsèquement probabiliste, et que ce soit seulement la loi des grands nombres qui le fasse apparaître déterministe)

Ça existe, le hasard ?

Au niveau de la “réalité physique” :

- La physique classique (mécanique newtonienne ou même relativiste) est purement déterministe : en théorie du moins, si un système est décrit totalement dans les moindres détails, son évolution dans le temps est entièrement déterminée.
- La physique quantique est intrinsèquement probabiliste : elle se contente de décrire des probabilités d’observer tel ou tel événement ; principe de Heisenberg : borne inférieure absolue sur le produit des incertitudes sur la position et la vitesse d’une particule.
- (à échelle microscopique, il semblerait que le monde soit intrinsèquement probabiliste, et que ce soit seulement la loi des grands nombres qui le fasse apparaître déterministe)
- Plus prosaïquement, approche “physique statistique” (dont thermodynamique) : décrire parfaitement un système composé de très grands nombres de particules est irréaliste, *tout se passe comme si* on était face à un système probabiliste (dont le comportement apparaît déterministe à grande échelle).

“Vrais” générateurs aléatoires

- `www.random.org` : nombres aléatoires (entiers, bits,...) obtenus par amplification de bruits atmosphériques captés par radio

“Vrais” générateurs aléatoires

- `www.random.org` : nombres aléatoires (entiers, bits,...) obtenus par amplification de bruits atmosphériques captés par radio
- Plus généralement : tous dispositifs physiques qui mesurent les variations de grandeurs, typiquement à petite échelle

“Vrais” générateurs aléatoires

- `www.random.org` : nombres aléatoires (entiers, bits,...) obtenus par amplification de bruits atmosphériques captés par radio
- Plus généralement : tous dispositifs physiques qui mesurent les variations de grandeurs, typiquement à petite échelle
- (D’après Wikipedia) une dizaine de constructeurs vendent des appareils (typiquement connectables par port USB) qui fournissent des bits aléatoires avec un débit allant de 32 kbit/s à plusieurs dizaines de Mbit/s, à des tarifs allant de quelques dizaines à plusieurs milliers d’euros

“Accumulateurs d’entropie” : générateurs *in silico*

- Générateurs de nombres non reproductibles : à part en enregistrant une séquence, on n’a pas de moyen de la retrouver

“Accumulateurs d’entropie” : générateurs *in silico*

- Générateurs de nombres non reproductibles : à part en enregistrant une séquence, on n’a pas de moyen de la retrouver
- **Hardware :** de plus en plus souvent, les processeurs et/ou cartes mères ont des dispositifs permettant de lire des bits aléatoires dans des registres dédiés

“Accumulateurs d’entropie” : générateurs *in silico*

- Générateurs de nombres non reproductibles : à part en enregistrant une séquence, on n’a pas de moyen de la retrouver
- **Hardware :** de plus en plus souvent, les processeurs et/ou cartes mères ont des dispositifs permettant de lire des bits aléatoires dans des registres dédiés
- **Software :** buffers de bits aléatoires remplis de manière logicielle lors d’événements imprévisibles (bits de poids faible de l’horloge lors d’événements liés à un changement de contexte sur un système multitâches, par exemple) (`/dev/random` sur certains Unix)

“Accumulateurs d’entropie” : générateurs *in silico*

- Générateurs de nombres non reproductibles : à part en enregistrant une séquence, on n’a pas de moyen de la retrouver
- **Hardware :** de plus en plus souvent, les processeurs et/ou cartes mères ont des dispositifs permettant de lire des bits aléatoires dans des registres dédiés
- **Software :** buffers de bits aléatoires remplis de manière logicielle lors d’événements imprévisibles (bits de poids faible de l’horloge lors d’événements liés à un changement de contexte sur un système multitâches, par exemple) (`/dev/random` sur certains Unix)
- **Typiquement :** “corrigés” pour devenir “plus uniformes” (cf. schéma de von Neumann pour obtenir des Bernoulli $1/2$) ; l’indépendance est “garantie” par un débit assez faible

“Accumulateurs d’entropie” : générateurs *in silico*

- Générateurs de nombres non reproductibles : à part en enregistrant une séquence, on n’a pas de moyen de la retrouver
- **Hardware :** de plus en plus souvent, les processeurs et/ou cartes mères ont des dispositifs permettant de lire des bits aléatoires dans des registres dédiés
- **Software :** buffers de bits aléatoires remplis de manière logicielle lors d’événements imprévisibles (bits de poids faible de l’horloge lors d’événements liés à un changement de contexte sur un système multitâches, par exemple) (`/dev/random` sur certains Unix)
- **Typiquement :** “corrigés” pour devenir “plus uniformes” (cf. schéma de von Neumann pour obtenir des Bernoulli 1/2) ; l’indépendance est “garantie” par un débit assez faible
- **Avantage :** réellement imprévisible

"Accumulateurs d'entropie" : générateurs *in silico*

- Générateurs de nombres non reproductibles : à part en enregistrant une séquence, on n'a pas de moyen de la retrouver
- **Hardware :** de plus en plus souvent, les processeurs et/ou cartes mères ont des dispositifs permettant de lire des bits aléatoires dans des registres dédiés
- **Software :** buffers de bits aléatoires remplis de manière logicielle lors d'événements imprévisibles (bits de poids faible de l'horloge lors d'événements liés à un changement de contexte sur un système multitâches, par exemple) (`/dev/random` sur certains Unix)
- **Typiquement :** "corrigés" pour devenir "plus uniformes" (cf. schéma de von Neumann pour obtenir des Bernoulli $1/2$) ; l'indépendance est "garantie" par un débit assez faible
- **Avantage :** réellement imprévisible
- **Inconvénient :** la séquence ne peut pas être "rejouée" (c'est un problème si, par exemple, on veut tester un programme)

Générateurs pseudo-aléatoires

- Tout le contraire : *séquences* de nombres qui ne sont pas du tout aléatoires, mais “ressemblent” à des “vraies séquences aléatoires”

Générateurs pseudo-aléatoires

- Tout le contraire : *séquences* de nombres qui ne sont pas du tout aléatoires, mais “ressemblent” à des “vraies séquences aléatoires”
- À peu près tous les générateurs pseudo-aléatoires sont construits par *itération* d’une fonction sur un domaine fini

Générateurs pseudo-aléatoires

- Tout le contraire : *séquences* de nombres qui ne sont pas du tout aléatoires, mais “ressemblent” à des “vraies séquences aléatoires”
- À peu près tous les générateurs pseudo-aléatoires sont construits par *itération* d'une fonction sur un domaine fini
- 4 paramètres : deux entiers m et k (typiquement $m > k$), une fonction $f : \{0, 1\}^m \to \{0, 1\}^m$, une fonction $g : \{0, 1\}^m \to \{0, 1\}^k$

Générateurs pseudo-aléatoires

- Tout le contraire : *séquences* de nombres qui ne sont pas du tout aléatoires, mais "ressemblent" à des "vraies séquences aléatoires"
- À peu près tous les générateurs pseudo-aléatoires sont construits par *itération* d'une fonction sur un domaine fini
- 4 paramètres : deux entiers m et k (typiquement $m > k$), une fonction $f : \{0,1\}^m \to \{0,1\}^m$, une fonction $g : \{0,1\}^m \to \{0,1\}^k$
- Le générateur est initialisé par $X_0 \in \{0,1\}^m$ (seed), puis à chaque appel : $X_{n+1} = f(X_n)$, $Y_{n+1} = g(X_{n+1})$ est retourné

Générateurs pseudo-aléatoires

- Tout le contraire : *séquences* de nombres qui ne sont pas du tout aléatoires, mais "ressemblent" à des "vraies séquences aléatoires"
- À peu près tous les générateurs pseudo-aléatoires sont construits par *itération* d'une fonction sur un domaine fini
- 4 paramètres : deux entiers m et k (typiquement $m > k$), une fonction $f : \{0,1\}^m \to \{0,1\}^m$, une fonction $g : \{0,1\}^m \to \{0,1\}^k$
- Le générateur est initialisé par $X_0 \in \{0,1\}^m$ (seed), puis à chaque appel : $X_{n+1} = f(X_n)$, $Y_{n+1} = g(X_{n+1})$ est retourné
- Les générateurs les plus basiques ont $k = m$ et $g = \mathrm{id}$: $Y_{n+1} = f(Y_n)$ (on peut prévoir toute la suite de la séquence si on connaît bien la fonction f et qu'on observe un Y).

Propriétés d'un générateur pseudo-aléatoire

- La suite $(X_n)_{n\geq 0}$ est *ultimement périodique* : plus précisément, il existe forcément n_0 et p, avec $n_0 + p \leq 2^m$, tels que, pour tout $n \geq n_0$, $X_{n+p} = X_n$ (donc la même chose est vraie de Y) **- un générateur avec très peu de mémoire aura une faible période ; il faut éviter qu'une application utilise un nombre de tirages qui soit non négligeable devant** p

Propriétés d'un générateur pseudo-aléatoire

- La suite $(X_n)_{n \geq 0}$ est *ultimement périodique* : plus précisément, il existe forcément n_0 et p, avec $n_0 + p \leq 2^m$, tels que, pour tout $n \geq n_0$, $X_{n+p} = X_n$ (donc la même chose est vraie de Y) **- un générateur avec très peu de mémoire aura une faible période ; il faut éviter qu'une application utilise un nombre de tirages qui soit non négligeable devant p**
- Une forte période est indispensable, mais pas suffisante : $f(x) = x + 1 \mod 2^m$ est typiquement une mauvaise idée (selon g).

Propriétés d'un générateur pseudo-aléatoire

- La suite $(X_n)_{n\geq 0}$ est *ultimement périodique* : plus précisément, il existe forcément n_0 et p, avec $n_0 + p \leq 2^m$, tels que, pour tout $n \geq n_0$, $X_{n+p} = X_n$ (donc la même chose est vraie de Y) **- un générateur avec très peu de mémoire aura une faible période ; il faut éviter qu'une application utilise un nombre de tirages qui soit non négligeable devant p**
- Une forte période est indispensable, mais pas suffisante : $f(x) = x + 1 \mod 2^m$ est typiquement une mauvaise idée (selon g).
- La fonction g devrait être, dans la mesure du possible, **équilibrée** : chacune des 2^k valeurs de g devrait avoir (environ) 2^{m-k} antécédents par g (de sorte que, quand X parcourt une éventuelle période de 2^m, les 2^k valeurs soient retournées avec fréquences identiques)

Exemples de générateurs utilisés (historiquement)

- "élévation au carré" : $f(x)$ est obtenu en prenant, sur les $2m$ bits de x^2, les m bits centraux (on oublie les $m/2$ bits de poids faible et les $m/2$ de poids fort)

Exemples de générateurs utilisés (historiquement)

- "élévation au carré" : $f(x)$ est obtenu en prenant, sur les $2m$ bits de x^2, les m bits centraux (on oublie les $m/2$ bits de poids faible et les $m/2$ de poids fort)
- "congruentiels linéaires" : $f(x) = a.x + b \mod 2^m$ (avec, typiquement, a et b impairs (pourquoi ?), et a plutôt grand (pourquoi ?)

Exemples de générateurs utilisés (historiquement)

- "élévation au carré" : $f(x)$ est obtenu en prenant, sur les $2m$ bits de x^2, les m bits centraux (on oublie les $m/2$ bits de poids faible et les $m/2$ de poids fort)
- "congruentiels linéaires" : $f(x) = a.x + b \mod 2^m$ (avec, typiquement, a et b impairs (pourquoi ?), et a plutôt grand (pourquoi ?)
- De plus en plus : f est issu d'une "fonction de hachage" cryptographique, choisie pour être facilement calculable

Exemples de générateurs utilisés (historiquement)

- “élévation au carré” : $f(x)$ est obtenu en prenant, sur les $2m$ bits de x^2, les m bits centraux (on oublie les $m/2$ bits de poids faible et les $m/2$ de poids fort)
- “congruentiels linéaires” : $f(x) = a.x + b \mod 2^m$ (avec, typiquement, a et b impairs (pourquoi ?), et a plutôt grand (pourquoi ?)
- De plus en plus : f est issu d’une “fonction de hachage” cryptographique, choisie pour être facilement calculable
- Exemple de l’**algorithme K** de Knuth : fonction f particulière (de $\{0, \ldots, 9\}^{10}$ dans lui-même)

1. Poser $Y = \lfloor X/10^9 \rfloor$ (premier chiffre de X)
2. Poser $Z = \lfloor X/10^8 \rfloor \mod 10$ (deuxième chiffre de X). Sauter à l'étape $(3 + Z)$.
3. Si $X < 5.10^9$, poser $X = X + 5.10^9$
4. Poser $X = \lfloor X^2/10^5 \rfloor \mod 10^{10}$ (middle square)
5. Poser $X = (1001001001X) \mod 10^{10}$
6. Si $X < 10^9$, poser $X = X + 9814955677$, sinon $X = 10^{10} - X$
7. Échanger les 5 chiffres de poids fort et de poids faible de X
8. Poser $X = (1001001001X) \mod 10^{10}$
9. Diminuer de 1 chaque chiffre décimal non nul de X
10. Si $X < 10^5$, $X = X^2 + 99999$; sinon, $X = X - 99999$
11. Tant que $X < 10^9$, $X = 10X$
12. $X = \lfloor X(X-1)/10^5 \rfloor \mod 10^{10}$
13. Si $Y > 0$, décrémenter Y et retourner à l'étape 2. Sinon, retourner X.

Le danger. . .

- L'algorithme K de Knuth est **catastrophique** : il se trouve avoir très peu de périodes, toutes très courtes (dont un point fixe !)

Le danger. . .

- L'algorithme K de Knuth est **catastrophique** : il se trouve avoir très peu de périodes, toutes très courtes (dont un point fixe !)
- S'il y a une leçon à en tirer, c'est celle-ci : *une méthode pour tirer des nombres aléatoire, ne devrait pas être choisie au hasard*

Test d'un générateur (pseudo-)aléatoire

- Pour les générateurs pseudo-aléatoires : vérification théoriques (période longue, équilibre)

Test d'un générateur (pseudo-)aléatoire

- Pour les générateurs pseudo-aléatoires : vérification théoriques (période longue, équilibre)
- Tests statistiques :

Test d'un générateur (pseudo-)aléatoire

- Pour les générateurs pseudo-aléatoires : vérification théoriques (période longue, équilibre)
- Tests statistiques :
  - Fréquences : bit par bit, les valeurs 0 et 1 devraient avoir la même fréquence (sur 100 tirages : entre 45 et 55 bits à 1, 95% du temps)

Test d'un générateur (pseudo-)aléatoire

- Pour les générateurs pseudo-aléatoires : vérification théoriques (période longue, équilibre)
- Tests statistiques :
  - Fréquences : bit par bit, les valeurs 0 et 1 devraient avoir la même fréquence (sur 100 tirages : entre 45 et 55 bits à 1, 95% du temps)
  - Fréquences des blocs : des blocs de longueur k, devraient avoir des fréquences de l'ordre de $1/2^k$ (au moins à tester jusqu'à $k = 3$)

Test d'un générateur (pseudo-)aléatoire

- Pour les générateurs pseudo-aléatoires : vérification théoriques (période longue, équilibre)
- Tests statistiques :
  - Fréquences : bit par bit, les valeurs 0 et 1 devraient avoir la même fréquence (sur 100 tirages : entre 45 et 55 bits à 1, 95% du temps)
  - Fréquences des blocs : des blocs de longueur k, devraient avoir des fréquences de l'ordre de $1/2^k$ (au moins à tester jusqu'à $k = 3$)
  - "Runs" : le nombre et la longueur des sous-suites de bits identiques (sur n bits : $n/2$ runs, le plus long de longueur $\log_2(n)$)

Tests de générateurs (suite)

- Au lieu de tester les générateurs comme des suites de bits aléatoires, on peut leur faire tirer des nombres (entiers, ou flottants de précision fixée - typiquement, la taille de mot machine).

Tests de générateurs (suite)

- Au lieu de tester les générateurs comme des suites de bits aléatoires, on peut leur faire tirer des nombres (entiers, ou flottants de précision fixée - typiquement, la taille de mot machine).
- On peut alors, pour des séquences pas trop longues (ordre de grandeur : $2^{k/2} = \sqrt{2^k}$ si on prend des entiers sur k bits), négliger l'influence des répétitions d'entiers sur une même séquence ("paradoxe des anniversaires")

Tests de générateurs (suite)

- Au lieu de tester les générateurs comme des suites de bits aléatoires, on peut leur faire tirer des nombres (entiers, ou flottants de précision fixée - typiquement, la taille de mot machine).
- On peut alors, pour des séquences pas trop longues (ordre de grandeur : $2^{k/2} = \sqrt{2^k}$ si on prend des entiers sur k bits), négliger l'influence des répétitions d'entiers sur une même séquence ("paradoxe des anniversaires")
- On peut ensuite prendre des séquences de n nombres et, en les *normalisant*, les transformer en permutations sur $[n]$, qui devraient être uniformes, et tester des statistiques bien connues sur les permutations aléatoires uniformes

Exemples de tests sur suites de nombres

- **Séquences croissantes :** la probabilité que les k premieres valeurs d'une séquence soient croissantes, devrait être $1/k!$

Exemples de tests sur suites de nombres

- **Séquences croissantes :** la probabilité que les k premieres valeurs d'une séquence soient croissantes, devrait être $1/k!$
- **Records :** le nombre de "records" (valeurs plus grandes que toutes les précédentes) dans une permutation aléatoire uniforme de taille n, a pour espérance H_n (somme de Bernoulli indépendantes de paramètres $1/i$).

Exemples de tests sur suites de nombres

- **Séquences croissantes :** la probabilité que les k premieres valeurs d'une séquence soient croissantes, devrait être $1/k!$
- **Records :** le nombre de "records" (valeurs plus grandes que toutes les précédentes) dans une permutation aléatoire uniforme de taille n, a pour espérance H_n (somme de Bernoulli indépendantes de paramètres $1/i$).
- **Test de Kolmogorov-Smirnov :** test d'hypothèse permettant de tester l'hypothèse "les n nombres (sur $[0, 1]$: calculs en flottants) suivent la loi uniforme", en comparant à x la proportion de ceux qui sont inférieurs à x

De manière générale. . .

- Une batterie de tests statistiques ne permettra jamais de *valider* un générateur aléatoire, seulement d'en *rejeter* certains.

De manière générale...

- Une batterie de tests statistiques ne permettra jamais de *valider* un générateur aléatoire, seulement d'en *rejeter* certains.
- Plus on multiplie les tests, plus il faut s'attendre à ce qu'un générateur donné "échoue" à certains.

De manière générale...

- Une batterie de tests statistiques ne permettra jamais de *valider* un générateur aléatoire, seulement d'en *rejeter* certains.
- Plus on multiplie les tests, plus il faut s'attendre à ce qu'un générateur donné "échoue" à certains.
- Les générateurs pseudo-aléatoires, surtout les plus anciens, ont tendance à fournir des mots de bits dans lesquels on peut faire "plus confiance" aux bits de poids fort qu'aux bits de poids faible : si `Random()` est censé fournir un entier entre 0 et N (avec N grand), il est sensiblement plus dangereux d'implémenter `RandomInt(a,b)` par $a + (\texttt{Random()} \bmod (b-a+1))$ que par $a + \mathrm{int}((b-a+1)\texttt{Random()})$.

Mersenne Twister (Matsumoto, Nishimura, 1997)

- Une famille de générateurs devenue un quasi-standard de fait (générateurs par défaut dans de nombreux langages et systèmes de calcul, parmi lesquels PYTHON, certaines versions de COMMON LISP, R, MAPLE, MATLAB, certaines bibliothèques GNU...)
- **Caractéristiques importantes :**

Mersenne Twister (Matsumoto, Nishimura, 1997)

- Une famille de générateurs devenue un quasi-standard de fait (générateurs par défaut dans de nombreux langages et systèmes de calcul, parmi lesquels Python, certaines versions de Common LISP, R, Maple, Matlab, certaines bibliothèques GNU...)
- **Caractéristiques importantes :**
  - une période énorme, très supérieure aux besoins raisonnables des applications ($2^{19937} - 1$ pour MT19937)

Mersenne Twister (Matsumoto, Nishimura, 1997)

- Une famille de générateurs devenue un quasi-standard de fait (générateurs par défaut dans de nombreux langages et systèmes de calcul, parmi lesquels PYTHON, certaines versions de COMMON LISP, R, MAPLE, MATLAB, certaines bibliothèques GNU...)
- **Caractéristiques importantes :**
  - une période énorme, très supérieure aux besoins raisonnables des applications ($2^{19937} - 1$ pour MT19937)
  - très bonnes propriétés statistiques

Mersenne Twister (Matsumoto, Nishimura, 1997)

- Une famille de générateurs devenue un quasi-standard de fait (générateurs par défaut dans de nombreux langages et systèmes de calcul, parmi lesquels PYTHON, certaines versions de COMMON LISP, R, MAPLE, MATLAB, certaines bibliothèques GNU...)
- **Caractéristiques importantes :**
  - une période énorme, très supérieure aux besoins raisonnables des applications ($2^{19937} - 1$ pour MT19937)
  - très bonnes propriétés statistiques
  - modérément lourd en temps et en mémoire (624 mots mémoire de 64 bits)

Mersenne Twister (Matsumoto, Nishimura, 1997)

- Une famille de générateurs devenue un quasi-standard de fait (générateurs par défaut dans de nombreux langages et systèmes de calcul, parmi lesquels PYTHON, certaines versions de COMMON LISP, R, MAPLE, MATLAB, certaines bibliothèques GNU...)
- **Caractéristiques importantes :**
  - une période énorme, très supérieure aux besoins raisonnables des applications ($2^{19937} - 1$ pour MT19937)
  - très bonnes propriétés statistiques
  - modérément lourd en temps et en mémoire (624 mots mémoire de 64 bits)
  - non adapté aux applications cryptologiques : la connaissance de “seulement” 624 valeurs consécutives permet de prédire la suite

Mersenne Twister (Matsumoto, Nishimura, 1997)

- Une famille de générateurs devenue un quasi-standard de fait (générateurs par défaut dans de nombreux langages et systèmes de calcul, parmi lesquels PYTHON, certaines versions de COMMON LISP, R, MAPLE, MATLAB, certaines bibliothèques GNU...)
- **Caractéristiques importantes :**
  - une période énorme, très supérieure aux besoins raisonnables des applications ($2^{19937} - 1$ pour MT19937)
  - très bonnes propriétés statistiques
  - modérément lourd en temps et en mémoire (624 mots mémoire de 64 bits)
  - non adapté aux applications cryptologiques : la connaissance de “seulement” 624 valeurs consécutives permet de prédire la suite
- Le code C de l’article original ne fait que quelques dizaines de lignes de code

Propriété statistique de Mersenne Twister

- MT19937 fournit une séquence d'entiers sur 64 bits, dont les 32 bits de poids fort sont *bien distribués* en un sens
- Notation : $t_v(x)$ représente l'entier sur v bits codé par les bits de poids fort de x (entier sur 64 bits) ; $t_v(x) = \lfloor x 2^{v-64} \rfloor$
- La suite $X_1, X_2, \ldots, X_p$ fournie par MT19937, de période $p = 2^{19937} - 1$, satisfait la propriété suivante : *pour tout $k \leq 623$, les p vecteurs de $32k$ bits $(t_{32}(X_i), t_{32}(X_{i+1}), \ldots, t_{32}(X_{i+k-1}))$ prennent chacune des 2^{32k} valeurs possibles autant de fois, sauf le vecteur intégralement nul qui n'est pris qu'une fois de moins*