Notes de cours
Semaine 9

Cours Turing

1 Générateurs de nombres (pseudo-)aléatoires [1]

La semaine dernière, nous avons vu le système de clé à usage unique et comment sa sécurité repose sur la génération d'une clé K aléatoire. Il existe un grand nombre d'applications en informatique, et plus particulièrement en cryptographie, qui reposent sur la génération de nombres aléatoires. Mais comment générer des nombres aléatoires à partir d'un ordinateur, qui est fondamentalement une machine déterministe ?

Pour simuler le hasard, une première idée est d'utiliser la représentation binaire du temps indiqué par l'horloge de l'ordinateur (par exemple, le samedi 18 novembre 2023 à 9 heures, 3 minutes, 45 secondes, 17 centièmes et...). Ceci fournit une séquence de bits qu'on peut considérer comme plus ou moins aléatoire (en considérant au besoin une sous-séquence de celle-ci) et qui représente un nombre entier donné. Mais comment choisir ensuite d'autres nombres entiers aléatoires ? Si on consulte l'horloge de l'ordinateur à intervalles réguliers pour trouver d'autres nombres, forcément qu'un motif répétitif fera irruption dans la séquence et endommagera la côté aléatoire de celle-ci.

On peut cependant retenir le premier nombre ainsi trouvé comme *graine* (ou *seed* en anglais) à fournir à un algorithme qui, à partir de là, génère une séquence de nombres qu'on espère "les plus aléatoires possibles". Encore une fois, vu que les ordinateurs sont des machines déterministes, l'algorithme, quel qu'il soit, ne pourra pas fournir de vrais nombres aléatoires : on parle donc de générateurs *pseudo*-aléaoires. En voici quelques exemples.

1.1 La méthode des carrés tronqués (Von Neumann, vers 1950)

Une première idée pour générer des séquences de (grands) nombres aléatoires est la suivante. A partir d'un nombre à 8 chiffres, par exemple $x = 30472901$, calculons son carré :

$$x^2 = (0)928597695355801$$

et ne retenons que les 8 chiffres du milieu de celui-ci (en rouge) pour le prochain nombre, et ainsi de suite... Cette idée, pour ingénieuse qu'elle soit, génère cependant une séquence de

1. Cette section reprend des éléments du rapport "Générateurs de nombres aléatoires", par J.-C. Barros, Y. Dethurrens, D. Kessler et J.-F. Ravoux, juillet 2021

nombres qui est loin d'être aléatoire : au bout d'un moment, la séquence ainsi produite tombe sur un nombre qui se répète (comme par exemple $x = 60 \to x^2 = 3600$) ou effectue une boucle à travers quelques valeurs seulement (comme par exemple $x = 57 \to x^2 = 3249 \to (0)576$).

1.2 Les "générateurs à congruence linéaire" (Lehmer, vers 1950)

La deuxième idée (qui a un nom bien savant…) est la suivante : on choisit tout d'abord deux nombres a et b fixés, ainsi qu'un nombre m (tous des nombres entiers positifs), et à partir d'un nombre x, on calcule la nombre suivant en effectuant l'opération :

$$(a \cdot x + b) \,(\text{mod } m)$$

(où pour rappel, $z \,(\text{mod } m)$ désigne le reste de la division de z par m, donc un nombre compris entre 0 et $m - 1$). Par exemple, en choisissant $a = 4$, $b = 3$ et $m = 9$, on obtient la suite de nombres

$$x = 1 \to 7 \,(\text{mod } 9) = 7 \to 31 \,(\text{mod } 9) = 4 \to 19 \,(\text{mod } 9) = 1$$

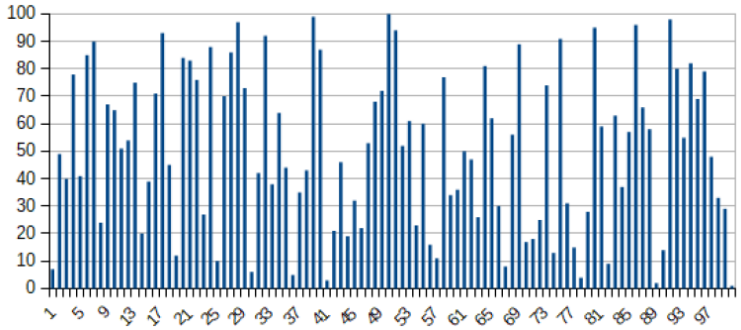
ou encore

$$x = 5 \to 23 \,(\text{mod } 9) = 5$$

Comme on le voit, cette méthode a aussi des défauts ! Mais notez qu'en choisissant $a = 4$, $b = 2$ et $m = 9$, on obtient le cycle le plus long possible pour cette valeur de m :

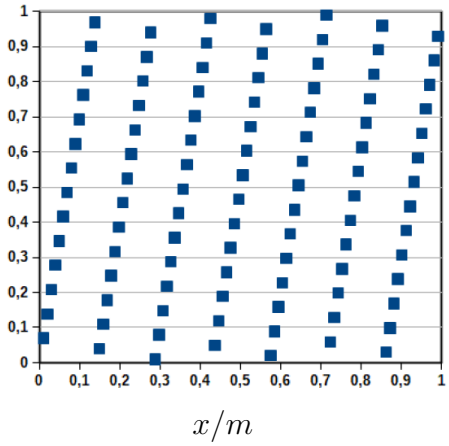
$$x = 1 \to 6 \to 8 \to 7 \to 3 \to 5 \to 4 \to 0 \to 2 \to 1$$

Il en va de même si on choisit $m = 101$, $a = 7$ et $b = 0$: la séquence des nombres ainsi générée explore toutes les valeurs possibles entre 0 et 100 avant de revenir au point de départ, comme le montre le graphique suivant :



et les nombres semblent explorés dans un ordre aléatoire. Mais si on représente maintenant la succession de ces nombres en deux dimensions :

$$\frac{(a \cdot x + b) \pmod m}{m}$$



x/m

on note alors le caractère grandement prévisible de cette séquence de nombres !

En choisissant des valeurs (nettement !) plus grandes de m, comme par exemple $m = 2^{32}$ (et $a = 129$, $b = 907633385$), on peut bien sûr obtenir des séquences de nombres plus intéressantes, mais le caractère prévisible observé ci-dessus ne disparaît pas.

1.3 Opérations binaires sur des nombres entiers en Python

Pour introduire la prochaine idée, nous avons besoin de quelques opérations binaires sur les nombres entiers en Python. Rappelons tout d'abord différentes représentations des nombres entiers : si x est un nombre entier (de type int), alors bin(x) et hex(x) permettent d'afficher (au format str) les représentations binaires et hexadécimales, respectivement, de ce même nombre entier. Par exemple, si $x = 156$, alors

$$\begin{cases} \text{bin}(x) = \text{“}0b10011100\text{”} & \text{car } 156 = 128 + 16 + 8 + 4 = 2^7 + 2^4 + 2^3 + 2^2 \\ \text{hex}(x) = \text{“}0x9c\text{”} & \text{car } 156 = 9 \cdot 16 + 12 \end{cases}$$

A noter pour toutes les opérations listées ci-dessous, il n'y a pas besoin de convertir les nombres en binaire avant d'effectuer ces opérations : Python utilise directement la représentation binaire des nombres dans ce cas !

1. L'opération XOR (notée x^y en Python) que nous avons vue la semaine dernière, s'effectue bit à bit sur la représentation binaire des deux nombres entiers x et y. Ainsi, si par exemple, $x = 156$ et $y = 17$, nous obtenons pour $z = x$^y :

```
  1 0 0 1 1 1 0 0 (x)
^ 0 0 0 1 0 0 0 1 (y)
--------------------
= 1 0 0 0 1 1 0 1 (z)
```

et donc $z = 128 + 8 + 4 + 1 = 141$.

2. De même, nous pouvons définir les opérations binaires ET (notée &) et OU (notée |) qui donnent respectivement sur les mêmes nombres $x = 156$ et $y = 17$ que ci-dessus :

```
  1 0 0 1 1 1 0 0        1 0 0 1 1 1 0 0
& 0 0 0 1 0 0 0 1      | 0 0 0 1 0 0 0 1
-----------------      -----------------
= 0 0 0 1 0 0 0 0      = 1 0 0 1 1 1 0 1
```

donc $x \,\&\, y = 16$ et $x \mid y = 128 + 16 + 8 + 4 + 1 = 157$.

3. Finalement, nous aurons encore besoin de deux autres opérations, notées << ("décalage à gauche") et >> ("décalage à droite") dont la fonction est de décaler (comme leur nom l'indique...) les bits qui composent un nombre d'un certain nombre de positions.

Par exemple, $x << 3$ veut dire décaler de 3 positions vers la gauche les bits qui composent le nombre x (en rajoutant des 0). Si par exemple $x = 156 = 0b10011100$ en binaire, alors $x << 3 = 0b10011100000 = 1248 = 156 \cdot 8$, vu que chaque décalage vers la gauche en binaire revient à multiplier le nombre par 2.

De même, $x >> 3$ décale de 3 positions vers la droite les bits du nombre x, en laissant éventuellement tomber les 1 qui se trouveraient aux 3 dernières places. Par exemple, si $x = 156 = 0b10011100$, alors $x >> 3 = 0b10011 = 19$ (qui n'est rien d'autre que le quotient de 156 par 8, où le reste de la division est négligé).

1.4 Algorithme Xorshift (Marsaglia, 2003)

Dans sa version la plus simple, l'idée de l'algorithme est la suivante : partir d'une "graine" d'une longueur de 32 bits, et effectuer un XOR de celle-ci avec une version décalée vers la gauche d'elle-même pour générer le prochain nombre. Ceci ne suffit toutefois pas (vous comprendrez pourquoi en faisant l'exercice correspondant) ; il faut encore effectuer à deux reprises une opération similaire pour obtenir une suite de nombres qui soit proche d'une suite aléatoire. Ceci donne l'algorithme suivant (en supposant qu'une valeur initiale a été attribuée à x) :

```
x = x^(x << 13)
x = x^(x >> 17)
x = x^(x << 5)
```

(à répéter n fois)

A noter que de nombreuses autres versions de l'algorithme existent, qui ont pour but de perfectionner celui-ci. Notamment, plus l'algorithme est sophistiqué, plus celui-ci passe un nombre important de tests qui vérifient le caractère aléatoire de la séquence ainsi produite : mais rentrer dans ces détails nous emmènerait un peu trop loin...

1.5 Et encore...

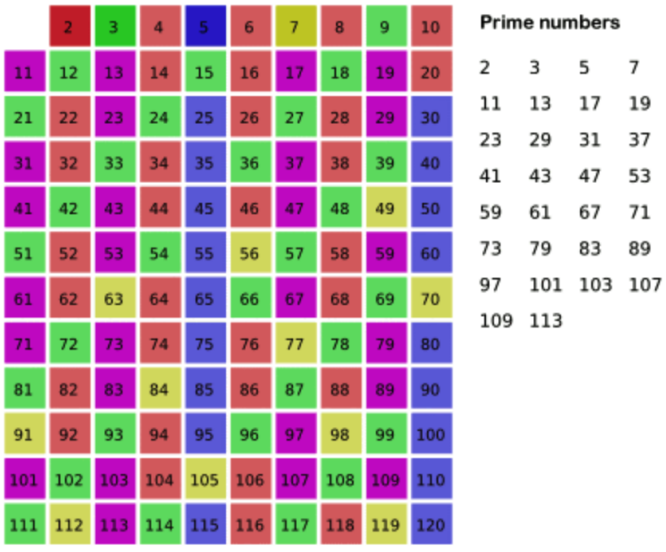
Pour produire des séquences de nombres pseudo-aléatoires, de nombreuses autres méthodes ont été inventées : citons notamment le "Mersenne twister", ainsi que d'autres algorithmes reposant sur la théorie du chaos, ou même le bruit atmosphérique ! (cf. `random.org`). Mais le générateur ultime de "vrais" nombres aléatoires reste à l'heure actuelle celui qui repose sur les propriétés quantiques de la matière au niveau microscopique, se basant en particulier sur la polarisation des photons, qui par nature se comporte de manière aléatoire lorsqu'on la mesure... A noter que cette dernière méthode reste beaucoup plus lente (et aussi plus coûteuse) que les précédentes.

2 La recherche de grands nombres premiers

Bon nombre d'algorithmes de cryptographie à clé publique, que nous découvrirons dans les prochaines leçons, reposent sur un ingrédient essentiel, à savoir l'utilisation de grands nombres premiers (pour rappel, un nombre premier est un nombre qui n'admet que deux diviseurs distincts : 1 et lui-même). Par "grands nombres premiers", on entend ici de *vraiment grands* nombres premiers, par exemple des nombres à 100 chiffres. La tâche de trouver de tels nombres est un peu ardue, mais nous allons progresser pas à pas.

2.1 Le crible d'Eratosthène (environ 200 avant J.-C.)

Cette (première !) méthode pour trouver des nombres premiers est la suivante : l'idée est de dresser la liste de tous les nombres, puis de rayer systématiquement tous les nombres qui sont des multiples de 2, de 3, de 5, etc. (notez que 4 étant un multiple de 2, il n'y pas besoin de rayer les multiples de 4, qui sont donc tous aussi des multiples de 2) : le nombres qui restent sont les nombres premiers. Voici une illustration de cette méthode, où les nombres premiers sont ceux colorés en rose (auxquels il faut encore ajouter les premiers nombres de chaque autre couleur) :



Cette méthode est certes très intéressante pour établir la liste des nombres premiers existants, mais devient de plus en plus lente lorsqu'on cherche de plus en plus grands nombres : ça n'est donc pas ainsi qu'il faut procéder pour trouver des nombres premiers à 100 chiffres...

2.2 Le théorème des nombres premiers (Hadamard et de la Vallée Poussin, 1896)

Pour de plus grands nombres, une question qui vient tout de suite à l'esprit est la suivante : est-ce que des nombres premiers à 100 chiffres existent ? (car si tel n'est pas le cas, mieux vaut arrêter tout de suite...). A priori, vu qu'un nombre à 100 chiffres est de l'ordre de 10^{100}, il a donc aussi de l'ordre de 10^{100} diviseurs potentiels : il est donc justifié de se poser la question ! C'est cependant un fait qui est connu depuis Euclide : il existent une infinité de nombres premiers.

Mais vient maintenant la deuxième question : vu que plus un nombre grandit, plus il a de diviseurs potentiels, il doit donc être vrai que les nombres premiers se *raréfient* plus on va vers de grands nombres. Par exemple :

- parmi les 10 premiers nombres, on trouve 4 nombres premiers (donc 40%) ;

- parmi les 100 premiers nombres, on trouve 25 nombres premiers (donc 25%) ;

- parmi les 1000 premiers nombres, on trouve 168 nombres premiers (donc 17% environ) ;

- etc.

Si les nombres premiers se raréfiaient trop lorsque qu'on va vers de grandes valeurs, la recherche de grands nombres premiers serait problématique... Heureusement, le *théorème des nombres premiers* nous dit que cette raréfaction est plutôt lente, à savoir :

$\pi(N)$, le nombre de nombres premiers plus petits ou égaux à un nombre donné N,
est approximativement donné par $\pi(N) \simeq \frac{N}{\ln(N)}$

où $\ln(N)$ est le *logarithme népérien de* N. Ceci veut dire que la *proportion* de nombres premiers plus petits ou égaux à N est de l'ordre de $\frac{1}{\ln(N)}$.

La fonction $\ln(N)$ est une fonction qui grandit très lentement avec N (par exemple, $\ln(1000) \simeq 6.9$, tandis que $\ln(1000000) \simeq 13.8$), donc la proportion $\frac{1}{\ln(N)}$ décroît très lentement avec N.

A nouveau, si N est un nombre à 100 chiffres, qu'implique ce résultat ? Dans ce cas, $N \simeq 10^{100}$ et on peut calculer que $\ln(N) \simeq 230$: même si N est gigantesque, $\ln(N)$ a une valeur assez faible : ceci veut dire en pratique que si on tire complètement au hasard un nombre à 100 chiffres, on a environ une chance sur 230 de tomber sur un nombre premier ; c'est une probabilité certes assez faible, mais suffisamment grande pour ne pas se décourager, car en effectuant 230 essais, on obtient une probabilité non-négligeable de tomber au moins une fois sur un nombre premier. Reste maintenant la question épineuse de trouver une manière efficace de *tester* si un nombre donné N est premier : pour cela, nous vous renvoyons aux exercices, ainsi qu'à la semaine prochaine !