TP 10. Représentation des nombres.

CONSIGNES

- Lors de l'écriture d'une fonction, on utilisera un seul `return`.

Observations

Absorption

Question 1. Tester dans le shell ces 3 propositions et discuter les résultats.

Python shell
```
>>> 1.0 + (2**53 - 2**53)
>>> (1.0 + 2**53) - 2**53
>>> (1 + 2**53) - 2**53
```

Des erreurs d'arrondi

Question 2. Tester dans le shell cette proposition et discuter le résultat.

Python shell
```
>>> (0.1+0.2) - 0.3 == 0
```

Phénomène de cancellation

Question 3. Tester dans le shell ces 2 propositions et discuter les résultats.

Python shell
```
>>> 1/1000-1/1001
>>> 1/(1000*1001)
```

Recommandation : ne jamais tester l'égalité entre deux nombres flottants, mais tester si leur distance est inférieure à un nombre très petit.

Détermination du nombre de bit de la mantisse d'un flottant

On se propose de vérifier que le stockage de la mantisse d'un flottant `python` s'effectue sur 52 bits.

On note $mc = m - 1$, `m` étant la mantisse du flottant. `mc` est la valeur stockée en mémoire en binaire.

L'idée est de se servir du nombre 0.5 dont on connait parfaitement la décomposition binaire :

$$0.5 = \frac{1}{2} = 1 \times \frac{1}{2} + 0 \times \frac{1}{4} + 0 \times \frac{1}{8} ...$$

On observe qu'en divisant `mc` successivement par 2 on obtient :

$mc = 0.5$	$m = 1.5$	stockée en mémoire sous la forme 1000...000
$mc = 0.25$	$m = 1.25$	stockée en mémoire sous la forme 0100...000
$mc = 0.125$	$m = 1.125$	stockée en mémoire sous la forme 0010...000
...		
$mc = 0.00...1$	$m = 1.00...1$	stockée en mémoire sous la forme 0000...001

Au bout d'un nombre suffisamment grand de divisions par 2 le chiffre 1 disparait complètement. En comptant le nombre de divisions par 2 nécessaires pour aboutir à 0, on a accès au nombre de bits disponibles pour coder `mc`.
On propose l'algorithme suivant :

Pseudo Code

initialisation (à compléter)
tant que $1 + mc \neq 1$ **faire**
  $mc \leftarrow mc/2$
  $i \leftarrow i + 1$
fin
retourner i

Question 4. Commenter ou compléter l'algorithme proposé :

- compléter la partie initialisation
- justifier le type de boucle choisie
- repérer la condition d'arrêt
- invariant de boucle
- la boucle a-t-elle une fin ?
- l'algorithme effectue t-il ce que l'on attend ?

Question 5. Implémenter cet algorithme dans `python`. Conclure quant au nombre de bits disponibles pour coder la mantisse d'un flottant.
Il est possible d'appliquer la méthode `hex()` sur un flottant pour avoir sa représentation en hexadécimal.

Python shell
```
>>> f=5.25
>>> f.hex()
'0x1.5000000000000p+2'
```

Cela dit que 5.25 est représentée par le nombre $1.(5000000000000)_{16} \times 2^2$ dont la mantisse est 1.5000000000000 et l'exposant 2 .

Question 6. Déterminer la mantisse de $\sqrt{2}$ à partir de son expression hexadécimale.

Génération de nombres pseudo aléatoires

On se propose d'étudier un algorithme permettant de générer des nombres pseudo-aléatoires.

Générer des nombres aléatoires

Un générateur de nombres aléatoires, `Random Number Generator` (RNG) en anglais, est un dispositif capable de produire une séquence de nombres dont on ne peut pas « facilement » tirer des propriétés déterministes. Cette fonctionnalité est présente dans Python avec la bibliothèque `random`.
Des méthodes pour obtenir des nombres aléatoires existent depuis très longtemps et sont utilisées dans les jeux de hasard : dés, roulette, tirage au sort, mélange des cartes, etc...

Ces générateurs ont une utilité dans de nombreux domaines. Outre les jeux, on peut citer :

- la simulation (phénomènes physiques aléatoires) ;
- l'échantillonnage ;
- la prise de décision ;
- la sécurité informatique (cryptologie, génération de clé).

Les critères suivants permettent de définir la qualité d'un générateur pseudo aléatoire :

- **la vitesse** : il faut que le calcul du nombre pseudo-aléatoire suivant soit rapide. Il n'est pas rare de devoir générer des millions de nombres.
- **la méthode ne doit pas souffrir de faille grave** : l'histoire des générateurs pseudo-aléatoire est pleine d'algorithmes qui se « coincent » lorsqu'ils arrivent sur un nombre particulier. Cette situation est source de bogues informatiques qui semblent eux aléatoires et qui sont par conséquent très difficiles à mettre en évidence.
- **les nombres produits ne doivent pas faire apparaître de suite logique**, quelle que soit la façon de les regarder. Ce critère est le plus difficile à quantifier car il dépend fortement de l'application.
- **chaque nombre doit apparaitre de manière équiprobable** : c'est-à-dire qu'on doit avoir un nombre de chance équivalent d'obtenir chaque nombre.

On dit qu'un générateur pseudo-aléatoire est acceptable s'il a passé avec succès toute une série de tests de statistiques généraux.

Nous allons travailler avec l'algorithme de génération de nombres pseudo aléatoires défini par D.H.Lehmer en 1948 :

$$U_{n+1} = (a \times U_n + c) mod(m).$$

Le premier terme U_0 est appelé "graine" (seed en anglais). Les choix du multiplicateur `a`, de l'incrément `c`, du module `m`, et de U_0 conditionnent la pertinence des nombres obtenus.

Pour la suite du TP, nous prendrons les valeurs suivantes :

$$\begin{cases} U_0 = 13 \\ U_{n+1} = f(U_n) \\ \text{où f est une fonction qui a } x \text{ associe } (16805 \times x + 1) mod 2^{15} \end{cases}$$

a mod b désigne le reste dans la division euclidienne de a par b.

Question 7. Écrire une fonction `f(u)` qui à partir du terme U_n, renvoie le terme suivant U_{n+1} de la suite. Cette fonction prendra pour argument `u`. Afficher les 1000 premiers termes de la suite. Les nombres générés vous paraissent-ils aléatoires ?

Génération de booléens

Nous allons nous servir des nombres pseudo-aléatoires générés pour générer des booléens, par exemple pour simuler une suite de tirages à `pile` ou `face`.
Nous avons deux situations possibles : soit `pile`, soit `face` que l'on peut rapprocher du "0" et du "1" de la représentation binaire.
La méthode consiste à convertir les termes de la suite décrite précédemment en nombres binaires. On extrait alors le n$^{\text{ième}}$ bit du nombre généré (on compte le n$^{\text{ième}}$ bit à partir de la droite).
Si ce bit est à "1" alors le tirage est `pile`, `face` dans l'autre cas.

Question 8. Écrire la fonction `binaire(e:int)` qui convertit un entier en base 2. Cette fonction aura pour argument l'entier `e` à convertir et retournera une chaine de caractère constituée de 0 et de 1.
Vérifier votre travail en comparant avec la fonction `bin` de `python` sur plusieurs exemples.

Question 9. Écrire une fonction `booleen(e:int,n:int)->str` qui convertit un entier en base 2 et renvoie le n$^{\text{ième}}$ bit sous la forme '0' ou '1' (n$^{\text{ième}}$ bit compté à partir de la droite). Cette fonction aura pour argument un entier `e` et la position `n` du bit à extraire.
Attention, il faut traiter les cas où le nombre obtenu est codé sur moins de n bits !

Question 10. On choisit d'utiliser le bit de poids faible (bit des unités) de chaque élément de la suite U_n. Vérifier l'équiprobabilité de la méthode en comptant le nombre de fois que sort le booléen 1 (tirage `face`) sur un test sur 10000 tirages. Afficher la liste des 100 premiers termes.
Conclure quant à la pertinence du choix du bit de poids le plus faible.

Question 11. Répondre à la question précédente en choisissant le 9$^{\text{ième}}$ bit.

Génération d'un entier quelconque

Question 12. Loin d'être réellement aléatoire, la suite de Lehmer est en fait périodique. On propose d'observer cette propriété. Déterminer au bout de combien de tirages le nombre initial 13 réapparait. En déduire la période (apparente) de la suite.

Question 13. Vérifier que chacun des nombres de l'intervalle des entiers `[0,32768[` n'apparaissent qu'une seule fois sur une période. Pour cela, réaliser un programme qui affiche le nombre d'apparitions s'il est différent de 1, et qui affiche **Tous les nombres apparaissent une seule fois** si c'est le cas.
Conclure quant à l'équiprobabilité.

Remarque : Pour créer une liste ne contenant que des zéros, on peut utiliser la syntaxe suivante :

Python shell
```
>>> liste = [0]*7 # creation d'une liste avec 7 elements qui valent tous 0
>>> liste [0,0,0,0,0,0,0]
```

Question 14. (*optionnelle*) Tester la fonction `randint` de `python`. Cette fonction est disponible en important la bibliothèque `random`. Vous pouvez notamment évaluer sur un échantillon suffisamment grand (100000 tirages par exemple) le nombre d'apparition de quelques entiers.

Le 9$^{\text{ième}}$ bit permet de générer une séquence suffisamment aléatoire de `pile` et de `face` suivant les critères donnés en introduction. Malheureusement, puisque l'on part toujours de $U_0 = 13$, la séquence générée est toujours la même. On souhaite donc créer un entier U_0 compris entre 0 et $2^{15} - 1$, qui serait aléatoire.

Question 15. Proposer une méthode pour créer un tel entier, en se servant de la fonction `time.perf_counter()` ou `time.perf_counter_ns()` (voir en bas de page).
Écrire une fonction `graine()` qui retourne cet entier aléatoire.
Écrire une fonction `aleatoire(n:int)` qui génère une liste de n `pile` et `face` (0 et 1) obtenue avec le 9$^{\text{ième}}$ bit, en partant de cet entier aléatoire. Cette fonction prendra comme argument le nombre `n` de booléens voulus (taille de la liste à retourner).

`time.perf_counter() → float`
Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration.
`time.perf_counter_ns() → int:`
Similar to perf_counter(), but return time as nanoseconds (new in version 3.7).
`time.process_time() → float`
Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process.
`time.process_time_ns() → int`
Similar to process_time() but return time as nanoseconds.