Informatique en CPGE (2018-2019)
TP 5 : représentation des nombres

1 Nombres entiers naturels

Exercice 1 : division euclidienne.

Ecrire une fonction `divise`, prenant en paramètres un entier naturel `a` et un entier naturel `b` non nul, qui calcule et renvoie le quotient et le reste obtenus par la division euclidienne de `a` par `b`. Il est interdit d'utiliser les opérateurs // et % ni la fonction `divmod`. (`divmod(a,b)` renvoie `(a//b,a%b)`)

Exercice 2 : codage d'un entier naturel.

1. Ecrire une fonction qui renvoie l'écriture en base deux d'un nombre entier naturel exprimé en base dix. Le paramètre en entrée est de type `int` et la valeur renvoyée en sortie de type `str`.
2. Ecrire une fonction `test` qui prend en paramètre une chaîne de caractères représentant l'écriture binaire d'un entier naturel et renvoie `True` s'il s'agit bien d'une écriture correcte en base deux et `False` sinon.
3. Ecrire une fonction qui renvoie l'écriture en base dix d'un nombre entier naturel exprimé en base deux. Le paramètre en entrée est de type `str` et la valeur renvoyée en sortie de type `int`.

2 Nombres entiers relatifs

Exercice 3 : opérations sur les bits.

1. L'opération `~a` inverse les valeurs des bits du nombre `a`. Pour différentes valeurs de `a`, tester les instructions `~a`, `-(a+1)`, `~(~a)`, `~a+1`, `~(a-1)`, et commenter les résultats.
   Expliquer en particulier pourquoi `~a+a` vaut -1.
2. L'opérateur `&` compare deux nombres bit à bit, écrit 1 chaque fois que les deux bits sont égaux à 1 et écrit 0 sinon. Par exemple `3 & 2` donne `2`, `5 & 2` donne `0`.
   Expliquer pourquoi `~a & a` vaut 0.
3. L'opérateur `^` compare deux nombres bit à bit, écrit 1 chaque fois que les deux bits correspondants sont différents et écrit 0 sinon. Par exemple `4 ^ 3` vaut 7, `7 ^ 2` vaut 5.
   Expliquer pourquoi `a ^ b ^ b` vaut `a`.

Exercice 4 : codage d'un entier relatif.

Ecrire une fonction qui renvoie le codage en binaire d'un entier relatif exprimé en base dix. Le nombre en entrée est de type `int` et la sortie de type `str`.

La fonction prend un deuxième paramètre qui représente le nombre de bits utilisés pour coder les nombres et renvoie `False` si le nombre ne peut pas être codé.

Rappel : si le codage se fait sur n bits, on peut coder les nombres positifs r de 0 à $2^{n-1}-1$ (par l'écriture en binaire de r) et les négatifs r de -2^{n-1} à -1 (par l'écriture en binaire de $r+2^n$). Le programme doit donc tester si $r \in [0\,;2^{n-1}-1]$ ou si $r \in [-2^{n-1}\,;-1]$. On peut alors utiliser l'exercice 2.

Attention : par exemple, sur 8 bits, si $r=5$, le programme doit afficher en sortie 00000101.

3 Nombres réels

Exercice 5 : approximations d'un réel.

On considère la suite (u_n) définie par $u_0=3$ et $u_{n+1}=\frac{1}{2}(u_n+\frac{2}{u_n})$. On démontre que cette suite est décroissante et minorée, donc convergente. Ecrire un algorithme qui permet de calculer u_{100}, le programmer et l'exécuter. Emettre une conjecture sur la limite de cette suite.

Que penser du test `sqrt(2)*sqrt(2)==2` ?

En utilisant la fonction `Decimal` du module `decimal`, donner l'écriture décimale du nombre `a=sqrt(2)*sqrt(2)`.

Note 1 : pour comparer deux nombres l'instruction `x==y` ne peut pas convenir ; tester par exemple `1+2.22e-16==1` et `1+1e-16==1`. Donc plutôt que tester si deux flottants sont égaux, on testera si deux flottants sont assez proches l'un de l'autre, par exemple avec le test : `abs(x-y)<0.0001`.

Note 2 : il y a un risque d'avoir des nombres flottants trop grands. L'expression `10.0**400` affiche une erreur "overflow", l'expression `10.0**100*10.0**300` affiche "inf". Attention, la commande`10**400` affiche bien "1" suivi de 40 zéros. (Les nombres de type `int` sont illimités en Python)

Exercice 6 : le codage d'un flottant x est en norme IEEE 754 sur 64 bits soit 1 bit pour le signe s, 11 bits pour l'exposant décalé e et 52 bits pour la mantisse tronquée m : $x = (-1)^s \times (1, m) \times 2^{e-1023}$.

1. Comment est codé le nombre réel 1,0 ? Quel est le plus petit flottant α strictement supérieur à 1 que l'on peut coder ? Que vaut $\alpha - 1$?
2. Dans l'interpréteur Python, importer le module **sys** (instruction `import sys`) ;
   écrire `sys.float_info` et analyser la réponse.
3. Comment est codé le plus grand flottant ? Que valent sa mantisse et son exposant ?

Exercice 7 : codage d'un flottant.

Ecrire une fonction qui détermine et renvoie le codage d'un flottant exprimé en base dix. L'entrée sera de type `float` et la sortie de type `str`.

Les étapes :

1. La variable x a pour valeur le flottant entré en paramètre de la fonction.
2. On détermine le signe de x et que l'on stocke dans s='0' ou s='1' et on change $x = |x|$.
3. On calcule l'exposant et la mantisse. Pour cela si $x >= 2$ on fait des divisions par 2 successives, si $x < 1$ on fait des multiplications par 2 successives, en remplaçant à chaque fois la valeur de x par le résultat obtenu, et dans les deux cas jusqu'à obtenir un nombre x tel que $1 \leq x < 2$; l'exposant est alors le nombre de divisions ou de multiplications effectuées et la mantisse est le nombre x final.
4. On calcule l'exposant décalé que l'on code en binaire sur 11 bits (voir exercice 2). Le résultat est stocké dans une chaîne b.
5. On calcule la mantisse tronquée $x = x - 1$ que l'on doit alors écrire en binaire sur 52 bits et que l'on stocke dans une chaîne m. Pour cela on multiplie x par 2 ($x = 2 * x$) ; si $x \geq 1$, on ajoute '1' à m et on retranche 1 à x, sinon on ajoute '0' à m ; on reproduit ce schéma 52 fois.
6. On renvoie la chaîne concaténée s + b + m.

Exercice 8 : accumulations d'erreurs d'arrondis.

Commenter le code suivant :

```
>>> (0.1+0.1+0.1-0.3)*10**20
5551.115123125783
```

1. En utilisant la fonction `Decimal` du module `decimal`, donner la représentation décimale de 0.1, puis vérifier avec le programme de l'exercice 7 que la représentation sur 64 bits de 0,1 est
   0 011 1111 1011 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010.
2. Comparer les quantités `Decimal(0.3)`, `Decimal(0.1+0.1+0.1)`
   et `Decimal(0.1)+Decimal(0.1)+Decimal(0.1)`.

Note : pour travailler avec des fractions exactes, on utilise le module `fractions` qui permet de manipuler des objets modélisant les fractions. On utilise la fonction `Fraction` de la manière suivante :

```
from fractions import Fraction as f
>>> print(f(2,3))
2/3
>>> f(2,3)+f(5,2)
Fraction(19, 6)
>>> 0.1==f(1,10)
False
>>> f(1,10)+f(1,10)+f(1,10)
Fraction(3, 10)
```