REPRÉSENTATION DES NOMBRES

Exercice 1 : Représentation des entiers

1. Donner la représentation binaire de 76 en base 2 sur 8 bits.
2. Écrire une fonction `bin` prenant en argument deux entiers naturels k et n, et renvoyant la représentation binaire de k sur n bits, sous forme de liste dans l'ordre des puissances décroissantes de 2. On supposera que k est représentable sur n bits.

   Par exemple, `bin(13,8)` doit renvoyer `[0, 0, 0, 0, 1, 1, 0, 1]`.
3. La méthode de Horner permet de calculer la valeur d'un nombre représenté en binaire, sans avoir à calculer la suite des puissances de 2. Elle se base sur l'identité :

$$\sum_{i=0}^{n} a_i 2^i = (\ldots(a_n \times 2 + a_{n-1}) \times 2 + \ldots) \times 2 + a_1) \times 2 + a_0$$

   Par exemple, pour calculer la valeur du nombre représenté par 1110 1001, on part de 1, on multiplie par 2 (2), on ajoute 1 (3), on multiplie par 2 (6), on ajoute 1 (7), on multiplie par 2 (14), on ajoute 0 (14), on multiplie par 2 (28), on ajoute 1 (29), on multiplie par 2 (58), on ajoute 0 et on multiplie par 2 (116), on ajoute 0 et on multiplie par 2 (232), et on ajoute 1. Le nombre représenté est donc 233.

   Appliquer cet algorithme pour déterminer la valeur du nombre représenté par 1101 0011.
4. Écrire une fonction `Horner` prenant en argument une liste de bits et implémentant cet algorithme.

Exercice 2 : Représentation en complément à 2

1. Rappeler l'intervalle représentable en complément à 2 sur 8 bits et donner la représentation de -4.
2. Écrire une fonction `vers_complement_a_2` prenant en argument un entier relatif k et un entier naturel n, et renvoyant la représentation de k en complément à 2 sur n bits sous forme de liste, et sa fonction réciproque `depuis_complement_a_2`.
3. Calculer l'opposé directement sur la représentation 1101 1001.
4. Écrire une fonction `opposé` prenant en argument une liste et calculant la représentation opposée. On travaillera directement sur la représentation, sans utiliser les fonctions précédentes.

Exercice 3 : Représentation des flottants

1. Rappeler le principe de représentation d'un nombre flottant en Python.
2. Doit-on s'attendre à ce que le booléen `0.1 + 0.7 == 0.8` soit forcément vrai ? Vérifier en Python.
3. On considère le bloc d'instructions suivant :

```
a = 1.0
while 10*a != float("inf"):
        a = a*10
print(a)
```

   Que va-t-il afficher ? Tester en Python.
4. En suivant le même principe, obtenir la valeur de la plus petite puissance de 10 positive représentable en Python.
5. On considère maintenant cette suite d'instructions :

```
a = 1.0
while a != a+1 :
        a = a*10
print(a)
```

   Interpréter son résultat.

Exercice 4 : Entiers multiprécision en Python

1. La taille des entiers n'est pas limitée en Python, ce qu'on peut observer avec le bloc d'instructions suivant ((qu'on interrompra avant de saturer la mémoire de l'ordinateur avec le raccourci ctrl+I) :

```
a = 1
while True :
        a = a*10
        print(a)
```

   En pratique, si un entier dépasse la taille maximale utilisable directement par le processeur, ce nombre est découpé en deux ou plusieurs parties et Python s'occupe des différentes opérations à effectuer. Chaque calcul est alors plus coûteux en temps et en espace mémoire.
   La fonction `time` de la bibliothèque `time` permet de calculer un temps depuis un moment de référence. Ainsi, en écrivant

```
t1 = time()
#calcul
t2 = time()
```

   on obtient avec la différence `t2-t1` le temps pris par le calcul entre les 2 appels à `time`. Écrire une fonction `C` prenant en argument un entier n et calculant le temps pris pour réaliser un million de fois l'addition $10^n + 10^n$.

2. Tracer la courbe représentative de `C`, pour des valeurs de n allant de 0 à 1000 avec un pas de 100.

Exercice 5 : Calculs en complément à 2

1. Additionner les représentations suivantes en complément à 2 sur 8 bits :
   (a) 1101 1010 et 1011 0010
   (b) 0101 0110 et 0110 0011
   (c) 0111 1011 et 1011 1000
2. Dans quel(s) cas y-a-t-il eu dépassement arithmétique ?
3. Écrire une fonction `addition` prenant en argument deux listes représentant des nombres en complément à 2 (sur un même nombre de bit) et calculant leur somme en travaillant directement sur les représentations. En plus de renvoyer son résultat, la fonction affichera à l'écran "Attention : dépassement arithmétique" le cas échéant.
4. En déduire une fonction réalisant la soustraction en complément à 2.

Exercice 6 : Annulation catastrophique

1. On veut calculer les racines d'une équation du second degré : $ax^2+bx+c=0$ pour laquelle le discriminant $\Delta > 0$. Voici une première façon de faire :

```
def racines1(a,b,c):
    Delta = b**2-4*a*c
    if Delta > 0:
        s = Delta**(1/2)
        return (-b-s)/(2*a), (-b+s)/(2*a)
```

   On décide de l'appliquer pour le trinôme $ax^2+\frac{1}{a}x-a=0$, pour $a>0$. Le cours nous dit alors que les solutions sont

$$x_1 = \frac{-\frac{1}{a}-\sqrt{\frac{1}{a^2}+4a^2}}{2a} \quad \text{et} \quad x_2 = \frac{-\frac{1}{a}+\sqrt{\frac{1}{a^2}+4a^2}}{2a},$$

   et les relations coefficients/racines montrent que dans tous les cas $x_1x_2 = \frac{c}{a} = -1$. Tester pour $a = 7\times10^{-i}$ pour $1 \leq i \leq 8$. Affichez à chaque fois le produit des racines. Qu'observe-t-on ?
2. Tester les lignes de code suivantes. À quoi vous attendez vous ? Que remarquez vous ?

```
from math import sqrt
M = sqrt(2)*(1+10**(-14))
m = sqrt(2)
print(M-m)
```

   L'annulation catastrophique ou *cancellation* est le nom donné à la perte de précision qui se produit lors de la soustraction de deux flottants très proches. En effet, il est alors possible de perdre de nombreux chiffres utiles, alors même que le résultat théorique peut être stocké dans un flottant.
3. Proposer une fonction `racines2` limitant le problème observé dans la version précédente.