# TP 11 : Représentation de nombres

## Introduction

Dans ce TP, nous allons coder des représentations de nombres entiers et utiliser des flottants pour des approximations.
On écrira un entier en base $b$ comme une liste, que nous lirons de droite à gauche (dans ce TP !) pour faciliter la programmation. En base 2, la liste `[0,0,1,1]` représente $1100_2$, qui s'écrit

$$0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3$$

soit 12. Cette lecture n'est pas standard : d'autres codages existent : avec des listes de gauche à droite, avec des chaînes de caractères...si vous finissez assez tôt, vous pourrez adapter les fonctions pour des chaînes de caractères.

## I Représentation de nombres en base b

**Exercice 1** *Base 2 vers base 10*
Écrire une fonction `bin_vers_dix` qui prend en entrée une liste de 0 et 1 et calcule, en base 10, le nombre représenté par cette liste en base 2. On s'appuiera sur la définition avec une boucle `for`.

**Attention :** pour simplifier la tâche, la liste sera lue de droite à gauche :

```
def bin_vers_dix(liste) :
    s = 0
    for k in range(___) :

    return s
```

N'oubliez pas de **tester** votre fonction !

**Exercice 2** *Base b vers base 10*
Adapter la fonction `bin_vers_dix` pour prendre en entrée un entier $b$ et une liste de nombres entre 0 et $b-1$ et calcule, en base 10, le nombre représenté par cette liste en base $b$.

**Attention :** pour simplifier la tâche, la liste sera lue de droite à gauche : la liste `[0,0,1,1]` représente $0 \times b^0 + 0 \times b^1 + 1 \times b^2 + 1 \times b^3$.

```
def base_b_vers_dix(b,liste) :
    s = 0
    for k in range(___) :

    return s
```

**Testez** votre fonction !

**Exercice 3** *Schéma de Hörner*
Écrire une fonction `horner_b_dix` qui prend en entrée une liste de 0 et 1 et calcule, en base 10, le nombre représenté par cette liste en base $b$ en utilisant l'algorithme de Hörner.
On rappelle que l'idée de cet algorithme est que :

$$a_n b^n + \cdots + a_1 b + a_0 = (\cdots((a_n b + a_{n-1})b + a_{n-2})b + \cdots)b + a_0$$

**Attention :** cette fois la lecture de droite à gauche de la liste est plus handicapante. `def horner_b_dix(b,liste) :`

**Exercice 4** *Bonus - Hexadécimal vers base 10*
Adapter la fonction `base_b_vers_dix` pour évaluer des nombres écrits en hexadécimal (et donc avec des lettres dans la liste des entrées !)
On pourra écrire une fonction auxiliaire `lettre_hexa_vers_dix` qui renvoie le chiffre en base 10 correspondant au chiffre hexadécimal (lettre ou chiffre).

```
def lettre_hexa_vers_dix(u):
    if u == 'A' :
        return ___
    elif u == 'B' :
```

```
def hexa_vers_dix(liste) :
```

## II Opérations en binaire

On va programmer maintenant la somme et le produit en binaire. Même si les programmes que nous écrirons ne sont pas optimisés, ils donnent une très bonne idée de ce qui se passe lors d'un calcul de l'ordinateur. On commence par ajuster la longueur de deux listes, ce qui est primordial pour une addition, puis on code l'addition et le produit en binaire.

**Exercice 5** *Ajuster la longueur*
Écrire une fonction `ajuste_longueur` qui prend en entrée deux listes et complète la plus courte avec des zéros pour qu'elles aient la même longueur.
Rappel : la commande `li.append(0)` ajoute un zéro à la fin d'une liste.

```
def ajuste_longueur(liste1,liste2) :
    n = len(liste1)
    m = len(liste2)
    li1 = liste1.copy()
    li2 = liste2.copy()
```

**Exercice 6** *Somme*
Écrire une fonction `somme_bin` qui prend en entrée deux listes représentant des nombres en binaire et renvoie une liste représentant la somme. On pourra commencer par ajuster les longueurs des deux listes, et on pensera à gérer les retenues (surtout pour le dernier chiffre de la somme).

```
def somme_bin(li1,li2):
    liste1 , liste2 = ajuste_longueur(li1,li2)
    somme = []

    retenue = 0

    for k in range(___) :
```

On pensera à **tester** la fonction avec des sommes connues en écriture décimale.

**Exercice 7** *Produit*
Écrire une fonction récursive `produit` qui calcule le produit de deux nombres écrits par des listes en binaire. On pourra utiliser la fonction `somme_bin`.
Rappel : le symbole + permet de concaténer des listes (et en particulier d'ajouter un zéro au début d'une liste.)

```
def produit(liste1,liste2):
    if liste2 == [] :
        return ___
    elif liste2[0] == 0 :
        return _____
    elif liste2[0] == 1 :
        return _____
```

On pensera à **tester** la fonction avec des produits connus en écriture décimale.

# III Flottants et approximations

**Exercice 8** *Une suite récurrente*
On définit une suite récurrente $(u_n)$ par $u_0 = \frac{1}{12}$ et pour $n \in \mathbb{N}$ : $u_{n+1} = 13u_n - 1$.

1. Montrer que la suite est constante égale à $\frac{1}{12}$
2. Écrire une fonction **itérative** `suite1(n)` qui prend un entier $n$ en paramètre et qui renvoie la valeur de $u_n$.
3. Écrire une fonction **récursive** `suite2(n)` qui prend un entier $n$ en paramètre et qui renvoie la valeur de $u_n$.
4. Afficher les valeurs de `suite1(k)` et `suite2(k)` pour $k \in [|0, 30|]$.
5. Expliquer pourquoi ce n'est pas le résultat attendu.
6. Écrire une fonction `suite3` qui corrige ce problème à l'aide du module fractions et de la fonction Fraction. (On pourra demander l'aide sur la console pour savoir comment utiliser cette fonction).

**Exercice 9** *Algorithme de Briggs*
Les flottants sont utilisés pour des calculs approchés. Par exemple, l'algorithme de Briggs donne une valeur approchée du logarithme d'un nombre. On utilise l'approximation $\log(u) \simeq u - 1$ si $u$ est proche de 1.
Si $x^{1/2^n} - 1 \simeq 0$, alors $\log(x^{1/2^n}) \simeq x^{1/2^n} - 1$, ce qui donne l'approximation $\log(x) \simeq 2^n(x^{1/2^n} - 1)$.
Programmer cet algorithme, et comparer les résultats avec ceux de `math.log`
On pourra prendre comme seuil, dans un premier temps $x - 1 < 0,0001$.

```
def briggs(x) :
    n = 0
    while abs(x-1) > 0.0001 :
```

**Exercice 10** *Flottant mystère*
Que fait le programme suivant ? Se termine-t-il ? Quel est le résultat donné ?

```
x = 1
n = 0
while x/2 > 0:
    n = n+1
    x = x/2
print(n,x)
```