Informatique Pour Tous

TD 24.3 – Codage
Conversions numériques : binaire entier relatif et float

1 / Conversion décimal → binaire en complément à 2

On donne ci-dessous la structure d'un code d'une fonction `into_uint(K)` permettant de convertir un nombre entier `K` en écriture binaire (la convention, écriture sous forme de liste) est la même qu'au TP/DM précédent.

```
def into_uint(K):    # K nombre entier positif

    # Etape 1 déterminer la plus petite puissance de 2 qui soit > K (strict)
    puiss2 = ...     # initialisation
    while ...
        puiss2 *= 2

    puiss2 = puiss2 // 2  # on peut donc écrire K comme une somme
                          # des puissances de 2 inférieures

    # Etape 2 écrire la liste en partant de la plus grande puissance de 2 jusqu'à 1
    L = []
    r = K     # par la suite, r désignera le reste à écrire en base 2
    while puiss2 > 1 :
        if ...    # on peut soustraire la puissance de 2 considérée au reste
            r = ...
            L.append(...
        else :
            L.append(...
        puiss2 = ...      # s'inspirer de ce qui précède

    # Etape 3 Attention ! on est sortis de la boucle sans pouvoir regarder puiss == 1
    L.append(... # ça peut être tentant de regarder la parité de K, mais y'a plus élégant
    return L
```

Q1 – Copier-coller, compléter le code ci-dessus.

Tests :

```
print(into_uint(0))     # doit afficher [0]
print(into_uint(1))     # doit afficher [1]
print(into_uint(10))    # doit afficher [1, 0, 1, 0]
print(into_uint(147))   # doit afficher [1, 0, 0, 1, 0, 0, 1, 1])
```

Q2 – Écrire une fonction `complement_gauche(N_bits, L)` prenant en argument une liste `L` et un nombre de bits `N_bits`, dont on admet qu'il est supérieur ou égal à la longueur de `L`. Cette fonction renverra une liste de longueur `N_bits` correspondant à la liste `L` qu'on a complété avec le bon nombre de `0` à gauche.

Tests :

```
print(complement_gauche(4,[1,0]))     # doit afficher [0,0,1,0]
print(complement_gauche(4,[1,0,1,0]))     # doit afficher [1,0,1,0]
```

Challenge : petit challenge pas trop difficile : faire tenir cette fonction sur 2 lignes : `def complem…` `return …`

Algorithme vu en cours : On rappelle que pour écrire un nombre K en complément à 2 sur un nombre de bits donné

- Si $K \geq 0$
  - Écrire K en binaire naturel, normalement
  - (Remarque : le 1er bit sera alors nécessairement = 0, ce qui est cohérent car la valeur codée est positive)

- Sinon $K < 0$
  - Écrire $|K+1|$ (ou, c'est équivalent, $|K|-1$) en binaire naturel
  - Faire le complémentaire de chacun des bits ! **Exemple :** 0 1 0 1 1 → 1 0 1 0 0
  - (Remarque : le 1er bit sera alors nécessairement = 1, ce qui est cohérent car la valeur codée est négative)

On impose donc, logiquement, la trame suivante, pour une fonction `into_int(Nbits, K)` permettant de convertir un entier *K* en binaire complément à 2 sur `Nbits`.

```
def into_int(Nbits, K):
    if K >= 0 :
        ...      # conversion de K en naturel
        return ...   # on complète pour une écriture sur Nbits

    else :
        ...     # conversion de |K|-1 = -K-1 en naturel
        L = ...     # on complète pour une écriture sur Nbits
        for i in range(len(L)):
            L[i] = ...   # on fait le complément à 2 de chaque bit, un à un
        return L
```

Aide : pour le complément à 2 bit à bit, que je vous demande d'essayer de faire tenir en une seule ligne, je vous conseille de regarder ceci :

```
>>> a, b = 1, 0
>>> not(a)    # ou not a   (c'est idem)
False
>>> not b    # ou not(b)   (c'est idem)
True

>>> int(False)
0
>>> int(True)
1
```

Q3 – Copier-coller le corps de la fonction `into_int(Nbits, K)`, compléter les trous.

Tests :
```
print(into_int(8, -89))     # affiche [1, 0, 1, 0, 0, 1, 1, 1]  cf cours
print(into_int(8,-128))    # affiche [1, 0, 0, 0, 0, 0, 0, 0]  cf cours
print(into_int(8,127))    # affiche [0, 1, 1, 1, 1, 1, 1, 1]  cf cours
```

2 / Conversion décimal → binaire par excès

On a vu dans le cours qu'en binaire **naturel**, *N* bits permettent d'écrire des valeurs entre 0 et $2^N - 1$. Le binaire par excès consiste à décaler toutes ces valeurs de moitié (arrondi à l'inférieur) afin de pouvoir coder des valeurs tant négatives que positives. Le décalage (valeur entière) est donc :

$$\Delta = \mathrm{E}\left[\frac{2^N - 1}{2}\right] = E\left[2^{N-1} - \frac{1}{2}\right] = 2^{N-1} - 1$$

($\mathrm{E}(x)$ désignant ici la partie entière de x)

De la sorte, convertir un entier relatif K en binaire par excès revient à convertir $K + \Delta$ (forcément positif) en binaire naturel (en complétant, si besoin, au nombre de bits imposé).

Q4 – Écrire une fonction `bin_exces(Nbits, K)` prenant en argument un nombre de bits `Nbits` et un entier relatif K, et renvoyant le codage de ce nombre en binaire par excès sur `Nbits`.

Remarque : comme dans tous les sujet d'infos, vous pouvez vous servir de toutes les fonctions auxiliaires qui précèdent (notamment `complement_gauche()`).

Tests :
```
print(bin_exces(8,-127))   # affiche [0, 0, 0, 0, 0, 0, 0, 0] cf cours
print(bin_exces(8,128))    # affiche [1, 1, 1, 1, 1, 1, 1, 1] cf cours
print(bin_exces(12,53))    # affiche [1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0]
```

3 / Conversion *float* [partie plus difficile]

Pour convertir un nombre x en flottant, la 1ère étape est de l'approcher par la somme de puissances de 2 (puissances pouvant être négatives) la plus proche, à $\pm\varepsilon$ près. On se fixe ici $\varepsilon = 5 \cdot 10^{-8}$.

Ainsi, nous avons par exemple vu en cours que :

- $(10.25)_{10} = 2^3 + 2^1 + 2^{-2} = (1010{,}01)_2$
- $(0.625)_2 = 2^{-1} + 2^{-3} = (0{,}101)_2$

Ces deux exemples sont « exacts », mais à $\pm\varepsilon$ près on peut aussi écrire :

- $1/3 \approx (0{,}0101010101\ldots\ldots)_2$: le nombre de bits après la virgule est de 22 pour avoir la précision de $5 \cdot 10^{-8}$

→ On veut écrire une fonction `decomposition_2(x)` qui renvoie deux listes : la listes des puissances de 2 positives, et la liste des puissances de 2 négatives.

Exemples : pour vos tests
```
print(decomposition_2(10.25))    # doit afficher ([1, 0, 1, 0], [0, 1])
print(decomposition_2(0.625))    # doit afficher ([0], [1, 0, 1])
print(decomposition_2(5.0))  # affiche ([1, 0, 1], []) ou ([1, 0, 1], [0])
```

Aide 1 : la liste de gauche correspond à la partie entière de x décomposée en base 2. Facile.

Aide 2 : pour la liste de droite, l'algorithme ressemblera beaucoup à celui de la 1ère fonction donnée en page 1, étape 2, sauf qu'elle s'exécute tant que le reste (qu'on prendra toujours positif) demeure supérieur à ε. Vous aurez donc :
```
while r > 5e-8 :
```

Q5 – Écrire et tester la fonction `decomposition_2(x)` ainsi décrite.

Maintenant qu'on a décrit un nombre x en deux listes : une liste `L_G` correspondant à sa partie entière et une liste `L_D` correspondant au reste $0 \leq x - \mathrm{E}[x] < 1$ qui s'écrit comme somme de puissances de 2 négatives, on veut déterminer le décalage de bits (vers la gauche ou la droite) qui permettrait d'écrire ce nombre en écriture scientifique (avec la caractéristique, 1er bit non nul, forcément égal à 1 en binaire).

En d'autres termes, on souhaite ici déterminer l'exposant e tel que l'on puisse écrire le nombre décrit sous la forme `[1],[...]` $\times 2^e$ où la seconde liste correspond à la mantisse. La fonction qu'on souhaite coder sera notée `ecriture_scientifique(L_G, L_D)`, elle renverra la liste de droite (correspondant à la mantisse) et le décalage (correspondant à l'exposant).

Exemples : reprenons les écritures sous la forme `L_G, L_D` des 3 valeurs `10.25, 0.625, 5.0` vues ci-dessus
```
print(ecriture_scientifique([1, 0, 1, 0], [0, 1])) # affiche ([0, 1, 0, 0, 1], 3)
print(ecriture_scientifique([0], [1, 0, 1]))       # affiche ([0, 1], -1)
print(ecriture_scientifique([1, 0, 1], []))        # affiche ([0, 1], 2)
```

La trame imposée est la suivante :
```
def ecriture_scientifique(L_G, L_D):
    if L_G != [0] :   # c'est donc que L_G est de la forme [1, ...]
        dec = ...  # valeur du décalage (exposant) : on sait s'il est positif ou négatif
        return ..., dec

    else :  # c'est donc que L_D est de la forme [0,...,0,1...], comptons le nbre de 0
        compt = ...    # initialisation du compteur
        while ...
            compt += 1
        dec = ... # valeur du décalage (exposant) : on sait s'il est positif ou négatif
        return ..., dec
```

Q6 – Copier-coller et compléter les `...` de la fonction `ecriture_scientifique(L_G, L_D)` ainsi décrite, et la tester sur les trois exemples donnés au-dessus de la trame.

On rappelle que l'écriture d'un nombre en *float* (virgule flottante, en simple précision) sur 32 bits est, d'après la norme IEEE 754 sous la forme $(s, e, m)_2$ avec :

- *s* (bit de poids fort) : bit de signe. 1 si le nombre est négatif, 0 sinon.
  Remarque : comme nous l'avons vu au-dessus de la question **Q3**, `int(booléen)` vaut
  $\begin{cases} 0 \text{ si booléen} == \text{True} \\ 1 \text{ si booléen} == \text{False} \end{cases}$
  Or, `x<0` est un test logique, renvoie donc un résultat booléen.
  → Ainsi, plutôt que d'envisager une structure `if x<0 : ... else : ...`, on peut écrire
- *e* l'exposant, codé en binaire par excès sur 8 bits
- *m* la mantisse, codée en binaire sur 23 bits. Si nécessaire, on complètera à droite avec des 0 pour atteindre 23 bits d'expression. Nous n'avons pas écrit de fonction permettant de compléter à gauche, mais vous pourrez vous inspirer du code que vous avez écrit en **Q2**.

Q7 – En vous appuyant sur les fonctions codées en **Q4**, **Q5** et **Q6**, écrire la fonction `into_float8(x)` qui prenne en argument un nombre *x* et qui le convertisse en *float* sur 32 bits.

Tests :

```
print(into_float8(10.25))
    # (0, [1, 0, 0, 0, 0, 0, 1, 0], [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
print(into_float8(0.625))
    # (0, [0, 1, 1, 1, 1, 1, 1, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
print(into_float8(-5))
    # (1, [1, 0, 0, 0, 0, 0, 0, 1], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
print(into_float8(-1/3))
    # (1, [0, 1, 1, 1, 1, 1, 0, 1], [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
```