Lycée Les 3 Sources **Bourg-Lès-Valence**
NSI 1re – Thème 1 Année **2024-25**

Python – Arithmétique en binaire

Préparation

Le fichier `arithmetique.py` se trouve dans le dossier de votre groupe dans `Echange`. Il contient le squelette des différentes fonctions de cette feuille. Vous pouvez le recopier dans vos documents pour compléter les fonctions à trous. Ou alors, vous pouvez créer une nouvelle feuille et essayer d'écrire intégralement les fonctions, en allant voir éventuellement le fichier si vous êtes bloqué.
L'objectif est de programmer l'addition et le complément à 2 sur les nombres binaires. Pour cela, vous devrez respecter les contraintes suivantes :

- Les nombres binaires sont représentés par des chaînes de caractères, comme `"0111"`.
- Vous ne pouvez utiliser le symbole + uniquement pour concaténer des chaînes de caractères.
- Même si pour les tests, vous pouvez vous restreindre aux nombres représentés sur 4 bits, vos fonctions doivent marcher pour un nombre arbitraire de bits.
- On notera généralement `nb_bin` les nombres binaires.

Pour plusieurs fonctions, vous aurez besoin de parcourir et de construire les chaînes de droite à gauche. Vous pouvez vous inspirer des deux fonctions ci-dessous, où `nb_bin` est un nombre en binaire :

```python
def copie_inverse1(nb_bin):
    res = ""
    i = len(nb_bin)
    while i > 0:
        i = i - 1 # laisser au début de la boucle
        res = nb_bin[i] + res
    return res

def copie_inverse2(nb_bin):
    res = ""
    for i in range(len(nb_bin)-1, -1, -1):
        res = nb_bin[i] + res
    return res
```

Lorsqu'il faut convertir un chiffre en un texte, ou un texte en chiffre, vous pouvez utiliser les fonctions suivantes :

```python
>>> str(0)   # chiffre -> texte
'0'
>>> str(1)
'1'
>>> int("1")   # texte -> chiffre
1
>>> int("0")
0
```

Échauffement

Exercice 1 : Écrire une fonction `conversion(nb)` qui prend un entier `nb` et renvoie son écriture en binaire.

$19 : 2 = 9$ reste 1
$9 : 2 = 4$ reste 1
$4 : 2 = 2$ reste 0
$2 : 2 = 1$ reste 0
$1 : 2 = 0$ reste 1
$1\,0\,0\,1\,1$

```
>>> conversion(13)
'1101'
>>> conversion(6)
'110'
```

Exercice 2 : Écrire une fonction `deconversion(nb_bin)` qui prend un nombre en binaire et renvoie la valeur décimale.

$0 \times 2 + 1 = 1$
$1 \times 2 + 0 = 2$
$2 \times 2 + 0 = 4$
$4 \times 2 + 1 = 9$
$9 \times 2 + 1 = 19$

```
>>> deconversion("1101")
13
>>> deconversion("110")
6
```

Fonctions intermédiaires

Afin de pouvoir implémenter l'addition et le complément à 2, il faut d'abord réaliser quelques fonctions qui seront nécessaires.

Exercice 3 : Écrire une fonction `complement_a_1(nb_bin)` qui renvoie le complément à 1 de `nb`. C'est-à-dire l'inverse bit à bit.

```
>>> complement_a_1("0111")
'1000'
>>> complement_a_1("1011")
'0100'
```

Exercice 4 : Écrire une fonction `ajouter_1(nb_bin)` qui renvoie le nombre binaire correspondant à `nb + 1`. Par contre, le nombre de bits n'est pas augmenté. S'il reste une retenue à la fin, elle n'est pas ajoutée à gauche. Il faut parcourir les bits de droite à gauche.
Vous pouvez utiliser une variable `retenue` que vous initialiserez à `"1"` avant de rentrer dans la boucle. On peut remarquer que tant qu'il y a une retenue, il faut modifier les bits de `nb_bin`, alors qu'on change plus rien dès qu'il n'y a plus de retenue.

```
>>> ajouter_1("0000")
'0001'
>>> ajouter_1("1011")
'1100'
>>> ajouter_1("1111")
'0000'
```

L'addition s'il vous plaît

Pour faire l'addition en binaire, il faut parcourir les deux nombres de droite à gauche en faisant l'addition bit à bit, en tenant compte d'un éventuelle retenue. Le tableau indique le résultat `b3` et la retenue `r2` obtenue en fonction des deux bits `b1` et `b2` additionnés et de la retenue initiale `r1`.

b1	b2	r1	b3	r2
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Exercice 5 : Écrire une fonction `addition(nb1, nb2)` qui renvoie la somme des nombres binaires `nb1` et `nb2`. S'il reste une retenue à la fin, elle n'est pas ajoutée à gauche. Le résultat a le même nombre de bits que `nb1` et `nb2`.

Il pourrait être judicieux de mettre l'assertion `assert len(nb1) == len(nb2)` en début de fonction pour vérifier que les deux nombres ont bien le même nombre de bits.
Vous pouvez vous inspirer de `ajouter_1(nb_bin)` et utiliser une variable `retenue`.

```
>>> addition("0101", "0011")
'1000'
>>> addition("1101", "0010")
'1111'
>>> addition("1001", "1000")
'0001'
```

Complément à 2

Exercice 6 : Écrire une fonction `complement_a_2(nb_bin)` qui renvoie le complément à 2 du nombre binaire `nb_bin`.
Pensez à vous inspirer des fonctions d'exemple.

```
>>> complement_a_2("0011")
'1101'
>>> complement_a_2("1101")
'0011'
```

Exercice 7 : Écrire une fonction `deconversion_a_2(nb)` qui renvoie l'entier relatif correspondant au nombre binaire `nb` codé avec le complément à 2. Il faut regarder le signe du nombre pour déterminer s'il faut utiliser le complément à 2 avant de faire la conversion vers le binaire et ensuite appliquer le signe si nécessaire.
Vous pouvez vous inspirer de la fonction `deconversion` précédemment définie.

```
>>> deconversion_a_2("0011")
3
>>> deconversion_a_2("1101")
-3
```

Exercice 8 : Écrire une fonction `conversion_a_2(nb, k)` qui renvoie le nombre binaire en complément à 2 sur `k` correspondant à l'entier relatif `nb`. Il faut regarder le signe de `nb`, le transformer en nombre entier s'il ne l'est pas encore, le convertir en binaire et utiliser le complément à 2 s'il était négatif.
Vous pouvez vous inspirer de la fonction `conversion` précédemment définie.

```
>>> conversion_a_2(3, 4)
'0011'
>>> conversion_a_2(-3, 4)
'1101'
```

Pour aller plus loin

Exercice 9 : Afin de tester vos fonctions, écrire une nouvelle fonction `tests(a, b, c, d, k)` qui affiche toutes les sommes $n_1 + n_2$ avec $a \leq n_1 < b$ et $c \leq n_2 < d$, en utilisant des nombres binaires en `k` bits en complément à 2.
Chaque ligne devra afficher les valeurs n_1, n_2, n_1 en binaire, n_2 en binaire, $n_1 + n_2$ en binaire et $n_1 + n_2$.
La somme en base 10 doit être obtenue depuis une déconversion de la valeur binaire.

```
>>> tests(-2, 3, -2, 3, 4)
-2 -2 1110 1110 1100 -4
-2 -1 1110 1111 1101 -3
-2 0 1110 0000 1110 -2
-2 1 1110 0001 1111 -1
...
2 0 0010 0000 0010 2
2 1 0010 0001 0011 3
2 2 0010 0010 0100 4
```

Exercice 10 : Écrire une fonction `multiplication(nb1, nb2)` qui renvoie le produit des nombre positifs `nb1` par `nb2`.

```
>>> multiplication("1101", "1011")
'10001111'
>>> multiplication("0011", "0011")
'00001001'
>>> multiplication("1111", "1111")
'11100001'
```

Exercice 11 : Écrire une fonction `multiplication_a_2(nb1, nb2)` qui renvoie le produit des nombre relatifs `nb1` par `nb2`. Le résultat sera exprimé à l'aide du complément à 2 sur le nombre suffisant de bits.

```
>>> multiplication_a_2("0011", "1001")
'11101011'
>>> multiplication_a_2("1011", "0110")
'11100010'
>>> multiplication_a_2("1111", "1111")
'00000001'
```