# V - TP 2 Manipulations de bits

## 1 Complément à 2 en python

L'objectif de cet partie est d'écrire en python un ensemble de fonction permettant de manipuler les représentations en complément à 2 des entiers relatifs.

**Question 1.** **1.** La fonction `combien_bits` prend en argument un entier `n` positif renvoie le nombre de bits nécessaires à l'écriture en base 2 de `n`.

Pour cela, on compte le nombre de fois successives que l'on doit diviser `n` par 2 avant d'obtenir un nombre inférieur ou égal à 1.

Recopier et compléter le code de la fonction `combien_bits`.

Code python

```
def combien_bits(n):
    """ int -> int
    n est un entier positif
    Renvoie le nombre de bits nécessaires à l'écriture en base 2 de n
    """
    # si n <= 1 alors il s'écrit sur 1 bit
    N = 1
    while n > 1:
        N = ...
        n = ...
    return ...
```

**2.** La fonction `base2` prend en argument un entier `n` positif et renvoie la **liste** correspondant à l'écriture en base 2 de `n`. Le bit de poids fort de `n` est le premier élément de la liste.

On rappelle que l'on obtient la liste des chiffres de `n` dans son écriture en base 2 en calculant successivement son quotient et son reste dans la division euclidienne de `n` par 2. Les restes successifs correspondant aux chiffres de `n` dans son écriture en base 2 **de la droite vers la gauche**.

Recopier et compléter le code de la fonction `combien_bits`.

Code python

```
def base2(n):
    """ int -> [int]
    n est un entier positif
    Renvoie l'écriture en base 2 de n """
    N = combien_bits(n)
    bits = [-1 for i in range(N)]
    for i in range(...):
        n, b = n//2, n%2
        bits[...] = ...
    return ...
```

On donne le code de la fonction `base10` qui prend en argument une liste de `bits` et qui renvoie l'entier positif dont l'écriture en base 2 est `bits`. On pourra utiliser cette fonction sans justification supplémentaire dans la suite du TP.

Code python

```python
def base10(bits):
    """ [int] -> int """
    N = len(bits)
    puiss2 = 1
    n = 0
    for i in range(N-1, -1, -1):
        n = n + bits[i]*puiss2
        puiss2 = 2*puiss2
    return n
```

Code python

```python
print(base10([1, 0, 1]))
print(base10([1, 1, 0]))
```

Résultat

```
5
6
```

**3.** La fonction `inverser` prend en argument une liste de `bits` et renvoie la liste de bits où les `0` ont été remplacés par des `1` et vice-versa.

Votre fonction ne modifiera pas la liste `bits`.

Code python

```python
def inverser(bits):
    """ [int] -> [int]
    Inverse les bits de la liste """
    N = len(bits)
    inv = [0 for i in range(N)]
    # À compléter
    return inv
```

Code python

```python
print(inverser([1, 1, 0, 1]))
```

Résultat

```
[0, 0, 1, 0]
```

**4.** La fonction `ajouter1` prend en argument une liste de `bits` correspondant à un nombre `n` et renvoie la liste de bits correspondant au nombre `n + 1`.

Votre fonction ne modifiera pas la liste `bits`. La liste renvoyée sera de même taille que la liste initiale. Les dépassements de capacités seront ignorés.

Pour cela, on remarquera que pour ajouter 1 à un nombre en base 2 :

- on parcourt les chiffres **de la droite vers la gauche** ;
- si jamais le ième chiffre est `0` alors on le passe à `1` et on renvoie immédiatement le résultat ;
- sinon on met le ième chiffre à `0` et on continue ;
- si on a parcouru tous les chiffres on renvoie le résultat dans tous les cas.

Code python

```python
def ajouter1(bits):
    """ [int] -> [int] """
    N = len(bits)
    plusun = [b for b in bits]
    # À compléter
    return plusun
```

Code python

```
print(ajouter1([1, 1, 0, 0]))
print(ajouter1([1, 0, 1, 1]))
print(ajouter1([1, 1, 1, 1]))
```

Résultat

```
[1, 1, 0, 1]
[1, 1, 0, 0]
[0, 0, 0, 0]
```

**5.** À l'aide des fonctions écrites précédemment, écrire une fonction `complement_a_2` qui prend en argument un entier **relatif** `n` et un entier `N` et qui renvoie la liste de bits correspondant à sa représentation en complément à 2 sur `N` bits.

On pourra utiliser sans justification supplémentaire la fonction `ecriture_base2` qui prend en argument deux entiers positifs `n` et `N` et qui renvoie l'écriture en base 2 de `n` sur `N` bits (en complétant à gauche l'écriture de `n` par des zéros).

Code python

```
def ecriture_base2(n, N):
    bits = base2(n)
    taille = len(bits)
    return [0]*(N - taille) + bits
```

Code python

```
def complement_a_2(n, N):
    """ int, int -> [int]
    Renvoie l'écriture en complément à 2 de n sur N bits """
    pass
```

Code python

```
print(complement_a_2(-1, 3))
print(complement_a_2(3, 3))
```

Résultat

```
[1, 1, 1]
[0, 1, 1]
```

La fonction `base10_cp2` prend en argument une liste de `bits` correspondant à la représentation à l'aide du complément à 2 de `n` sur `N` bits et renvoie le nombre `n` correspondant. On ne demande pas de comprendre cette fonction. Elle pourra être utilisée sans justification supplémentaire lors de vos tests.

Code python

```
def base10_cp2(bits, N):
    if bits[0] == 0:
        return base10(bits)
    return -2**N + base10(bits)
```

Code python

```
print(base10_cp2([0, 1, 1], 3))
print(base10_cp2([1, 1, 1], 3))
print(base10_cp2([1, 0, 0], 3))
```

Résultat

```
3
-1
-4
```

**6.** **a.** Écrire une fonction `somme` qui prend en argument deux listes `bits1` et `bits2` correspondant à la représentation en complément à 2 sur `N` bits de deux entiers `n1` et `n2` et qui renvoie la liste correspondant à la représentation en complément à 2 sur `N` bits de l'entier `n1 + n2`. Les dépassements de capacités seront ignorés.

Pour cela, vous utiliserez l'algorithme suivant :

- un tableau `s` rempli uniquement de `0` et de même taille que `bits1` et `bits2` est initialisé ;
- initialement il n'y a aucune `retenue` ;
- on parcourt les chiffres de `bits1` et `bits2` de la **droite vers la gauche** :
  - on détermine le $i$-ème chiffre de `s` en ajoutant le $i$-ème chiffre de `bits1` au $i$-ème chiffre de `bits2` et à la `retenue` éventuelle ;
  - si une retenue est générée on met `retenue` à `1`, sinon à `0`

Recopier et compléter le code de la fonction `somme` ci-dessous.

Code python

```
def somme(bits1, bits2):
    """ [int], [int] -> [int]
    bits1 et bits2 sont de même taille
    renvoie la somme chiffre à chiffre de bits1 avec bits2 en
    ↪ gérant les retenues """
    N = len(bits1)
    s = [...]
    retenue = ...
    for i in range(...):
        # À compléter
    return s
```

Code python

```
N = 8
b1 = complement_a_2(42, N)
b2 = complement_a_2(-14, N)
s = somme(b1, b2)
print(base10_cp2(s, N))
```

Résultat

```
28
```

**b.** Afficher toutes les représentations en complément à 2 des entiers pouvant s'écrire sur $N = 8$ bits. À côté de chaque représentation sera indiqué l'entier correspondant.

Résultat

```
0 [0, 0, 0, 0, 0, 0, 0, 0]
...
-99 [1, 0, 0, 1, 1, 1, 0, 1]
...
-1 [1, 1, 1, 1, 1, 1, 1, 1]
```

**c.** Vérifier les réponses apportées à l'exercice 2 de la planche 1 à l'aide des fonctions écrites dans cette partie.

## 2 Minuscules et majuscules

Étant donnée une chaîne de caractères, l'objectif est de produire une copie de cette chaîne convertie en « minuscules ».
Par exemple, la chaîne `"Les algorithmes de Bellman-Ford et de Dijkstra"` sera convertie en `"les algorithmes de bellman-ford et de dijkstra"`.

**Remarque.** On rappelle qu'un caractère est encodé par un nombre entier que l'on obtient avec la fonction `ord`. Par exemple `ord('A')` est évalué à 65. Les codes des caractères alphabétiques non accentués en majuscule se suivent : `ord('A')` vaut 65, `ord('B')` vaut 66, etc.
Réciproquement, étant donné un entier positif, on obtient le caractère encodé par cet entier avec la fonction `chr`. Par exemple `chr(65)` est évalué à `'A'`.

**Question 2.** **1.** Que vaut `ord(c_min) - ord(c_maj)` lorsque :

**a.** `c_min, c_maj = "a", "A"`
**b.** `c_min, c_maj = "f", "F"`
**c.** `c_min, c_maj = "u", "U"`
**d.** `c_min, c_maj = "z", "Z"`

**2.** Compléter la fonction `minuscule` qui prend en paramètre une chaîne de caractères `chaine` et renvoie une nouvelle chaîne qui est la copie de la chaîne `chaine` convertie en minuscules.

On se limite à convertir les caractères allant de `'A'` à `'Z'`, les autres caractères étant laissés inchangés.

Code python

```
def minuscule(chaine):
    copie = ...
    for caractere in chaine:
        code = ...
        if ...("A") <= ... <= ...:
            code = ...
        copie = ...
    return ...

# Tests

assert minuscule("ABCDE") == "abcde"
chaine = "Les algorithmes de Bellman-Ford et de Dijkstra."
assert minuscule(chaine) == "les algorithmes de bellman-ford et de
↪ dijkstra."
```

## 3 Chiffrement de césar

Le chiffrement de César transforme un message en changeant chaque lettre par une autre obtenue par décalage circulaire dans l'alphabet de la lettre d'origine. Par exemple, avec un décalage de 3, le `'A'` se transforme en `'D'`, le `'B'` en `'E'`, ..., le `'X'` en `'A'`, le `'Y'` en `'B'` et le `'Z'` en `'C'`.
Les autres caractères (`'!'`, `'?'`...) ne sont pas transformés et sont simplement recopiés tels quels dans le message codé.
Dans cet exercice, nous considérerons que les messages n'utilisent que des lettres majuscules, non accentuées.

On fournit les deux fonctions qu'il est possible d'utiliser sans justification supplémentaire.

- `indice` : renvoie l'indice dans l'alphabet d'une lettre majuscule en commençant à 0.
- `majuscule` : renvoie la lettre majuscule d'indice donné.

Code python

```python
def indice(caractere):
    "Renvoie l'indice de `caractere` qui doit être une majuscule"
    return ord(caractere) - ord('A')

def majuscule(i):
    """Renvoie la majuscule d'indice donnée
    majuscule(0) renvoie 'A'
    majuscule(25) renvoie 'Z'
    """
    return chr(ord('A') + i)
```

**Remarque.** Pour opérer un décalage circulaire d'un indice, on utilise l'opération modulo qui renvoie un résultat de inclus à exclu.
Par exemple, pour décaler de 8 la lettre 'Z'

Code python

```python
print(indice('Z'))
print(25 + 8)
print(33 % 26)
print(majuscule(7))
```

Résultat

```
25
33
7
H
```

**Question 3.** Écrire la fonction `cesar` qui prend en paramètres une chaine de caractères `message` et un nombre entier `decalage` et renvoie le nouveau message chiffré avec le chiffre de César utilisant ce `decalage`.
On constate que pour déchiffrer un message, il suffit d'utiliser la clé opposée à celle du chiffrement.

Code python

```python
def cesar(message, decalage):
    resultat = ''
    for caractere in ...:
        if 'A' <= caractere <= ...:
            i = ...(caractere)
            i = (i + ...) ...
            resultat += ...(i)
        else:
            resultat += ...
    return ...

# Tests

assert cesar('HELLO WORLD!', 5) == 'MJQQT BTWQI!'
assert cesar('MJQQT BTWQI!', -5) == 'HELLO WORLD!'

assert cesar('BONJOUR LE MONDE !', 23) == 'YLKGLRO IB JLKAB !'
assert cesar('YLKGLRO IB JLKAB !', -23) == 'BONJOUR LE MONDE !'
```